

UNICAMP

Universidade Estadual de Campinas



Faculdade de Engenharia Elétrica e de Computação

TESE DE DOUTORADO

WAGNER LUIZ ALVES DE OLIVEIRA

UMA ABORDAGEM PARA A MODELAGEM DE SISTEMAS DIGITAIS

ORIENTADORES: PROF. DR. NORIAN MARRANGHELLO

PROF. DR. FURIO DAMIANI

COMISSÃO JULGADORA: NORIAN MARRANGHELLO – FEEC/UNICAMP – PRESIDENTE

AKEBO YAMAKAMI – FEEC/UNICAMP

ALEXANDRE CÉSAR RODRIGUES DA SILVA – FEIS/UNESP

ALICE MARIA BASTOS HUBINGER TOKARNIA – FEEC/UNICAMP

JOSÉ RAIMUNDO DE OLIVEIRA – FEEC/UNICAMP

WANG JIANG CHAU – EPUSP

RESUMO

O projeto de sistemas digitais alcançou um elevado grau de complexidade, inviabilizando sua consecução sem o uso de ferramentas de CAD. O ponto de partida de tais ferramentas consiste numa visão conceitual do sistema pretendido (dada por um ou mais modelos conceituais), a qual é capturada para tratamento computacional por uma ou mais linguagens de especificação. Várias dessas linguagens foram desenvolvidas visando capturar tantas características de *hardware* e de *software* quanto possível, de acordo com diferentes metodologias de projeto. Rede de Petri é uma classe de modelos conceituais utilizada na modelagem de diversos tipos de sistemas computacionais paralelos. Algumas extensões de rede de Petri foram propostas visando à descrição, de forma tão acurada quanto possível, de características de sistemas digitais. Entretanto, somente duas destas extensões possuem um número maior de características necessárias à descrição integral de tais sistemas. O presente trabalho apresenta uma extensão de rede de Petri desenvolvida para superar as limitações das demais extensões na representação de sistemas digitais. O trabalho apresenta, também, uma metodologia de coprojeto *hardware/software* na qual a extensão proposta pode ser usada como linguagem de modelagem interna. Tal plataforma visa a descrição, simulação, análise, validação e síntese em alto nível de sistemas digitais embutidos.

Palavras-Chave: Redes de Petri, Sistemas Digitais, Sistemas Digitais Embutidos, Coprojeto *Hardware/Software*.

ABSTRACT

Digital system design has reached a high degree of complexity that prevents its realization without CAD tools. The starting point of such tools consists on a conceptual view of the intended system (given by one or more conceptual models), which is captured for computational handling by one or more specification languages. Several of such languages were developed aiming to capture as many hardware and software characteristics as possible, according to different design methodologies. Petri net is a class of conceptual models for parallel system modeling. Some Petri net extensions have been proposed aiming at describing digital systems characteristics as accurately as possible. However, only two of them have nearly all features needed to describe such systems in full. This work presents a Petri net extension developed to overcome the restrictions for digital system modeling through Petri net extensions. A hardware/software codesign methodology in which the proposed extension can be used as the internal modeling language is presented as well. Such a framework aims embedded digital system description, simulation, analysis, validation, and high-level synthesis.

Keywords: Petri Nets, Digital Systems, Digital Embedded Systems, Hardware/Software Codesign.

**DEDICO ESTE TRABALHO AOS MEUS PAIS, OSMAR E CONCEIÇÃO,
MEUS IRMÃOS, SOLANGE E TADEU,
MEUS SOGROS, JOÃO E APARECIDA,
E À PESSOA QUE ESTARÁ SEMPRE AO MEU LADO, MARISA.**

**AGRADEÇO AOS PROFESSORES FURIO DAMIANI E
NORIAN MARRANGHELLO POR TEREM ME DADO ESTA OPORTUNIDADE.**

**AGRADEÇO À FAPESP PELO APOIO PRESTADO
AO DESENVOLVIMENTO DO TRABALHO AQUI EXPOSTO.**

FIAT LUX

ÍNDICE

INTRODUÇÃO	1

PARTE I – SISTEMAS DIGITAIS

CAPÍTULO 1: SISTEMAS DIGITAIS	7
1.1. INTRODUÇÃO	7
1.2. EVOLUÇÃO HISTÓRICA DE METODOLOGIAS DE PROJETO RECENTES	10
1.2.1. CAPTURA E SIMULAÇÃO	10
1.2.2. DESCRIÇÃO E SÍNTESE	11
1.2.3. METODOLOGIAS EM NÍVEL DE SISTEMA	12
CAPÍTULO 2: MODELOS CONCEITUAIS	15
2.1. INTRODUÇÃO	15
2.2. TAXONOMIA DE MODELOS CONCEITUAIS COMPUTACIONAIS SEGUNDO GAJSKI	15
2.2.1. MODELOS DE ESTADO	15
2.2.2. MODELOS DE ATIVIDADES	17
2.2.3. MODELOS ESTRUTURAIS	18
2.2.4. MODELOS DE ESTRUTURAS DE DADOS	18
2.2.5. MODELOS HETEROGÊNEOS	19
2.3. TAXONOMIA E TERMINOLOGIA DE MODELOS PARA SISTEMAS DIGITAIS – VSI ALLIANCE	21
2.3.1. DEFINIÇÃO DA TAXONOMIA PROPOSTA	21
2.3.1.1. EIXO DE PRECISÃO TEMPORAL	23
2.3.1.2. EIXO DE PRECISÃO DE DADOS	23
2.3.1.3. EIXO DE PRECISÃO FUNCIONAL	24
2.3.1.4. EIXO DE PRECISÃO ESTRUTURAL	24
2.3.1.5. EIXO DO NÍVEL DE ABSTRAÇÃO DE PROGRAMAÇÃO (EIXO DE SOFTWARE)	24
2.3.2. CONCEITOS GERAIS DE MODELAGEM	25
2.3.2.1. CLASSE DE MODELOS PRIMÁRIOS	26
2.3.2.1.1. MODELO COMPORTAMENTAL	26
2.3.2.1.2. MODELO FUNCIONAL	27
2.3.2.1.3. MODELO ESTRUTURAL	27
2.3.2.2. CLASSE DE MODELOS ESPECIALIZADOS	28
2.3.2.2.1. MODELOS DE DESEMPENHO	28
2.3.2.2.2. MODELOS DE INTERFACE	29
2.3.2.2.3. MODELOS DE NÍVEL MISTO	29
2.3.3. MODELOS PARA SISTEMAS DIGITAIS	30
2.3.3.1. MODELOS DE SISTEMA	30
2.3.3.1.1. ESPECIFICAÇÃO EXECUTÁVEL	30
2.3.3.1.2. MODELOS DE EQUAÇÃO MATEMÁTICA	31
2.3.3.1.3. MODELOS ALGORÍTMICOS	31
2.3.3.2. MODELOS ARQUITETURAIS	32
2.3.3.2.1. MODELO DE DESEMPENHO BASEADO EM <i>TOKEN</i>	32
2.3.3.2.2. MODELOS DE DESEMPENHO COMPORTAMENTAL ABSTRATO	33
2.3.3.2.3. PRIMITIVAS DE TAREFA COM GRAFO DE FLUXO DE DADOS	34
2.3.3.2.4. MODELOS ISA	35
2.3.3.3. MODELOS DE <i>HARDWARE</i>	35
2.3.3.3.1. MODELO COMPORTAMENTAL DETALHADO	36
2.3.3.3.2. MODELOS RTL	36
2.3.3.3.3. MODELOS EM NÍVEL LÓGICO	37
2.3.3.3.4. MODELOS EM NÍVEL DE PORTAS	38
2.3.3.3.5. MODELOS EM NÍVEL DE CHAVES	38
2.3.3.3.6. MODELOS EM NÍVEL DE CIRCUITO	39

CAPÍTULO 3: ARQUITETURAS	41
3.1. INTRODUÇÃO	41
3.2. ARQUITETURAS PARA APLICAÇÕES ESPECÍFICAS	41
3.2.1. ARQUITETURAS PARA CONTROLADORES SIMPLES	41
3.2.2. ARQUITETURAS <i>DATAPATH</i>	42
3.3. ARQUITETURAS PARA PROCESSADORES DE PROPÓSITO GERAL	43
3.3.1. FSM COM <i>DATAPATH</i>	43
3.3.2. ARQUITETURAS CISC/RISC	44
3.3.2.1. CISC	45
3.3.2.2. RISC	46
3.4. ARQUITETURAS PARA PROCESSAMENTO PARALELO	49
3.4.1. MÁQUINAS SIMD	50
3.4.1.1. MÁQUINAS SM-SIMD	51
3.4.1.2. MÁQUINAS DM-SIMD	52
3.4.2. MÁQUINAS MISD	54
3.4.3. MÁQUINAS MIMD	54
3.4.3.1. MÁQUINAS SM-MIMD	55
3.4.3.2. MÁQUINAS DM-MIMD	57
CAPÍTULO 4: LINGUAGENS DE ESPECIFICAÇÃO	59
4.1. INTRODUÇÃO	59
4.2. VHDL	61
4.3. VERILOG	64
4.4. CSP	66
4.5. STATECHARTS	68
4.6. SDL	69
4.7. SILAGE	72
CAPÍTULO 5: COPROJETO <i>HARDWARE/SOFTWARE</i> DE SISTEMAS EMBUTIDOS	75
5.1. INTRODUÇÃO	75
5.2. ETAPAS DE COPROJETO	77
5.3. ESPECIFICAÇÃO	81
5.4. ALOCAÇÃO, PARTICIONAMENTO <i>HARDWARE/SOFTWARE</i> E ESCALONAMENTO	83
5.5. TÉCNICAS DE VERIFICAÇÃO	92

PARTE II – REDES DE PETRI

CAPÍTULO 6: REDES DE PETRI	97
6.1. INTRODUÇÃO	97
6.2. COMPOSIÇÃO BÁSICA DE REDES DE PETRI	98
6.3. REDES DE PETRI ELEMENTARES	99
6.4. EXEMPLO DE MODELAGEM DE UMA BIBLIOTECA EM RPE	101
CAPÍTULO 7: REDES DE PETRI LUGAR/TRANSIÇÃO	103
7.1. REGRAS DE FUNCIONAMENTO	103
7.2. PROPRIEDADES GERAIS DE UMA RP LUGAR/TRANSIÇÃO	107
7.2.1. LIMITABILIDADE	107
7.2.2. VIVACIDADE	107
7.2.3. REINICIABILIDADE	108
7.3. EXEMPLO DE MODELAGEM DE UMA BIBLIOTECA EM RP L/T	110

CAPÍTULO 8: REDES DE PETRI TEMPORIZADAS	111
8.1. TIPOS DE REDES DE PETRI TEMPORIZADAS	111
8.2. RP TEMPORIZADAS COM VALORES DE TEMPO ASSOCIADOS ÀS TRANSIÇÕES	112
8.2.1. DESCRIÇÃO DAS RP TEMPORIZADAS DE MERLIN	112
8.2.2. DESCRIÇÃO DAS RP TEMPORIZADAS DE RAMCHANDANI	112
8.3. RP TEMPORIZADAS COM VALORES DE TEMPO ASSOCIADOS AOS LUGARES	113
8.4. RP TEMPORIZADAS DE MERLIN (RPTM)	113
8.5. MÉTODO DE ENUMERAÇÃO DE CLASSES DE ESTADOS PARA RPTM	117
CAPÍTULO 9: REDES DE PETRI COLORIDAS	123
9.1. INTRODUÇÃO	123
9.2. DEFINIÇÃO FORMAL	129
9.3. PROPRIEDADES DINÂMICAS	134
9.4. GRAFO DE DISPARO	137

PARTE III – REDE DE PETRI PARA SISTEMAS DIGITAIS

CAPÍTULO 10: REDES DE PETRI E SISTEMAS DIGITAIS	145
10.1. INTRODUÇÃO	145
10.2. REDES DE PETRI NO PROJETO DE SISTEMAS DIGITAIS	147
10.3. PLACE CHART NETS	149
10.4. REDE DE PETRI PARA SISTEMAS EMBUTIDOS	151
10.4.1. DESCRIÇÃO FORMAL	152
10.4.2. EXEMPLO DE MODELAGEM	154
CAPÍTULO 11: REDE DE PETRI PARA SISTEMAS DIGITAIS (RPSD)	157
11.1. INTRODUÇÃO	157
11.2. EXTENSÃO RPSD	159
11.2.1. DESCRIÇÃO FUNCIONAL	159
11.2.1.1. LUGARES	161
11.2.1.1.1. TIPO I: ESTADO LOCAL	161
11.2.1.1.2. TIPO II: ARMAZENAMENTO	164
11.2.1.1.3. TIPO III: FUNCIONAL	166
11.2.1.2. RAMOS	178
11.2.1.3. TRANSIÇÕES	180
11.2.1.4. INICIALIZAÇÃO DE LUGARES	183
11.2.2. DESCRIÇÃO FORMAL	186
11.3. RPSD E CARACTERÍSTICAS DE MODELAGEM DE SISTEMAS DIGITAIS	192
11.4. EXEMPLO DE MODELAGEM	202
CAPÍTULO 12: EQUIVALÊNCIA ENTRE REDE DE PETRI PARA SISTEMAS DIGITAIS E OUTRAS EXTENSÕES DE REDES DE PETRI	213
12.1. INTRODUÇÃO	213
12.2. EQUIVALÊNCIA DE LUGARES	213
12.2.1. TIPO I: ESTADO LOCAL	213
12.2.2. TIPO II: ARMAZENAMENTO	221
12.2.3. TIPO III: FUNCIONAL	224
12.3. EQUIVALÊNCIA DE TRANSIÇÕES E RAMOS	239
12.3.1. CONEXÕES COM LUGARES DO TIPO I	240
12.3.2. CONEXÕES COM LUGARES DO TIPO II	246
12.3.3. CONEXÕES COM LUGARES DO TIPO III	254

CAPÍTULO 13: MÉTODOS DE ANÁLISE BASEADOS EM SIMULAÇÃO	275
13.1. INTRODUÇÃO	275
13.2. ALGORITMO DE SIMULAÇÃO PARA REDE DE PETRI PARA SISTEMAS DIGITAIS	275
13.3. ANÁLISE DE PROPRIEDADES.....	291
13.3.1. LIMITABILIDADE	291
13.3.2. REINICIABILIDADE	292
13.3.3. VIVACIDADE	292
13.3.4. TRANSIÇÕES NÃO DISPARADAS.....	298
13.3.5. <i>TOKENS</i> NÃO UTILIZADOS EM LUGARES DO TIPO I.....	299
13.3.6. ENDEREÇOS NÃO ACESSADOS EM LUGARES DO TIPO II.....	299
13.3.7. QUANTIDADE DE ACESSOS EM LUGARES DO TIPO II.....	300
13.3.8. VERIFICAÇÃO DA ADEQUAÇÃO DAS LATÊNCIAS EM LUGARES DO TIPO II.....	301
13.3.9. VERIFICAÇÃO DE SUBDIMENSIONAMENTO EM LUGARES DO TIPO III.....	302
13.3.10. VERIFICAÇÃO DE SUPERDIMENSIONAMENTO EM LUGARES DO TIPO III.....	303
 CAPÍTULO 14: MÉTODOS DE ANÁLISE BASEADOS EM INVARIANTES.....	305
14.1. INTRODUÇÃO	305
14.2. ANÁLISE DE INVARIANTES EM REDES DE PETRI.....	305
14.3. APLICABILIDADE DA ANÁLISE DE INVARIANTES EM RPSD.....	308
 CAPÍTULO 15: METODOLOGIA DE COPROJETO <i>HARDWARE/SOFTWARE</i> BASEADA EM RPSD	311
15.1. INTRODUÇÃO	311
15.2. FLUXO DE PROJETO	311
 CONCLUSÕES	317
 REFERÊNCIAS BIBLIOGRÁFICAS / INTERNET.....	319
 APÊNDICE I: TÓPICOS DA TEORIA DE GRAFOS.....	325
 APÊNDICE II: VALIDAÇÃO DE RPSD COMO REDE DE PETRI.....	333

ÍNDICE DE FIGURAS

Figura 2.1. Eixos da taxonomia [4]	22
Figura 2.2. Símbolos para indicação dos níveis de resolução	26
Figura 2.3. Níveis de resolução – modelo comportamental	26
Figura 2.4. Níveis de resolução – modelo funcional	27
Figura 2.5. Níveis de resolução – modelo estrutural	27
Figura 2.6. Níveis de resolução – modelos de desempenho	28
Figura 2.7. Níveis de resolução – modelos de interface	29
Figura 2.8. Níveis de resolução – modelos de nível misto	29
Figura 2.9. Níveis de resolução – especificação executável	30
Figura 2.10. Níveis de resolução – modelos de equação matemática	31
Figura 2.11. Níveis de resolução – modelos algorítmicos	31
Figura 2.12. Níveis de resolução – modelo de desempenho baseado em <i>token</i>	32
Figura 2.13. Níveis de resolução – modelos de desempenho comportamental abstrato	33
Figura 2.14. Nível de resolução – primitivas de tarefa com grafo de fluxo de dados	34
Figura 2.15. Níveis de resolução – modelos ISA	35
Figura 2.16. Níveis de resolução – modelo comportamental detalhado	36
Figura 2.17. Níveis de resolução – modelos RTL	36
Figura 2.18. Níveis de resolução – modelos em nível lógico	37
Figura 2.19. Níveis de resolução – modelos em nível de portas	38
Figura 2.20. Níveis de resolução – modelos em nível de chaves	38
Figura 2.21. Níveis de resolução – modelos em nível de circuito	39
Figura 3.1. Arquiteturas para um controlador baseado em FSM	42
Figura 3.2. Exemplo de arquitetura FSMD	43
Figura 3.3. Arquitetura CISC com unidade de controle microprogramada	46
Figura 3.4. Arquitetura RISC com unidade de controle <i>hardwired</i>	47
Figura 3.5. Arquitetura básica de uma máquina SISD	49
Figura 3.6. Arquitetura básica de uma máquina SIMD	50
Figura 3.7. Arquitetura de uma máquina vetorial com 1 processador vetorial	51
Figura 3.8. Arquitetura genérica de uma máquina SIMD com memória distribuída	53
Figura 3.9. Arquitetura básica de uma máquina MISD	54
Figura 3.10. Arquitetura básica de uma máquina MIMD	55
Figura 3.11. Exemplos de estruturas de interconexão para máquinas SM-MIMD	56
Figura 3.12. Conexões hipercubo de 1, 2, 3 e 4 dimensões	58
Figura 4.1. Técnicas de sincronização em Statecharts	70
Figura 5.1. Metodologia genérica de coprojetado (adaptação de Kalavade e Lee [27])	78
Figura 5.2. CFG do particionamento espacial do sistema	89
Figura 5.3. Resultados gerados pelo algoritmo de particionamento temporal	91
Figura 6.1. Redes de Petri equivalentes: 2 lugares e 2 transições	98
Figura 6.2. Marcações distintas para uma RPE	100
Figura 6.3. Rede de Petri Elementar modelando uma biblioteca com 1 livro	101
Figura 6.4. Marcação decorrente da inicial após o disparo de <u>UsuárioPedeLivro</u>	101
Figura 6.5. Máquina de senhas da RPE da figura 6.3	102
Figura 6.6. Rede de Petri Elementar modelando uma biblioteca com 2 livros	102

Figura 7.1. Transição 2 habilitada e transição 1 desabilitada	104
Figura 7.2. Marcação alcançada com o disparo da transição 2	104
Figura 7.3. Matrizes de incidência direta (A) e reversa (B)	105
Figura 7.4. Rede L/T não limitada	107
Figura 7.5. Rede L/T não viva e sua máquina de senhas	108
Figura 7.6. Máquina de senhas da rede da figura 7.5.a: – rede viva para nova marcação inicial	108
Figura 7.7. Rede L/T limitada, viva e não reiniciável e sua máquina de senhas	109
Figura 7.8. Máquina de senhas da rede da figura 7.7.a com nova marcação inicial	109
Figura 7.9. RP L/T modelando uma biblioteca com 2 livros	110
Figura 8.1. Exemplo de RPTM	115
Figura 8.2. Transformação de estados em classes de estados	117
Figura 8.3. Grafo das classes de estados para a RPTM da figura 8.1	121
Figura 9.1. Modelo RPC de um sistema de banco de dados distribuído	124
Figura 9.2. Grafo de disparo para o sistema de banco de dados com 3 gerenciadores	137
Figura 9.3. Grafo de disparo com simetria para o SBD com 3 gerenciadores	140
Figura 10.1. Parte de uma PCN modelando um protocolo de comunicação	150
Figura 10.2. Rede de Petri elementar equivalente à PCN da figura 10.1.....	151
Figura 10.3. Modelo PNES para o procedimento de atendimento de chamadas	155
Figura 10.4. Refinamento do supermodo <u>Announcement</u> da figura 10.3	156
Figura 11.1. Elementos constituintes de RPSD.....	160
Figura 11.2. Componentes de um lugar tipo I – Estado Local	161
Figura 11.3. Seqüências de disparo de <u>process</u> : a) Cronologicamente; b) Aleatoriamente	163
Figura 11.4. Componentes de um lugar tipo II – Armazenamento	164
Figura 11.5. Transferência de dados entre lugares de armazenamento	166
Figura 11.6. RPSD com comportamento similar à rede da figura 11.5	167
Figura 11.7. Componentes de um lugar tipo III – Funcional	168
Figura 11.8. Formatação para rotina associada a um lugar do tipo III	171
Figura 11.9. Utilização de lugar funcional para comparação entre 2 números inteiros	175
Figura 11.10. Rede da figura 11.9 após o disparo de <u>Inserir1Argumento</u>	176
Figura 11.11. Seqüência de marcações para o lugar funcional da rede da figura 11.9	177
Figura 11.12. Componentes de uma transição	181
Figura 11.13. Diferentes representações para um lugar tipo I – Estado Local	184
Figura 11.14. Diferentes inicializações para um lugar tipo II – Armazenamento	185
Figura 11.15. Referência a conjuntos descritos na classe <u>anotações</u>	186
Figura 11.16. Concorrência orientada a dados	192
Figura 11.17. Concorrência orientada a controle	193
Figura 11.18. Instâncias hierárquicas do lugar <u>Comparador</u> da figura 11.9 (4ª marcação da figura 11.11).....	196
Figura 11.19. Modelo RPSD de um <i>pipeline</i> de 4 estágios, com transição preemptiva	198
Figura 11.20. Modelagem de interrupção em RPSD	200
Figura 11.21. Instâncias ativas do lugar <u>FiatLux</u> da RPSD da figura 11.20	200
Figura 11.22. Modelo RPSD do processador proposto	204
Figura 11.23. Formatos de instrução e modos de endereçamento do processador	206
Figura 12.1. Correspondência entre lugar tipo I para 1 <i>token</i> e lugares de RP L/T	214
Figura 12.2. Correspondência entre lugar tipo I para 2 <i>tokens</i> e lugares de RP L/T	216
Figura 12.3. Correspondência entre lugar tipo I para n <i>tokens</i> e lugares de RP L/T	217
Figura 12.4. Efeito da remoção de <i>tokens</i> sobre a RP L/T associada a um lugar do tipo I.....	220
Figura 12.5. Correspondência entre lugares tipo II para 1 <i>token</i> e lugares de RP L/T.....	222

Figura 12.6. Correspondência entre lugares tipo II para n <i>tokens</i> e lugares de RP L/T	223
Figura 12.7. Representação dos repositórios de um lugar funcional através de RP L/T.....	226
Figura 12.8. Lugar funcional com 3 parâmetros de entrada e 2 de saída	228
Figura 12.9. Lugar da figura 12.8 após a remoção do <i>sample</i> presente na linha 2 de R_o	229
Figura 12.10. Representação de uma rotina através de uma RPTM	231
Figura 12.11. Representações equivalentes para um lugar funcional	232
Figura 12.12. Representação completa de um lugar funcional	233
Figura 12.13. RPTM descrevendo uma conexão com ramo de entrada comum	242
Figura 12.14. RPTM descrevendo uma conexão com ramo de saída comum	245
Figura 12.15. RPTM descrevendo uma conexão com ramo de leitura	249
Figura 12.16. RPTM descrevendo uma conexão com ramo de escrita	250
Figura 12.17. RPTM descrevendo uma conexão com ramo coletor	257
Figura 12.18. RPTM descrevendo uma conexão com ramo fornecedor	267
Figura 12.19. RPTM descrevendo uma conexão com ramo preemptivo	271
Figura 13.1. Estrutura de dados INFO_LUGARES	277
Figura 13.2. Estrutura de dados INFO_TRANSICOES	281
Figura 13.3. Estrutura de dados GRAFO_CLASSES	282
Figura 13.4. Estrutura de dados CLASSES_PROVISORIAS	283
Figura 13.5. Fluxograma do algoritmo de simulação para RPSD	286
Figura 13.6. Fluxograma do algoritmo de verificação de reiniciabilidade	294
Figura 13.7. Fluxograma do algoritmo de verificação de vivacidade	296
Figura 13.8. Trecho da estrutura de dados INFO_LUGARES	304
Figura 14.1. Modelo em RP L/T do problema dos processos de leitura/escrita	306
Figura 15.1. Fluxo de projeto da metodologia proposta	311
Figura I.1. Grafos não orientado (a) e orientado (b).....	327
Figura I.2. Grafo parcial (a) e subgrafo (b).....	327
Figura I.3. Grafos rotulados	328
Figura I.4. Partição de grafos	328
Figura I.5. Exemplos de percursos	329
Figura I.6. Grafos não conexos	330
Figura I.7. Grafos s-conexos.....	330
Figura I.8. Grafos sf-conexos	330
Figura I.9. Grafos f-conexos.....	331
Figura I.10. Componentes f-conexas	331
Figura II.1. Edição de uma rede no programa JARP	336
Figura II.2. Simulação de uma rede no programa JARP	337
Figura II.3. Estado da rede da figura II.2 caso <u>T1</u> dispare	338
Figura II.4. Estado da rede da figura II.2 caso <u>T3</u> dispare	338
Figura II.5. Modelo JARP correspondente à RPTM da figura 12.13	339
Figura II.6. Tela inicial do simulador ARP 2.4	340
Figura II.7. Carregamento da rede produzida pelo programa JARP	340
Figura II.8. Instruções de edição	341
Figura II.9. Edição da rede – adição de intervalos de disparo	342
Figura II.10. Compilação da rede.....	344
Figura II.11. Menu <u>aNalysis</u>	344
Figura II.12. Definição da marcação inicial M_0	345
Figura II.13. Análise de propriedades para a rede <u>R12-13.PN</u>	345

Figura II.14. Auxílio para compreensão dos termos utilizados	346
Figura II.15. Informações para a construção do grafo de classes	349
Figura II.16. Classes geradas	349
Figura II.17. Grafo de classes para a rede das figuras 12.13 / II.5	350
Figura II.18. Modelo JARP correspondente à RPTM da figura 12.14	352
Figura II.19. Grafo de classes para a rede das figuras 12.14 / II.18	355
Figura II.20. Modelo JARP correspondente à RPTM da figura 12.15	357
Figura II.21. Grafo de classes para a rede das figuras 12.15 / II.20	364
Figura II.22. Modelo JARP correspondente à RPTM da figura 12.16	365
Figura II.23. Grafo de classes para a rede das figuras 12.16 / II.22	374
Figura II.24. Modelo JARP correspondente à RPTM da figura 12.17	375
Figura II.25. Classe C_0	382
Figura II.26. Classe C_1	383
Figura II.27. Classe C_2	383
Figura II.28. Classe C_3	384
Figura II.29. Classe C_4	385
Figura II.30. Classe C_5	385
Figura II.31. Classe C_6	386
Figura II.32. Classe C_7	387
Figura II.33. Classe C_8	388
Figura II.34. Classe C_9	388
Figura II.35. Classe C_{10}	388
Figura II.36. Classe C_{11}	389
Figura II.37. Classe C_{12}	389
Figura II.38. Classe C_{13}	389
Figura II.39. Classe C_{14}	390
Figura II.40. Classe C_{15}	390
Figura II.41. Classe C_{16}	391
Figura II.42. Classe C_{17}	391
Figura II.43. Classe C_{18}	392
Figura II.44. Classe C_{19}	393
Figura II.45. Classe C_{20}	394
Figura II.46. Classe C_{21}	394
Figura II.47. Classe C_{22}	394
Figura II.48. Classe C_{23}	395
Figura II.49. Classe C_{24}	395
Figura II.50. Classe C_{25}	396
Figura II.51. Classe C_{26}	396
Figura II.52. Classe C_{27}	396
Figura II.53. Classe C_{28}	397
Figura II.54. Classe C_{29}	397
Figura II.55. Classe C_{30}	397
Figura II.56. Classe C_{31}	398
Figura II.57. Classe C_{32}	398
Figura II.58. Classe C_{33}	398
Figura II.59. Classe C_{34}	399
Figura II.60. Classe C_{35}	400

Figura II.61. Modelo JARP correspondente à RPTM da figura 12.18	401
Figura II.62. Grafo de classes para a rede das figuras 12.18 / II.61	406
Figura II.63. Modelo JARP correspondente à RPTM da figura 12.19	408
Figura II.64. Classe E_1 – equivalente à classe C_3 (figura II.28)	419
Figura II.65. Classe E_2 – equivalente à classe C_4 (figura II.29)	419
Figura II.66. Classe E_3 – equivalente à classe C_5 (figura II.30)	420
Figura II.67. Classe E_4 – equivalente à classe C_6 (figura II.31)	420
Figura II.68. Classe E_5 – equivalente à classe C_6 (figura II.31)	421
Figura II.69. Classe E_6 – equivalente à classe C_{12} (figura II.37)	422
Figura II.70. Classe E_7	422
Figura II.71. Classe E_8	423
Figura II.72. Classe E_9	424
Figura II.73. Classe E_{10}	424
Figura II.74. Classe E_{11}	425
Figura II.75. Classe E_{12}	426
Figura II.76. Classe E_{13}	426
Figura II.77. Classe E_{14}	426
Figura II.78. Classe E_{15}	427
Figura II.79. Classe E_{16}	428
Figura II.80. Classe E_{17}	428
Figura II.81. Classe E_{18}	429
Figura II.82. Classe E_{19}	429
Figura II.83. Classe E_{20} – equivalente à classe E_{15} (figura II.78)	430
Figura II.84. Classe E_{21} – equivalente à classe E_{17} (figura II.80)	430
Figura II.85. Classe E_{22}	430
Figura II.86. Classe E_{23}	431

ÍNDICE DE LISTAGENS

Listagem II.1. Modelo ARP da figura II.5	343
Listagem II.2. Análise de propriedades: transcrição da figura II.13	346
Listagem II.3. Modelo ARP da figura II.18	354
Listagem II.4. Análise de propriedades para a rede da figura II.18	355
Listagem II.5. Modelo ARP da figura II.20	358
Listagem II.6. Análise de propriedades para a rede da figura II.20	362
Listagem II.7. Modelo ARP da figura II.22	366
Listagem II.8. Análise de propriedades para a rede da figura II.22	372
Listagem II.9. Modelo ARP da figura II.24	376
Listagem II.10. Modelo ARP da figura II.61	402
Listagem II.11. Análise de propriedades para a rede da figura II.61	404
Listagem II.12. Modelo ARP da figura II.63	409

ÍNDICE DE TABELAS

Tabela 1.1. Relação entre tipos de representação e níveis de abstração [3]	10
Tabela 3.1. Quadro comparativo CISC $\times$ RISC	48
Tabela 5.1. Tempos de execução nos processadores	90
Tabela 8.1. Classes de estados para a RPTM da figura 8.1	121
Tabela 9.1. Limites inteiro e multi-conjunto do sistema de banco de dados	134
Tabela 9.2. Tamanho dos grafos de disparo integral e com simetria para o SBD	140
Tabela 11.1. Conjunto de registradores para o processador proposto	203
Tabela 11.2. Conjunto de instruções para o processador proposto	205
Tabela 11.3. Ciclo de execução de instrução para alguns exemplos	211

INTRODUÇÃO

O projeto de sistemas digitais alcançou um elevado grau de complexidade, inviabilizando sua consecução sem a utilização de ferramentas de CAD. Na tentativa de modelar as características dos componentes de *hardware* e de *software* destes sistemas, da forma mais acurada possível, e de prover mecanismos necessários a um grau maior de automação do processo de projeto dos mesmos, vários modelos conceituais têm sido capturados por linguagens de especificação empregadas nessas ferramentas, dentro de metodologias de projeto distintas.

Rede de Petri e suas extensões (simplicadamente, redes de Petri) constituem uma classe de modelos conceituais utilizada na modelagem de diversos tipos de sistemas computacionais paralelos. Por exemplo, tem-se usado redes de Petri para modelar protocolos de comunicação, processos de manufatura, arquiteturas de computadores etc. Tais modelos possibilitam a representação de sistemas paralelos, os quais podem então ser simulados, por exemplo, nos níveis funcional e lógico, incluindo ou não aspectos de temporização, com um grau razoável de simplicidade.

Mais recentemente, redes de Petri têm demonstrado ser eficientes na representação de determinados aspectos de sistemas de eventos discretos. Este fato deve-se, em grande parte, à existência de um conjunto de técnicas para análise estrutural e dinâmica de redes de Petri, permitindo a validação formal de importantes propriedades de um modelo, tais como inexistência de *livelocks* e *deadlocks*, reiniciabilidade do sistema modelado (após a realização de um conjunto de tarefas), limitabilidade dos recursos modelados, dentre outras. Entretanto, embora apresentem características potenciais para a modelagem de sistemas digitais, as extensões de rede de Petri formalmente definidas para esta finalidade não apresentam os requisitos necessários em sua totalidade.

Em resumo, rede de Petri e suas extensões representam um amplo campo de estudo, o qual pode ser explorado no que se refere à área de sistemas digitais, buscando-se soluções alternativas no tocante à modelagem, simulação, análise e síntese de tais sistemas.

O objetivo principal deste trabalho de doutorado consistiu no desenvolvimento de uma extensão de rede de Petri, elaborada para superar as limitações das demais extensões na representação de sistemas digitais, inserindo-a numa metodologia para a modelagem, simulação, análise e síntese em alto nível de sistemas digitais, mais especificamente, de sistemas embutidos. Tal extensão foi denominada **Rede de Petri para Sistemas Digitais – RPSD**. Como objetivo secundário deste trabalho, foi realizado um estudo abrangente das atuais metodologias de projeto de sistemas embutidos, incluindo modelos conceituais, arquiteturas e linguagens de especificação.

A tese é dividida em três partes, descritas na sequência.

Parte I: Sistemas Digitais

Esta parte trata de definições, conceitos, metodologias, modelos computacionais, arquiteturas e linguagens de especificação, inerentes ao projeto de sistemas digitais.

O capítulo 1 aborda sistemas digitais. São apresentadas metodologias de projeto utilizadas nas últimas três décadas, envolvendo captura/simulação e descrição/síntese. O capítulo encerra-se com uma breve descrição sobre metodologias recentes em nível de sistema, conhecidas por metodologias de coprojeto *hardware/software*, as quais são retomadas no capítulo 5.

Na sequência, o capítulo 2 apresenta duas taxonomias para modelos conceituais computacionais. Tais modelos podem ser aplicados em diferentes fases de metodologias de projeto para sistemas digitais, permitindo representar características funcionais, temporais e estruturais de diversos componentes e tipos destes sistemas.

No capítulo 3 são abordadas arquiteturas, as quais complementam modelos de especificação ao descrever como estes podem ser implementados.

Para que possa ser tratado por ferramentas de CAD (*Computer Aided Design*) e de CAE (*Computer Aided Engineering*), um modelo conceitual deve ser transformado em uma especificação através de uma linguagem de especificação. Há vários tipos de linguagens de especificação, cada qual podendo capturar características de um ou mais modelos. O capítulo 4 apresenta uma visão geral das principais linguagens empregadas na especificação de sistemas digitais.

O capítulo 5 apresenta como diferentes metodologias de coprojeto *hardware/software* tratam as etapas principais do fluxo de projeto de um sistema embutido.

Parte II: Redes de Petri

Extensões de rede de Petri comuns e as utilizadas como base para a extensão que desenvolvemos, são apresentadas nesta parte.

As definições e conceitos básicos de redes de Petri são apresentados no capítulo 6. Inicialmente é dado o desenvolvimento cronológico da teoria de redes de Petri. Em seguida, por meio da rede de Petri Elementar, são apresentados a composição e conceitos básicos usados pelas principais extensões de rede de Petri. O capítulo mostra, ainda, a pouca capacidade de redes Elementares quando da modelagem de sistemas maiores.

O capítulo 7 apresenta a rede de Petri Lugar/Transição, com ênfase na formalização matemática do modelo. Tal formalização é importante, uma vez que as demais extensões de rede de Petri utilizam a rede Lugar/Transição como base.

O capítulo 8 traz três extensões temporizadas de rede de Petri: as redes de Sifakis, de Ramchandani e de Merlin. A ênfase é dada para este último modelo, o qual é amplamente utilizado por ser uma extensão fortemente temporizada e por trabalhar com intervalos de tempo, o que possibilita a especificação e verificação de tempo em sistemas. Na parte III da tese, as características da extensão de Merlin são agregadas à extensão que desenvolvemos neste trabalho de doutorado, para a especificação de sistemas digitais.

Encerrando a parte II, o capítulo 9 apresenta uma extensão de rede de Petri bastante utilizada em aplicações práticas, qual seja, as redes de Petri Coloridas. Este modelo é o principal representante das chamadas redes de alto nível.

Parte III: Rede de Petri para Sistemas Digitais

Esta parte relaciona redes de Petri com sistemas digitais e apresenta a extensão que desenvolvemos, inserindo-a numa metodologia de coprojeto *hardware/software*.

No capítulo 10, é mostrado como diferentes extensões de rede de Petri atendem, não simultaneamente, aos requisitos necessários à modelagem de sistemas digitais. O capítulo fornece, ainda, exemplos do uso de redes de Petri no projeto desses sistemas. São abordadas duas extensões voltadas para sistemas digitais: PCN e PNES. Como PCN pode ser considerada como um subconjunto de PNES, ênfase maior é dada sobre esta última.

Como as extensões de rede de Petri formalmente definidas para a representação de sistemas digitais não apresentam os requisitos necessários em sua totalidade, o capítulo 11 apresenta a extensão que desenvolvemos para esta finalidade, qual seja, a de atender a todos os requisitos necessários em um único modelo, adequado para a especificação de sistemas digitais, particularmente, sistemas embutidos. Esta extensão, denominada **Rede de Petri para Sistemas Digitais – RPSD**, constitui o núcleo de uma metodologia para a modelagem, simulação, análise e síntese de alto nível de sistemas digitais. Através de exemplos, é mostrado o poder descritivo da extensão proposta. O capítulo apresenta, também, a descrição formal de **RPSD**.

O capítulo 12 mostra como os diferentes componentes de **RPSD** (lugares, transições e ramos) podem ser representados através de redes de Petri de baixo nível, como as redes Lugar/Transição e as redes Temporizadas de Merlin. O objetivo da equivalência de representações é mostrar que a extensão proposta é uma extensão legítima de redes de Petri, o que permite o desenvolvimento de técnicas de verificação (formal ou por simulação) de propriedades, baseadas em redes de Petri.

O capítulo 13 apresenta um algoritmo para simulação de modelos em **RPSD**, capaz de gerar o grafo de classes de um dado modelo. Baseado neste algoritmo, é discutido um conjunto de técnicas dinâmicas de análise de propriedades.

Os conceitos inerentes à análise de invariantes em redes de Petri são apresentados no início do capítulo 14. Na sequência, é mostrado como esse tipo de análise é limitada em modelos **RPSD**, além de indicações de como poderia ser feita.

Finalmente, o capítulo 15 mostra as linhas gerais de uma metodologia de coprojeto *hardware/software* baseada em **RPSD**, proposta para desenvolvimento futuro, sob supervisão do Grupo de Micro-Eletrônica do Departamento de Ciências da Computação e Estatística do IBILCE/UNESP (GME/DCCE/IBILCE/UNESP).

Após as partes I, II e III, a tese apresenta as conclusões, apontando os resultados alcançados e as diretrizes para trabalhos futuros. Seguem-se as referências bibliográficas e de *internet*. A tese encerra-se com dois apêndices. Pontos da teoria dos grafos, inerentes à compreensão do algoritmo de geração do grafo de classes de **RPSD**, dado no capítulo 13, são descritos no apêndice I. O apêndice II valida, através de simulação, as redes descritas no capítulo 12, utilizadas na demonstração de equivalência de **RPSD** com redes de Petri de baixo nível.

PARTE I – SISTEMAS DIGITAIS

CAPÍTULO 1: SISTEMAS DIGITAIS

1.1. INTRODUÇÃO

Sistemas digitais podem ser divididos em dois grandes grupos [1], quais sejam:

- a) **Sistemas de computação de propósito geral;**
- b) **Sistemas embutidos.**

No primeiro grupo, situam-se sistemas que executam *software* genérico, sendo suportados por um grande número de sistemas operacionais e linguagens de programação. Exemplos incluem desde *palm tops* até supercomputadores. No segundo grupo, encontram-se sistemas dedicados (específicos) a uma dada aplicação. Exemplos de sistemas embutidos vão desde um microcontrolador de uma lavadora de roupas até um sistema de controle e navegação de uma aeronave.

Desses dois grupos, o que vem despertando maior interesse da comunidade acadêmica e industrial é o de sistemas embutidos, o qual forma um mercado que já é maior e cresce mais rapidamente que o relacionado ao de computadores de propósito geral [2]. É sobre o grupo de sistemas embutidos que este trabalho enfatiza. Ressalta-se que, neste texto, sempre que houver a ocorrência da palavra sistema, não acompanhada de qualquer qualificação, estar-se-á referindo a sistema digital.

Há três grandes grupos de sistemas embutidos:

- a) **Sistemas transformacionais:** são sistemas que realizam um padrão repetitivo de processamento, para conjuntos diferentes de dados que alimentam suas entradas periodicamente. O principal exemplo desta categoria é o de sistemas **DSP** (*Digital Signal Processing* – Processamento Digital de Sinais), amplamente utilizados na área de comunicação digital e sistemas multimídia;
- b) **Sistemas reativos:** são aqueles que reagem ao ambiente, através da execução de funções e da ativação de atuadores, em resposta a determinados estímulos de entrada (capturados por sensores);
- c) **Sistemas de tempo real:** classificação atribuída aos sistemas que, em determinados momentos, são obrigados a responder (realizar determinado processamento) dentro de intervalos de tempo bastante restritos.

Sistemas digitais podem ser constituídos por unidades de vários tipos, tais como:

- a) **Processadores padrão:** denominados processadores de prateleira (ou comerciais), possuem um conjunto de instruções genérico;
- b) **Coprocessadores:** possuem um conjunto de funções implementadas, visando acelerar determinados tipos de computação, como, por exemplo, processamento matemático, gerenciamento de memória e coerência de cache;
- c) **ASICs** (*Application-Specific Integrated Circuits* – Circuitos Integrados Específicos a uma Aplicação): também conhecidos como *hardware* dedicado, oferecem grande desempenho, uma vez que implementam um grupo de funções de acordo com as características de uma dada aplicação. Entretanto, impõem um aumento de tempo e de custo ao projeto. Uma

alternativa ao projeto de ASICs é a substituição destes por ASICs baseados em *core* (núcleo de processador). Nesta tecnologia, os fabricantes de processadores tornam os *cores* de seus processadores disponíveis como mega células, as quais podem ser configuradas por projetistas em seus projetos;

- d) **ASIPs** (*Application-Specific Instruction Processors* – Processadores com Conjunto de Instruções Específico a uma Aplicação): são processadores especializados às necessidades de um produto específico ou uma família de produtos. Ao gastar silício onde realmente importa, tais processadores são menores e mais simples que os processadores padrão de mesmo nível tecnológico, sendo capazes de executar em frequências mais altas de *clock*, com um menor consumo de energia.

O **projeto** de um sistema digital envolve um conjunto de tarefas, visando transformar uma especificação, do sistema a ser tratado, em um plano de manufatura. Esta especificação representa as funções do sistema e as restrições que devem ser atendidas. O projeto deve seguir uma **metodologia**, a qual determina as tarefas a serem realizadas, a ordem de execução destas e as ferramentas a serem empregadas durante o processo de projeto.

As diferentes atividades de um projeto envolvem diferentes profissionais. Por exemplo [3], a especificação de um sistema pode ser elaborada com base em pesquisas de mercado, encomendadas e analisadas por profissionais de *marketing*. Um arquiteto chefe fica encarregado de construir um diagrama de blocos inicial da arquitetura do sistema. Em seguida, faz-se escolhas referentes à tecnologia de fabricação e às ferramentas de CAD (*Computer Aided Design*) e de CASE (*Computer Aided Software Engineering*) empregadas para dar suporte ao projeto. Na sequência, os projetos de *hardware* e de *software* são executados, de forma conjunta (recebendo a denominação de coprojeto *hardware/software*) ou separada. Estratégias de teste são definidas e executadas por engenheiros de teste. Por fim, engenheiros de manufatura são responsáveis pelo desenvolvimento do plano de manufatura.

Assim, há diferentes **representações** (visões) de projeto, de acordo com os profissionais envolvidos. Tradicionalmente, estas representações são assim qualificadas:

- a) **Comportamental**: enfatiza a funcionalidade do sistema, isentando-a de qualquer forma de implementação. Pode incluir especificações temporais;
- b) **Estrutural**: representa os componentes que implementarão a funcionalidade do sistema, assim como as interconexões entre os mesmos, sem entrar em detalhes sobre suas características físicas (dimensionalidade);
- c) **Física**: representa a dimensionalidade dos componentes da arquitetura, mas não a funcionalidade dos mesmos. As características físicas de um componente envolvem sua localização, dimensões, peso, consumo de energia e potência dissipada.

Usualmente, estas formas de representação são adotadas ao longo do **processo de projeto** de um sistema digital. Entretanto, dentro de cada uma destas formas, o sistema pode ser descrito em diferentes **níveis de abstração** (granulosidade), de acordo com os objetos utilizados nestas descrições. Estes objetos correspondem a componentes do tipo processador, registrador, porta ou transistor. Assim, os níveis de abstração (em ordem decrescente) e seus respectivos componentes estruturais são:

- 1º) **Processador**: processadores padrão, ASICs, controladores e memórias, dentre outros;
- 2º) **Registrador** (ou RTL, de *Register Transfer Level*): somadores, comparadores, seletores, multiplexadores, demultiplexadores, ULAs (unidades lógico-aritméticas) e registradores (isolados, em bancos, formando filas ou de deslocamento), dentre outros;

- 3º) **Portas** (ou lógico): portas lógicas (AND, OR, NOT, XOR etc.) e *flip-flops*;
- 4º) **Transistor**: transistores, resistores e capacitores.

Associações diversas entre componentes estruturais de um mesmo nível de abstração, formam objetos físicos diferentes (tratados na representação física do sistema). Por outro lado, os objetos físicos de um nível de abstração são os constituintes dos componentes estruturais do nível de abstração superior. Exemplificando, componentes do nível transistor são utilizados para formar tanto células analógicas quanto células digitais, as quais são representadas fisicamente pelos *layouts* dos componentes utilizados na sua definição (transistores, resistores, capacitores). Os componentes estruturais do nível de abstração imediatamente superior, quais sejam, portas lógicas e *flip-flops*, são definidos a partir destas células.

Continuando, a associação de portas lógicas e de *flip-flops* permite a formação de módulos aritméticos ou de armazenamento. Estes módulos são os constituintes físicos dos componentes estruturais no nível de registrador (somadores, comparadores, registradores etc.). Por sua vez, estes componentes são associados, formando *microchips* (processadores, memórias, controladores) e ASICs (*hardware* dedicado). *Microchips* e ASICs, por sua vez, são os componentes estruturais do nível de abstração de processador. Tais componentes permitem a montagem de PCBs (*printed-circuit board* – placa de circuito impresso) e MCMs (*multi-chip module* – módulo de múltiplos *chips*).

Tendo relacionado as representações física e estrutural com os diversos níveis de abstração, falta relacionar estes níveis com a representação comportamental.

No nível de transistor, a representação comportamental descreve a funcionalidade de um circuito através de um conjunto de equações diferenciais ou através de diagramas tensão-corrente. A descrição comportamental no nível de portas é dada por meio de equações booleanas (no caso de circuitos combinacionais) ou de FSMs (no caso de circuitos sequenciais); FSM é a sigla para *Finite-State Machine* (máquina de estados finitos – seção 2.2.1). Para o nível de registrador, o comportamento pode ser descrito através de fluxogramas, algoritmos ou FSMs generalizadas, por exemplo. Finalmente, no nível mais alto de abstração, o de processador, o comportamento do sistema pode ser descrito de diversas formas como, por exemplo, linguagem natural, algoritmos, programas numa linguagem de programação ou especificações executáveis em uma dada HDL (*hardware description language* – linguagem de descrição de *hardware*).

O exposto acima pode ser sintetizado pela tabela 1.1 [3], a qual traz o relacionamento entre os tipos de representação e os níveis de abstração, na descrição de um sistema digital.

Quanto maior o nível de abstração, menor o número de componentes na descrição. Inversamente, quanto menor o nível de abstração, maior o número de componentes. Desta forma, a compreensão da funcionalidade do sistema será melhor quando do deslocamento para níveis maiores de abstração, nos quais trabalha-se com um número menor de componentes ("grãos maiores"), facilitando o gerenciamento dos mesmos.

Tabela 1.1. Relação entre tipos de representação e níveis de abstração [3].

Níveis de Abstração	Tipos de Representação		
	Comportamental	Estrutural	Física
Processador	programas, especificações executáveis	processadores, ASICs, controladores, memórias	PCBs, MCMs
Registrador (RTL)	algoritmos, fluxogramas, FSMs generalizadas	somadores, comparadores, registradores	<i>microchips</i> , ASICs
Portas (Lógico)	equações booleanas, FSMs	portas lógicas, <i>flip-flops</i>	módulos
Transistor	equações diferenciais, diagramas $v-i$	transistores, resistores, capacitores	células analógicas e digitais

1.2. EVOLUÇÃO HISTÓRICA DE METODOLOGIAS DE PROJETO RECENTES

No processo de projeto de um sistema digital, diversas metodologias podem ser empregadas, de acordo com o sistema alvo. Uma metodologia de projeto define quais, e em que ordem são executadas, as tarefas que constituem o processo de projeto, assim como as ferramentas de suporte a estas tarefas.

Nesta seção, são abordadas as metodologias mais utilizadas nos últimos anos, no projeto de sistemas digitais. As seções 1.2.1 e 1.2.2 tratam de metodologias de projeto utilizadas em nível de *chip*, enquanto a seção 1.2.3 trata dos requisitos para uma metodologia de projeto em nível de **sistema**, envolvendo tanto o projeto de *software* quanto o de *hardware*.

1.2.1. CAPTURA E SIMULAÇÃO

A metodologia de captura e simulação (*capture-and-simulate*) é baseada na captura (entrada de dados no computador) de uma representação esquemática, para propósitos de simulação (verificação funcional). Suas etapas podem ser colocadas nesta ordem:

- 1^a) **Especificação** dos requisitos do projeto por profissionais da área de *marketing*;
- 2^a) **Projeto arquitetural**, gerando um diagrama de blocos do sistema;
- 3^a) **Projeto lógico e de layout**, produzindo um esquemático lógico ou de circuito;
- 4^a) **Captura do esquemático** (seguindo formatação exigida pelas ferramentas de simulação);
- 5^a) **Simulação**, verificando funcionalidade, aspectos temporais e cobertura de falhas;
- 6^a) **Síntese**, através de ferramentas de projeto físico (projeto dedicado, FPGAs).

1.2.2. DESCRIÇÃO E SÍNTESE

Na metodologia de descrição e síntese (*describe-and-synthesize*) o sistema é descrito através de uma representação comportamental no nível lógico ou no nível de transferência entre registradores (RTL). Esta descrição comportamental é, então, automaticamente sintetizada por ferramentas de CAD.

Descrições comportamentais no nível lógico são sintetizadas através de um conjunto de técnicas denominado **síntese lógica**. Neste nível (lógico), a associação de portas lógicas e de *flip-flops* permite a representação adequada tanto de **circuitos combinacionais** como de **circuitos sequenciais**.

A descrição comportamental de um circuito combinacional (ULA, por exemplo) é dada através de um conjunto de equações booleanas. Esta descrição é submetida a duas fases:

- 1ª) **Minimização lógica**, na qual são aplicadas técnicas de redução sobre as equações booleanas; e
- 2ª) **Mapeamento tecnológico**, onde as funções booleanas, obtidas a partir da minimização lógica, são mapeadas para portas pertencentes a uma dada biblioteca de portas lógicas.

Por outro lado, a descrição comportamental de um circuito sequencial (por exemplo, uma unidade de controle) é representada por uma FSM (máquina de estados finitos). Tal descrição passa por três fases:

- 1ª) **Minimização de estados**, onde se aplicam técnicas de redução sobre os estados da FSM, com a conseqüente configuração final da função próximo estado e da função de saída;
- 2ª) **Minimização lógica**, tal como descrita acima, aplicada sobre as funções da FSM minimizada; e
- 3ª) **Mapeamento tecnológico**, tal como descrito acima, aplicado sobre as funções minimizadas obtidas da fase anterior.

Descrições comportamentais RTL são representadas através de HDLs (*hardware description language* – linguagem de descrição de *hardware*). Uma descrição comportamental RTL é convertida para uma descrição estrutural, também no nível RT, através de uma série de tarefas que recebe a denominação de **síntese comportamental**. A seqüência de tarefas que compõem a síntese comportamental é assim descrita:

- 1ª) **Alocação**: Nesta tarefa, é feita a escolha dos componentes estruturais, com base em análises que consideram a relação custo/desempenho de diversas configurações, envolvendo unidades funcionais, de armazenamento e de interconexão, assim como diferentes opções de estágios de *pipeline* em determinadas unidades funcionais;
- 2ª) **Escalonamento**: Tarefa responsável pelo particionamento da descrição comportamental em intervalos de tempo – denominados etapas de controle. Transferências de dados entre registradores são executadas durante o período de duração determinado para uma etapa de controle; e
- 3ª) **Atribuição** (*binding*): Esta tarefa faz a associação entre as operações de computação e de transferência de dados e as unidades funcionais.

A descrição estrutural resultante da síntese comportamental alimenta as etapas referentes à síntese lógica (descrita anteriormente).

1.2.3. METODOLOGIAS EM NÍVEL DE SISTEMA

Na última década, as metodologias de projeto baseadas na captura e simulação deram lugar às metodologias baseadas na descrição comportamental e síntese automática. Entretanto, a descrição comportamental nestas metodologias está mais voltada para o projeto em nível RTL, ou seja, para o projeto de *chips* (processadores, ASICs, memórias). Projetos mais complexos, que envolvam um sistema completo (constituído por vários elementos de *hardware* e de *software*), exigem novas metodologias, nas quais o nível de abstração para a representação comportamental seja o de processador (veja tabela 1.1).

Pelas metodologias mais utilizadas até hoje, a especificação de um sistema é obtida a partir de requisitos iniciais, sendo transformada em um diagrama de blocos (descrição estrutural da arquitetura do sistema). Desta forma, configura-se um problema em potencial: a qualidade da representação obtida está extremamente vinculada à experiência do projetista, o qual é responsável por esta transformação (de requisitos para arquitetura). Outro problema é que nem todas as funcionalidades do sistema são conhecidas durante as fases iniciais do projeto e, alterações na especificação devido a erros e inconsistências descobertos em fases mais adiantadas, provocam um grande aumento no tempo e no custo do projeto.

Em seus primeiros passos, as novas metodologias de projeto em nível de sistema, conhecidas por **metodologias de coprojeto *hardware/software*** (tratadas no capítulo 5), buscam enfatizar a importância de uma especificação bem detalhada (senão completa) nas primeiras etapas do projeto, antes que quaisquer decisões de projeto (tais como definição da arquitetura e escolhas tecnológicas referentes aos componentes que irão implementar a arquitetura) tenham sido tomadas. Neste contexto, um dos aspectos mais importantes na definição de uma metodologia em nível de sistemas, recai sobre a escolha de seu modelo conceitual (ou modelos conceituais) e de sua linguagem de especificação (ou linguagens de especificação).

Um **modelo conceitual** é um sistema formal constituído por um conjunto de objetos e de regras de composição entre os objetos; sua finalidade é a de representar, de forma não ambígua, as características funcionais e/ou temporais de uma dada classe de sistemas, além de possíveis restrições que se façam necessárias (dependendo do modelo). Cada modelo possui uma capacidade de representação, que o torna mais adequado ou menos, de acordo com o tipo de sistema. Além da ausência de ambigüidade, são características desejáveis de um modelo:

- a) Boa cobertura, ou seja, capacidade de descrever todo o sistema;
- b) Facilidade de alteração, uma vez que mudanças no projeto inicial são frequentes;
- c) Facilidade de compreensão, para permitir a troca de informações entre os projetistas; e
- d) Facilidade de tratamento, para que o modelo seja adequado às tarefas de projeto, funcionando como um amigo (que auxilie nas tarefas de forma natural) e não como um inimigo (que coloque uma série de obstáculos ao trabalho das pessoas envolvidas).

Para que possa ser tratado por ferramentas de CAD (*Computer Aided Design*) e de CAE (*Computer Aided Engineering*), um modelo deve ser transformado em uma especificação por meio de uma **linguagem de especificação**. Há vários tipos de linguagens de especificação, cada qual podendo capturar características de um ou mais modelos. São características desejáveis de uma linguagem de especificação:

- a) Capacidade de capturar o maior número de características de uma classe de sistemas;
- b) Possibilitar o uso de ferramentas de síntese, tanto para *hardware* quanto para *software*;

- c) Ter boa legibilidade; e
- d) Gerar uma especificação executável, a qual possa ser utilizada para propósitos de demonstração da funcionalidade do produto, documentação (servindo de guia para as diferentes equipes envolvidas no processo de projeto), análise de propriedades (verificação automática), exploração do espaço de projeto (alternativas de projeto) e síntese do projeto.

Além dos modelos formais e linguagens de especificação adotadas, uma metodologia em nível de sistema deve prover também capacidades que permitam ao projetista **explorar e refinar várias alternativas de projeto**. Estas capacidades incluem:

- a) Interação com o projetista, permitindo que este escolha de forma sistemática os recursos alocados (processadores, memórias, ASICs, tipos de barramento, interfaces), assim como defina limites para parâmetros de restrições do projeto;
- b) Análise automática e seleção dentre alternativas de particionamento das funções do sistema em componentes de *hardware* ou de *software*, com base em restrições como, por exemplo, custo, desempenho, consumo de energia e potência dissipada; e
- c) Refinamento (detalhamento) da especificação inicial, gerando, pelo menos, a mesma informação de um diagrama de blocos (arquitetura).

Observações relevantes sobre modelos conceituais, arquiteturas e linguagens de especificação encontram-se nos próximos três capítulos. Metodologias em nível de sistema são tratadas no capítulo 5.

CAPÍTULO 2: MODELOS CONCEITUAIS

2.1. INTRODUÇÃO

Há vários tipos de modelos conceituais. Diferentes modelos podem ser aplicados em diferentes fases do projeto, de acordo com a ênfase que se deseja ressaltar. A natureza da aplicação é que determina qual ou quais modelos são mais adequados para determinados aspectos de representação. Por exemplo, aplicações de tempo real necessitam de modelos que permitam a especificação de restrições temporais (como intervalos de tempo para um certo processamento); já aplicações de bancos de dados exigem modelos que enfatizem estruturas de dados.

Gajski *et al.* [3] propõem uma classificação envolvendo modelos computacionais, enquanto o SLD DWG (*System-Level Design Development Working Group*) da VSI Alliance [4], apresenta taxonomia e terminologia diretamente voltadas para sistemas digitais, com base na versão 0.2.4.2 do documento de terminologia e taxonomia para modelagem em VHDL [5], produzido pelo grupo de trabalho RTWG (*RASSP Taxonomy Working Group*) da iniciativa RASSP (*Rapid-prototyping of Application Specific Signal Processor*). Ambas terminologias encontram-se brevemente descritas nas seções 2.2 e 2.3.

2.2. TAXONOMIA DE MODELOS CONCEITUAIS COMPUTACIONAIS SEGUNDO GAJSKI

Gajski classificou modelos computacionais conceituais em 5 grupos, quais sejam:

- a) Modelos de estado;
- b) Modelos de atividades;
- c) Modelos estruturais;
- d) Modelos de estruturas de dados;
- e) Modelos heterogêneos.

2.2.1. MODELOS DE ESTADO

Os modelos de estado apresentam, na sua composição básica, **estados** e **transições** entre estes estados, ativadas por **eventos externos** ao sistema. São modelos adequados para a descrição de **sistemas de controle** (onde o sistema transita por vários modos de operação, em resposta a interações com seu ambiente) e de **sistemas reativos de tempo real** (uma vez que apresentam boa captura do comportamento temporal do sistema), desde que tais sistemas não sejam muito complexos. O controle eletrônico de uma lavadora de roupas, o controle de um elevador e o circuito de um semáforo de trânsito, são exemplos de sistemas de controle. Sistemas de exploração espacial e de controle de tráfego aéreo, são exemplos de sistemas reativos de tempo real.

Dois modelos de estado recebem atenção especial, quais sejam, as **máquinas de estados finitos** (FSM, de *Finite-State Machine*) e as **redes de Petri**. Estes modelos possuem,

ainda, várias extensões. Nesta seção, são abordadas FSMs, enquanto redes de Petri, o modelo sobre o qual este trabalho está centrado, são tratadas na parte II desta tese.

Máquinas de estados finitos são compostas por um conjunto de estados, um conjunto de transições e um conjunto de saídas (associadas às transições, ou aos estados, de acordo com o tipo da FSM). São formalmente definidas por:

$$\begin{aligned} &< S, I, O, f: S \times I \rightarrow S, h: S \times I \rightarrow O >, \text{FSM de Mealy (orientada a transição),} \\ &\quad \text{ou} \\ &< S, I, O, f: S \times I \rightarrow S, h: S \rightarrow O >, \text{FSM de Moore (orientada a estado),} \end{aligned}$$

onde: S = conjunto de estados,
 I = conjunto de sinais de entrada,
 O = conjunto de sinais de saída,
 f = função próximo estado,
 h = função de saída.

Em geral, máquinas Moore requerem mais estados que máquinas Mealy. Isto ocorre quando um estado está associado a várias saídas: como as máquinas Mealy associam as saídas às transições (pela função $h: S \times I \rightarrow O$), não ocorrerá um aumento no número de estados, e sim de transições, caso se deseje incluir novas saídas a partir de um estado; já nas máquinas Moore, um novo sinal de saída o_i , que se deseje incluir a partir de um estado s_x , representará a adição de um novo estado s_y (disponibilizador da saída o_i) e de uma nova transição f_{xy} , ligando o estado s_x ao estado s_y .

O principal problema referente ao uso de FSMs recai sobre a explosão no número de estados, conforme a complexidade do sistema aumenta. Este problema tem origem tanto na representação de números inteiros utilizando-se de *bits*, como na representação de concorrência e de hierarquia.

FSMDs (*FSM with Datapath*) [6] surgiram para resolver o problema da representação de dados em baixo nível (associada às FSMs), através da introdução de variáveis, expressões e condições. Embora suportem a representação de sistemas de controle e de computação, FSMDs, assim como FSMs, também não são adequadas para sistemas mais complexos, devido à ausência de representação explícita para concorrência e hierarquia.

O problema de representação de concorrência em FSMs refere-se ao fato destas expressarem somente um estado ativo por vez. Assim, caso deseje-se que uma FSM possa representar 2 estados ativos por vez (ou seja, concorrentes 2 a 2), ter-se-ia que fazer uma nova FSM, onde cada estado corresponderia a um elemento do produto cartesiano $S \times S$. Quanto à hierarquia, como FSMs são planas, ocorrerá uma explosão no número de transições (máquinas Mealy) ou no número de estados (máquinas Moore), caso deseje-se associar (via sinal de entrada ou via sinal de saída) cada um dos estados de um subconjunto G de S a um dado estado s_i , o que poderia ser representado por um único arco de transição (máquina Mealy) ou um único estado (e sua transição correspondente, numa máquina Moore), caso FSMs não fossem planas (ou seja, admitissem o agrupamento hierárquico de estados e de transições).

Para resolver os problemas de FSMs relativos à concorrência e hierarquia, foram criadas, como modelo da linguagem de especificação *Statecharts* [7], as **HCFSM**

(*Hierarchical Concurrent Finite-State Machine*), as quais permitem decomposição de estados em subestados, execução paralela de subestados concorrentes, transições entre diferentes níveis hierárquicos (associadas a condições e a execução de ações) e comunicação através de variáveis globais (mecanismo de *broadcast*).

As extensões de FSMs permitem a modelagem de sistemas de controle complexos, devido à possibilidade de atribuição de valores a variáveis, à concorrência e à hierarquia. No entanto, elas não se mostram adequadas para outros tipos de sistemas complexos, possuidores de estruturas de dados sofisticadas e de atividades (computações) associadas a cada estado.

2.2.2. MODELOS DE ATIVIDADES

Modelos de atividades enfatizam as **atividades** do sistema modelado, onde a **dependência** entre atividades é especificada ou por **seqüenciamento** de execução, ou por **passagem de dados** entre elas. Estes modelos são adequados para **sistemas transformacionais**, onde a saída do sistema é obtida por uma série de transformações executada sobre a entrada. Como exemplos de sistemas transformacionais, tem-se os sistemas de comunicação digital e os sistemas de processamento digital de sinais (DSP, de *Digital Signal Processing*), que compreendem desde *modems* até filtros de imagem.

Os dois principais representantes destes modelos são o **grafo de fluxo de dados** (DFG, de *Data Flow Graph*) e o **grafo de fluxo de controle** (CFG, de *Control Flow Graph*), também conhecido como fluxograma (do inglês *Flowchart*). Devido à maior ocorrência em textos relativos a esta área, são utilizadas, neste texto, as siglas em inglês, **DFG** e **CFG**, para designar estes modelos.

Os **nós** do **DFG** descrevem as **atividades** responsáveis pelas **transformações** de dados, enquanto os **arcos** representam o **fluxo** de dados entre os nós (estabelecendo, portanto, uma **dependência de dados** entre as atividades modeladas). Há 3 tipos de nós:

- a) Nó extremidade: utilizado para representar a entrada e a saída de dados;
- b) Nó atividade: utilizado para descrever uma transformação, de acordo com a granulosidade desejada (programa, rotinas, tarefas, funções, instruções, operações aritméticas ou lógicas);
e
- c) Nó de armazenamento: utilizado para descrever formas de armazenamento, tais como arquivos, registros e variáveis.

Ao permitir hierarquia (um nó pode ser representado por um DFG subjacente), o DFG pode ser empregado na descrição de sistemas transformacionais complexos, tais como DSP. Além disso, pode ser usado em várias fases do processo de projeto, como modelo auxiliar. No entanto, peca nos aspectos relativos a tempo e controle. Como a representação de comportamento temporal está limitada às dependências de dados entre atividades e, como inexiste outra forma de determinar o seqüenciamento entre estas, o modelo DFG é inadequado para a especificação de sistemas computacionais embutidos.

Outro representante dos modelos de atividades é o **CFG** (ou fluxograma). Este tipo de grafo apresenta nós semelhantes aos do DFG, sendo assim classificados:

- a) Nó extremidade: representa o início ou o fim do processamento descrito pelo fluxograma;

- b) Nó de computação: assim como o nó atividade do DFG, descreve uma transformação de acordo com a granulosidade desejada; e
- c) Nó de decisão: nó responsável pelo controle de desvios no seqüenciamento do fluxo de execução entre atividades.

Os **arcos** do grafo indicam a seqüência no **fluxo de execução** das atividades: a transição de um nó origem a um nó destino do arco se dá quando do término da atividade do nó origem. Eventos externos não interferem no seqüenciamento entre as atividades, o que torna CFG inadequados como modelo único para a representação de sistemas reativos de controle.

2.2.3. MODELOS ESTRUTURAIS

Modelos estruturais são compostos por um conjunto de **módulos físicos** (de características bem definidas) e um conjunto de **interconexões** entre estes módulos. O **diagrama de conectividade de componentes (CCD, de Component-Connectivity Diagram)** é uma classe de modelos estruturais, voltados para a descrição estrutural de um sistema, omitindo informações funcionais e temporais. Modelos CCD descrevem os componentes do sistema por meio de nós de um grafo, assim como as interconexões entre estes componentes (barramentos, fios) através de arestas ligando os nós. Como exemplo de CCD, tem-se:

- a) Diagrama de blocos: onde os componentes são processadores, memórias, ASICs, FPGAs, entre outros, e as interconexões entre estes componentes omitem quaisquer detalhes (como tipo de barramento, interface de comunicação etc.);
- b) Esquemático em nível de registrador: formado por componentes representantes de registradores, ULAs, comparadores, multiplexadores, entre outros, e os sinais de controle relativos a estes componentes, são omitidos;
- c) Esquemático em nível de portas: nos quais os componentes são portas lógicas, de acordo com a disponibilidade de uma biblioteca.

Por não terem capacidade de expressar características funcionais, CCDs são geralmente usados nas fases finais do processo de projeto, para representação de **detalhes de implementação**.

2.2.4. MODELOS DE ESTRUTURAS DE DADOS

Modelos de estruturas de dados enfatizam os **dados** de uma computação, através de suas **classes, atributos** e formas de **relacionamento** e de **dependência**. Modelos deste tipo são mais adequados no projeto de **sistemas de informação** (bancos de dados), nos quais a representação dos dados é o aspecto mais importante do projeto, e onde a funcionalidade e os aspectos temporais não necessitam estar explicitamente expressos, ocupando um segundo plano na descrição. A área de Engenharia de *Software* apresenta muitos destes modelos como, por exemplo, o diagrama entidade-relacionamento (**DER**) e o **diagrama de Jackson**.

Os componentes de um **DER** [8] são **entidades** e **relacionamentos** entre estas entidades. Uma entidade representa um dado, caracterizado por um **conjunto de atributos**. Por exemplo, uma entidade *Cliente* pode ter como atributos *Nome*, *RG* e *CIC*. Por outro lado, um relacionamento representa conexões entre duas ou mais entidades. Exemplificando, um

relacionamento *Adquiriu Produto* pode estar relacionando a entidade *Cliente* a uma entidade *Produto*. Relacionamentos podem ser descritos por meio de tabelas. São graficamente representados por arestas.

Como DERs permitem a visualização das relações entre os dados armazenados no sistema, são muito utilizados na descrição de sistemas de bancos de dados.

O **diagrama de Jackson** [9] permite visualizar a **composição dos dados**, a qual é descrita por uma **estrutura de árvore**. Nesta estrutura, os dados primários, ou seja, aqueles que compõem os dados mais complexos, são representados por nós do tipo folha, enquanto os dados compostos são obtidos por operações de composição (**AND**), seleção (**OR**) e iteração (*) sobre os dados primários, sendo representados pelos nós não-folha na estrutura da árvore.

Desta forma, o diagrama de Jackson é adequado para representar dados que tenham uma estrutura mais complexa, em oposição ao DER, que é mais adequado para representar relacionamentos complexos entre dados. Assim como o DER, o diagrama de Jackson é inadequado para representar características funcionais e temporais do sistema modelado, sendo mais apropriado para a área de sistemas de informação, onde é utilizado dentro de uma metodologia denominada **JSD** (*Jackson Systems Development*), a qual não será tratada aqui, por diferir da linha adotada neste trabalho, voltada para a especificação de sistemas computacionais embutidos.

2.2.5. MODELOS HETEROGÊNEOS

A classe heterogênea é formada por modelos que combinam, de **forma integrada**, exemplares dos tipos citados anteriormente, oferecendo **diferentes perspectivas** para sistemas mais complexos.

O primeiro modelo a ser tratado é o de fluxo de dados e de controle (**CDFG**, *Control and Data Flow Graph*) [10], o qual é uma combinação dos modelos **DFG** e **CFG**. Neste modelo, o fluxo de execução é ditado tanto por **desvios no controle** de seqüência (característica tomada do CFG) quanto por **dependências de dados** (como no DFG). Mais especificamente, os fluxos de dados entre os nós de um CFG podem ser expressos por DFGs ou, em outras palavras, o CFG expressa o seqüenciamento entre DFGs.

Modelos ao estilo de CDFGs podem apresentar construções de controle adicionais, àquelas comumente encontradas em linguagens de programação, permitindo-os modelar ações de controle e atividades complexas, como as presentes em sistemas reativos de tempo real. Exemplo de um modelo desse tipo pode ser encontrado na referência [11], consistindo de um DFG agregado a um CFG, o qual é especificado por uma FSM; este CFG pode responder a eventos externos e internos, podendo controlar a execução do DFG através de sinais de controle, tais como *enable* (para iniciar uma atividade) e *disable* (para parar uma atividade).

Ao corrigir a incapacidade de DFGs de apresentar construções de controle e, também, a incapacidade de CFGs de representar dependências entre dados e eventos externos, CDFGs são bem adequados para a modelagem de sistemas reativos de tempo real.

Outra classe de modelos heterogêneos bastante conhecida é a composta por **linguagens de programação**. Uma linguagem de programação é um **modelo textual**

(linguagens visuais também fornecem representações textuais), capaz de representar tanto **dados** como **atividades** e **construções de controle**. Sua principal desvantagem para a especificação de sistemas embutidos, é a ausência de modelagem explícita dos estados de um sistema. De acordo com a especificação da sequência de execução, linguagens de programação podem ser classificadas em 2 grupos:

- a) **Linguagens imperativas**: são aquelas em que a sequência de execução é totalmente especificada. De particular importância para a área de *hardware*, neste grupo encontram-se as linguagens capazes de explorar processamento concorrente, tais como **CSP** (concorrência suportada pela troca de mensagens), **ADA** (suporte através de sinais de sincronização) e **VHDL** (suporte através de memória compartilhada);
- b) **Linguagens declarativas**: são linguagens onde a sequência de execução é definida de acordo com o objetivo final da computação (parte-se da definição de onde se quer chegar). Como exemplos, têm-se **LISP** e **PROLOG**, linguagens muito utilizadas na área de *Inteligência Artificial*.

Finalizando esta seção, é apresentado o modelo **PSM** (*Program-State Machine*) [12], o qual é um modelo heterogêneo que integra **HCFSM** e o paradigma de linguagens de programação. Os componentes principais deste modelo são denominados **estados-programa**, os quais descrevem **modos de computação** (processamento ou estado do sistema), sendo dispostos hierarquicamente. Em qualquer momento de execução, somente um estado-programa estará ativo, ou seja, executando as computações associadas (ou vigorando os estados associados). Há dois tipos de estados-programa, quais sejam, compostos e folhas.

Denomina-se **estado-programa composto** aquele que pode ser decomposto ou em subestados-programa **seqüenciais** (ativados um por vez, enquanto o estado-programa estiver ativo) ou em subestados-programa **concorrentes** (ativados ao mesmo tempo, assim que o estado-programa for ativado). A decomposição seqüencial de um estado-programa utiliza-se de um conjunto de **arcos de transição**, denotando o seqüenciamento entre seus subestados-programa componentes. Estes arcos são de 2 tipos:

- a) **TOC** (*transition-on-completion*): são arcos percorridos (ou seja, a transição é acionada, disparada) somente quando o subestado-programa ligado à sua origem tiver encerrado sua computação e, também, a condição booleana associada ao arco for verdadeira;
- b) **TI** (*transition-immediately*): são arcos percorridos imediatamente após a condição booleana associada ao arco tornar-se verdadeira, a despeito da computação descrita pelo subestado-programa ligado à sua origem ter sido finalizada ou não.

Por outro lado, **estados-programa folhas** são aqueles descritos por sentenças de programação, situados no nível hierárquico mais baixo.

Comparando-se PSMs com HCFSMs, ressalta-se que PSMs são capazes de representar os estados, dados e atividades de um sistema em um único modelo, mostrando-se, dessa forma, mais adequadas do que HCFSMs na modelagem de sistemas que tenham dados e atividades complexos associados a estados. Quando comparadas a linguagens de programação, PSMs possuem o ponto forte de representar diretamente os estados do sistema. Aliás, HCFSMs e linguagens de programação podem ser representadas com PSMs: um programa pode ser visto como uma PSM contendo um único nó folha, o qual traz o código do programa, escrito através das sentenças de programação da PSM; no outro extremo, uma HCFSM pode ser vista como uma PSM onde os seus nós folha não contêm qualquer sentença de programação (uma vez que HCFSMs não admitem construções de programação).

2.3. TAXONOMIA E TERMINOLOGIA DE MODELOS PARA SISTEMAS DIGITAIS – VSI ALLIANCE

O Grupo de Trabalho para Desenvolvimento de Projeto em Nível de Sistema (SLD DWG), da VSI Alliance, apresenta a seguinte taxonomia para modelos associados ao **domínio de sistemas digitais** (versão 2.1 – julho/2001 [4]):

- a) **Modelos de sistema;**
- b) **Modelos arquiteturais;**
- c) **Modelos de *hardware*;**
- d) **Modelos de *software*.**

Para uma boa compreensão da capacidade representativa de cada categoria de modelo, o SLD DWG elaborou uma forma de classificação (taxonomia) baseada no trabalho desenvolvido pelo RTWG, de taxonomia e terminologia para modelagem em VHDL [5]. O SLD DWG estendeu a taxonomia do RTWG, passando-a do domínio de modelos VHDL para o domínio de modelos utilizados no projeto de sistemas digitais em geral.

A seção 2.3.1 trata das definições inerentes ao **modelo de taxonomia** proposto pelo grupo. Na seção 2.3.2 são dados **conceitos comuns de modelagem**, enquanto na seção 2.3.3 são apresentados os **modelos envolvidos no projeto de sistemas digitais**. Todas as classes de modelos são acompanhadas da descrição de seus atributos, de acordo com os eixos de taxonomia definidos a seguir.

2.3.1. DEFINIÇÃO DA TAXONOMIA PROPOSTA

A taxonomia (de modelos) procura enquadrar os diversos modelos, de acordo com um conjunto de **atributos** relevantes para os projetistas de sistemas digitais, criando **tipos de modelos**, para os quais a taxonomia visa estabelecer **definições formais concisas e não ambíguas**.

Os atributos da taxonomia proposta pelo SLD DWG são representados através de **cinco eixos**, os quais caracterizam de forma explícita a resolução de detalhes relativa a um modelo. Estes eixos são:

- i) **Precisão Temporal;**
- ii) **Precisão de Dados;**
- iii) **Precisão Funcional;**
- iv) **Precisão Estrutural; e**
- v) **Nível de Abstração de Programação (*Software*).**

Os detalhes de resolução dos quatro primeiros eixos citados são aplicados em duas diferentes visões, quais sejam, a **interna** e a **externa**. A distinção entre estas visões é importante na seleção, uso e construção de modelos, uma vez que ela possibilita clareza e precisão [13]. Outras terminologias geralmente misturam atributos relacionados à parte interna do modelo com atributos similares, visíveis através de interface. A **resolução interna** faz referência à maneira como um modelo descreve os aspectos temporais, funcionais, estruturais e de granulosidade de dados, dos elementos contidos **dentro dos limites do dispositivo modelado**. Por outro lado, a **resolução externa** faz referência à forma como um modelo descreve a **interface** (detalhes de E/S) do dispositivo modelado, para os outros dispositivos que possam estar conectados ao dispositivo em foco. Os detalhes externos podem

incluir aspectos temporais e funcionais (comumente denominados em conjunto como protocolo), assim como aspectos estruturais (portas do dispositivo) e de granulosidade de dados (valores de sinais).

Considerando-se a duplicação decorrente das resoluções interna e externa, os quatro primeiros eixos correspondem a oito eixos. Como nenhum deles trata do aspecto de coprojeto *hardware/software* do modelo, foi criado um nono eixo (quinto item da lista anterior) para representar o nível de programabilidade em *software* de um modelo voltado para *hardware*. A referência [4] afirma que este eixo serve também para um propósito oposto, qual seja, o de representar o nível de abstração de um componente de *software* em termos do modelo complementar em *hardware* que o interpretará; entretanto, nenhum dos modelos citados adota tal significado.

A figura 2.1 [4] apresenta os eixos de taxonomia, sendo os níveis de precisão descritos logo na seqüência. Ressalta-se que, quanto maior o nível de precisão (resolução) menor o nível de abstração, e vice-versa.

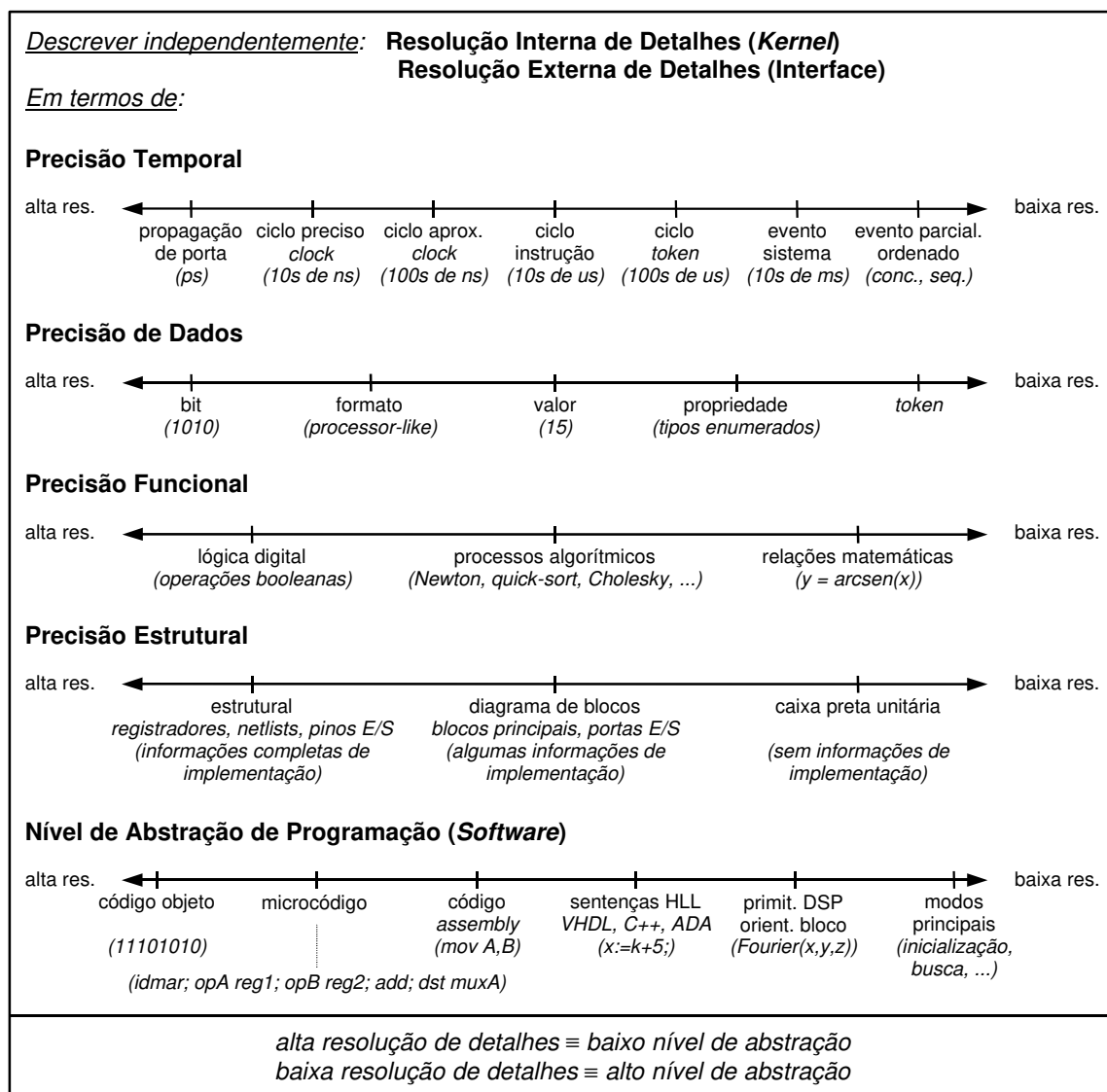


Figura 2.1. Eixos da taxonomia [4].

2.3.1.1. EIXO DE PRECISÃO TEMPORAL

O eixo de precisão temporal representa o **grau de precisão de eventos** que são modelados de acordo com uma **escala de tempo**. Seus níveis de precisão, em ordem crescente, são:

- a) **Evento parcialmente/completamente ordenado**: eventos descritos neste nível têm uma relação de ordenação parcial ou completa, especificada através do início e do término de cada evento, não em função de unidades de tempo, mas sim em função de relações de precedência entre eventos;
- b) **Evento do sistema**: neste nível, os tempos de início e de término das funções principais do sistema são indicados por unidades de tempo na ordem de milhares ou milhões de ciclos de *clock*;
- c) **Ciclo de *token* (ciclo de dados)**: a precisão de eventos neste nível é definida em termos do fluxo de *tokens* de dados ou de controle entre uma dada unidade de processamento funcional e outra. Um exemplo é a quantidade de quadros por segundo que um sistema de processamento de imagens pode processar.
- d) **Ciclo de instrução**: eventos neste nível são especificados em termos do processamento de um fluxo de instrução. Este nível é mais preciso do que o anterior, uma vez que várias instruções podem ser utilizadas para processar um *token* de dados. Um ciclo de instrução compreende vários ciclos de *clock*;
- e) **Ciclo aproximado de *clock***: enquanto o nível anterior trata de fluxo de instrução, este nível trata de cada instrução de um fluxo, contabilizando um número aproximado de ciclos de *clock* (do sistema) para cada instrução;
- f) **Ciclo preciso de *clock***: neste nível, o número de ciclos de *clock* para cada instrução é preciso. Eventos que ocorram durante o processamento de uma instrução são indicados com precisão exata, sobre qual ciclo de *clock* eles ocorrerão;
- g) **Propagação de porta**: neste nível, os tempos de início e de término de um evento são descritos através de unidades de tempo precisas (ns ou ps), dentro dos ciclos de *clock*. A precisão neste nível está associada aos modelos de interconexão e de nível de circuito utilizados.

O conceito de seqüência em tempo zero (**ciclo delta**), o qual possibilita concorrência instantânea, existe para todos os níveis citados.

2.3.1.2. EIXO DE PRECISÃO DE DADOS

Este eixo representa a **granulosidade dos valores** descritos por um modelo, ou seja, o **formato** de seus valores. Quanto maior for a abstração da representação de um valor (maior granulosidade), menor será o número de detalhes de implementação envolvidos. Os níveis de precisão de dados, em ordem crescente, são:

- a) ***Token***: neste nível, dados são representados sem quaisquer detalhes de implementação (estrutura, tamanho, valores, entre outros) e de quantidade de informação;
- b) **Propriedade**: formato de dados geralmente derivado de formatos previamente definidos. Como principal exemplo, os tipos enumerados;
- c) **Valor**: neste nível, dados podem assumir valores tais como real ou inteiro, sem que detalhes de implementação (que caracterizem a representação do valor como ponto fixo ou como ponto flutuante, entre outros) estejam presentes;

- d) **Formato (*processor-like*)**: neste nível, os dados assumem valores que são representados de acordo com as características de implementação de um dado processador, tal como *Motorola*, *Intel*, *AMD* etc. Estas características de implementação fornecem detalhes sobre o formato de representação (ponto fixo ou ponto flutuante, número de *bytes* da mantissa, do expoente etc.);
- e) **Bit**: neste nível, a representação final de um valor (*bit a bit*) pode ser usada. Cada *bit* pode ser representado binariamente ou por multi-valorização. Exemplos: 0, 1, X, Z.

Estas representações de formato podem aparecer de forma composta em um modelo.

2.3.1.3. EIXO DE PRECISÃO FUNCIONAL

Este eixo representa o **nível de detalhes** com o qual um modelo descreve a **funcionalidade** de um componente ou sistema. Seus níveis de resolução são:

- a) **Relações matemáticas**: neste nível, a funcionalidade é descrita através de um conjunto de equações matemáticas, sem qualquer relação de seqüenciamento;
- b) **Processos algorítmicos**: neste nível, um algoritmo descreve a funcionalidade, impondo relações de seqüenciamento, através da ordenação de operações e do estabelecimento de um fluxo de controle. Detalhes sobre como o algoritmo será implementado não constam deste nível;
- c) **Lógica digital**: nível no qual a funcionalidade é especificada através de operadores booleanos.

2.3.1.4. EIXO DE PRECISÃO ESTRUTURAL

Este eixo representa o **nível de detalhes** com o qual um modelo descreve a maneira **como um componente é construído** a partir de suas **partes constituintes** (módulos). Os níveis de precisão são caracterizados pela quantidade de informações de implementação. Assim, para exemplificar como estes níveis estariam associados a um modelo, considere a representação estrutural de um circuito integrado (CI):

- a) **Sem informações de implementação**: CI representado por um único grande bloco;
- b) **Com algumas informações de implementação**: CI representado como um conjunto de blocos interligados, denotando, por exemplo, unidades funcionais (como ULAs), de armazenamento (como registradores e arquivos de registradores) e de controle (unidade de controle, por exemplo);
- c) **Informações completas de implementação**: CI representado como um conjunto composto pela interconexão de unidades simples (como portas lógicas), unidades compostas (como *flip-flops*) e unidades mais complexas (tais como registradores, multiplicadores, somadores e seletores).

2.3.1.5. EIXO DO NÍVEL DE ABSTRAÇÃO DE PROGRAMAÇÃO (EIXO DE SOFTWARE)

Este eixo corresponde ao **nível de granulosidade** das **instruções de software interpretadas e executadas** por um modelo de **componente em hardware**. Em ordem crescente de resolução (decrecente de abstração) os níveis que compõem este eixo são:

- a) **Modos principais:** neste nível, o *software* é especificado em termos de modos de trabalho (computação) fundamentais. Exemplos destes são os modos de busca, inicialização e espera, dentre outros;
- b) **Primitivas DSP orientadas a bloco:** o *software* é expresso neste nível através da execução de blocos e da chamada a funções, com seus respectivos parâmetros;
- c) **Linguagem de alto nível (HLL – *High Level Language*):** este nível utiliza-se de sentenças escritas em linguagens de alto nível (C/C++, VHDL, Verilog, ADA, dentre outras), as quais são independentes de máquina (redirecionáveis). Geralmente, linguagens de alto nível fornecem construções de controle e tratamento de variáveis de dados através de nomes simbólicos, de acordo com uma gramática;
- d) **Código *assembly*:** neste nível, o *software* é descrito através de um código baseado em mnemônicos, o qual possui uma expressão sintática limitada, geralmente restrita a um conjunto de registradores e de operadores lógicos ou aritméticos simples. Variáveis de dados são tipicamente relacionadas a endereços específicos de memória. Código *assembly* pode ser gerado manualmente ou através do uso de compiladores;
- e) **Microcódigo:** este nível representa o *software* como um conjunto de micro-instruções de controle executáveis, responsáveis pelo controle das diversas fases de execução e das estruturas internas de um processador. Enquanto o código *assembly* especifica uma operação por ciclo de instrução, o microcódigo é responsável pela configuração das várias unidades do processador (multiplexadores, barramentos, unidades funcionais e de armazenamento, dentre outras) em cada ciclo de *clock* (através de sinais de controle), para executar o tipo de operação geralmente especificada por uma única instrução *assembly*. Código *assembly* pode ser traduzido para microcódigo. Neste processo, cada instrução *assembly* é tipicamente decomposta em uma rotina de microcódigo, a qual é um conjunto de instruções em microcódigo;
- f) **Código objeto:** neste nível, o *software* é descrito como um conjunto de instruções executáveis por uma máquina (daí o sinônimo código de máquina). Não incluindo representação simbólica, o código objeto é descrito em uma forma aceitável pela lógica digital do processador, geralmente através de números binários compostos pelos algarismos 0 e 1. O código objeto pode ser produzido por compiladores de linguagens de alto nível, montadores e sistemas de geração de microcódigo.

2.3.2. CONCEITOS GERAIS DE MODELAGEM

Nesta seção, são fornecidos conceitos gerais de modelagem, os quais encontram-se presentes em muitos níveis e tipos de modelos. A referência [4] traz estes conceitos divididos em três classes de modelos, quais sejam:

- a) **Modelos primários;**
- b) **Modelos especializados;** e
- c) **Modelos computacionais.**

Como a classe de **modelos computacionais** já foi tratada na taxonomia proposta por Gajski (seção 2.2), as duas seções seguintes (2.3.2.1 e 2.3.2.2) tratam de conceitos inerentes às classes de **modelos primários** e **especializados**. A figura 2.2 descreve os símbolos utilizados nas figuras 2.3 a 2.8, as quais iniciam, respectivamente, as subseções 2.3.2.1.1 a 2.3.2.2.3, detalhando o potencial de representação de cada eixo, para cada visão do sistema no modelo considerado, de acordo com os eixos de taxonomia da figura 2.1. Os símbolos

utilizados nas figuras indicam se os níveis de resolução interna e externa são atendidos (total ou parcialmente), ou não, para os eixos da taxonomia.

Símbolo	Descrição
	Modelo representa informações no(s) nível (níveis) de resolução assinalado(s)
	Modelo representa parcialmente/opcionalmente informações no(s) nível (níveis) de resolução assinalado(s)
	Modelo não representa informações no eixo

Figura 2.2. Símbolos para indicação dos níveis de resolução.

2.3.2.1. CLASSE DE MODELOS PRIMÁRIOS

Todos os modelos podem ser classificados de acordo com **aspectos primários**, quais sejam, os aspectos **comportamentais**, **funcionais** e **estruturais**. Esta classificação encontra-se nas três subseções seguintes.

2.3.2.1.1. MODELO COMPORTAMENTAL

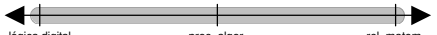
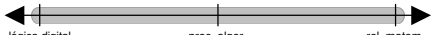
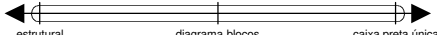
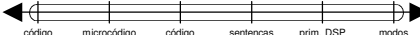
Eixo	Resolução Interna	Resolução Externa
Temporal		
Dados		
Funcional		
Estrutural		
Software		

Figura 2.3. Níveis de resolução – modelo comportamental.

Um modelo é classificado como comportamental quando representa a **funcionalidade** e a **temporização** de um sistema, ou um de seus componentes, **sem associar** uma **implementação** específica. Há modelos comportamentais para diferentes níveis de abstração. Estes níveis são definidos de acordo com a capacidade representacional de detalhes de implementação. Por exemplo, um modelo comportamental pode descrever as características temporais e funcionais de um processador executando um algoritmo abstrato, enquanto outro modelo pode descrever o processador num nível menos abstrato (maior resolução de detalhes), como no nível de conjunto de instruções. A granulosidade interna e externa dos valores de dados depende do nível de abstração do modelo. Modelos comportamentais podem ser resumidos como modelos com temporização.

2.3.2.1.2. MODELO FUNCIONAL

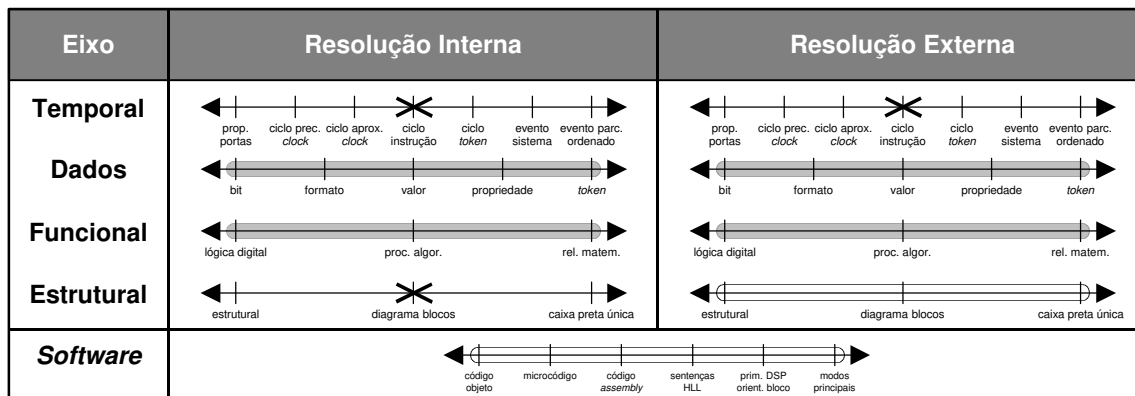


Figura 2.4. Níveis de resolução – modelo funcional.

A diferença entre um modelo classificado como comportamental de um modelo **funcional**, é que este último **não possui temporização**. Há modelos funcionais para os diferentes níveis de abstração, sendo estes níveis definidos de acordo com a precisão dos detalhes de implementação. Por exemplo, um modelo funcional pode descrever um algoritmo qualquer, enquanto outro tipo de modelo funcional pode descrever a função da unidade lógico-aritmética na execução daquele algoritmo.

Um modelo funcional pode ser descrito por meio de funções matemáticas cujos domínios e contra-domínios são constituídos por variáveis tomadas da interface do componente modelado. Se os conjuntos domínio e contra-domínio forem disjuntos, então o primeiro é denominado como **entradas** e o segundo como **saídas** do componente modelado. Um modelo funcional é denominado **parcial** se a união dos conjuntos domínio e contra-domínio não for equivalente ao conjunto formado por todas as variáveis de interface do componente modelado.

2.3.2.1.3. MODELO ESTRUTURAL

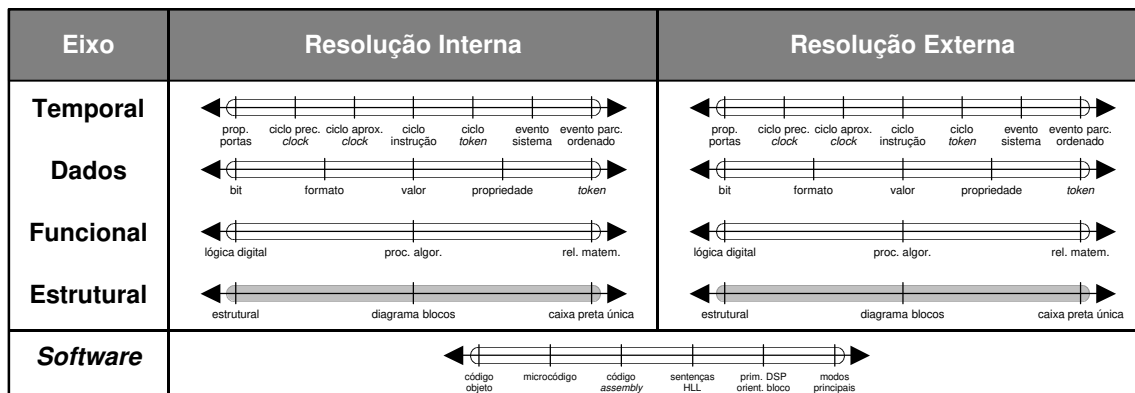


Figura 2.5. Níveis de resolução – modelo estrutural.

Um modelo **estrutural** representa o componente ou sistema através da **interconexão** entre suas **partes constituintes**, as quais também podem representar componentes ou sistemas, descritos através de modelos comportamentais, funcionais ou estruturais. Esta

hierarquia pode representar, por exemplo, a organização de um conjunto de módulos de *software* ou a organização física de uma implementação específica. Um modelo estrutural pode ser simulado somente se os componentes presentes no nível hierárquico mais baixo, formado pelos chamados nós folha, forem descritos através de modelos comportamentais ou funcionais. Desta forma, a resolução dos eixos temporal, funcional e de dados dependerá dos modelos adotados para os nós folha de um modelo decomposto estruturalmente. Já a resolução do eixo estrutural dependerá da granulosidade dos blocos adotados. Assim como ocorre com os modelos das classes comportamental e funcional, os modelos desta classe apresentam-se distribuídos em diferentes níveis de abstração.

2.3.2.2. CLASSE DE MODELOS ESPECIALIZADOS

Modelos especializados são aqueles que visam atender propósitos específicos, podendo abranger vários níveis de abstração. Há três classes desses modelos, quais sejam:

- Modelos de desempenho;**
- Modelos de interface;** e
- Modelos de nível misto.**

2.3.2.2.1. MODELOS DE DESEMPENHO

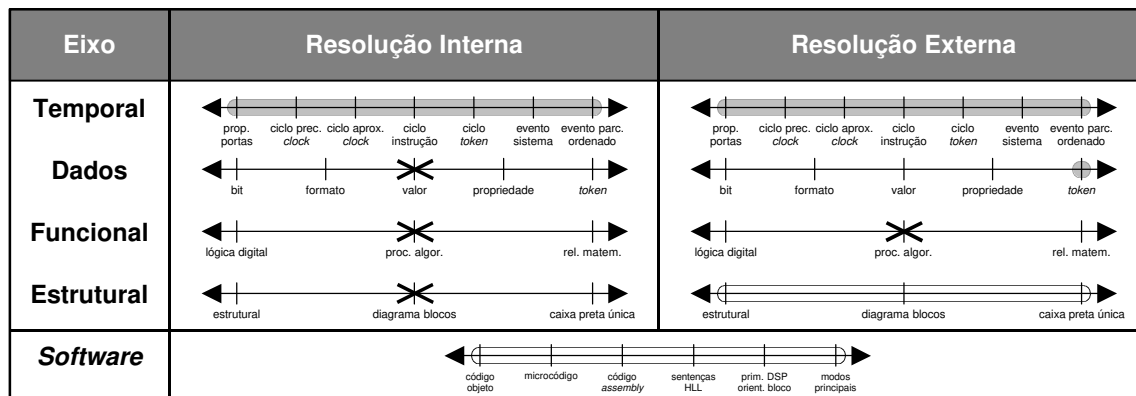


Figura 2.6. Níveis de resolução – modelos de desempenho.

Desempenho é um conjunto de **medidas de qualidade de um projeto**, utilizadas para **quantificar temporalmente a reação** do sistema (ou componente) a **estímulos**. Medidas de desempenho incluem tempo de resposta, *throughput* (vazão de transferência de dados) e a taxa de utilização de recursos. Há modelos de desempenho para diferentes níveis de abstração. Em geral, descrevem o tempo exigido para a realização de tarefas simples, tais como o acesso à memória de uma CPU. Modelos de desempenho são também conhecidos pela denominação de modelos **não interpretados**.

2.3.2.2.2. MODELOS DE INTERFACE

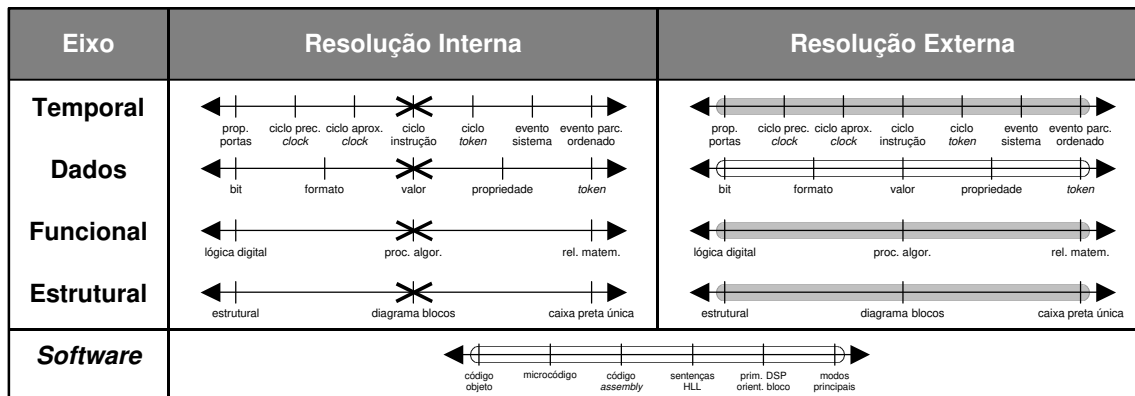


Figura 2.7. Níveis de resolução – modelos de interface.

Modelos de **interface** descrevem a maneira **como** um **componente troca informações com outros componentes**, presentes no ambiente que o cerca. Detalhes **temporais**, **funcionais** e **estruturais externos** (portas e parâmetros são exemplos) são fornecidos por um modelo de interface, para descrever a forma como é realizada esta troca de informações. Valores de dados externos não são modelados, a menos que representem informação de controle. Detalhes da visão interna do componente não são fornecidos. Há modelos de interface para diferentes níveis de abstração do detalhamento da interface. Modelos de interface são, às vezes, **erroneamente** denominados **barramento funcional** ou **interface comportamental**, os quais são, de fato, um **subconjunto** de modelos de interface de baixo nível de abstração.

2.3.2.2.3. MODELOS DE NÍVEL MISTO

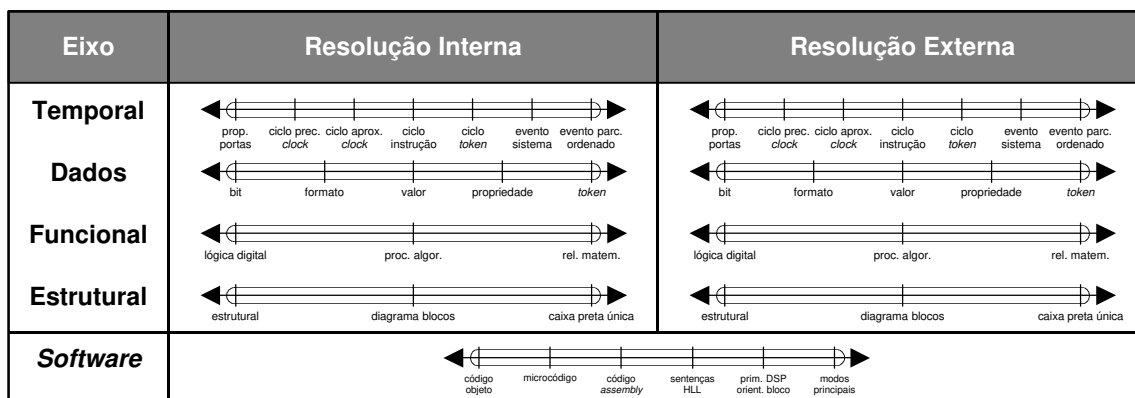


Figura 2.8. Níveis de resolução – modelos de nível misto.

Um modelo de **nível misto** (ou modelo **híbrido**) é uma **combinação** de modelos que se encontram em diferentes níveis de abstração.

2.3.3. MODELOS PARA SISTEMAS DIGITAIS

Nas seções anteriores, foram vistas características do modelo de taxonomia criado pelo SLD DWG da VSI Alliance [4], além de um conjunto de conceitos comuns de modelagem. A referência citada apresenta as seguintes classes de modelos, para utilização no projeto de sistemas digitais:

- a) **Modelos de sistema;**
- b) **Modelos arquiteturais;**
- c) **Modelos de *hardware*;**
- d) **Modelos de *software*.**

A classe de modelos de *software* está em construção na versão atual do documento, correspondendo às formas de representação de *software* descritas na seção 2.3.1.5. As demais classes são abordadas na sequência. As figuras 2.9 a 2.21 detalham o potencial de representação de cada eixo, para cada visão do sistema no modelo considerado, de acordo com as convenções adotadas anteriormente (sintetizadas nas figuras 2.1 e 2.2).

2.3.3.1. MODELOS DE SISTEMA

Modelos de **sistema** são utilizados para descrever **sistemas digitais** com **alto grau de abstração**, **não incluindo** quaisquer informações acerca das **estruturas** de *hardware* e de *software* utilizadas na **implementação** de um sistema. Esta classe é composta por modelos algorítmicos, de especificação executável e de equação matemática.

2.3.3.1.1. ESPECIFICAÇÃO EXECUTÁVEL

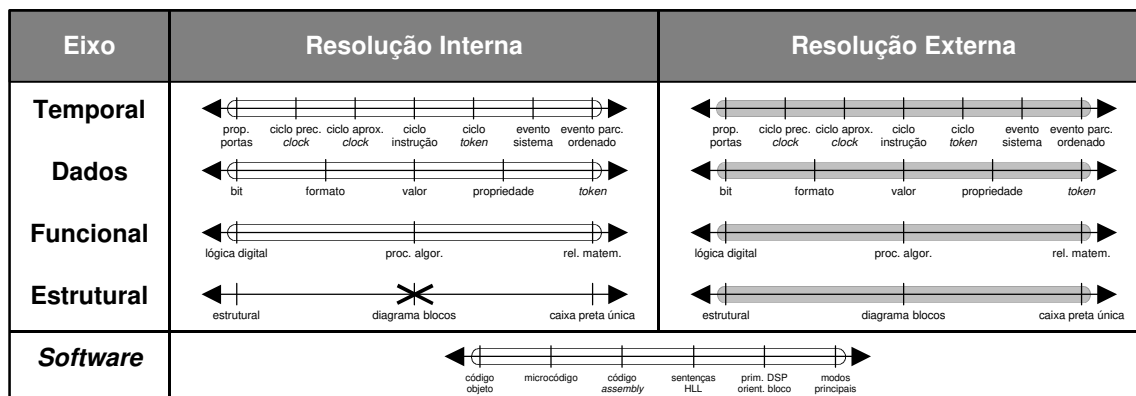


Figura 2.9. Níveis de resolução – especificação executável.

Especificação executável é uma **descrição comportamental** (função e tempo) de um objeto (sistema ou componente) de um projeto, **do ponto de vista da interface** desse objeto, quando da execução do mesmo em um ambiente de simulação por computador. Além da descrição comportamental, a especificação executável pode incluir, também, descrições de aspectos elétricos ou físicos do projeto em foco, tais como área, consumo e potência dissipada, os quais devem ser aferidos. Especificações executáveis podem descrever um objeto em um nível de abstração qualquer, tal como o nível de componentes de *software* e/ou *hardware*, o nível arquitetural e o nível de sistemas multiprocessadores. Conforme já relatado neste texto, especificações executáveis podem ser utilizadas para propósitos de demonstração

de funcionalidade de um produto, documentação para as equipes envolvidas no processo de projeto, análise de propriedades de um objeto (verificação automática), exploração do espaço de projeto (alternativas de projeto) e síntese do projeto.

2.3.3.1.2. MODELOS DE EQUAÇÃO MATEMÁTICA

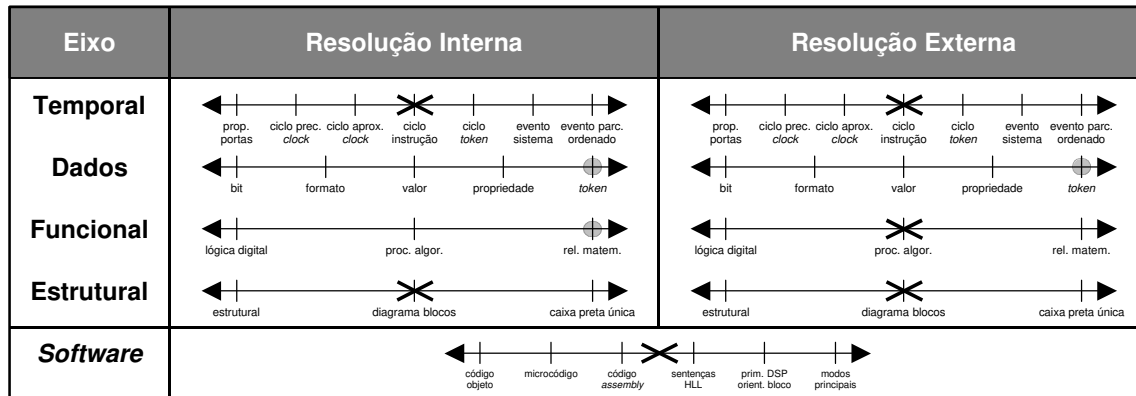


Figura 2.10. Níveis de resolução – modelos de equação matemática.

Modelos de **equação matemática** descrevem a **relação funcional** entre **valores de entrada e de saída** de um sistema, ou de um de seus componentes, através do uso de expressões algébricas. Ao contrário de uma descrição algorítmica, o modelo matemático não impõe uma seqüência específica de operações para a implementação de uma função. A principal atribuição de um modelo matemático é a de testar equações matemáticas e seus respectivos parâmetros quanto às exigências que devem ser atendidas por um sistema ou por um de seus componentes. Exemplos de descrições matemáticas para funções de um sistema:

$$x = \sqrt{r},$$

$$y = \cos(s),$$

$$z = \arctg(t).$$

As funções acima representam relações matemáticas bem definidas, sem indicar quais métodos seriam empregados para calcular a raiz quadrada, o co-seno e o arco tangente dos números representados pelas variáveis de dados.

2.3.3.1.3. MODELOS ALGORÍTMICOS

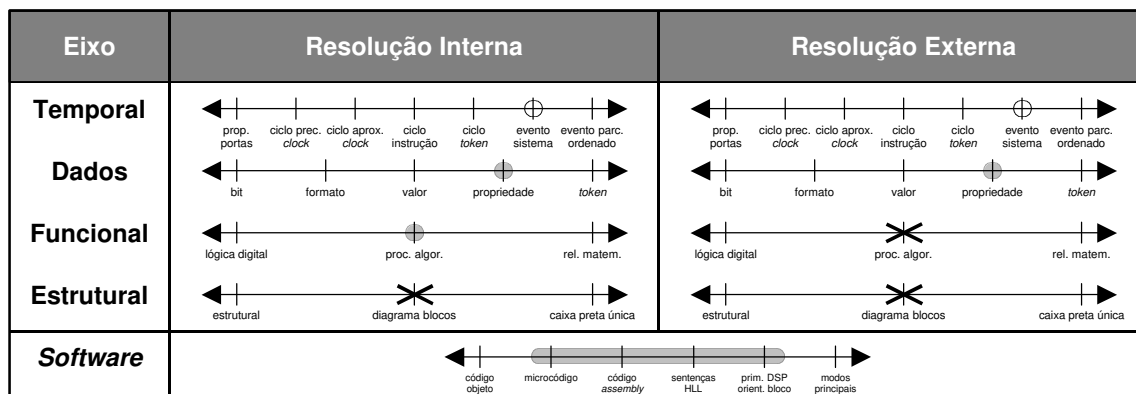


Figura 2.11. Níveis de resolução – modelos algorítmicos.

Um modelo **algorítmico** possibilita a **descrição** do **procedimento** adotado na implementação de uma função, através da especificação de uma **seqüência de operações aritméticas e construções condicionais e iterativas (loops)**. Trata-se de uma descrição menos abstrata do que a descrição matemática, uma vez que fornece detalhes de implementação de funções. Além disso, transforma dados reais. São exemplos, os algoritmos *quick-sort* (para ordenação de valores), a decomposição matricial de Cholesky, os métodos para encontrar raízes de um polinômio (bissecção, falsa posição, iterativo linear, Newton, Briot-Ruffine, dentre outros), transformadas de Fourier, entre tantos outros.

O principal objetivo de um modelo algorítmico é avaliar o quão bem um algoritmo satisfaz as exigências numéricas de um sistema, na implementação de uma tarefa matemática. São também úteis na avaliação de efeitos numéricos, decorrentes de definições acerca de precisão finita e formato de representação de parâmetros (ponto fixo ou flutuante), sobre os resultados de uma função.

2.3.3.2. MODELOS ARQUITETURAIS

Esta seção apresenta termos utilizados para descrever modelos abstratos da arquitetura de *hardware* e de *software* de um sistema. Modelos arquiteturais descrevem somente a estrutura básica de uma aplicação e os componentes em *hardware* para os quais a aplicação é mapeada. Detalhes adicionais ao processo de definição da arquitetura para uma aplicação, são relegados para outros modelos, mais detalhados, de *hardware* e de *software*. São quatro os modelos arquiteturais tratados nesta seção:

- Modelo de desempenho baseado em token;**
- Modelos de desempenho comportamental abstrato;**
- Primitivas de tarefa com grafo de fluxo de dados; e**
- Modelos ISA (*Instruction Set Architecture*).**

2.3.3.2.1. MODELO DE DESEMPENHO BASEADO EM TOKEN

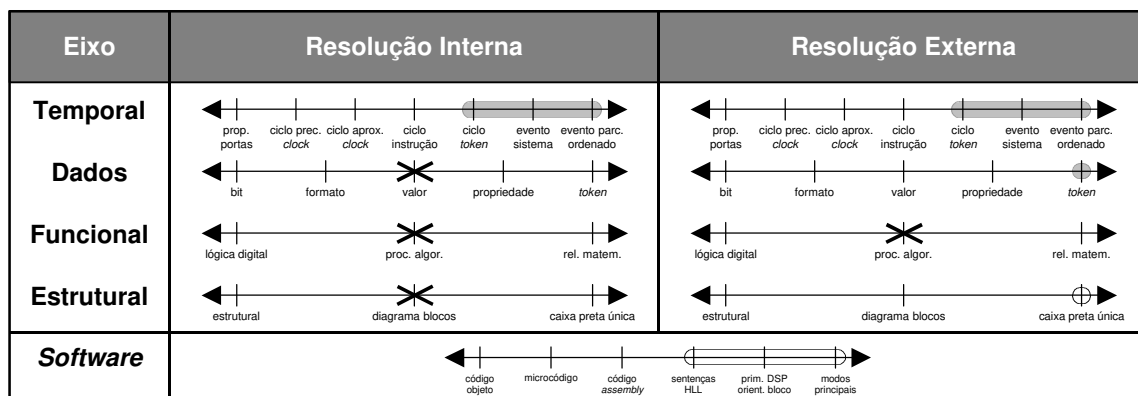


Figura 2.12. Níveis de resolução – modelo de desempenho baseado em token.

O modelo de **desempenho baseado em token** pertence à classe de modelos de desempenho (item 2.3.2.2.1), sendo voltado para a **modelagem de desempenho da arquitetura de um sistema**. As medidas de desempenho adotadas por este modelo incluem o tempo de resposta e o *throughput* do sistema, além da taxa de utilização de seus recursos.

O modelo de desempenho baseado em *token* descreve o tempo gasto por um sistema na execução de funções importantes. A granulosidade dos dados não é tratada, exceto para informações de controle. A estrutura do sistema é descrita, na visão externa, no nível de nós principais (componentes interligados, tais como chaves, elementos de processamento, memórias, unidades de E/S); internamente, a estrutura destes nós principais não costuma ser descrita.

O objetivo principal deste modelo é avaliar o quão bem um determinado grupo de componentes (e sua interconexão) atendem às exigências de latência e de *throughput* de processamento de um sistema. A definição de quais componentes e de como eles devem estar interligados, passa pelas seguintes escolhas:

- Número e tipos de componentes;
- Tamanho e largura da banda de comunicação de memórias e *buffers*;
- Topologia da rede (barramento, anel, cubo, árvore, ponto-a-ponto etc.);
- Protocolos;
- Particionamento, mapeamento e escalonamento de tarefas da aplicação nos elementos de processamento (ASICs, FPGAs, de propósito geral);
- Esquema de controle do fluxo.

2.3.3.2.2. MODELOS DE DESEMPENHO COMPORTAMENTAL ABSTRATO

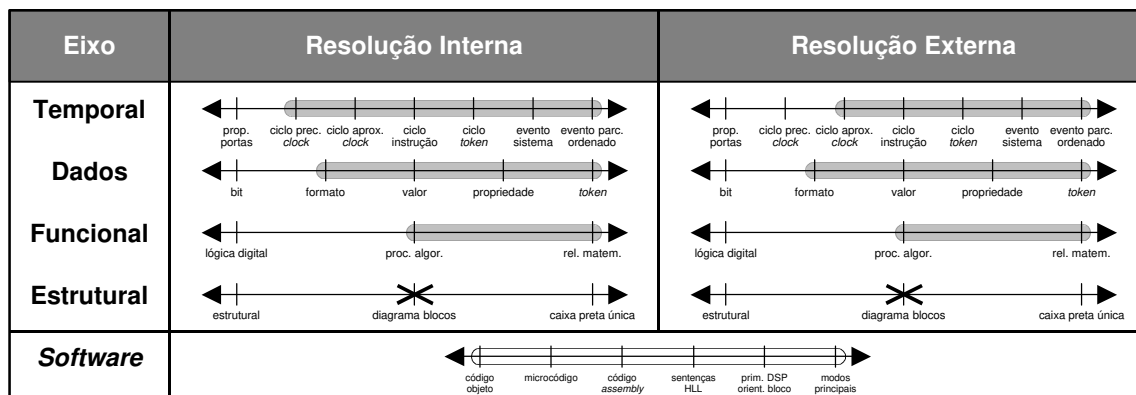


Figura 2.13. Níveis de resolução – modelos de desempenho comportamental abstrato.

O modelo de **desempenho comportamental abstrato** é um **modelo composto**, o qual é formado por **um modelo comportamental** (para descrever a aplicação de um sistema), além de **um conjunto de modelos de desempenho** (para descrever os efeitos que escolhas arquiteturais têm sobre o sistema em foco).

Seu modelo comportamental é expresso no nível de passagem de *tokens*, sendo os valores de dados contidos dentro dos *tokens* tanto modelados quanto processados funcionalmente, ao contrário do que ocorre no modelo de desempenho baseado em *token*. Desta forma, o comportamento da aplicação é modelado de forma acurada, com os *tokens* representando dados reais que são transmitidos entre as funções do sistema, assim como mensagens de controle que são passadas entre componentes da arquitetura. A interface destes componentes é modelada abstratamente, não descrevendo o número de pinos das portas. Assim, ao invés de descrever a interface de memória de um microprocessador através de suas

linhas de controle e barramentos de dados e de endereços (com seus respectivos tamanhos), o modelo comportamental permite a descrição da interface como uma única porta, associada a um tipo de dados arbitrariamente complexo.

Os modelos de desempenho abstratos são utilizados para explicar os efeitos no desempenho de um sistema, devido a escolhas arquiteturais (definição dos blocos funcionais da aplicação e dos mecanismos de comunicação entre estes blocos). Estes modelos permitem a modelagem de estimativas de tempo de processamento para cada bloco funcional, além dos atrasos no protocolo de comunicação entre estes blocos. Outras estimativas incluem a disputa por recursos computacionais (processadores, por exemplo) e de comunicação (tal como barramentos). Geralmente, os modelos de desempenho não descrevem estruturas internas dos componentes computacionais e de comunicação; entretanto, permitem a descrição em vários níveis de abstração dos protocolos de comunicação envolvidos. Protocolos descritos em alto nível fazem uso de transferência de *tokens* inteiros, ao passo que, se forem descritos em um nível baixo de abstração, farão uso de *tokens* decompostos em uma sequência de transações de comunicação/barramento.

O objetivo principal de um modelo de desempenho comportamental abstrato é verificar a adequação de uma arquitetura, perante o comportamento pretendido para a aplicação de um sistema. Desta forma, este modelo auxilia na escolha de recursos computacionais e de comunicação da arquitetura de um sistema, através da avaliação de configurações envolvendo diferentes números e tipos de processadores, de unidades de *hardware* dedicado, de barramentos de comunicação e protocolos, de capacidades de memórias e a distribuição destas. Outros objetivos deste modelo são:

- a) Estabelecimento e verificação conjunta das exigências funcionais e temporais de um componente, de forma a assegurar a consistência deste com as exigências do sistema como um todo;
- b) Documentação da implementação pretendida para o sistema;
- c) Reutilização do projeto, quando da alteração na implementação de componentes ou interfaces;
- d) Compreensão e otimização do projeto de sistemas complexos;
- e) Geração de vetores de teste, para utilização no projeto detalhado de componentes.

2.3.3.2.3. PRIMITIVAS DE TAREFA COM GRAFO DE FLUXO DE DADOS

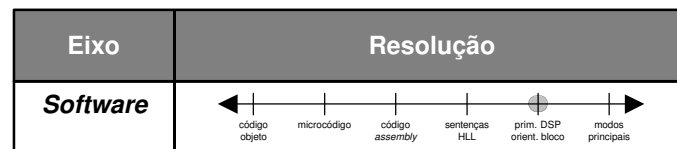


Figura 2.14. Nível de resolução – primitivas de tarefa com grafo de fluxo de dados.

No nível de **primitivas de tarefa**, o *software* é descrito através de **blocos funcionais**, os quais **independem de implementação em hardware ou em software**. **Grafos de fluxo de dados** podem ser utilizados para representar graficamente as primitivas de tarefa de um *software*.

2.3.3.2.4. MODELOS ISA

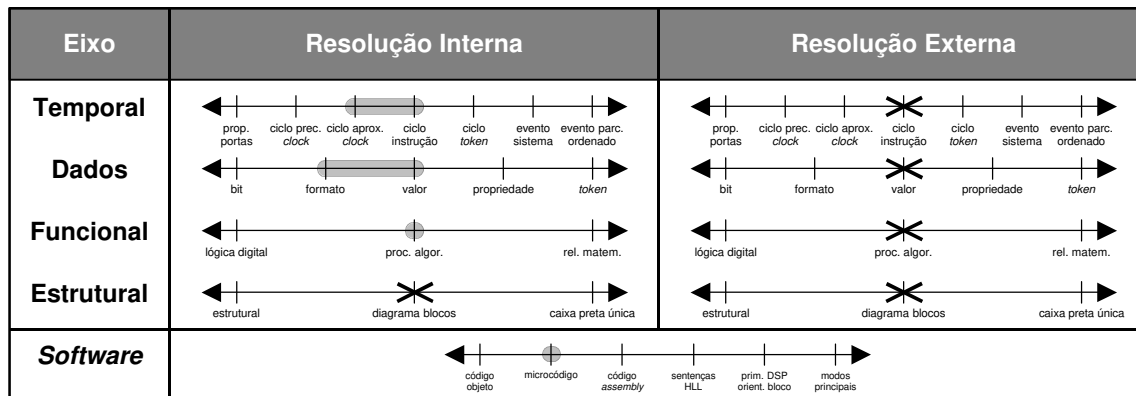


Figura 2.15. Níveis de resolução – modelos ISA.

Um modelo **ISA** (*Instruction Set Architecture*) descreve a função do **conjunto de instruções** adotado por um **processador**, assim como a **operação da memória** e do conjunto de **registradores** do processador. Qualquer código objeto para um dado processador poderá ser executado por um modelo ISA do mesmo, gerando os mesmos resultados, desde que o modelo parta dos mesmos estados iniciais e seja alimentado com as mesmas entradas do processador real (E/S simulada). Modelos de processador sem portas externas são classificados como um modelo ISA; já os modelos com portas de E/S externas são classificados como um modelo comportamental. A resolução temporal de um modelo ISA é a de ciclos de instrução. Valores de dados são do tipo *bit-true*, de acordo com os tamanhos de palavras referentes aos registradores e memória internos. Modelos ISA não contêm detalhes de implementação da estrutura interna.

Modelos ISA são importantes para os seguintes propósitos:

- Desenvolvimento eficiente do *software* residente de um processador, antes que este último possa estar pronto;
- Projeto eficiente do conjunto de instruções e da arquitetura dos registradores de um processador, visando uma aplicação específica;
- Documentação da funcionalidade das instruções de um processador;
- Medição de tempos de execução de rotinas em *software*.

2.3.3.3. MODELOS DE *HARDWARE*

Modelos de *hardware* são utilizados para descrever o *hardware* em níveis específicos de abstração. Esta seção apresenta os seguintes modelos de *hardware*:

- Modelo comportamental detalhado;**
- Modelos RTL (*Register Transfer Level*);**
- Modelos em nível lógico;**
- Modelos em nível de portas;**
- Modelos em nível de chaves;**
- Modelos em nível de circuito.**

2.3.3.3.1. MODELO COMPORTAMENTAL DETALHADO

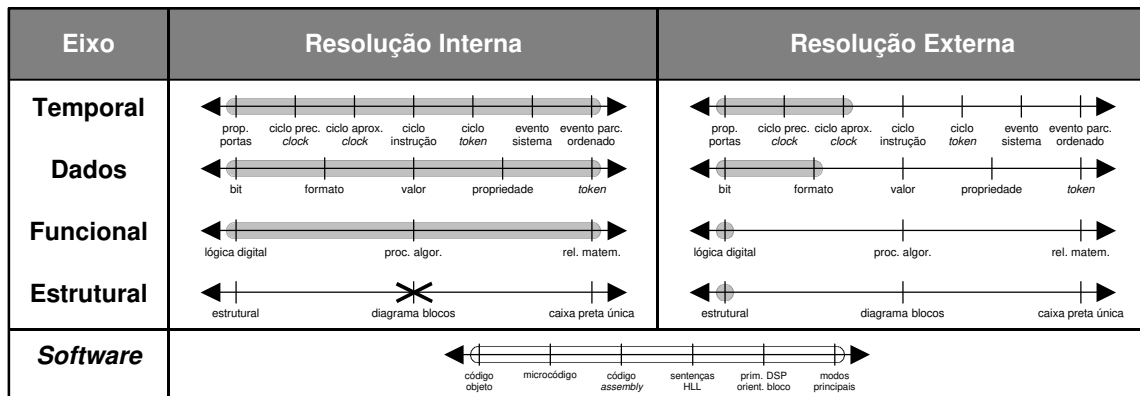


Figura 2.16. Níveis de resolução – modelo comportamental detalhado.

O modelo **comportamental detalhado** (também denominado **modelo funcional completo**) trata-se de um **modelo comportamental** que descreve a **interface** de um componente no **nível de pinos**. Este modelo apresenta toda a funcionalidade e temporização documentada de um componente, sem especificar a implementação da estrutura interna do mesmo.

Objetivos deste modelo incluem:

- Desenvolvimento e teste da estrutura, temporização e funcionamento da interface de um componente, especialmente no projeto em nível de placa;
- Exame de interações entre *drivers* de *software* e *hardware*;
- Substituição de estimativas iniciais por valores de tempo, em modelos de nível mais alto, aumentado, assim, a acurácia destes modelos.

2.3.3.3.2. MODELOS RTL

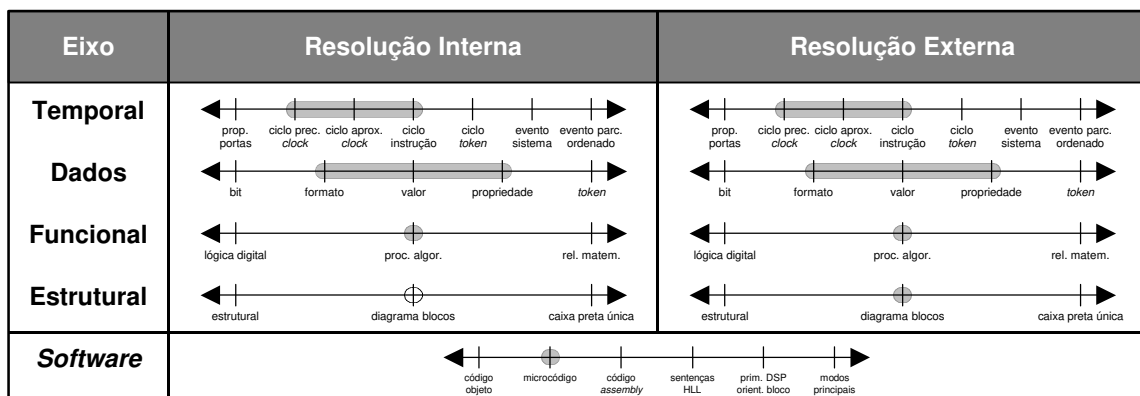


Figura 2.17. Níveis de resolução – modelos RTL.

Um modelo **RTL** (*Register Transfer Level* – nível de transferência entre registradores) descreve um sistema em termos de **registradores**, **circuitos combinacionais** e de **controle** e **barramentos** entre estes componentes. Circuitos de controle são geralmente implementados como FSMs (máquinas de estados finitos, seção 2.2.1). Modelos deste tipo oferecem alguns

detalhes (não explicitamente descritos) de implementação da estrutura interna, derivados de transformações de registradores. Linguagens de descrição de *hardware*, como VHDL, capturam modelos RTL.

Modelos RTL têm por objetivos:

- Desenvolvimento e teste da lógica de controle e da arquitetura interna de um componente do tipo CI (circuito integrado), de forma a satisfazer a funcionalidade pretendida e as restrições temporais inerentes ao CI projetado;
- Especificação do projeto em um formato neutro, o qual possa ser redirecionável para linhas de processo ou tecnologias específicas, através de **síntese automática (síntese comportamental + síntese lógica**, veja seção 1.2.2);
- Geração de vetores de teste.

2.3.3.3. MODELOS EM NÍVEL LÓGICO

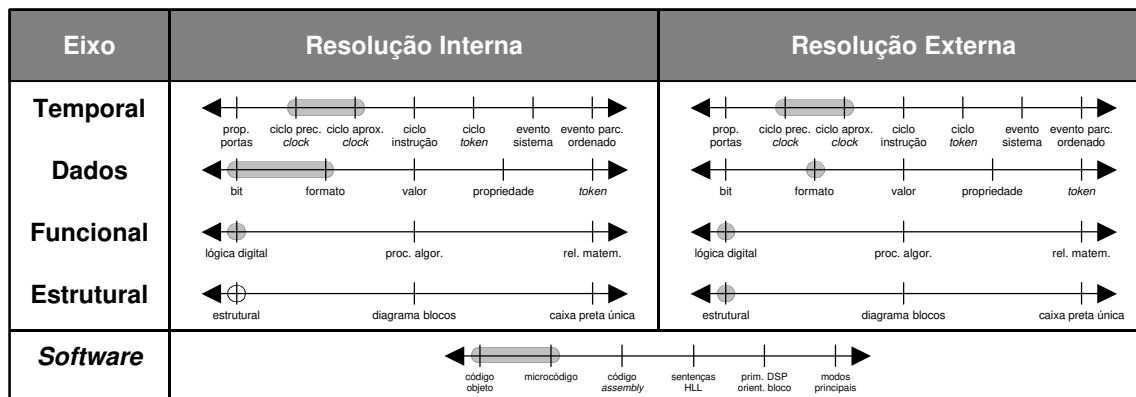


Figura 2.18. Níveis de resolução – modelos em nível lógico.

Modelos em **nível lógico** descrevem um componente em termos de **funções lógicas booleanas** e **dispositivos de memória simples**, tais como *flip-flops*. Estes modelos não descrevem a implementação em termos de portas lógicas; para chegar-se neste nível, deve-se reduzir (ou transformar) as expressões lógicas do modelo (em nível lógico) para formas funcionais equivalentes, visando implementação em termos de blocos lógicos.

O objetivo principal de um modelo em nível lógico é o desenvolvimento de expressões lógicas para posterior transformação em portas lógicas; naturalmente, o modelo deve ser utilizado para verificar se estas expressões lógicas estão de acordo com a funcionalidade pretendida para o componente. Outro objetivo é a representação do componente em um formato neutro, o qual possa ser redirecionável para linhas de processos ou tecnologias específicas, através de **síntese lógica** (veja seção 1.2.2).

2.3.3.3.4. MODELOS EM NÍVEL DE PORTAS

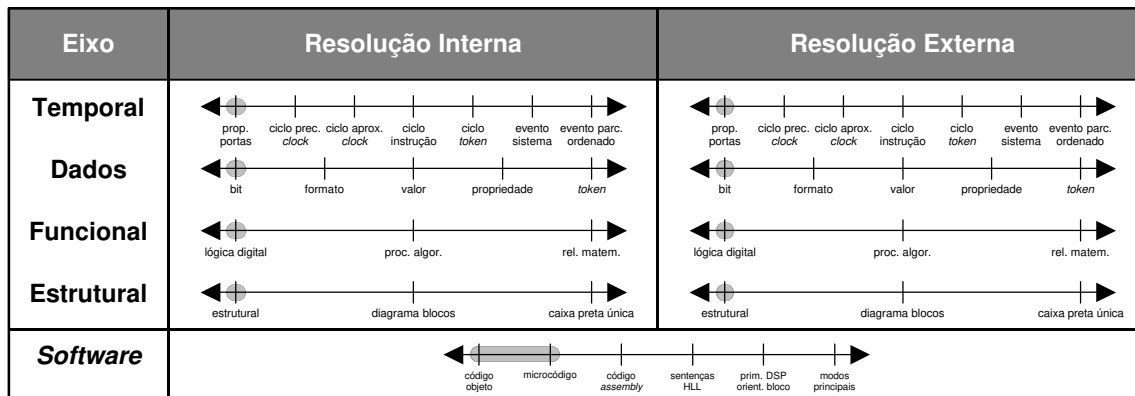


Figura 2.19. Níveis de resolução – modelos em nível de portas.

Modelos em **nível de portas** descrevem a **função**, a **temporização** e a **estrutura** de um **componente** em termos da **interconexão de blocos lógicos**, os quais implementam funções booleanas simples, tais como AND, OR, NOT, XOR, NAND, NOR. Um modelo em nível de portas descreve a estrutura real e os agrupamentos de portas lógicas, utilizados na implementação do componente em foco.

O objetivo principal de um modelo em nível de portas é a especificação da implementação de um componente de CI, através da interconexão de elementos tomados de uma dada biblioteca de família lógica. Outro objetivo é a geração de vetores de teste, os quais são utilizados para:

- Determinar de forma precisa a reação temporal do circuito a estímulos (permitindo aumentar a acurácia de modelos mais abstratos);
- Verificar se o projeto atende às exigências funcionais e temporais; e
- Otimizar a implementação em nível de portas do projeto lógico.

2.3.3.3.5. MODELOS EM NÍVEL DE CHAVES

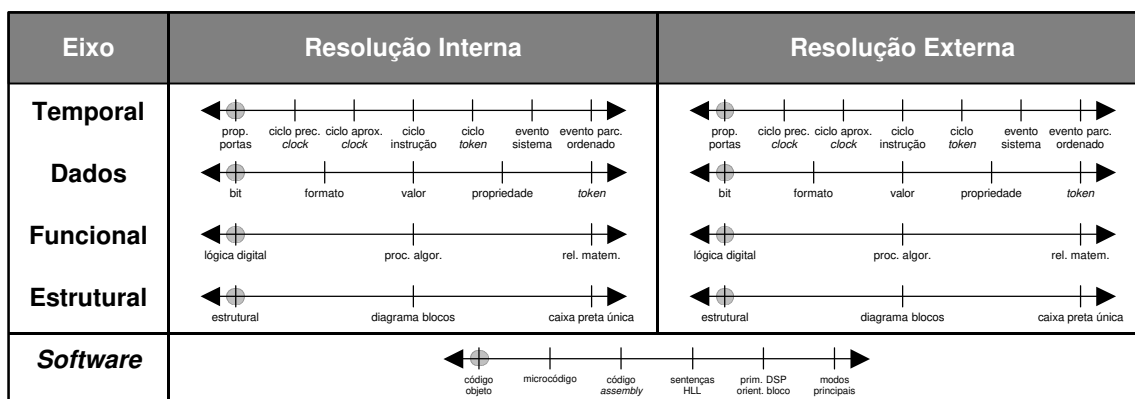


Figura 2.20. Níveis de resolução – modelos em nível de chaves.

Um modelo em **nível de chaves** descreve a **interconexão dos transistores** que compõem um circuito lógico. Estes transistores são modelados como **chaves liga/desliga controladas por voltagem**.

O objetivo principal de um modelo em nível de chaves é a determinação eficiente da reação de uma porção de um CI (porção esta formada por uma porta ou por um conjunto de portas) a estímulos, possibilitando melhor acurácia de modelos mais abstratos. Esta determinação é mais eficiente, porém mais grosseira, do que a gerada por modelos em nível de circuito.

2.3.3.3.6. MODELOS EM NÍVEL DE CIRCUITO

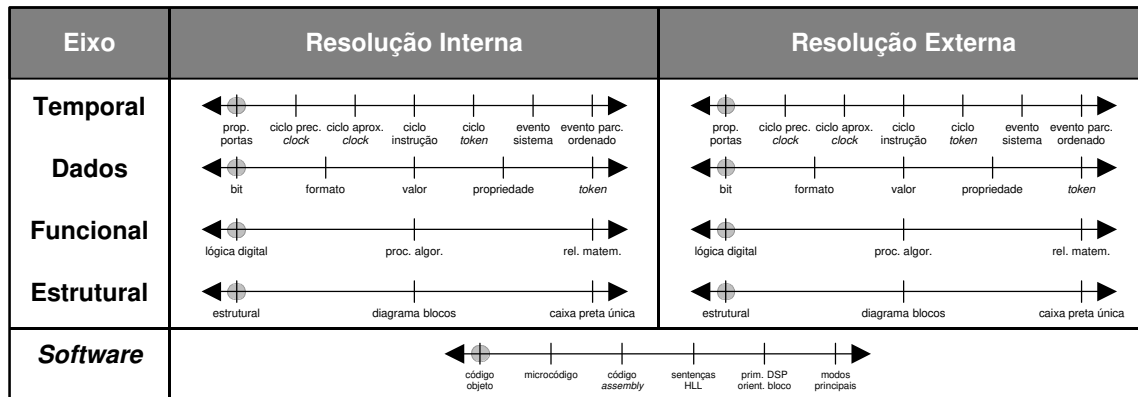


Figura 2.21. Níveis de resolução – modelos em nível de circuito.

Modelos em **nível de circuito** descrevem a **funcionalidade** de um circuito em termos de **diagramas voltagem-corrente** dos resistores, capacitores, diodos, transistores e indutores que integram o circuito, além da **interconexão** entre estes componentes.

O objetivo principal de um modelo deste tipo é a determinação da reação de uma porção de um CI, visando tanto otimizar o projeto desta porção (de acordo com suas exigências), como para determinar de forma acurada os tempos mínimo e máximo de propagação nesta porção.

CAPÍTULO 3: ARQUITETURAS

3.1. INTRODUÇÃO

Neste capítulo, são abordadas arquiteturas, as quais complementam modelos de especificação (capítulo 2), ao descrever como estes podem ser implementados. O objetivo de uma arquitetura é descrever quais os componentes utilizados na implementação de um sistema, a quantidade de cada componente e as formas de conexão entre os mesmos.

Há diversos tipos de arquiteturas, abrangendo desde controladores simples até sistemas voltados para processamento paralelo. Neste texto, é adotada a seguinte taxonomia [3]:

- a) **Arquiteturas para aplicações específicas;**
- b) **Arquiteturas para processadores de propósito geral;**
- c) **Arquiteturas para processamento paralelo.**

No primeiro grupo, situam-se arquiteturas que implementam sistemas de controle simples e sistemas DSP (*Digital Signal Processing*). No segundo grupo, encontra-se um modelo de arquitetura básico utilizado na implementação de processadores de propósito geral. Finalmente, no último grupo encontram-se arquiteturas utilizadas na implementação de processadores e sistemas que exploram o paralelismo na execução de tarefas, apresentando diferenças quanto ao mecanismo de controle, à organização do sistema de memória e às estruturas de interconexão empregadas.

3.2. ARQUITETURAS PARA APLICAÇÕES ESPECÍFICAS

3.2.1. ARQUITETURAS PARA CONTROLADORES SIMPLES

Um **controlador simples** (que não exija manipulação de dados complexa) pode ser implementado a partir de uma arquitetura diretamente derivada da máquina de estados finitos (FSM, de *Finite-State Machine*) representativa do mesmo. Conforme visto na seção 2.2.1, FSMs são formalmente definidas pela quintupla:

$$\langle S, I, O, f: S \times I \rightarrow S, h: S \times I \rightarrow O \rangle, \text{ FSM de Mealy (orientada a transição),}$$

ou

$$\langle S, I, O, f: S \times I \rightarrow S, h: S \rightarrow O \rangle, \text{ FSM de Moore (orientada a estado),}$$

onde: S = conjunto de estados,
 I = conjunto de sinais de entrada,
 O = conjunto de sinais de saída,
 f = função próximo estado,
 h = função de saída.

A arquitetura de um controlador simples (figura 3.1) é composta por um **registrador** (para armazenar o **estado atual** da FSM) e **dois blocos combinacionais**, representando as

funções **próximo estado** (f) e **de saída** (h) da FSM. O bloco que implementa a função f tem como saída o registrador, enquanto o bloco relativo à função h é responsável pela geração dos **sinais de saída** definidos em O . Na figura 3.1.a, as entradas de ambos os blocos são fomentadas pelo estado atual da FSM e pelo conjunto de sinais de entrada (I), uma vez que a função de saída de uma FSM de Mealy possui dois parâmetros ($h: S \times I \rightarrow O$). Por outro lado, na figura 3.1.b, o bloco referente à função de saída possui uma única entrada, qual seja, o estado atual da FSM, uma vez que FSMs de Moore possuem um único parâmetro em sua função de saída ($h: S \rightarrow O$).

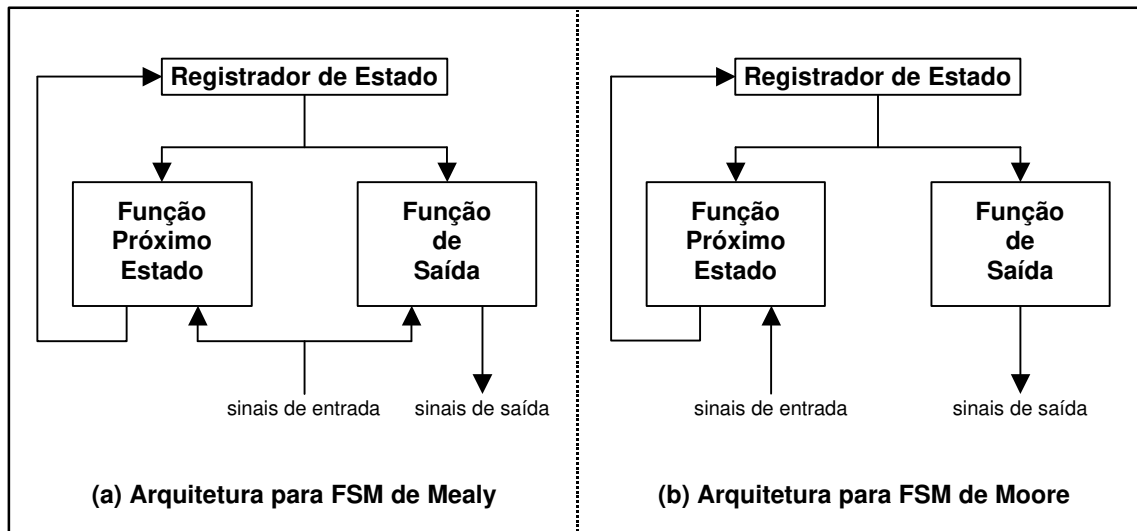


Figura 3.1. Arquiteturas para um controlador baseado em FSM.

Conforme visto na seção 1.2.2, a síntese de um controlador passa por três fases:

- 1ª) **Minimização de estados**, onde se aplicam técnicas de redução sobre os estados da FSM, com a conseqüente configuração final da função próximo estado (f) e da função de saída (h);
- 2ª) **Minimização lógica**, na qual são aplicadas técnicas de redução sobre as funções booleanas f e h da FSM minimizada; e
- 3ª) **Mapeamento tecnológico**, onde as funções booleanas reduzidas, obtidas a partir da minimização lógica de f e h , são mapeadas para portas pertencentes a uma dada biblioteca de portas lógicas.

3.2.2. ARQUITETURAS *DATAPATH*

Arquiteturas *datapath* são utilizadas em **sistemas transformacionais**, nos quais **transformações** (operações) pré-definidas são **repetidas** para diferentes **conjuntos de dados** (*samples de dados*), fornecidos **periodicamente** ao sistema. Nesta categoria, encontram-se os **sistemas de comunicação digital** e os **sistemas de processamento digital de sinais** (DSP, de *Digital Signal Processing*), os quais são usados para filtragem digital, compressão de sinais de vídeo, processamento de imagens, dentre outras aplicações.

A configuração usual de uma arquitetura *datapath* corresponde a um **conjunto de unidades lógico-aritméticas de alto desempenho**, conectadas em paralelo e **encadeadas em pipeline** para atingir alta produtividade (taxa de processamento – *throughput*).

Em relação ao **controle do fluxo de dados** dentro do *datapath*, dois casos são possíveis:

Caso I: **Não necessidade de um controlador.** Este caso ocorre em duas situações:

- I.1) Quando a arquitetura associa uma unidade lógico-aritmética exclusiva para cada operação realizada sobre um fluxo de dados; ou
- I.2) Quando um sinal de *clock* é responsável pelo posicionamento (alimentação) de dados em cada estágio do *pipeline*;

Caso II: **Necessidade de um controlador simples.** Este caso ocorre em três situações:

- II.1) Quando o número de unidades lógico-aritméticas é menor do que o número de operações sobre um fluxo de dados;
- II.2) Quando as unidades lógico-aritméticas executam funções diferentes ao longo do processamento de um fluxo de dados; ou
- II.3) Quando a arquitetura executa vários algoritmos.

Arquiteturas *datapath* podem ser implementadas como ASICs (*Application-Specific Integrated Circuits* – Circuitos Integrados Específicos a uma Aplicação). Entretanto, como estes impõem um aumento de tempo e de custo ao projeto, uma alternativa para a redução de custos é o uso de processadores DSP disponíveis no mercado. Tais processadores geralmente são dotados de *conversores A/D e D/A*, barramentos separados para instruções e dados, e funções pré-programadas (em ROM) como, por exemplo, para o cálculo de transformadas de Fourier e para filtragem digital.

3.3. ARQUITETURAS PARA PROCESSADORES DE PROPÓSITO GERAL

3.3.1. FSM COM *DATAPATH*

FSM com *datapath* (figura 3.2), ou **FSMD**, é um tipo de arquitetura que **combina** um **controlador** baseado em **FSM** com um *datapath* que pode ser constituído por unidades lógico-aritméticas, multiplicadores, registradores de deslocamento, seletores, barramentos e unidades de armazenamento (memórias, registradores).

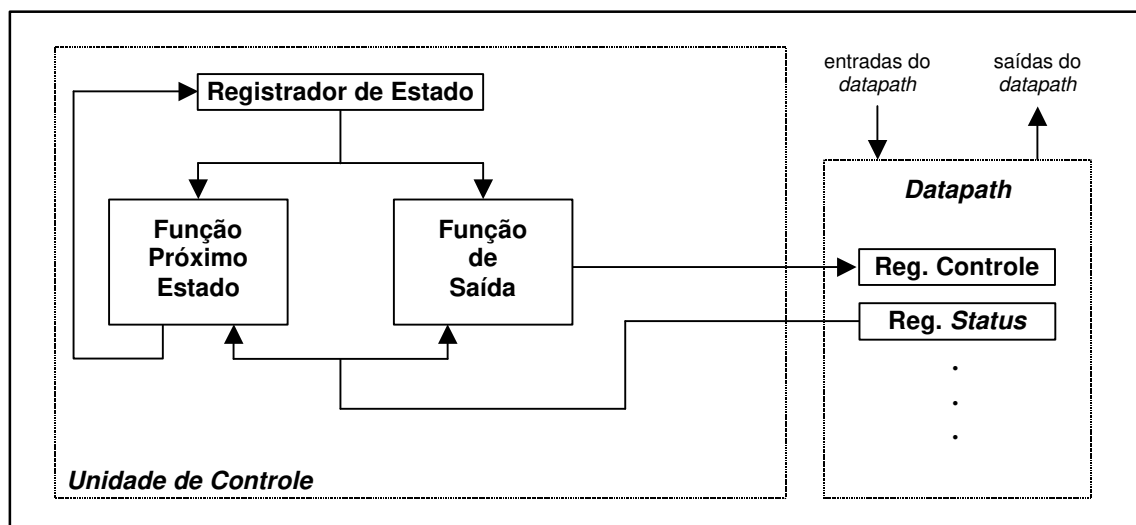


Figura 3.2. Exemplo de arquitetura FSMD.

Na figura 3.2, o componente FSM (*Unidade de Controle*) atua como um controlador, (conforme a arquitetura descrita na seção 3.2.1), no qual o conjunto de sinais de entrada corresponde ao estado do *Datapath* (**registrador Status**), enquanto o conjunto de sinais de saída corresponde aos sinais de controle enviados para o *Datapath* (**registrador Controle**). As entradas e saídas do componente *Datapath* poderão estar conectadas às unidades de memória, as quais fornecem dados para computação no *Datapath* e armazenam resultados obtidos.

A arquitetura FSMD engloba as arquiteturas descritas nas duas seções anteriores, sendo bastante utilizada no projeto de ASICs, além de constituir-se na arquitetura básica para processadores de propósito geral, os quais possuem uma unidade de controle e um *datapath* (para manipulação de dados), juntamente com uma memória para armazenamento de dados e programas.

3.3.2. ARQUITETURAS CISC/RISC

Para compreender o **contexto histórico e tecnológico** no qual arquiteturas **CISC** e **RISC surgiram**, é necessário compreender o estado-da-arte das tecnologias empregadas no projeto de **circuitos VLSI**, no projeto de **componentes de armazenamento** (memória) e no desenvolvimento de **compiladores**, entre o final dos anos 70 e início dos anos 80. Estas três tecnologias definiram o ambiente no qual pesquisadores trabalharam na construção de máquinas mais rápidas [14]:

- a) **Tecnologia de Armazenamento:** Na década de 70, computadores utilizavam memória de núcleo magnético para armazenar o código do programa, a qual, além de custosa, era muito lenta. Logo após o surgimento de memórias RAM, ocorreu uma melhora em relação à velocidade, mas o custo continuava alto. Meios de armazenamento secundário também eram caros e lentos, o que deteriorava em muito o desempenho dos computadores. Neste contexto, a preocupação em relação ao tamanho do código era justificável (código bom era código compacto). A busca na redução do tamanho do código estava diretamente relacionada à diminuição do custo total do sistema;
- b) **Tecnologia de Compiladores:** O trabalho de um compilador é o de traduzir sentenças (*statements*) de programação escritas em uma linguagem de alto nível (HLL, de *High Level Language*) em código na linguagem *assembly*. Um aplicativo chamado *assembler* é responsável pela conversão do código *assembly* gerado pelo compilador em código de máquina. Nos anos 70, o processo de compilação era lento e a saída gerada não era otimizada;
- c) **Tecnologia VLSI:** Na década de 70, a tecnologia empregada no projeto de circuitos VLSI (*Very Large Scale Integration*) alcançava densidades muito baixas em relação às de hoje. Em 1981, quando Petterson e Sequin propuseram o projeto *RISC I*, que mais tarde seria a base para a arquitetura *SPARC* da *Sun Microsystems*, era obtida a densidade de um milhão de transistores em um único *chip* [15]. Devido à baixa densidade de transistores, máquinas CISC da época (como *VAX*) tinham suas unidades funcionais "espalhadas" em vários *chips*, o que deteriorava o desempenho por causa de transferências de dados entre *chips*.

Vejamos, na sequência, detalhes de ambas arquiteturas.

3.3.2.1. CISC

A arquitetura **CISC** (*Complex Instruction Set Computer – computador com conjunto de instruções complexo*) surgiu no contexto anteriormente descrito, visando dois objetivos:

- 1º) **Redução no número de instruções do código compilado**, com a conseqüente redução no número de acessos à memória (menos leitura de instruções). Isto fazia-se importante num cenário onde a memória era muito mais lenta em comparação com o processador;
- 2º) **Simplificação no desenvolvimento de compiladores**, com o uso de instruções mais parecidas com as sentenças das linguagens de alto nível, pretendendo-se diminuir a complexidade na construção do compilador.

Para suportar o uso de instruções mais complexas, a arquitetura CISC exige um ***datapath* mais complexo**, assim como um **controlador microprogramado**. Nesse tipo de controlador, uma **linha** da memória microprogramada corresponde a um **conjunto de sinais de controle (palavra de controle)** para o *datapath*. Assim, cada instrução é executada de acordo com uma dada seqüência de palavras de controle.

A figura 3.3 ilustra uma arquitetura CISC com unidade de controle microprogramada. Quando uma instrução é lida da **memória** (*fetch*), ela é armazenada no **registrador de instrução**, o qual é utilizado pelo **circuito de seleção de endereço** para determinar a posição da primeira palavra de controle referente à instrução na **memória microprogramada**. Em seguida, o endereço dessa primeira palavra é armazenado no registrador **MPC** (*Microprogram Counter – contador de programa na memória microprogramada*). A palavra de controle é lida da **memória microprogramada** e, então, utilizada no ***datapath***, em operações como, por exemplo, transferência de dados entre registradores e seleção de operações na **unidade lógico-aritmética (ULA)**. Concorrentemente, o registrador **MPC** é incrementado para selecionar a próxima palavra de controle da instrução. Enquanto a última palavra da instrução está sendo executada no ***datapath***, uma nova instrução será lida da **memória**, repetindo-se o processo descrito neste parágrafo. Todo este processo é seqüenciado através do uso de um sinal de *clock*.

Instruções CISC distintas podem apresentar um **número diferente de palavras de controle**. Como cada palavra de controle é executada no *datapath* em um ciclo de *clock*, **instruções** consomem um **número de ciclos de clock variável**, o que **dificulta** a implementação de um ***pipeline* de instrução** (execução paralela de várias instruções, cada uma numa etapa), tornando a arquitetura inadequada para aplicações de alto desempenho. Outro problema de implementação refere-se ao uso da **memória microprogramada**, comparativamente **lenta** em relação aos demais componentes da arquitetura, impondo **ciclos de clock longos**.

Apesar de computadores CISC possuírem um **grande número de instruções e modos de endereçamento**, levantamentos estatísticos sobre aplicações comuns mostraram que, na prática, apenas um pequeno grupo de instruções e modos de endereçamento simples eram utilizados com mais freqüência, em detrimento de instruções e modos de endereçamento mais complexos, raramente utilizados. Isto se devia à dificuldade de mapear construções da linguagem de alto nível em instruções complexas, por causa de diferenças (ainda que pequenas) entre tais construções e o conjunto de instruções disponível. Desta forma, os projetistas de compiladores utilizavam um conjunto maior de instruções simples para representar uma dada construção na linguagem de alto nível. Esta estratégia de

implementação de compiladores crescia conforme o preço de memórias caía e sua velocidade aumentava. Além disso, instruções complexas tornavam a otimização uma tarefa custosa, dificultando a implementação de compiladores. Por causa destes problemas, surgiu um contraponto à arquitetura CISC, a RISC, descrita na próxima seção.

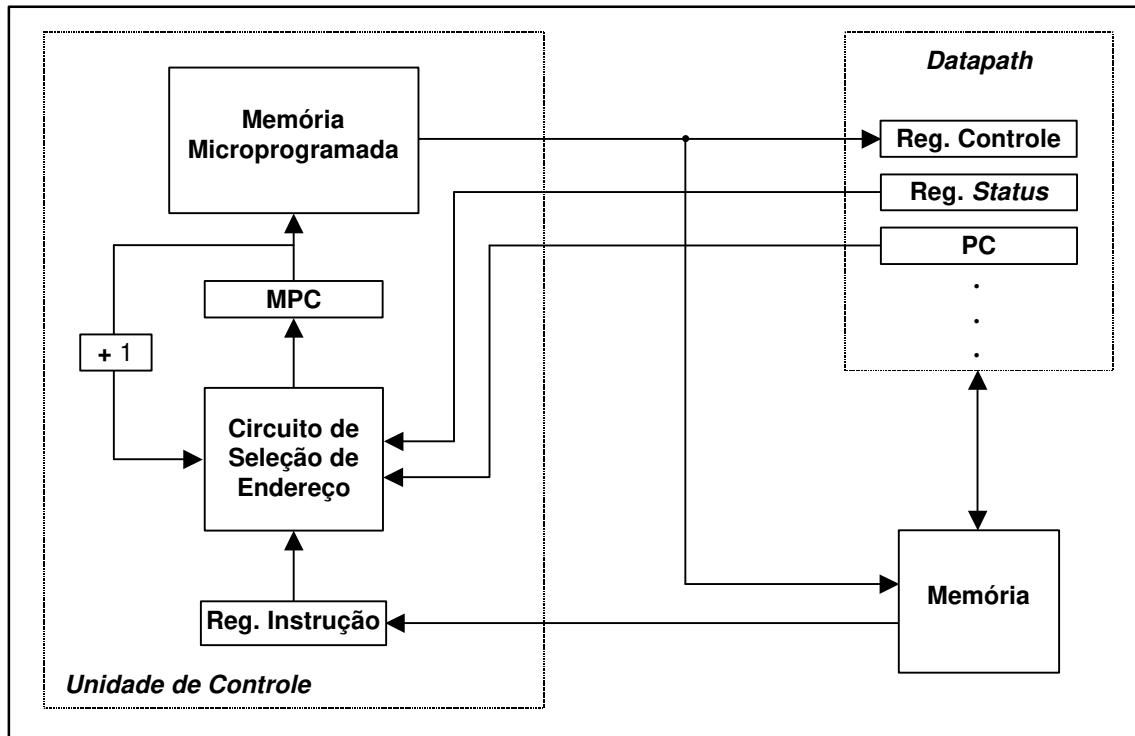


Figura 3.3. Arquitetura CISC com unidade de controle microprogramada.

3.3.2.2. RISC

O projeto da arquitetura **RISC** (*Reduced Instruction Set Computer* – computador com conjunto de instruções reduzido) surgiu em decorrência dos problemas enfrentados pela arquitetura CISC, visando três objetivos:

- 1º) **Redução do tamanho dos ciclos de clock;**
- 2º) **Diminuição no número de ciclos gastos por instrução;**
- 3º) **Implementação eficiente de *pipeline* de instrução.**

A arquitetura de um processador RISC (figura 3.4) apresenta um *datapath* com um grande número de **registradores** conectados a uma **ULA**. Nestes registradores, são armazenados os **operandos** e os **resultados** da execução das instruções. Operandos são trazidos da **memória** por meio de instruções **LOAD**, e resultados são transferidos dos registradores para a memória através de instruções **STORE**. A implementação de uma arquitetura RISC visa a construção de ***pipelines* de instrução curtos** (poucos estágios) e **eficientes**, buscando **reduzir** a **degradação** de desempenho devido a **instruções de desvio** (*branches*). Um exemplo de *pipeline* de instrução de três estágios seria:

- a) **Estágio 1:** leitura (*fetch*) de instrução (no **registrador de instrução**);
- b) **Estágio 2:** decodificação e leitura (*fetch*) de operandos no **banco de registradores**;
- c) **Estágio 3:** operação na **ULA** ou transferência de dados para ou do **cache de dados**.

Supondo que cada um destes estágios consuma 1 ciclo de *clock*, a execução de uma instrução consumiria 3 ciclos de *clock*.

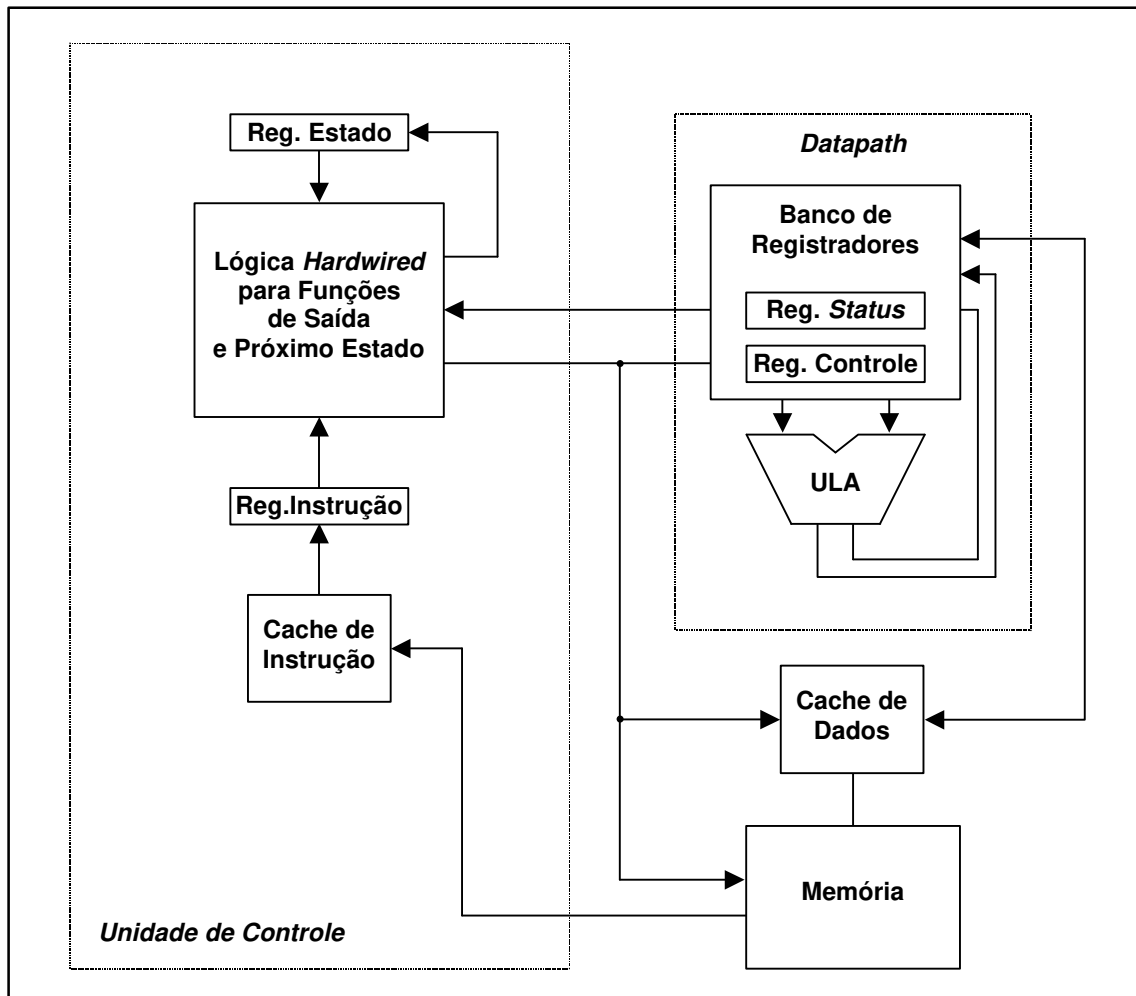


Figura 3.4. Arquitetura RISC com unidade de controle *hardwired*.

O *datapath* de uma arquitetura RISC pode ser **mais simplificado** do que o de uma arquitetura CISC, caso as operações envolvam somente operandos presentes nos registradores (utilização de **modos de endereçamento simples**). Em relação à unidade de controle, se cada operação for realizada em 1 ciclo de *clock* e cada instrução for limitada a um número pequeno de ciclos (3, no exemplo dado), a estrutura da unidade de controle seria simples, podendo ser implementada com um circuito combinacional. Tais simplificações resultariam em **ciclos de *clock* mais curtos** e, conseqüentemente, em uma **melhora de desempenho**.

Nas arquiteturas RISC, o **papel do compilador** é muito mais **importante** do que nas arquiteturas CISC. O sucesso do RISC depende de compiladores que busquem **otimizar** o código de forma eficiente, o que os torna mais complexos. Por exemplo, para minimizar a ociosidade no *pipeline* de instrução, devido à ocorrência de dependências, o compilador deve gerar um **código livre de dependências**, ou através do atraso no envio de instruções ou através da reordenação da seqüência de instruções. Além disso, como o número de instruções é reduzido, sentenças de programação mais complexas exigem que o compilador gere uma grande seqüência de instruções RISC.

Compiladores RISC tentam manter operandos que serão reutilizados em breve nos **registradores**, de maneira que **instruções registrador-registrador** possam ser usadas, ao invés de instruções de acesso à memória (LOAD/STORE). Por outro lado, **compiladores CISC tentam descobrir o modo de endereçamento ideal e o formato de instrução mais curto**, trabalhando com **operandos na memória**.

A tabela 3.1 traz um quadro comparativo [14] entre arquiteturas CISC e RISC.

Tabela 3.1. Quadro comparativo CISC × RISC.

Aspecto	CISC	RISC
Estratégia para redução de custo	Deslocar complexidade de <i>software</i> para <i>hardware</i>	Deslocar complexidade de <i>hardware</i> para <i>software</i>
Estratégia para aumento de desempenho	Favorecer a diminuição no tamanho do código, ao custo de um CPI (<i>Cycles per Instruction – ciclos por instrução</i>) mais elevado	Favorecer a diminuição do CPI, ao custo de um aumento no tamanho do código
Decisões de projeto	- Grande e variado conjunto de instruções, incluindo instruções simples e rápidas (para execução de tarefas básicas) e instruções complexas, realizadas em vários ciclos de clock (correspondendo a sentenças de programação em uma HLL) - Suporte para HLLs feito em <i>hardware</i> - Modos de endereçamento memória-memória - Unidade de controle microprogramada - Gasto de um número menor de transistores nos registradores	- Instruções simples de um único ciclo (execução somente de funções básicas). Instruções assembly correspondem a instruções microcódigo de uma máquina CISC - Suporte para HLLs feito em <i>software</i> - Modos de endereçamento simples. Excetuando-se LOAD e STORE (que acessam a memória), as demais operações são registrador-registrador - Unidade de controle <i>hardwired</i> - Gasto de um número maior de transistores em bancos de registradores - Uso de pipeline de instrução para diminuição do CPI

3.4. ARQUITETURAS PARA PROCESSAMENTO PARALELO

Até o momento, foram tratadas arquiteturas voltadas para máquinas com uma única unidade de processamento. Nesta seção, são abordadas arquiteturas com mais de um **elemento de processamento**, de acordo com a taxonomia proposta por Flynn [16], a qual baseia-se na forma de manipulação de fluxos de instruções e de dados. Chama-se **fluxo de instruções** a uma seqüência de instruções executadas por um único elemento (unidade) de processamento, enquanto **fluxo de dados** é uma seqüência de dados referenciados pelo fluxo de instrução. Flynn considera que um computador é composto por **módulos de memória** (que armazenam tanto dados quanto instruções), por uma ou mais **unidades de controle** (que decodificam as instruções) e por **unidades de processamento** (que executam um fluxo de instruções enviado pela(s) unidade(s) de controle). Os dados fluem bidirecionalmente entre os processadores e os módulos de memória. A taxonomia de Flynn compreende quatro classes arquiteturais básicas:

- i) **Máquinas SISD;**
- ii) **Máquinas SIMD;**
- iii) **Máquinas MISD; e**
- iv) **Máquinas MIMD.**

Como esta seção trata de processamento paralelo, as subseções seguintes abordam os três últimos modelos descritos acima. Apenas para não deixar a taxonomia de Flynn incompleta, será dada uma visão geral de máquinas SISD (uniprocessadoras).

Máquinas SISD (*Single Instruction Stream, Single Data Stream*) são sistemas convencionais com uma **única CPU** (*Central Processing Unit*, unidade de processamento central), comportando, desta forma, um **único fluxo de instrução**, o qual é **executado serialmente**. Atualmente, a maioria dos uniprocessadores SISD é dotada de *pipeline*. Uma máquina SISD pode ter mais de uma unidade funcional, mas todas devem estar sob a supervisão de uma única unidade de controle. Muitos *mainframes* possuem mais de uma CPU executando fluxos de instruções não co-relacionados. Neste caso, tais sistemas devem também ser considerados como um conjunto de máquinas SISD atuando sobre diferentes conjuntos de dados. Em sua grande maioria, *workstations* (*DEC, Hewlett-Packard, Sun Microsystems*) são máquinas SISD. A figura 3.5 mostra a arquitetura básica de uma máquina SISD.

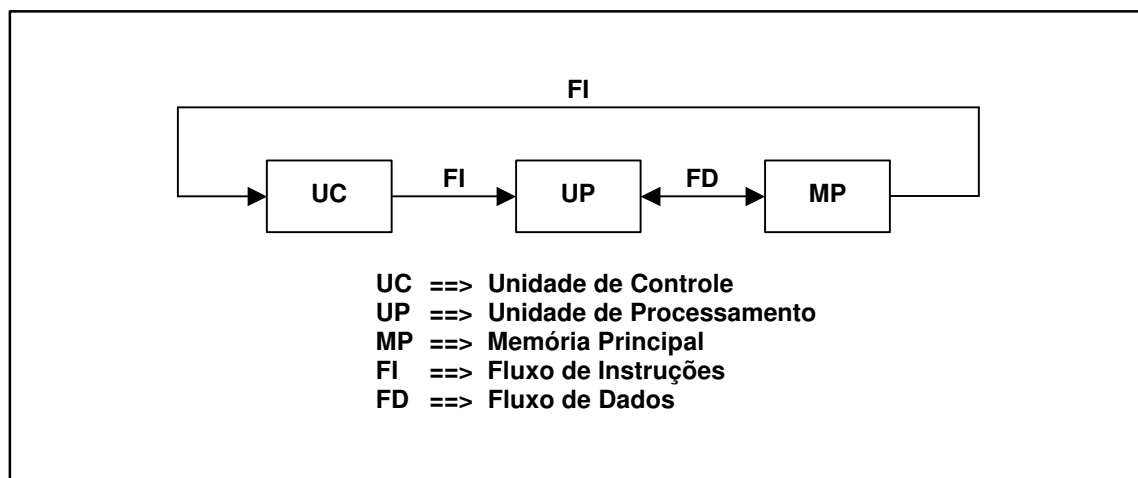


Figura 3.5. Arquitetura básica de uma máquina SISD.

Antes de entrar na descrição das máquinas em si, faz-se necessário introduzir conceitos referentes ao **sistema de memória**, para que se possa entender as subclasses de máquinas SIMD e MIMD.

Em máquinas com **sistema de memória compartilhada**, várias CPUs compartilham o mesmo espaço de endereçamento. **Processadores vetoriais com uma única CPU** são considerados máquinas **SM-SIMD** (*Shared-Memory SIMD*), enquanto **processadores vetoriais com várias CPUs** são exemplos de máquinas **SM-MIMD** (*Shared-Memory MIMD*). Por outro lado, em máquinas com **sistema de memória distribuída**, cada CPU possui sua **própria memória**, podendo **trocar dados** com as memórias associadas às outras CPUs através de uma **rede de interconexão**. Ao contrário de máquinas com memória compartilhada, o usuário deve estar informado sobre a **localização de dados** dentro das **memórias locais**, tendo que deslocar ou distribuir estes dados de forma explícita, quando necessário. Os termos **DM-SIMD** e **DM-MIMD** são utilizados neste texto para designar, respectivamente, máquinas SIMD e MIMD com sistema de memória distribuída (*Distributed Memory*). Para máquinas DM-MIMD uma subdivisão é possível: aquelas nas quais os processadores são conectados em uma topologia fixa e aquelas nas quais a topologia é flexível, podendo variar de tarefa para tarefa.

3.4.1. MÁQUINAS SIMD

Máquinas SIMD (*Single Instruction Stream, Multiple Data Stream*) possuem um **grande número de unidades de processamento**, denominadas **PE** (*Processing Element*), que executam uma **mesma instrução** sobre **diferentes fluxos de dados**. Nesta categoria, estão inclusos os chamados **processadores matriciais**. A figura 3.6 mostra a arquitetura básica de uma máquina SIMD.

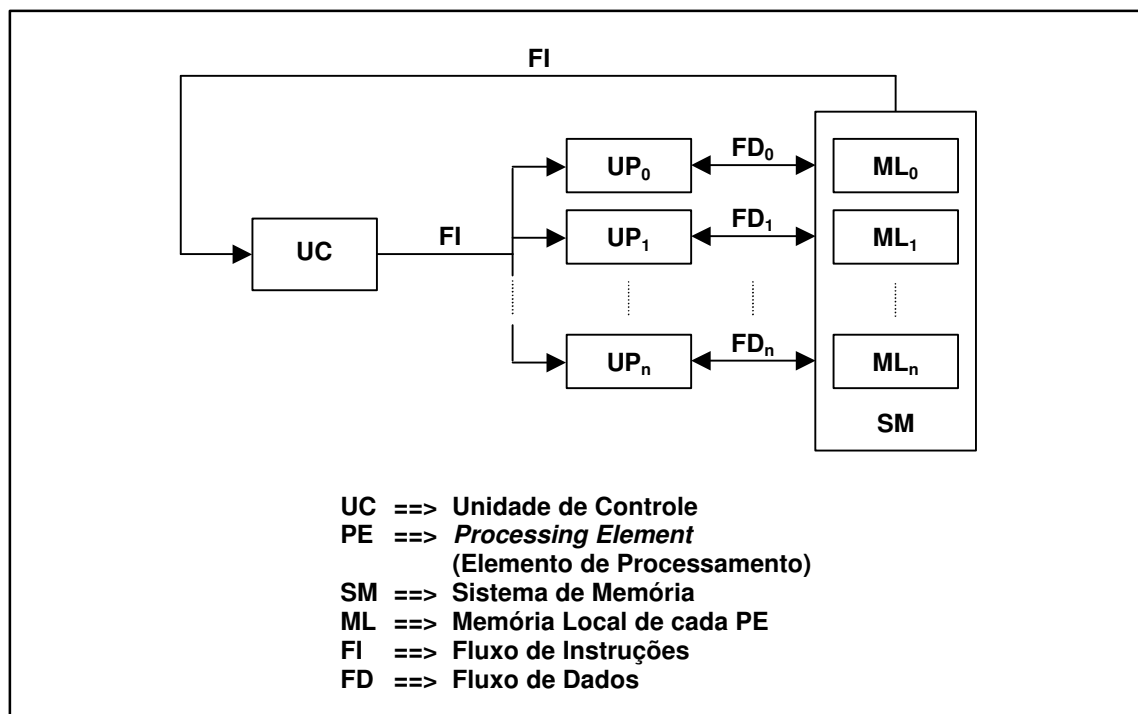


Figura 3.6. Arquitetura básica de uma máquina SIMD.

De acordo com o **sistema de memória**, máquinas SIMD podem ser subdivididas em duas categorias, quais sejam, **SM-SIMD** (*Shared-Memory SIMD*) e **DM-SIMD** (*Distributed Memory SIMD*).

3.4.1.1. MÁQUINAS SM-SIMD

O principal representante desta classe é a **máquina vetorial uniprocessadora** (figura 3.7), embora existam outros modelos conhecidos, como, por exemplo, as máquinas **VLIW** (*Very Long Instruction Word Computer*) [17]. **Máquinas vetoriais multiprocessadoras** são enquadradas na categoria de máquinas MIMD de memória compartilhada (SM-MIMD), a qual encontra-se descrita na seção 3.4.3.1. Nas considerações da presente seção, é utilizado o termo **máquina vetorial**, sem distinção entre uniprocessadora e multiprocessadora. A diferença básica entre ambas é que nas máquinas vetoriais multiprocessadoras existe mais de um **processador vetorial** (aqui chamado **VPU**, de *Vector Processing Unit*).

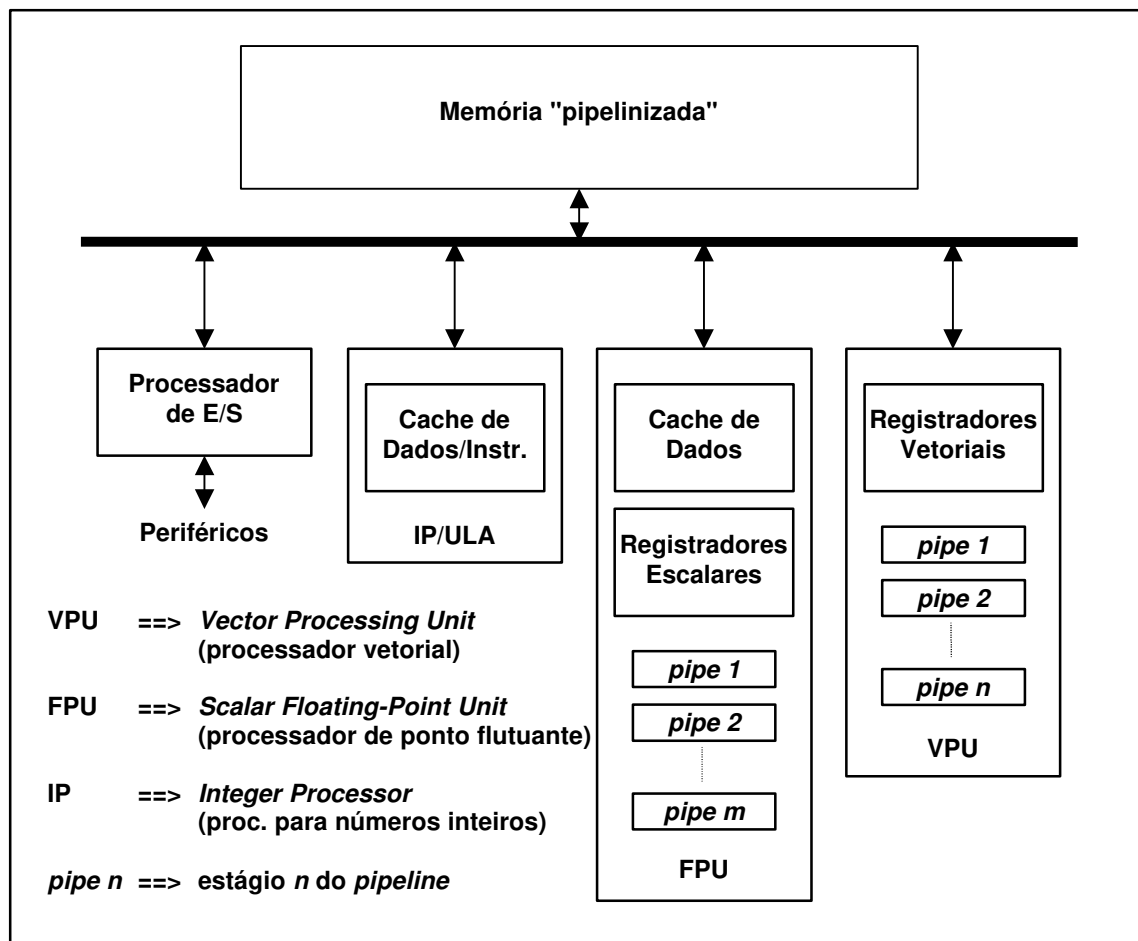


Figura 3.7. Arquitetura de uma máquina vetorial com 1 processador vetorial.

Máquinas vetoriais foram idealizadas no início da década de 70, com o objetivo de obter alto desempenho através da **redução do ciclo de clock a valores mínimos**. Naquela época, o ciclo de *clock* representava cerca de 10% do tempo gasto num acesso à memória ou numa operação aritmética, fazendo com que os projetistas implementassem unidades aritméticas e de memória com vários estágios de *pipeline*. Nas máquinas vetoriais, a **memória**

é decomposta em **bancos** com igual número de endereços, com acesso paralelo aos dados presentes no mesmo endereço de diferentes bancos (**vetor de dados**). Cada **vetor de dados** é **transferido serialmente** para/dos **registradores vetoriais** através do *pipeline* de memória. **Operandos e resultados de instruções vetoriais são vetores**, embora existam **registradores** e **processadores escalares** para operações escalares sobre números inteiros ou de ponto flutuante.

Máquinas vetoriais uniprocessadoras têm somente um **VPU**, podendo o sistema ter sua capacidade escalar de ponto flutuante compartilhada com este processador (como ocorre nos sistemas *Cray*). Geralmente, VPUs não possuem cache, pois a velocidade de execução pode diminuir com a ocorrência de estouro (*overflow*) do mesmo [18].

Embora tenham existido VPUs que carregavam seus operandos diretamente da memória e armazenavam os resultados imediatamente de volta à mesma, todos os VPUs de hoje usam **registradores vetoriais**. Estes não prejudicam a velocidade das operações, fornecendo maior flexibilidade na junção de operandos e na manipulação de resultados intermediários.

Na figura 3.7, detalhes de interconexão entre o VPU e a memória são omitidos. No entanto, esta interconexão é muito importante para a velocidade efetiva de uma operação vetorial: quando a largura da banda (*bandwidth*) entre a memória e o VPU é muito pequena, o VPU terá que esperar enquanto operações LOAD/STORE estiverem sendo executadas. Quando a razão (*operações aritméticas*) / (*operações LOAD/STORE*) não for alta o suficiente para compensar tais situações, fortes quedas de desempenho poderão ocorrer.

Máquinas vetoriais podem ser muito **rápidas**, desde que exista um **número suficiente de operações vetoriais no código**. Como as aplicações nem sempre são escritas objetivando vetorização, é necessário que os **compiladores** se encarreguem da **vetorização** de um código comum, o que muitas vezes se limita à decomposição de *loops* em operações vetoriais. Além disso, o desempenho de máquinas vetoriais tende a **degradar** durante **operações escalares** e quando do **acesso a posições não sequenciais em matrizes**.

3.4.1.2. MÁQUINAS DM-SIMD

Máquinas deste tipo são também conhecidas como **processadores matriciais** (*processor-array systems*). Estes processadores **executam a mesma instrução** ao mesmo tempo, sobre **diferentes itens de dados**. A não exigência de sincronização entre os processadores simplifica em muito o projeto de tais sistemas. Um **processador de controle** é responsável pelo envio das instruções que serão executadas pelos demais processadores do arranjo. As máquinas DM-SIMD atuais utilizam um **processador front-end** responsável pela interface com o usuário, o qual conecta-se ao processador de controle através de um *datapath*. A figura 3.8 [18] mostra o modelo genérico de uma máquina DM-SIMD. Esta figura poderia sugerir que todos os processadores estariam conectados em uma rede bidimensional. De fato, a topologia de interconexão destas máquinas sempre inclui redes bidimensionais. Entretanto, várias máquinas apresentam estruturas de interconexão mais complexas, com redes tridimensionais ou ligações diagonais.

Durante a execução de uma instrução, é possível que processadores do arranjo fiquem **ociosos**, o que acarretará em **queda do desempenho**. Outro fator prejudicial ao desempenho ocorre quando um processador *i* necessita de dados armazenados na memória de um

processador j . Este último terá que obter os dados (*fetch*) e enviá-los através da rede de **roteamento** ao processador i , o que poderá consumir bastante tempo. Por estes problemas, máquinas DM-SIMD devem ser empregadas em computações maciçamente paralelas como, por exemplo, aplicações para o processamento de imagens digitais.

Os **processadores** do arranjo **muitas vezes** são de tipos **simples**, operando *bit a bit* serialmente (*bit-serial type*), independentemente dos tipos dos itens de dados. Desta forma, operações sobre inteiros são produzidas por **rotinas** de *software* sobre estes processadores, o que consome um **número variável de ciclos** em conformidade com o **tamanho dos operandos**. Assim, um resultado inteiro de 32 *bits* será produzido duas vezes mais rápido do que um resultado inteiro de 64 *bits*. Situação similar ocorre com operações de ponto flutuante. Em alguns casos, coprocessadores de ponto flutuante são adicionados ao arranjo. A quantidade destes é, no entanto, de 8 a 16 vezes menor do que a de processadores do tipo *bit-serial*, devido ao aumento no custo. Uma vantagem apresentada por processadores *bit-serial* é que eles podem operar sobre operandos de qualquer tamanho. Esta característica é de grande valia durante a geração de números randômicos e no processamento de sinais, atividades nas quais operandos de 1 ou 8 *bits* são abundantes. Como o tempo de execução em processadores *bit-serial* é proporcional ao tamanho dos operandos, isto resulta em ganhos significativos de velocidade.

Processadores matriciais são fáceis de construir e de programar [3], desde que a estrutura natural do problema (aplicação) case-se com a topologia do arranjo. Assim, eles não podem ser considerados máquinas de propósito genérico, uma vez que nem todas as aplicações podem ser escritas facilmente para estas máquinas, a fim de explorar sua potencialidade.

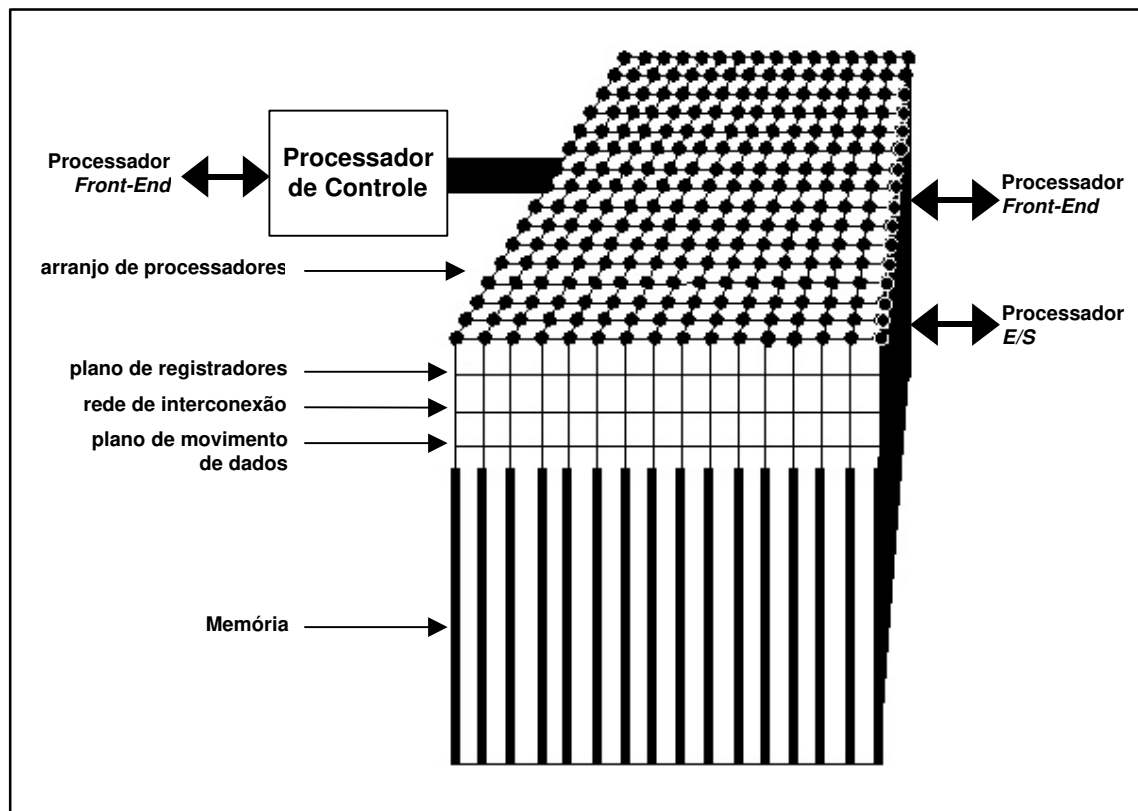


Figura 3.8. Arquitetura genérica de uma máquina SIMD com memória distribuída.

3.4.2. MÁQUINAS MISD

Em **máquinas MISD** (*Multiple Instruction Stream, Single Data Stream*) **vários fluxos de instruções** atuam sobre um **único fluxo de dados**. Teoricamente, existem vários processadores distintos executando conjuntos de instruções distintos sobre um único conjunto de dados. Até hoje não se conhece qualquer máquina que tenha sido construída com esta arquitetura, a qual é apresentada na figura 3.9.

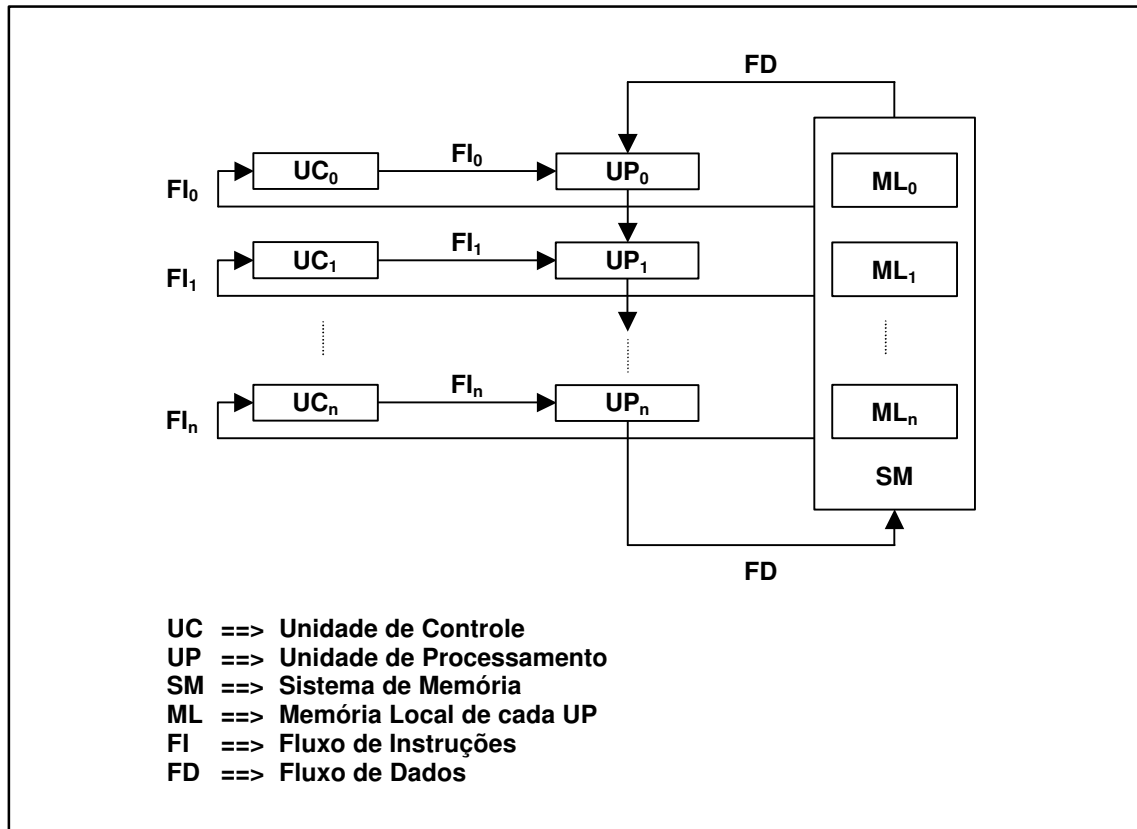


Figura 3.9. Arquitetura básica de uma máquina MISD.

3.4.3. MÁQUINAS MIMD

Máquinas MIMD (*Multiple Instruction Stream, Multiple Data Stream*) executam **vários fluxos de instruções** sobre **diferentes fluxos de dados**, ao mesmo tempo. Nesta categoria se inclui a **maioria** dos sistemas **multiprocessadores**. A diferença em relação a multiprocessadores SISD está no fato de que os processadores de um computador MIMD tratam diferentes partes de uma mesma tarefa, interagindo entre si. Um computador MIMD é dito **fortemente acoplado** se o grau de interação entre os processadores é alto; caso contrário, é considerado **fracamente acoplado**. A maioria dos computadores MIMD existentes é fracamente acoplada. A figura 3.10 mostra a arquitetura básica de uma máquina MIMD.

Tal como ocorre com máquinas SIMD, máquinas MIMD podem ser divididas em duas categorias, uma vez que os fluxos de dados são derivados ou de uma memória compartilhada pelos processadores ou de um sistema de memória distribuída. Estas categorias são descritas na sequência.

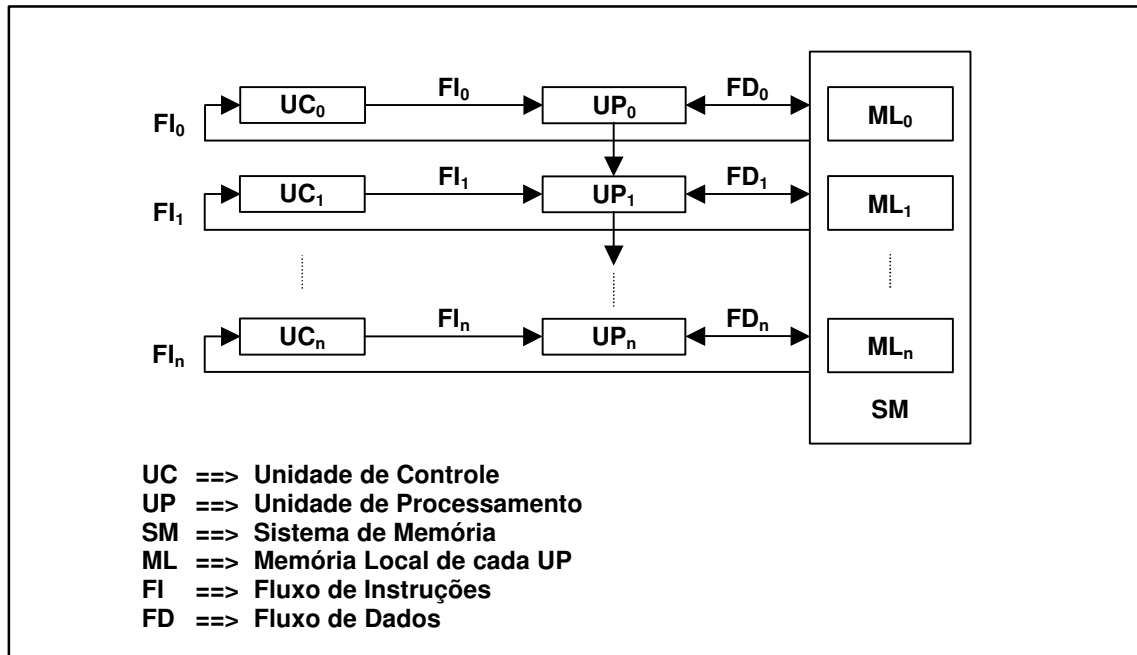


Figura 3.10. Arquitetura básica de uma máquina MIMD.

3.4.3.1. MÁQUINAS SM-MIMD

A máquina vetorial uniprocessadora mostrada na figura 3.7 pode ser considerada uma subclasse de máquinas SM-MIMD, constituindo-se num caso especial de um tipo mais genérico. Naquela figura, é possível a inserção de mais de um FPU, IP ou VPU, transformando aquela arquitetura em uma **máquina vetorial multiprocessadora**.

O principal problema em máquinas de memória compartilhada está na **interconexão** de CPUs e destas com a **memória**. Conforme mais CPUs forem **adicionadas**, o **ideal** seria a **largura da banda de memória (bandwidth) aumentar linearmente** com o número de **processadores**. Outra situação ideal seria a de processadores se comunicarem **diretamente** um com o outro, sem usar a memória como intermediária. Entretanto, esta situação **encareceria** em muito o **custo da interconexão**, a qual cresceria na ordem de $O(n^2)$ para um crescimento no número de processadores de ordem $O(n)$.

A figura 3.11 apresenta algumas alternativas de **estrutura de interconexão**. A estrutura **crossbar** utiliza n^2 conexões, a **rede Ω** utiliza $n \log_2 n$ conexões e o **barramento** utiliza somente uma conexão. Na estrutura **crossbar**, cada canal de comunicação é direto e não é compartilhado com outros elementos. Na rede Ω há $\log n$ estágios de chaveamento, limitando neste valor a quantidade de itens de dados que podem competir por alguma conexão. No barramento, todos os dados devem compartilhar o mesmo meio, de maneira que n itens de dados possam competir por vez [18].

A **conexão por barramento** é a **opção mais barata**, embora seja a mais propensa a causar **queda de desempenho** devido à ocorrência de **contenções**. Há projetos que associam caches à CPU para minimizar o tráfego no barramento. Contudo, isto leva a uma estrutura mais complicada, elevando o custo. Na prática, é difícil projetar barramentos que sejam

rápidos o suficiente para atender a crescente velocidade dos processadores, o que limita o número de processadores ligados desta forma a uma quantidade entre 10 e 20 unidades.

Em relação a custo e capacidade, a **rede Ω** constitui uma opção intermediária entre o barramento centralizado e a estrutura *crossbar*. Em máquinas com um grande número de processadores, as $n \log_2 n$ conexões tornam-se mais atrativas do que as n^2 conexões de uma estrutura *crossbar*, desde que as chaves nos níveis intermediários sejam suficientemente rápidas para atender a largura de banda exigida. Na rede Ω a largura dos canais de comunicação entre os processadores também é importante: uma rede que utilize 16 *links* paralelos terá uma largura de banda 16 vezes maior do que uma rede de igual topologia implementada com *links* seriais.

Em todas as máquinas vetoriais multiprocessadoras, estruturas *crossbar* são utilizadas [18]. Isto é possível porque o número de processadores nestas máquinas ainda é pequeno (32 na maioria deles). Problemas tecnológicos surgirão à medida que este número aumentar. Ficará mais difícil construir um *crossbar* de velocidade suficiente para atender a um grande número de processadores, os quais, por sua vez, terão sua velocidade individual incrementada, aumentando o problema de tornar a velocidade do *crossbar* compatível com a largura de banda exigida pelos processadores.

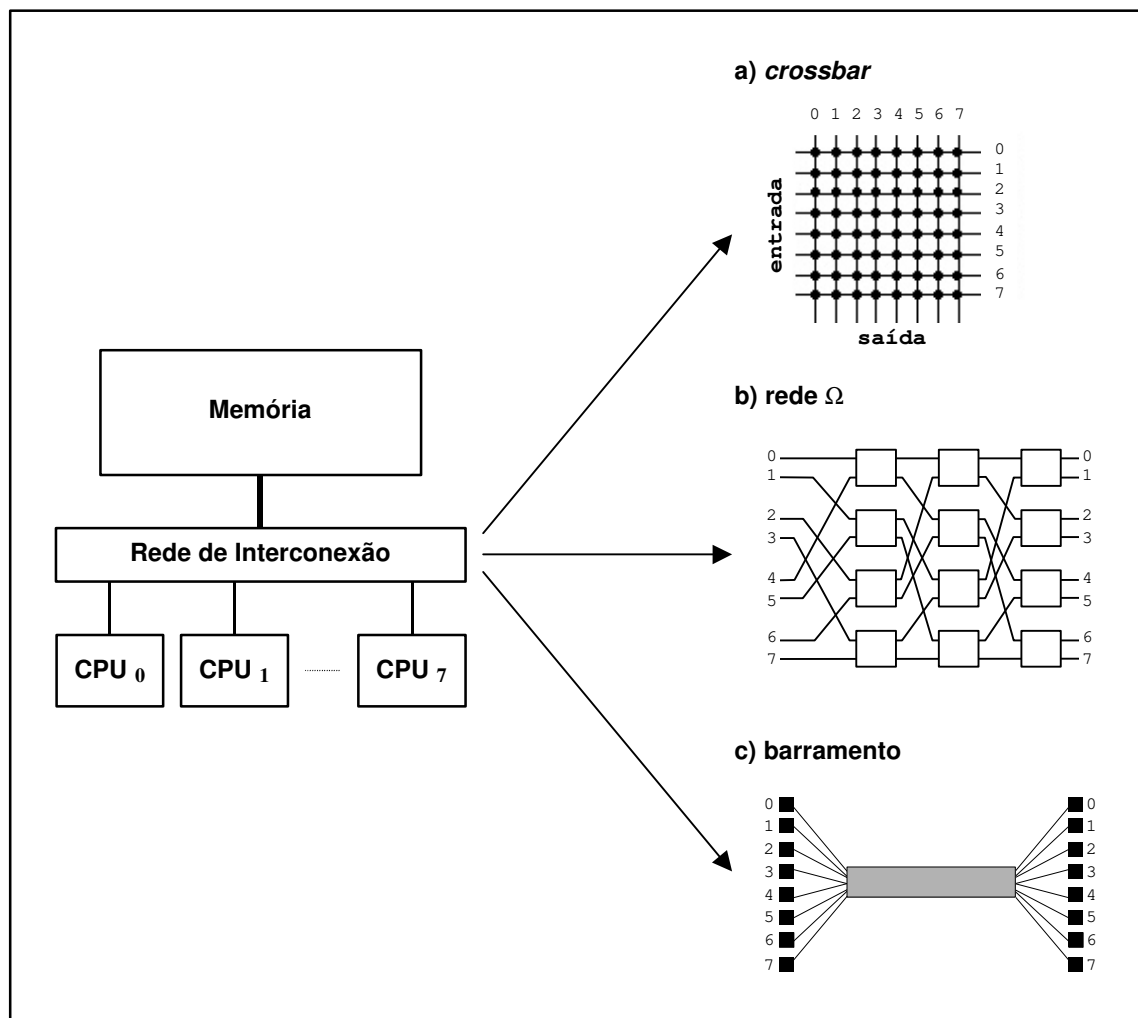


Figura 3.11. Exemplos de estruturas de interconexão para máquinas SM-MIMD.

Os sistemas discutidos aqui são máquinas MIMD e, como tais, permitem a execução de diferentes tarefas rodando em diferentes processadores simultaneamente. Muitas vezes faz-se necessária a **sincronização** entre tarefas e, novamente, a estrutura de interconexão é de suma importância. A maioria dos processadores vetoriais (VPUs) utiliza registradores de comunicação especiais dentro da CPU, os quais possibilitam comunicação direta com outras CPUs durante a sincronização. Alguns sistemas fazem sincronização através da memória compartilhada. Por ser muito mais lento, este mecanismo é aceitável em casos onde a necessidade por sincronização é pouco freqüente. Em sistemas baseados em barramento, a comunicação também é feita via barramento. Na maioria destes sistemas, o barramento de comunicação (visando propósitos de sincronização) é separado do barramento de dados.

3.4.3.2. MÁQUINAS DM-MIMD

Grande parte dos **computadores de alto desempenho** é de sistemas MIMD com **memória distribuída**. Estes sistemas são mais difíceis de tratar do que máquinas com memória compartilhada e máquinas DM-SIMD. Nos processadores matriciais (DM-SIMD), as estruturas de dados candidatas à paralelização (vetores e matrizes) são submetidas automaticamente ao arranjo de processadores, através de um sistema em *software*. Nos sistemas de memória compartilhada, a distribuição dos dados é transparente para o usuário. Já em sistemas DM-MIMD, a distribuição e a troca de dados entre os processadores devem ser explicitadas pelo usuário, o que não é uma tarefa trivial. Entretanto, tal situação tende a mudar com o surgimento de um padrão de *software* de comunicação nestes sistemas [19].

Sistemas DM-MIMD possuem **vantagens** claras sobre máquinas de memória compartilhada. O problema de **largura de banda** é **evitado**, uma vez que esta **aumenta** automaticamente com o aumento no **número** de **processadores**. Além disso, a **velocidade da memória** tem **menor impacto**, porque mais processadores podem ser configurados sem a preocupação com os problemas relativos à largura de banda. Para obterem desempenho máximo comparável com máquinas DM-MIMD, os processadores de máquinas com memória compartilhada deveriam ser muito rápidos, o que obrigaria o aumento de velocidade da memória, para que esta não estrangulasse o sistema.

As **desvantagens** de sistemas DM-MIMD residem em dois fatores. O primeiro é que a **comunicação entre processadores** é muito **mais lenta** do que em sistemas SM-MIMD, prejudicando o desempenho durante a execução de tarefas que exigem uma sobrecarga (*overhead*) de **sincronização**. O segundo fator é a **baixa velocidade** do **acesso remoto** (em comparação a acesso local) a **dados** que **não** estão **presentes** na **memória local** de um dado processador. Desta forma, quando a aplicação em execução exige uma troca freqüente de dados entre processadores ou requer muitas sincronizações, é provável que somente uma parcela do ganho teórico de velocidade esperado seja obtida.

Tal como nos sistemas SM-MIMD, aspectos cruciais no projeto de máquinas DM-MIMD recaem na **topologia** e na **velocidade da estrutura de conexão**. Uma das estruturas mais utilizadas é a **topologia hipercubo** (figura 3.12).

Uma característica interessante da topologia hipercubo é que, para um hipercubo com 2^d nós, o número máximo de *links* a serem percorridos entre dois nós (processadores) quaisquer é d . Desta forma, a dimensão da rede cresce logaritmicamente com o número de nós. Além disso, teoricamente, é possível simular qualquer outra topologia sobre um

hipercubo como, por exemplo, árvores, anéis, *meshes* bi- ou tri-dimensionais. Na prática, a topologia exata para hipercubos não interessa tanto, uma vez que os sistemas atuais empregam o chamado **roteamento wormhole**, no qual é montada dinamicamente uma conexão direta entre dois processadores que necessitam comunicar-se: quando a conexão está pronta, os dados são enviados de um processador para o outro, sem interferir na operação dos nós intermediários. Exceto pelo pequeno tempo gasto na montagem da conexão, o tempo de comunicação tende a ser virtualmente independente da distância entre os nós.

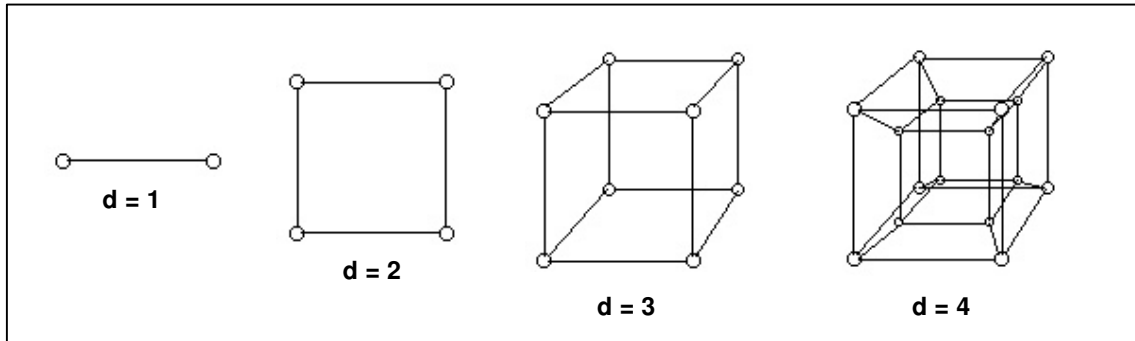


Figura 3.12. Conexões hipercubo de 1, 2, 3 e 4 dimensões.

Muitos dos sistemas DM-MIMD maciçamente paralelos adotam a estrutura *mesh* bi- ou tri-dimensional. A razão para isso parece ser que a maioria de simulações físicas em alta-escala pode ser eficientemente mapeada sobre esta topologia, além do fato de que estruturas mais sofisticadas dificilmente vingam [18]. Entretanto, há sistemas que mantêm uma ou mais redes agregadas ao *mesh* para tratar certos gargalos na distribuição e recuperação de dados [20].

Boa parte dos sistemas DM-MIMD emprega estrutura *crossbar*. *Crossbars* de 1 estágio são empregados em sistemas com um número relativamente pequeno de processadores (em torno de 64). Para sistemas com um grande número de processadores, são utilizados *crossbars* multi-estágio, nos quais as conexões de um *crossbar* no nível 1 estão também conectadas a um *crossbar* no nível 2, e assim por diante, ao invés de existirem ligações diretas entre nós muito distantes. Desta forma, é possível conectar alguns milhares de nós, utilizando-se somente de alguns estágios de chaveamento. Redes Ω também são utilizadas em sistemas DM-MIMD, dentre outros tipos de interconexão.

Tal como ocorre em máquinas SM-MIMD, a princípio um nó pode ser qualquer tipo de processador (escalar ou vetorial), trabalhando com memória local (com ou sem cache) ou acionando (na maioria dos casos) processadores especializados em comunicação, que conectam um nó com a sua vizinhança. Atualmente, a maioria dos nós é de processadores comerciais RISC, algumas vezes acompanhados de processadores vetoriais.

CAPÍTULO 4: LINGUAGENS DE ESPECIFICAÇÃO

4.1. INTRODUÇÃO

No capítulo 2 foram abordados diversos modelos conceituais, os quais podem ser aplicados em diferentes fases de metodologias de projeto para sistemas digitais. Para que possa ser tratado por ferramentas de CAD (*Computer Aided Design*) e de CAE (*Computer Aided Engineering*), um modelo deve ser transformado em uma especificação através de uma **linguagem de especificação**. Há vários tipos de linguagens de especificação, cada qual podendo capturar características de um ou mais modelos. São características desejáveis de uma linguagem de especificação:

- a) Capacidade de capturar o maior número de características de uma classe de sistemas;
- b) Possibilitar o uso de ferramentas de síntese, tanto para *hardware* quanto para *software*;
- c) Ter boa legibilidade; e
- d) Gerar uma especificação executável, a qual possa ser utilizada para propósitos de demonstração da funcionalidade do produto, documentação (servindo de guia para as diferentes equipes envolvidas no processo de projeto), análise de propriedades (verificação automática), exploração do espaço de projeto (alternativas de projeto) e síntese do projeto.

Seguindo a linha adotada para este trabalho de doutorado, nas seções deste capítulo, são tratadas linguagens de especificação empregadas no projeto de **sistemas digitais embutidos**. As linguagens são brevemente apresentadas com base na proporção em que elas capturam, ou não, as principais características encontradas nos diversos modelos conceituais utilizados no projeto de tais sistemas. Estas características, levantadas por Gajski [3], são:

- a) **Concorrência**: Este termo refere-se à possibilidade do modelo descrever a execução simultânea de comportamentos do sistema. Há duas formas de representação de concorrência. Na concorrência **orientada a dados**, não há uma ordem de execução explícita dos comportamentos envolvidos. A ordem de execução é determinada pela dependência de dados entre os comportamentos. Na concorrência **orientada a controle**, há segmentos de controle determinando a ordem de execução dos comportamentos envolvidos. Cada segmento de controle agrupa um conjunto de comportamentos executados sequencialmente. A concorrência caracteriza-se pela execução simultânea destes agrupamentos sequenciais (segmentos de controle). O conceito de concorrência pode ser aplicado em diferentes níveis de abstração (tarefas, sentenças, operações, bits);
- b) **Transição de estados**: Sistemas embutidos geralmente apresentam vários modos (estados) de comportamento. Transições entre tais modos podem ocorrer de forma não estruturada (em oposição a um sequenciamento linear pelos modos), mediante a detecção de certos eventos ou condições;
- c) **Hierarquia comportamental**: Esta característica refere-se à possibilidade de decompor o comportamento (funcionalidade) de um sistema (ou subsistemas componentes) em partes menores (subcomportamentos), de forma a facilitar o gerenciamento de sua complexidade (trabalhando-se em separado sobre cada parte). Tal **decomposição** denomina-se **paralela**, quando os subcomportamentos gerados são paralelos entre si (admitindo, assim, execução simultânea), ou **sequencial**, quando os subcomportamentos gerados são executados um após o outro;

- d) **Estruturas de programação:** São as estruturas (comandos/funções/operadores) utilizadas por linguagens de programação, apropriadas para a descrição de comportamentos através de algoritmos sequenciais;
- e) **Término comportamental:** Esta expressão refere-se à habilidade de um comportamento indicar a sua finalização (todas as computações internas do comportamento executadas), assim como à habilidade de outras partes do sistema detectarem este fato;
- f) **Tratamento de exceções:** Em um modelo de sistema, a ocorrência de um dado evento pode exigir que um modo de computação (comportamento ou estado local) seja imediatamente interrompido ou abortado, transferindo o fluxo de controle de execução do modelo para um outro modo de computação. Um evento desse tipo é chamado de exceção. Basicamente, há dois tipos de exceções: **preempção** (quando o modo atual é abortado) e **interrupção** (quando o modo atual é temporariamente suspenso, podendo ser retomado depois de decorrido um certo tempo, mediante dadas condições);
- g) **Representação de tempo:** Em geral, há duas formas de especificar informações de tempo em um sistema, quais sejam, a **temporização funcional** e a **temporização restritiva**. Na primeira, as informações de tempo podem interferir na funcionalidade do modelo (em sua simulação). Já na segunda, a especificação de restrições temporais impõe limites para a execução de um comportamento ou para a permanência do sistema modelado num dado estado (tais limites são úteis para ferramentas de síntese e de verificação);
- h) **Comunicação:** Geralmente, um sistema é composto por vários comportamentos que interagem entre si. Desta forma, faz-se necessária a adoção de um mecanismo de comunicação entre tais comportamentos. Mecanismos de comunicação enquadram-se em uma de duas categorias, quais sejam, a de modelos de comunicação por **memória compartilhada** (na qual existe um meio de armazenamento intermediário, representante direto de um dispositivo físico, o qual possibilita a escrita e a leitura de dados pelos comportamentos que o acessam), e a de modelos de comunicação por **passagem de mensagens** (na qual os detalhes de transferência de dados são substituídos pela comunicação através de um meio abstrato, denominado **canal**); e
- i) **Sincronização de processos:** Em um comportamento formado por vários processos concorrentes, um processo pode gerar dados ou eventos que necessitem ser reconhecidos por outros processos. Em casos assim, pode ser necessária a sincronização entre os processos que se comunicam, isto é, pode ser necessário que um dos processos fique suspenso até que o outro (ou outros) atinja um dado ponto em sua execução. Há dois métodos de sincronização comuns a sistemas embutidos. Na sincronização **dependente de controle**, a estrutura de controle do comportamento é a responsável pela sincronização entre seus processos, através da especificação de pontos de sincronização. Na sincronização **dependente de dados**, os processos envolvidos é que controlam sua própria execução, através da verificação de determinados valores de dados (ou eventos), de acordo com o mecanismo de comunicação adotado para tais processos (memória compartilhada ou passagem de mensagens).

Como o objetivo principal deste trabalho é a obtenção de um modelo conceitual baseado em redes de Petri, que sirva de suporte à criação de uma metodologia para a modelagem, simulação e análise em alto nível de sistemas digitais (embutidos, principalmente) e, como este modelo conceitual foi desenvolvido para abranger todas as características supracitadas, faz-se necessário compreender a maneira pela qual as principais linguagens de especificação tratam tais características, inclusive para propósitos de comparação, conforme será visto na seção 11.3. Esta comparação, envolvendo linguagens de especificação com o modelo conceitual RPSD (Rede de Petri para Sistemas Digitais – capítulo 11), faz sentido, na

medida em que se pretende implementar, como decorrência deste trabalho de doutorado, uma plataforma em *software* que permita capturar de forma direta descrições RPSD de sistemas digitais embutidos, para as suas simulação, análise e síntese. A metodologia que abrange tal plataforma será abordada no capítulo 15.

4.2. VHDL

A linguagem VHDL (*VHSIC Hardware Description Language*) foi desenvolvida entre o final dos anos 70 e início dos 80, dentro do programa VHSIC (*Very High Speed Integrated Circuit*) do Departamento de Defesa Norte-Americano, sendo padronizada pelo IEEE em 1987 [21]. Inicialmente, VHDL foi projetada como uma linguagem de documentação e simulação. Uma vez que a linguagem foi bem aceita na comunidade de projeto como uma linguagem de descrição, logo surgiu uma grande quantidade de ferramentas para a captura, simulação, depuração (*debugging*), verificação e síntese de projetos de *hardware* descritos em VHDL, no nível RTL (*Register Transfer Level*, nível de transferências entre registradores).

A unidade básica de abstração de *hardware* em VHDL é a **entidade** (*entity*), a qual é utilizada para identificar e representar uma porção unitária de um projeto maior, realizando uma função específica e possuindo entradas e saídas bem definidas. A funcionalidade de cada entidade é descrita através de sua **arquitetura** (*architecture*). VHDL permite que o projetista represente a arquitetura associada a uma entidade de três formas, ou de uma combinação destas. Estas formas são:

- a) **Seqüencial**: Arquitetura utiliza-se de sentenças seqüenciais de programação;
- b) **Comportamental** (ou **a fluxo de dados** – *dataflow*): Arquitetura utiliza-se de sentenças de programação que trabalham somente sobre sinais, os quais representam armazenadores de valores com conteúdo escalonado, conforme descrito adiante;
- c) **Estrutural**: Arquitetura descrita como uma interconexão de componentes, os quais correspondem a instâncias de pares entidade-arquitetura.

Na seqüência, é visto como VHDL trata das características necessárias a um bom modelo conceitual para a modelagem de sistemas digitais embutidos.

a) *Concorrência*

VHDL suporta concorrência em dois níveis, quais sejam: nível de **tarefas** e nível de **sentenças** (*statements*) de programação.

No nível de tarefas a concorrência é possível, uma vez que vários **processos** (descrevendo, cada qual, a arquitetura seqüencial de uma entidade) podem estar ativos num dado momento.

No nível de sentenças, a concorrência é obtida através do uso de sentenças que permitem atribuição **concorrente** de **sinais**, em arquiteturas descritas de forma comportamental (*dataflow*). Diferentemente de **variáveis** (utilizadas somente em processos), que armazenam um único valor e sofrem atribuição imediata (logo após a execução da sentença), **sinais** permitem a atribuição de diversos valores de forma escalonada (ao longo do tempo). Como exemplo, observe o seguinte trecho de sentenças

dentro de um processo, envolvendo os sinais *a* e *b*, além do operador de atribuição de sinais (**<=**):

```
a <= b;  
b <= a;  
wait on b;
```

Nele, as duas atribuições serão feitas ao mesmo tempo, trocando os valores dos sinais *a* e *b*. A cláusula **wait on** mantém o processo suspenso até que ocorra um evento sobre o sinal *b*. Pode-se, também, obter concorrência em nível de sentenças implicitamente, através da cláusula **after**, a qual permite escalonar atribuição futura sobre um sinal, podendo ocorrer atribuições concorrentes, como no trecho seguinte:

```
a <= b + c after 10 ns;  
wait for 10 ns;  
x := 50;
```

Neste trecho, a cláusula **after** adia em 10 ns a atribuição sobre *a*, e a cláusula **wait for** pára o processamento durante 10 ns, após o que será feita a atribuição sobre *x*. Desta forma, as atribuições sobre o sinal *a* e a variável *x* ocorrerão concorrentemente.

b) *Transição de estados*

Esta característica não é suportada em VHDL, pois a linguagem não trabalha com o conceito de estado.

c) *Hierarquia*

VHDL suporta tanto hierarquia estrutural como comportamental. A hierarquia **estrutural** é obtida através do uso de **blocos (blocks)** encadeados e de sentenças para a criação de instâncias de **componentes** (na descrição estrutural de uma arquitetura).

A hierarquia **comportamental** possui dois níveis. No nível topo, a especificação pode ser decomposta em um conjunto de processos que podem ser executados **concorrentemente**. O segundo nível consiste de uma decomposição **seqüencial** destes processos em procedimentos. No entanto, uma hierarquia comportamental verdadeira, na qual concorrência possa ser especificada em qualquer nível, não é suportada em VHDL.

d) *Estruturas de programação*

VHDL possui suporte completo para estruturas de programação, semelhantes às encontradas na linguagem de programação ADA. Uma ampla variedade de tipos de dados é fornecida, a qual mostra-se adequada para modelagem em alto nível, uma vez que se encontram tipos tais como inteiros, reais, enumerados, vetores, registros e ponteiros.

e) *Término comportamental*

Esta característica é suportada por VHDL, uma vez que o encerramento de um processo é bem definido (execução da última sentença).

f) *Tratamento de exceções*

VHDL não possui construtores capazes de encerrar um processo em resposta a uma exceção. Suporte parcial pode ser obtido, através do uso de uma expressão **guarda** (booleana), associada a um bloco, que encapsule um conjunto de atribuições concorrentes sobre sinais.

g) *Representação de tempo*

VHDL é limitada à especificação de temporização **funcional**. A cláusula **after** é utilizada para escalonar a atualização de um sinal no futuro. Já a cláusula **wait** é utilizada para suspender um processo temporariamente. Pode-se trabalhar com a noção de tempo global do sistema, através da expressão **now**.

A utilização de tempo para propósitos de **restrição** não é diretamente suportada em VHDL, embora restrições temporais possam ser especificadas indiretamente através do uso de **atributos**.

h) *Comunicação*

Comunicação entre processos em VHDL pode ser obtida através da adoção de um modelo de **memória compartilhada**, através do qual sinais e variáveis associados a um processo possam ser acessados por outros processos.

i) *Sincronização de processos*

Sincronização pode ser obtida por duas formas. Na primeira, através do uso da **lista de sensibilidade** de um processo. Os sinais presentes nesta são responsáveis pela ativação do processo correspondente, quando da ocorrência de um evento sobre qualquer um destes sinais. No exemplo seguinte, o processo *Interface* permanecerá suspenso até que um evento ocorra sobre um dos sinais (*Enable* ou *Acknowledge*) de sua lista de sensibilidade. Isto permite ao projetista sincronizar o processo *Interface* com outros processos que utilizem os sinais *Enable* ou *Acknowledge*.

```
Interface: process (Enable, Acknowledge)
begin
    ...
end process Interface;
```

Na segunda forma de sincronização, a cláusula **wait** pode ser empregada para suspender um processo até que seja detectada a ocorrência de um evento sobre um dos sinais explicitados, ou mediante a presença de uma dada condição (acompanhada da cláusula **until**<condição>). Por exemplo, o processo anterior poderia ser reescrito da seguinte forma:

```

Interface: process
          begin
              wait on Enable, Acknowledge
              ...
          end process Interface;

```

4.3. VERILOG

Verilog foi desenvolvida como uma linguagem de uso particular, voltada para a especificação e simulação de sistemas digitais [22]. Em 1990, foi liberada para o domínio público e, desde então, passou a ser amplamente utilizada como uma linguagem de descrição de *hardware*.

a) *Concorrência*

Assim como VHDL, Verilog permite concorrência em nível de **tarefas** (vários processos executando paralelamente, acionados dentro de uma estrutura **fork-join**) e concorrência em nível de **sentenças** (através do uso de operadores de atribuição simultânea).

b) *Transição de estados*

Esta característica não é suportada em Verilog.

c) *Hierarquia*

Ao permitir que um sistema possa ser especificado como uma hierarquia de **módulos** (*modules*) interconectados, Verilog suporta hierarquia **estrutural**. Cada módulo pode ser descrito ou através do uso de módulos de nível inferior, ou através da especificação de seu comportamento como um programa. Verilog também suporta hierarquia **comportamental**, ao permitir que um processo em qualquer nível hierárquico possa ser decomposto em um conjunto de subprocessos **paralelos** ou **seqüenciais**.

d) *Estruturas de programação*

Verilog apresenta um conjunto de estruturas de programação similar à linguagem C, permitindo uma boa descrição de processos. Uma característica poderosa nesta linguagem é a existência de uma **PLI** (*Programming Language Interface*) padronizada, a qual permite acessar a estrutura de dados interna do Verilog. Através desta interface, pode-se implementar extensões a um simulador Verilog comercial, tais como:

- um calculador de atraso em linguagem C para uma biblioteca de células;
- um gerador de *waveform* gráfico;
- um depurador (*debugging*) personalizado;
- modelos de simulação em linguagem C;
- interfaces *hardware/software*.

e) *Término comportamental*

Esta característica é suportada por Verilog tal qual ocorre com VHDL, uma vez que o encerramento de um processo é bem definido (execução da última sentença).

f) *Tratamento de exceções*

Verilog suporta preempção através da cláusula **disable**, a qual desabilita um determinado **bloco** de sentenças sequenciais e transfere o controle para a sentença seguinte ao bloco.

g) *Representação de tempo*

Especificação de tempo é suportada em Verilog através da modelagem de **atrasos** para portas e redes, além do **escalonamento** na atribuição sobre um sinal. Em relação à modelagem de atrasos, Verilog permite a definição de valores para atrasos associados à **subida** (*rise*, passagem do nível lógico **0**, **x** ou **z** para **1**), **descida** (*fall*, passagem do nível lógico **1**, **x** ou **z** para **0**) ou **desligamento** (*turn-off*, passagem do nível lógico **0**, **1** ou **x** para **z**) de um sinal. Para tal, utiliza-se o **senalizador de atraso** **#**. Se apenas um valor for especificado após **#**, o mesmo será utilizado para os três atrasos. Caso dois valores sejam especificados, eles serão utilizados, respectivamente, como atrasos de subida e de descida, sendo o atraso de desligamento implicitamente igual ao menor destes dois valores. Outra opção é explicitar os três valores. O uso de atrasos é dado no exemplo seguinte, o qual modela o comportamento de um multiplexador 2-1:

```
module multiplexor_2_to_1(out, cnt, a, b);

    output out;
    input cnt, a, b;
    wire not_cnt, a0_out, a1_out;

    not # 2 n0(not_cnt, cnt); /* Rise=2, Fall=2, Turn-Off=2 */
    and #(2,3) a0(a0_out, a, not_cnt); /* Rise=2, Fall=3, Turn-Off=2 */
    and #(2,3) a1(a1_out, b, cnt);
    or #(3,2) o0(out, a0_out, a1_out); /* Rise=3, Fall=2, Turn-Off=2 */

endmodule
```

O sinalizador de atraso **#** pode ser utilizado de forma similar à cláusula **after** de VHDL. Atribuições podem ser escalonadas como no exemplo abaixo, onde o valor de **x** será atualizado 25 unidades de tempo após a descida do sinal **clock**:

```
@(negedge) clock #25 x = y
```

h) *Comunicação*

Comunicação pode ser implementada em Verilog através de um modelo de **memória compartilhada**, utilizando-se fios (*wires*) que conectam portas à módulos, registradores e memórias, para estabelecer comunicação entre processos.

i) *Sincronização de processos*

Verilog pode suportar esta característica de várias formas. **Sincronização orientada a controle** pode ser implementada pelo uso de estruturas **fork-join**. Instruções de **controle de eventos** (que verificam mudanças no valor de um sinal) também podem ser utilizadas para sincronizar diferentes processos. No exemplo abaixo, a cláusula **wait** é utilizada para suspender o processo (no qual está inserida) até que o sinal *clock* assuma o valor lógico 0 (observe a equivalência com o último exemplo dado, no item g):

```
wait (clock = 0);  
#25 x = y;
```

4.4. CSP

A linguagem CSP (*Communicating Sequential Processes*, processos sequenciais que se comunicam) foi proposta por C.A.R. Hore em 1978 [23], visando superar as limitações que as linguagens de programação tradicionais apresentam ao rodar programas em máquinas multiprocessadoras. Posteriormente, outras linguagens foram derivadas de CSP como, por exemplo, Occam [24]. CSP permite especificar um programa como um conjunto de **processos concorrentes**, usando construtores que simplificam a especificação de comunicação e sincronização entre os processos. Além do seu uso como linguagem de programação, CSP tem sido utilizada para especificar sistemas de *hardware*.

a) *Concorrência*

Um programa CSP é formado por uma lista de **comandos**. O comando **parallel** permite que um processo possa ser dividido em um conjunto de subprocessos executados em paralelo. O comando encerra-se quando todos os seus subprocessos estiverem terminados. Cada subprocesso de um comando **parallel** pode ser descrito através de outro comando **parallel** e, assim, sucessivamente. Desta forma, CSP suporta concorrência em nível de **tarefas**.

b) *Transição de estados*

CSP não suporta esta característica, uma vez que não está baseada sobre a noção de estado.

c) *Hierarquia*

CSP suporta hierarquia **comportamental**, ao permitir o uso do comando **parallel** em vários níveis hierárquicos. Hierarquia **estrutural** não é suportada em CSP.

d) *Estruturas de programação*

Processos CSP são descritos através de estruturas de programação. Uma característica interessante é que sub-rotinas são implementadas como **co-rotinas**, as quais executam em paralelo com as rotinas que as chamaram (acionaram). Sub-rotinas **recursivas** podem ser

simuladas por meio de um **vetor de processos**, onde cada elemento do vetor representa um nível de recursão.

Estruturas de controle são implementadas através do comando **guarded**, o qual consiste de uma lista de **guardas** (condições booleanas) e uma lista de **comandos**. Estes comandos serão executados somente quando todas as guardas forem verdadeiras.

e) ***Término comportamental***

Esta característica é suportada, uma vez que a execução do último comando dentro de cada processo sinaliza o término do mesmo.

f) ***Tratamento de exceções***

Não há suporte para o tratamento de exceções em CSP.

g) ***Representação de tempo***

CSP não dispõe de mecanismos para a especificação de tempo.

h) ***Comunicação***

Ao contrário das linguagens vistas até agora (VHDL e Verilog), CSP não utiliza um modelo de memória compartilhada, uma vez que não possui variáveis globais. Assim, a comunicação entre processos concorrentes é obtida através de um mecanismo de **passagem de mensagens**, com o uso dos comandos **input** e **output**. Para que um processo **P1** envie dados para um processo **P2**, são necessárias as seguintes condições:

- a) O comando **output** de **P1** especificar **P2** como o destino dos dados a serem enviados;
- b) O comando **input** de **P2** especificar **P1** como a origem dos dados a serem recebidos;
- c) O tipo do receptor (variável) especificado no comando **input** de **P2** ser compatível com a expressão do comando **output** de **P1**.

i) ***Sincronização de processos***

O único mecanismo de sincronização em CSP é obtido através do **bloqueamento** (*blocking*) de um processo envolvido numa comunicação. Neste mecanismo, o processo que primeiro atingir seu comando de comunicação (**input/output**) ficará suspenso, até que o outro processo envolvido na comunicação fique pronto para a transferência de dados, isto é, alcance seu comando correspondente (**output/input**).

4.5. STATECHARTS

Estendendo FSM's com hierarquia, concorrência e comunicação, a linguagem Statecharts [7] foi projetada visando a especificação de **sistemas reativos**, os quais são sistemas predominantemente de controle, orientados a eventos. O objeto básico em Statecharts é o **estado** (*state*), o qual pode ser decomposto em um conjunto de subestados **seqüenciais** ou **concorrentes**. **Transições** entre estados são determinadas por uma combinação de **eventos** e **condições**.

a) *Concorrência*

Statecharts suporta concorrência em nível de **tarefas**, ao permitir que um estado possa ser decomposto em um conjunto de sub-estados concorrentes, os quais se encontram na condição de ativos (inativos) sempre que o estado pai (nível hierárquico superior) estiver ativo (inativo).

b) *Transição de estados*

Quando um estado é decomposto em um conjunto de subestados seqüenciais, a passagem de um estado para o outro (transição entre estados) é realizada mediante determinados **eventos** (que possam ocorrer no sistema modelado) ou **condições** (referentes a dados do sistema). Esta decomposição seqüencial de um estado pode ser representada por uma máquina de estados, formada pelos subestados do estado e arcos de transições. Ações instantâneas (sem atraso no tempo de computação) podem ser associadas às transições ou estados.

c) *Hierarquia*

Ao permitir que uma especificação possa ser decomposta em uma hierarquia de estados, Statecharts suporta hierarquia **comportamental**.

d) *Estruturas de programação*

Estruturas de programação não são suportadas em Statecharts.

e) *Término comportamental*

Statecharts não suporta término comportamental. Além disso, comportamento **não-determinístico** pode ser especificado, através da adoção de duas ou mais transições indo de um estado para outro. Neste caso, somente uma das transições será escolhida, de forma indeterminística.

f) *Tratamento de exceções*

Statecharts suporta a especificação de preempções, uma vez que pode-se associar uma transição que desative um determinado estado (e, conseqüentemente, todos os seus subestados) e ative um outro, mediante um evento que represente uma preempção.

g) *Representação de tempo*

Especificação de temporização **funcional** pode ser feita em Statecharts, através do uso de arcos de transições especiais denominados **timeout**. Estes arcos são rotulados com as quantidades de tempo mínima e máxima que um determinado estado possa perdurar.

h) *Comunicação*

Comunicação em Statecharts é obtida por meio de um mecanismo de **broadcast**, através do qual qualquer transição, geração de evento ou atualização de variável que ocorra em alguma parte do sistema modelado, será imediatamente sentida (percebida) nas demais partes do sistema.

i) *Sincronização de processos*

Statecharts permite o uso de mecanismos de sincronização tanto **dependente de controle** como **dependente de dados** (figura 4.1, adaptada de [3]). No primeiro tipo de mecanismo, é a estrutura de controle do comportamento a responsável pela sincronização de dois ou mais processos no sistema. No caso de Statecharts, a técnica empregada para sincronização dependente de controle é a de **inicialização**. Na figura 4.1.a, os subestados concorrentes **A** e **B** do estado **AB** são sincronizados quando da ocorrência do evento **e**, o qual inicializa os subestados **A** e **B**, cujos estados atuais passam a ser, respectivamente, **A1** e **B1**.

Na sincronização dependente de dados, há vários mecanismos que podem ser utilizados. A técnica de sincronização por **evento em comum** é mostrada na figura 4.1.b, na qual o evento **e** é responsável pela entrada simultânea dos processos (estados) **A** e **B** nos seus subprocessos (subestados) **A2** e **B3**. Já na parte **c**) da mesma figura, um exemplo de sincronização por **dados em comum** é ilustrado, no qual a atribuição sofrida pela variável **x** (em **A2**) constitui-se na condição associada ao arco de transição que promove a mudança de **B** do sub-estado **B1** para o **B2**. Finalmente, na figura 4.1.d, é mostrado um exemplo de sincronização por **detecção de status**, no qual a entrada no subestado **A2** de **A** (decorrente da ocorrência do evento **e**) é sentida (percebida) pela expressão **entered A2** associada ao arco de transição que conecta os subestados **B1** e **B2**.

4.6. SDL

O desenvolvimento da linguagem SDL (*Specification and Description Language*) começou em 1972. Um grupo de estudos dentro no CCITT (*Consultative Committee on International Telegraphy and Telephony*), hoje ITU-T (*International Telecommunication Union–Telecommunication Standardization Sector*), composto por 15 membros representando vários países e grandes companhias da área de telecomunicações (*Bellcore*, *Ericsson*, *Motorola*, dentre outras), voltou-se para o desenvolvimento de uma linguagem de especificação padronizada para a indústria de telecomunicações. A primeira versão da linguagem surgiu em 1976, seguida por novas versões em 1980, 1984, 1988, 1992 e 1996 [25]. Esta linguagem tem sido desenvolvida com o propósito de fornecer descrição e especificação não ambíguas do comportamento, estrutura e protocolos de sistemas de telecomunicação.

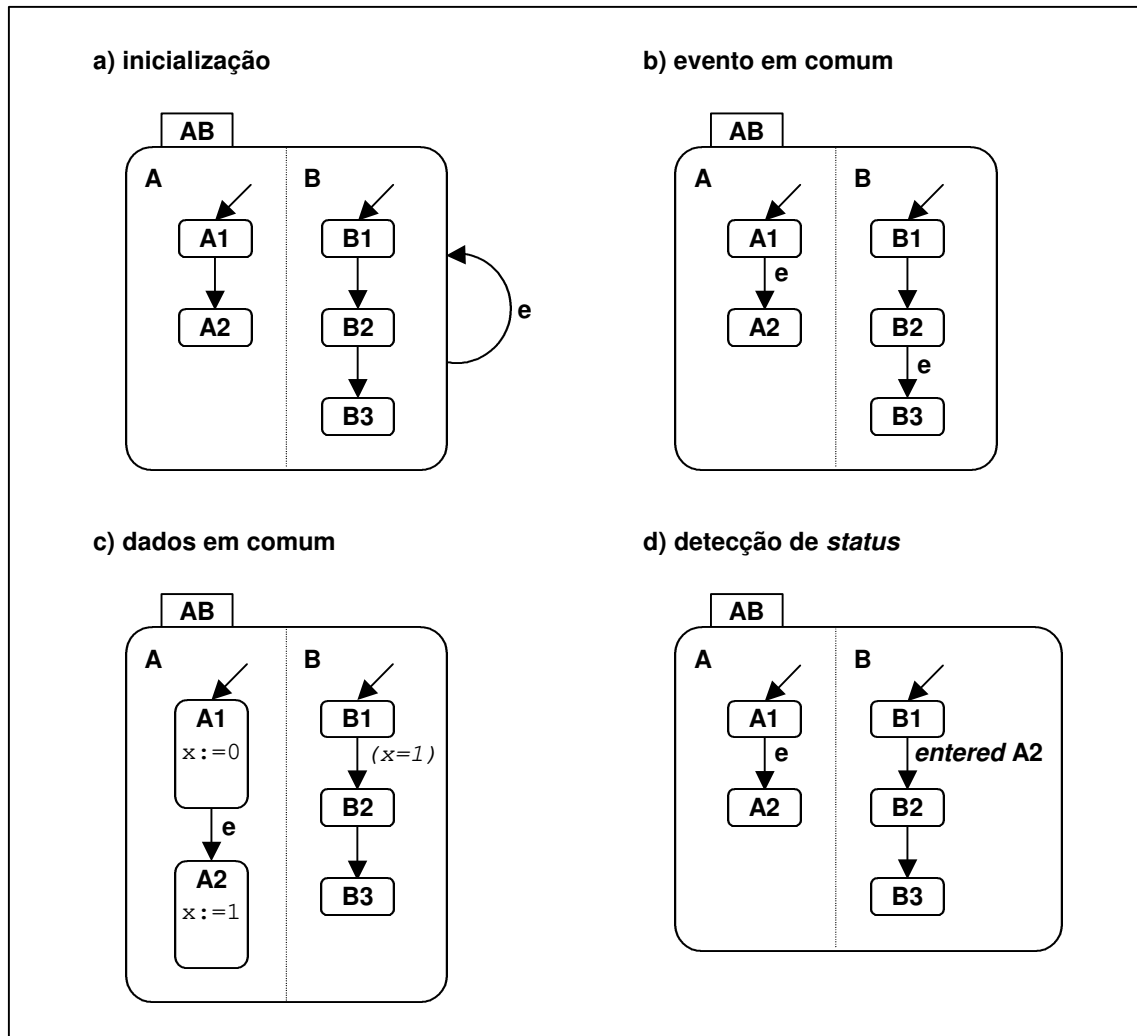


Figura 4.1. Técnicas de sincronização em Statecharts.

SDL permite duas formas equivalentes de representação, quais sejam, a **gráfica** (SDL/GR) e a **textual** (SDL/PR). SDL especifica um sistema como um conjunto **hierárquico** de **blocos**, os quais são representados, no nível mais baixo, por um **processo** correspondendo a uma **FSM** estendida com **variáveis** (utilizadas para registro de estados). O comportamento dinâmico de um sistema SDL encontra-se descrito nos processos. A hierarquia sistema/blocos é somente uma descrição estática da estrutura do sistema. Processos em SDL podem ser criados durante a inicialização do sistema ou mediante a ação de outros processos. Atingindo o seu ponto de término, um processo é encerrado. Durante a execução, mais de uma instância de um determinado tipo de processo pode existir. Cada instância tem um identificador único (**PId**). Os processos se comunicam assincronamente através do envio de mensagens discretas, denominadas **sinais**, sobre **rotas de sinais** (*signal routes*) entre os processos. Blocos se comunicam através de **canais**. A cada processo associa-se um **tempo de existência**, **parâmetros**, **variáveis** e uma **porta de entrada** (fila de sinais). Um processo está ou em um estado ou realizando uma transição entre dois estados. Uma transição é iniciada por um sinal da fila associada à porta de entrada do processo. Quando um sinal inicia uma transição, ele é removido da fila e consumido. Durante uma transição, pode ocorrer, por exemplo, a atualização de variáveis, a criação de instâncias de processos (a partir de tipos de processos previamente definidos) e o envio de sinais para outros processos.

a) *Concorrência*

Como mais de um processo pode estar ativo por vez, SDL suporta concorrência em nível de tarefas. O conceito de processos e de instâncias de processos, que trabalham de forma autônoma e concorrentemente, faz de SDL uma boa linguagem para a especificação de **sistemas de tempo real**.

b) *Transição de estados*

Característica suportada, conforme visto anteriormente.

c) *Hierarquia*

Hierarquia **estrutural** é suportada em SDL. O objeto principal da especificação é o **sistema** (*system*), o qual é descrito como uma hierarquia de **blocos** (*blocks*). O sistema é formado por um ou mais blocos, os quais são conectados entre si e com a fronteira do sistema através de **canais**, sobre os quais **sinais** são conduzidos de um ponto para outro. Blocos podem ser sucessivamente decompostos em outros blocos. Os blocos folha (nível hierárquico mais baixo) podem ter um ou mais **processos**, os quais se comunicam através de **rotas de sinais**. O suporte para hierarquia **comportamental** é limitado, uma vez que existe um único nível de processos, contido nos blocos folha. Entretanto, SDL suporta decomposição seqüencial de processos em procedimentos (estados de uma FSM estendida).

d) *Estruturas de programação*

SDL possui um conjunto pobre de estruturas de programação, ao lado de alguns construtores de tempo. Estruturas para a definição de dados são bem elaboradas.

e) *Término comportamental*

SDL suporta esta característica pois, uma vez criado um processo (o que ocorre durante a inicialização do sistema ou mediante a ação de outro processo), ele se mantém ativo até atingir o seu ponto de término (construtor **stop**).

f) *Tratamento de exceções*

SDL não suporta tratamento direto de exceções.

g) *Representação de tempo*

A especificação de tempo é suportada em SDL através do objeto **timer** (declarado dentro de um processo), o qual gera um sinal temporizado, depois de decorrido um tempo pré-estabelecido.

h) *Comunicação*

SDL não utiliza dados globais. Assim, a comunicação entre blocos e processos é obtida através de interfaces bem definidas, envolvendo **canais** (conectando blocos) e **rotas de sinais** (conectando processos entre si, e processos com a fronteira do bloco onde estão situados). Esta arquitetura de comunicação facilita o desenvolvimento em equipe, ao assegurar consistência entre diferentes partes do sistema.

i) *Sincronização de processos*

A sincronização em SDL é obtida ao permitir que um processo verifique e espere por uma atualização apropriada sobre um sinal.

4.7. SILAGE

A linguagem Silage [26] foi desenvolvida para tratar questões relativas à especificação de **sistemas DSP** (*Digital Signal Processing*, Processamento Digital de Sinais). Um sistema DSP pode ser conceitualizado como um conjunto de operações realizadas sobre um fluxo contínuo de valores de dados na sua entrada. Para suportar tais sistemas, Silage é uma linguagem **aplicativa**, isto é, ela somente especifica a aplicação de funções para a manipulação de valores de dados, sem fazer quaisquer atribuições sobre variáveis.

Silage é uma excelente linguagem para a especificação de descrições a fluxo de dados. Expressões Silage representam fluxos de valores. Assim, na expressão **a+b**, **a** e **b** representam um fluxo de números, ao invés de variáveis ou elementos de vetores (como é comum em outras linguagens). Um programa Silage recebe, de forma sincronizada, o conjunto de valores de entrada, transformando-o em um fluxo de valores de saída. Para tal, utiliza um conjunto de **definições**, cada qual derivando novos valores como uma função de outros valores. A ordem das definições não é relevante, uma vez que as mesmas não representam atribuições sobre variáveis.

A metodologia de coprojetos *hardware/software* **Ptolemy** [27] (seção 5.2) utiliza a linguagem Silage para **simulações bit-true** de componentes associados a *hardware* dedicado. Estas simulações são utilizadas durante a fase de **síntese de hardware** para análise de efeitos de precisão finita, ajuste do algoritmo para tamanhos limitados de palavras (de dados ou de controle) e para a determinação do tamanho ideal das palavras. O código Silage serve de entrada para **ferramentas de síntese** de *datapaths* personalizados. Estas ferramentas são capazes de **estimar** o consumo, a área e o caminho crítico. Blocos Silage são adicionados ao modelo de **simulação** do sistema (**Thor**) para representar o *hardware* dedicado.

a) *Concorrência*

Ao permitir a descrição de sistemas a fluxo de dados, Silage suporta concorrência orientada a dados.

b) *Transição de estados*

Esta característica não se aplica a programas Silage.

c) *Hierarquia*

Silage não suporta descrições hierarquizadas. Entretanto, permite recursão, embora em número limitado para cada função.

d) *Estruturas de programação*

Por se tratar de uma linguagem aplicativa, Silage não suporta construtores de linguagens de programação.

e) *Término comportamental*

Esta característica não se aplica a programas Silage.

f) *Tratamento de exceções*

Silage não possui tratamento de exceções.

g) *Representação de tempo*

Silage possui uma forma limitada de temporização funcional, obtida através de **recorrências**. Numa recorrência, um ou mais elementos do fluxo que está sendo tratado, depende de valores anteriores do fluxo. O operador de atraso @ é utilizado para representar valores anteriores num fluxo, como na sentença $x = x@1 + 1$, onde os valores do fluxo atual x devem ser maiores em uma unidade, quando comparados ao fluxo x anterior.

h) *Comunicação*

Característica não aplicável.

i) *Sincronização de processos*

Não se aplica a programas Silage.

CAPÍTULO 5: COPROJETO HARDWARE/SOFTWARE DE SISTEMAS EMBUTIDOS

5.1. INTRODUÇÃO

Conforme visto na introdução do capítulo 1, **sistemas digitais** podem ser divididos em **sistemas de computação de propósito geral** e em **sistemas embutidos**. Desses dois grupos, o que vem despertando maior interesse tanto da comunidade acadêmica quanto da industrial é o de sistemas embutidos, o qual forma um mercado que já é maior e cresce mais rapidamente que o relacionado aos computadores de propósito geral [2].

Sistemas embutidos podem ser divididos em três grandes grupos, quais sejam, o de **sistemas transformacionais**, o de **sistemas reativos** e o de **sistemas de tempo real**. Sistemas **transformacionais** são aqueles que realizam um **padrão repetitivo de processamento**, para **conjuntos diferentes de dados** que alimentam suas entradas periodicamente; nesta classe enquadram-se os sistemas DSP (*Digital Signal Processing* – Processamento Digital de Sinais), amplamente utilizados na área de comunicação digital e sistemas multimídia. Sistemas **reativos** são aqueles que **reagem ao ambiente**, através da **execução de funções** e da **ativação de atuadores**, em resposta a determinados **estímulos de entrada** (capturados por **sensores**). Por fim, sistemas de **tempo real** são aqueles que, em determinados momentos, são obrigados a **responder** (realizar certo processamento) dentro de **intervalos de tempo** bastante **restritos**.

Exemplos de sistemas embutidos incluem:

- a) Aparelhos simples, como um forno de microondas, onde o microprocessador fornece uma interface amigável e aspectos avançados;
- b) Um mecanismo de execução computacional intensiva, como uma impressora laser;
- c) Um dispositivo de mão, como um telefone celular, onde o consumo de energia e o tamanho representam aspectos críticos, ao mesmo tempo em que deve tratar do processamento digital de sinais e outras funções sofisticadas;
- d) Um controlador industrial em uma fábrica, no qual características de confiabilidade, capacidade de manutenção e facilidade de programação são desejáveis;
- e) Um controlador crítico em termos de segurança, como, por exemplo, um controlador de freio *ABS* ou um mecanismo de piloto automático.

Geralmente, um sistema embutido contém componentes de *hardware* e de *software* trabalhando em conjunto. O **projeto concorrente** dos componentes de *hardware* e de *software* de um sistema é denominado **coprojeto hardware/software** (também denominado **projeto misto hardware/software** ou **projeto em nível de sistema**). Os problemas envolvidos no coprojeto não são novos; projetistas têm desenvolvido sistemas *hardware/software* durante anos. O que há de novo são as **metodologias** associadas com o desenvolvimento concorrente de componentes de *hardware* e de *software* [28].

Em uma estratégia de **projeto tradicional**, os projetistas do sistema tomam decisões de **particionamento hardware/software** nos estágios iniciais do **ciclo de desenvolvimento do projeto (fluxo de projeto, processo de projeto)**. Após estas decisões de particionamento (determinação do que será implementado em *hardware* e em *software*), os projetos de

hardware e de *software* seguem de forma **independente** um do outro. Há **pequena interação** entre os dois projetos, devido à inexistência de um *framework* (modelo baseado em componentes computacionais) unificado para representação, validação e síntese de sistemas.

Os novos sistemas exigem uma estratégia de projeto **flexível**, na qual os projetos de *hardware* e de *software* caminhem em **paralelo**, de **forma interativa**, para assegurar que a implementação de um sistema não somente funcione adequadamente, mas também atenda aos **requisitos** do projeto, geralmente dentro de um **curto tempo de janela de mercado** [29]. Os projetistas podem determinar a melhor opção de implementação, após avaliarem diferentes **alternativas de implementação**, baseados em **critérios (requisitos)** de **desempenho** (taxa de processamento), **programabilidade**, **área** (de *hardware* e de *software*), **consumo**, **potência dissipada**, **custos** (de **desenvolvimento** e de **manufatura**), **confiabilidade**, **manutenção** e **evolução do projeto (famílias de produtos)**. Esta estratégia exige ferramentas que suportem uma representação *hardware/software* unificada, verificação em diferentes níveis de abstração e síntese *hardware/software* [27].

Há uma diversidade de metodologias de coprojeto *hardware/software*, abrangendo diferentes etapas do processo de projeto de sistemas embutidos. Kalavade e Lee [27] consideram a existência de duas classes de metodologias de coprojeto, quais sejam, a **abordagem unificada** e a **abordagem heterogênea**. A **vantagem da primeira** dependeria da existência de uma **linguagem de especificação unificada** em **nível de sistema**, com a qual fosse possível a **descrição de sistemas digitais** dos mais **variados tipos**. A partir de uma linguagem desse tipo, pode-se desenvolver um **ambiente integrado** de ferramentas de projeto, indo da **especificação** à **síntese**, passando pelas ferramentas de **verificação**. Entretanto, a **variedade de domínios de projeto** é **grande**, dificultando a concepção semântica de uma linguagem de especificação abrangente. A **vantagem da abordagem heterogênea** recai sobre a **diversidade de estilos de projeto**. O princípio da abordagem é aproveitar o **conhecimento** dos projetistas em seus **diferentes domínios de projeto** e as **ferramentas de CAD** anteriormente desenvolvidas.

Uma das dificuldades encontradas para o desenvolvimento de uma metodologia de coprojeto é a **diversidade de componentes** que podem ser usados para implementar os subsistemas. Entre estes componentes, estão:

- a) *Software/firmware*;
- b) ASICs;
- c) *Core-based* ASICs;
- d) ASIPs;
- e) FPGAs;
- f) Subsistemas multiprocessadores;
- g) Circuitos analógicos.

Resumindo, **coprojeto hardware/software** refere-se ao **projeto concorrente** dos componentes de *hardware* e de *software* de um **sistema**, de forma a **atender** a **funcionalidade** pretendida e os **requisitos de projeto**, geralmente em curto prazo de tempo; há uma grande variedade de componentes envolvidos, que podem ser utilizados de diferentes formas, proporcionando **várias alternativas de projetos**, as quais deverão ser **testadas** e **avaliadas**. Como diferentes aplicações apresentam particularidades específicas, uma **metodologia de coprojeto** encontra-se **restrita** a um determinado **grupo de aplicação**.

Geralmente, **nem todas as etapas** de uma metodologia são **automatizadas**, e nem sempre uma metodologia cobrirá todas as etapas. Além disso, devido à diversidade de aplicações, há **diferenças** entre os **fluxos de projeto** de metodologias propostas por grupos de pesquisadores em todo o mundo.

Neste capítulo, é dada uma **visão geral** das principais etapas de metodologias de coprojeto *hardware/software* de sistemas embutidos, além de mostrar como diferentes metodologias atendem tais etapas. A seção 5.2 descreve de forma geral as **etapas de coprojeto**, mostrando como elas estão relacionadas na metodologia **Ptolemy**. A **especificação** de sistemas *hardware/software* em diferentes metodologias é apresentada na seção 5.3. A seção 5.4 apresenta o problema do **particionamento hardware/software**, conjuntamente com os problemas de **escalonamento** e **alocação**. Técnicas envolvidas na **verificação (validação)** de um sistema encontram-se na seção 5.5.

A elaboração deste capítulo baseia-se nas notas de aula e referências bibliográficas da disciplina **IA542 – Coprojeto de Hardware-Software em Sistemas Embutidos**, ministrada na FEEC/Unicamp pela professora **Alice Maria Bastos H. Tokarnia**.

5.2. ETAPAS DE COPROJETO

As metodologias de coprojeto *hardware/software* de sistemas embutidos apresentam diferenças em relação ao fluxo (processo) de projeto adotado. Entretanto, na maioria dos casos, as etapas envolvidas são basicamente as mesmas, diferindo na forma como são tratadas: manualmente ou automaticamente. As **etapas** incluem:

- a) **Especificação do sistema;**
- b) **Particionamento hardware/software;**
- c) **Escalonamento/alocação;**
- d) **Síntese (*hardware, software, interface*);**
- e) **Verificação (verificação formal, simulação, prototipação);**
- f) **Avaliação do projeto.**

A figura 5.1 apresenta um **fluxo de projeto**, proposto por Kalavade e Lee [27], para uma metodologia genérica de coprojeto *hardware/software*, o qual não está voltado para um *framework* em particular, embora seja aplicado na metodologia **Ptolemy**. Vejamos, na seqüência, como a metodologia Ptolemy trata desse fluxo de projeto; nesta discussão, os **números entre parênteses** correspondem aos **estágios** mostrados na figura 5.1.

Dada a **especificação** de um sistema, o projetista desenvolve um **algoritmo** usando simulações funcionais em alto nível **(1)**, independentes de qualquer forma de implementação.

O projetista então **particiona** o algoritmo em *hardware* e em *software* **(2)**, guiado por requisitos de **velocidade, complexidade e flexibilidade**. Componentes que apresentam funções complexas (alto custo em *hardware*) ou que exijam flexibilidade (ou programabilidade, útil, por exemplo, em casos de configuração de parâmetros ou em expansão futura de funcionalidade), são associados para implementação em *software*. Operações que tenham requisitos críticos de velocidade de execução são associadas para implementação em *hardware*. Para **explorar o espaço (alternativas) de projeto**, o projetista deve repetir esta etapa a quantidade de vezes necessárias. No Ptolemy, esta etapa é feita manualmente.

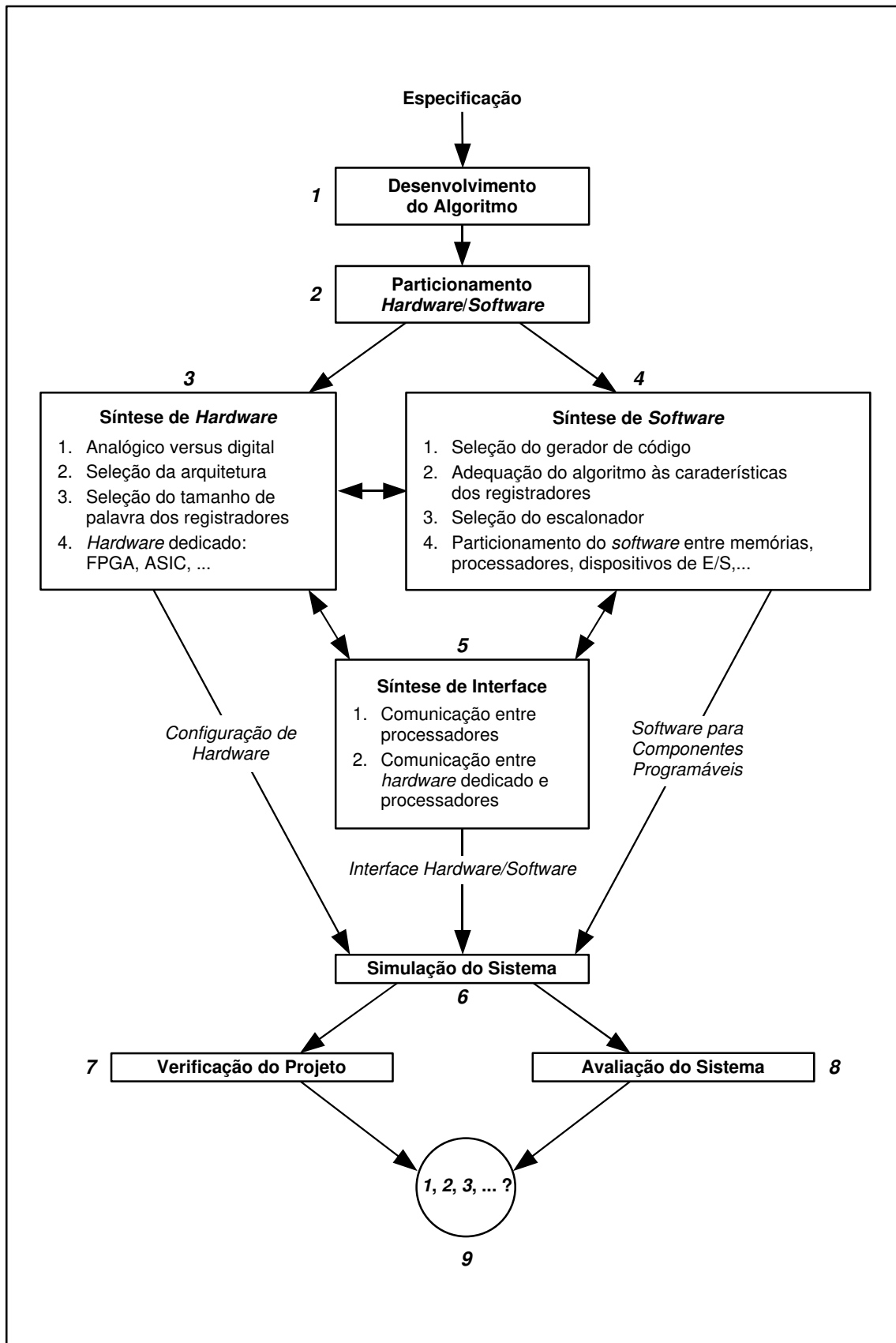


Figura 5.1. Metodologia genérica de coprojetado (adaptação de Kalavade e Lee [27]).

Após o particionamento, seguem-se as etapas de **síntese de *hardware* (3)**, ***software* (4)** e **interface (5)**, as quais estão profundamente amarradas, pois mudanças em uma delas trarão efeitos imediatos sobre as outras duas.

Decisões de projeto em ***hardware*** incluem a **seleção dos processadores programáveis** (afetando a seleção do gerador de código para módulos em *software*) e a determinação do **número de processadores** e de seu **esquema de interconexão** (influenciando o particionamento e escalonamento do código e a síntese da interface *hardware/software*); processadores programáveis incluem ASIPs e processadores de propósito geral. Na síntese de ***hardware dedicado***, as escolhas variam desde a geração de ***datapaths* personalizados** até a programação de **FPGAs**. Ao projetar *datapaths* personalizados, o projetista deve escolher o tamanho das palavras dos registradores.

Em relação ao ***software***, no caso de processadores de ponto fixo, algumas modificações algorítmicas podem ser necessárias para minimizar os efeitos da precisão finita. A síntese de *software* envolve o **particionamento** e o **escalonamento do código** da aplicação **entre os vários processadores**, além da **síntese do código** para a **comunicação** entre estes **processadores** e os demais **dispositivos**. Estas decisões dependem da **arquitetura** escolhida. Neste processo de particionamento de *software* e escalonamento (também denominado de particionamento temporal), o projetista toma **decisões** com base na **otimização de funções custo**. Exemplos destas funções representam o **custo de comunicação**, a **largura da banda de memória** e a **capacidade de memórias locais e compartilhadas**.

A síntese de **interface** envolve, **em relação ao *hardware***, a adição de *latches*, FIFOs ou decodificadores de endereços. **Em relação ao *software***, envolve a inserção de código para operações de E/S e de sincronização por semáforos.

Uma forma típica de tratamento deste problema cíclico (a **síntese**) é começar com um projeto e trabalhar sobre este **iterativamente** para **explorar diferentes opções**.

A próxima etapa é a de simulação heterogênea **(6)**. Em particular, o *hardware* simulado tem que rodar o *software* gerado. Isto envolve a **interação** de vários **simuladores** diferentes, caso várias linguagens de especificação tenham sido utilizadas. Uma descrição sobre como o ambiente Ptolemy trata desta etapa é dada mais adiante.

Os resultados da simulação são utilizados para **verificar (7)** se o projeto atende às **especificações da aplicação**. Tendo realizado a síntese de *hardware* e de *software* para uma determinada alternativa de projeto, o projetista pode **estimar** área, consumo, caminho crítico, taxas de utilização de componentes e barramentos, entre outros fatores. Após usar estas estimativas, para **avaliar o projeto** quanto às **restrições (8)**, o projetista poderá reparticionar o sistema ou tentar diferentes opções de síntese (retorno no fluxo de projeto, etapa **9**). Desta forma, o processo como um todo é **iterativo**. As etapas **(7)** e **(8)** são tratadas manualmente no Ptolemy.

Para o tratamento destas etapas na metodologia Ptolemy, a qual é uma abordagem heterogênea, seus criadores conceitualizaram **domínios de projeto**, os quais utilizam-se de determinados conjuntos de **ferramentas** e **representações** para auxílio ao projeto. Na referência [27] são descritos com algum detalhe os domínios **SDF**, **Thor** e **Silage**. Além destes, são mencionados os domínios **DDF** (*dynamic dataflow*) e **DE** (*discret event*).

SDF (*synchronous dataflow*) é utilizado nas fases de **desenvolvimento algorítmico**, **síntese de hardware** (modelagem funcional de componentes analógicos) e **síntese de software**, etapa na qual este domínio recebe um nome de acordo com o **gerador de código** designado, tal como **CGC** (*code generation in C*), **CG56** (*code generation in DSP56000 family*), **CG96** (*code generation in DSP96000 family*), entre outros. Por permitir a **modelagem funcional de componentes analógicos**, tais como filtros e conversores, este domínio é utilizado também na fase de **simulação do sistema**.

De distribuição livre, **Thor** é um **simulador funcional** para *hardware* digital, podendo simular desde portas lógicas até *chips* DSP programáveis. Permitindo simulação de sistemas multiprocessadores, é fundamental na seleção da arquitetura, durante a fase de **síntese de hardware**. No caso deste tipo de sistema (multiprocessadores), Thor é utilizado também na **síntese de interface**. Na fase de **simulação do sistema**, componentes analógicos (domínio SDF) são inseridos no modelo de simulação arquitetural Thor, através de *wormholes* (comunicação – interface – entre as partes descritas em domínios distintos).

O domínio **Silage** é utilizado para **simulações bit-true** de componentes associados a *hardware* dedicado. Estas simulações são utilizadas durante a fase de **síntese de hardware** para análise de efeitos de precisão finita, ajuste do algoritmo para tamanhos limitados de palavras (de dados ou de controle) e para a determinação do tamanho ideal das palavras. O código Silage serve de entrada para **ferramentas de síntese** de *datapaths* personalizados. Estas ferramentas são capazes de **estimar** o consumo, a área e o caminho crítico. Blocos Silage são adicionados ao modelo de **simulação** do sistema (Thor) para representar o *hardware* dedicado.

A manipulação de informações em diferentes domínios torna o sistema Ptolemy flexível e poderoso, devido ao aspecto da **modularidade**. Os domínios tratam de diferentes tipos de atividades (modelagem, simulação, geração de código, estimativas, síntese), as quais estão intrinsicamente amarradas e possuem suas próprias dificuldades de desenvolvimento. Isolando-se os domínios em módulos distintos, os quais podem ser associados com uma interface apropriada, obtém-se uma plataforma de projeto flexível e poderosa.

Ptolemy é flexível devido à fácil substituição de um domínio por um análogo (por exemplo, a substituição de um domínio de geração de código por outro) e, também, devido à facilidade de expansão (pode-se introduzir facilmente um novo domínio no sistema, bastando-se atender aos requisitos da interface denominada *EventHorizon*).

A plataforma Ptolemy é poderosa devido à possibilidade de poder testar várias alternativas de projeto (alteração de domínios) e, também, devido ao aprimoramento que pode ser dado de forma particularizada em um domínio qualquer, sem a necessidade de adequação dos demais domínios (desde que a interface entre os domínios seja mantida).

O exemplo da metodologia Ptolemy serve para demonstrar quão amarradas estão as tarefas que devem ser executadas ao longo do processo de projeto de um sistema embutido. Na sequência, são vistas particularidades a respeito destas tarefas, abrangendo outras metodologias.

5.3. ESPECIFICAÇÃO

Modelos conceituais (capítulo 2) são utilizados para representar **diferentes aspectos** inerentes ao sistema. Para que possam ser trabalhados em computador, estes modelos precisam ser **capturados**, cada qual, por alguma **linguagem de especificação**. A **especificação** gerada é então tratada pelas demais tarefas do fluxo de projeto.

A especificação de sistemas na metodologia **POLIS** está centrada sobre uma única representação baseada em FSM, denominada **CFSM** (*Co-design Finite-State Machine* – Máquina de Estados Finitos para Coprojeto) [30]. CFSM, assim como FSM, transforma um conjunto de entradas em um conjunto de saídas utilizando-se de um conjunto finito de estados internos. A diferença entre os dois modelos é que o modelo de comunicação síncrona de FSMs concorrentes (**HCFSM**, seção 2.2.1) é substituído, no modelo CFSM, por um tempo de reação finito, não-nulo. Este modelo de computação pode ser descrito como globalmente assíncrono, localmente síncrono.

Cada **elemento** de uma rede de CFSMs descreve um **componente** do sistema a ser modelado. A especificação CFSM é **independente de implementação** em *hardware* ou em *software*. Embora executem a mesma computação para cada uma das transições da CFSM, *hardware* e *software* exibem características de atraso diferentes. Uma implementação síncrona em *hardware* a partir de uma CFSM pode executar uma transição em 1 ciclo de *clock*, enquanto uma implementação em *software* exigirá mais do que 1 ciclo de *clock*.

CFSM é um modelo que permite a utilização de **técnicas de síntese e de verificação**, uma vez que muitos resultados teóricos e ferramentas para o modelo FSM podem ser adaptados para CFSM. A referência [30] mostra como uma abordagem para verificação formal baseada em autômatos pode ser aplicada para modelos CFSM. Numa abordagem baseada em autômatos, um sistema é modelado por um autômato de estados finitos, sendo a **linguagem** desse autômato (L_{sistema}) considerada o **comportamento do sistema**. A linguagem do autômato corresponde a um conjunto de **seqüências de entradas e saídas**, observável nas **portas** do sistema. Neste contexto, a tarefa da **verificação formal** é mostrar que todas estas seqüências são **aceitáveis**. As seqüências aceitáveis são especificadas como uma linguagem de um outro autômato ($L_{\text{aceitáveis}}$), de maneira que o problema da verificação fica reduzido à checagem da linguagem $L_{\text{aceitáveis}}$ estar contida na linguagem L_{sistema} .

Assim como outros modelos baseados em FSM, CFSM apresenta o problema de **explosão no número de estados**, dificultando a manipulação do modelo. A estratégia sugerida pelos autores em [30] para superar esse problema é realizar **agrupamentos** de determinados estados em um único estado (tratar um grupo de estados como um único estado). Desta forma, executa-se uma ascensão para um nível de abstração superior e conseqüentemente diminui-se a complexidade do problema.

A metodologia proposta por Wolf *et al.* [31] baseia-se na **especificação funcional** em nível de sistema. O projetista especifica o sistema a ser tratado através de **objetos e métodos**, classificados em 3 grupos: os de *hardware*, de *software* e de **coprojeto**. Neste último, enquadram-se os módulos do sistema compostos por objetos e métodos que poderiam ser implementados tanto em *hardware* quanto em *software*. Os módulos dos grupos de *hardware* e de *software* são descritos em **C++**, enquanto os módulos de coprojeto são descritos numa linguagem denominada **OOFs** (*Object-Oriented Functional Specification*).

Para ilustrar melhor a importância de uma especificação funcional, é apresentado, na sequência, o restante do fluxo de projeto da metodologia de Wolf *et al.*

Após a especificação, é feito o **particionamento**: escolhe-se, **manualmente**, para cada módulo de coprojeto, a sua melhor implementação. Os módulos a serem implementados em *hardware* são transformados (via tradutor) em código da HDL **Bestmap-C** (visando síntese) e em código C++ (visando simulação). Os módulos escolhidos como de *software* são transformados pelo tradutor em código C++.

A próxima etapa é a de **simulação**, onde os diversos fontes C++, gerados para os módulos e para as *templates* (modelos) de interface *hardware/software*, são compilados e executados, visando a verificação de sua correção.

A **avaliação de desempenho** do *software* gerado para os módulos de coprojeto é feita através de utilitários padrão C para medida de desempenho. Já para os módulos de coprojeto associados a *hardware*, a avaliação de desempenho é feita sobre resultados de estimativas de área e de desempenho fornecidas pelo *Bestmap-C*.

Os resultados da avaliação de desempenho podem determinar a necessidade de se tomar diferentes decisões de particionamento ou de se realizar a **síntese**. A síntese de *hardware* e de interfaces *hardware/software* é realizada a partir da descrição em *Bestmap-C*.

As etapas automatizadas da metodologia de Wolf *et al.* são as de simulação, avaliação de desempenho e síntese. Entretanto, estas etapas são automáticas somente dentro das atividades a que elas se destinam (não há integração), sendo a avaliação de desempenho muito restrita. Desta forma, a metodologia, apesar de ajudar no trabalho do projetista, relega ao mesmo decisões importantes, como a determinação da melhor escolha de particionamento, a verificação da funcionalidade do sistema (através dos resultados de simulação) e a escolha das interfaces *hardware/software*.

Pelo exposto em [31], pode-se concluir que a argumentação dos autores acerca da "**neutralidade**" da especificação funcional está bem ilustrada, uma vez que demonstraram (através de exemplos) como métodos funcionais em *OOFs* podem gerar tanto descrições em *software* quanto em *hardware*. Desta forma, a **especificação funcional permite um número maior de implementações** do que uma **especificação operacional**, a qual **tende** a adotar uma sintaxe de uma dada linguagem de programação (voltada para a descrição de *software*) ou a sintaxe de uma HDL (voltada para a descrição de *hardware*). Esta "**tendência**" pode **comprometer** a **qualidade** e a **quantidade** de formas de implementação em *hardware* e *software* que podem ser levadas em conta, devido à **especificidade de domínio** que linguagens de programação e HDLs possuem. Assim, uma **especificação funcional** coloca-se numa posição neutra, em relação à decisão de implementar um componente de um sistema em *hardware* ou *software*, simplificando **decisões de particionamento**, ao permitir que várias alternativas de implementação possam ser testadas.

Nesta seção foram vistos dois exemplos de especificação em nível de sistema. Ressalta-se, no entanto, que modelos conceituais de especificação (capítulo 2) podem ser utilizados em várias fases do fluxo de projeto de uma metodologia de coprojeto *hardware/software*, de acordo com a ênfase dada pela metodologia. Nas próximas seções, é vista a importância de modelos de especificação para o tratamento de questões relativas às tarefas de particionamento, escalonamento e alocação.

5.4. ALOCAÇÃO, PARTICIONAMENTO *HARDWARE/SOFTWARE* E ESCALONAMENTO

Dada uma biblioteca de componentes de sistema, tais como processadores, memórias e módulos IP (*Intellectual Property*) de terceiros, a tarefa de **alocação** é definida como a **seleção** do tipo e **número** destes componentes, assim como a determinação de sua **interconexão**, de tal forma que possam a **funcionalidade** do sistema ser **implementada**, as **restrições de projeto** ser **satisfeitas** e a **função objetivo de custo** ser **minimizada** [32]. Geralmente, esta tarefa é feita manualmente.

A tarefa de **particionamento** *hardware/software* tem por objetivo encontrar uma **implementação** que atenda os **requisitos de desempenho** a um **custo mínimo**. A idéia é explorar a **potencialidade** oferecida por componentes de *hardware* e de *software*. Geralmente, seções críticas do sistema em relação a tempo são associadas a implementações em componentes mais velozes (*hardware* dedicado), enquanto seções menos críticas (ou que exijam programabilidade) são associadas a *software* (executado por componentes geralmente mais lentos e baratos) [33].

Dado um conjunto de comportamentos e um possível conjunto de restrições de desempenho, a tarefa de **escalonamento** é determinar a **ordem** total dos **tempos de chamada** (para **execução**) das **tarefas** rodando num mesmo **PE** (*Processor Element* – Elemento Processador), ao mesmo tempo **respeitando** a ordem parcial imposta pelas **dependências** na **funcionalidade**, assim como **minimizando** o **overhead** (custo advindo) de **sincronização** entre os PEs e o **overhead** de **troca de contexto** dentro de cada PE [32].

As **metodologias** de coprojeto *hardware/software* **diferem** na **ordem** de execução destas **etapas**. Há metodologias que agrupam as tarefas de escalonamento e alocação numa única etapa, enquanto outras agrupam as tarefas de particionamento e alocação. Nesta seção, são vistos dois algoritmos. O primeiro deles é utilizado na metodologia Ptolemy, resultando no particionamento e escalonamento de tarefas de um sistema cujos componentes foram previamente alocados. O segundo algoritmo trata do chamado **particionamento temporal**, isto é, dado um **particionamento espacial** (ou comum, como explicado mais adiante) e uma alocação de componentes, o algoritmo faz o escalonamento das tarefas da aplicação, adotando técnicas de *pipeline* para explorar a concorrência entre estas, além de mapear as tarefas entre os componentes alocados.

O algoritmo de particionamento e escalonamento **GCLP** (*Global Criticality/Local Phase* – **Criticalidade Global/Fase Local**) foi proposto por Kalavade e Lee [34] para uso na metodologia Ptolemy. Seu aspecto chave é o **mecanismo adaptativo** das **funções objetivo de custo**, o qual é baseado na **combinação** entre **medidas globais** e **locais**. Como a minimização da área de *hardware* e as restrições de latência apresentam objetivos **contraditórios**, uma medida de **criticalidade global** (**GC**) do tempo é responsável pela seleção de uma função objetivo (dentre duas possíveis), de acordo com o tempo **restante** para execução da aplicação. Adicionalmente, características locais dos nós (nós = tarefas) são enfatizadas ao classificá-los em tipos **fase local** (**LP**). Um nó de fase local 1 (denominado **extremista**) é aquele que tem uma **preferência óbvia** por uma implementação em *hardware* ou em *software*. Um nó de fase local 2 (**repelente**) se **opõe** à implementação em *hardware* ou em *software*, quando comparado a outros nós. A cada **iteração** do algoritmo, os critérios globais e locais são relacionados por um mecanismo de **threshold** (limiar), de forma a determinar a melhor implementação (*hardware* ou *software*) para o nó em análise.

O **modelo** utilizado no algoritmo GCLP é um **grafo orientado acíclico** (**DAG**, *Direct Acyclical Graph*), representando a **aplicação**. Os **nós** do grafo representam **computações** (no nível de granulosidade de **tarefas** ou de **processos**), enquanto os **arcos** representam **comunicação** entre estas computações. A **arquitetura** corresponde a um **processador genérico** (para rodar os nós associados a *software*), um **core-based ASIC** (para implementar os nós associados a *hardware*) e uma **memória** (para armazenar o *software* e, quando for o caso, suportar comunicação via E/S mapeada para a memória). Assim, para que o algoritmo possa avaliar as alternativas de associação de nós à implementação em *hardware* ou em *software*, cada nó **i** possui 4 atributos, quais sejam:

- **ah_i** = **área** exigida para implementação do i-ésimo nó em *hardware*;
- **as_i** = **tamanho** do código exigido para implementação do i-ésimo nó em *software*;
- **th_i** = **tempo** de execução para implementação do i-ésimo nó em *hardware*;
- **ts_i** = **tempo** de execução para implementação do i-ésimo nó em *software*.

Para cada arco ligando um nó **i** a um nó **j**, associa-se um número inteiro não negativo **N_{i,j}** indicando o número de amostras (*samples*) de dados enviadas de **i** para **j**. Os custos associados à comunicação de uma amostra de dados através da interface *hardware/software* são fornecidos através dos números não negativos **ah_{comm}** (área de *hardware*), **as_{comm}** (área de *software*) e **t_{comm}** (tempo de comunicação).

As restrições que devem ser atendidas pelo algoritmo GCLP são:

- **T** = **latência** (tempo total disponível para execução da aplicação);
- **AH** = **capacidade** do recurso de *hardware* (área máxima para o ASIC);
- **AS** = **capacidade** do recurso de *software* (memória disponível).

Os resultados gerados pelo algoritmo são:

- **Mapeamento dos nós**: indicando implementação em *hardware* ou *software* para cada nó;
- **Escalonamento dos nós**: indicando o tempo inicial de execução para cada nó.

Os resultados do algoritmo GCLP devem atender às **restrições** do projeto, levando em conta os custos de comunicação, de forma que a área ocupada pelos nós mapeados para *hardware* seja mínima.

Cada **iteração** do algoritmo trata do **mapeamento** e **escalonamento** de um **nó**, de acordo com os valores **GC**, **LP** e **threshold**. Os valores **GC** e **threshold** são calculados a cada iteração; o valor **LP** é calculado previamente.

O cálculo da medida **GC** (*Global Criticality*) tem por finalidade indicar quão crítico o **tempo restante** para execução da aplicação está para os nós que ainda não foram escalonados. O valor de **GC** indica a **probabilidade** do nó a ser escalonado ser mapeado para *hardware*. Isto significa que, quanto maior for **GC**, mais restrito é o tempo para término de execução da aplicação, indicando que o nó em foco deve ser preferencialmente mapeado para *hardware*, o que dependerá da comparação com o valor **threshold**.

Em cada etapa do algoritmo, o **threshold** (**th**) é inicializado em 0,5. Este valor pode ser alterado, caso o algoritmo adote **características locais** dos nós (chamadas de **fase local**), quantificadas por um valor Δ . Este valor Δ é somado ao **threshold**, aumentando (caso Δ seja positivo) ou diminuindo (caso Δ seja negativo) seu valor inicial (0,5). Desta forma, o novo valor do **threshold** (**th'**) procurará **favorecer** a implementação do nó ou em *hardware* ou em *software*, com base nas características locais do nó. Há três casos a considerar:

Caso 1: $\Delta < 0$

Neste caso, $th' < 0,5$, indicando que as características locais do nó **favorecem** a sua implementação em *hardware*. **Para que isto ocorra**, a probabilidade do nó ser mapeado para *hardware* (GC) deve ser maior do que th' , para que a **função objetivo** escolhida seja a de **minimizar o tempo final**, o que favorecerá a escolha pela implementação em *hardware*. Isto é, a implementação em *hardware* é favorecida ao **diminuir-se** o valor inicial do *threshold*;

Caso 2: $\Delta > 0$

Neste caso, $th' > 0,5$, indicando que as características locais do nó **favorecem** a sua implementação em *software*. **Para que isto ocorra**, a probabilidade do nó ser mapeado para *hardware* (GC) deve ser menor do que th' , para que a **função objetivo** escolhida seja a de **minimizar a área em hardware**, o que favorecerá a escolha pela implementação em *software*. Isto é, a implementação em *software* é favorecida ao **eleva-se** o valor inicial do *threshold*.

Caso 3: $\Delta = 0$

Este caso ocorre quando o nó não é **nem** um **extremista** e **nem** um **repelente**, sendo classificado como nó **normal** (fase local 3). Desta forma, a probabilidade do nó ser mapeado para *hardware* (GC) é o valor que determina a escolha da **função objetivo**, a qual poderá ou **minimizar o tempo final** (mapeamento para *hardware*) ou **minimizar a área em hardware** (mapeamento para *software*).

As **características** dos módulos do sistema são incorporadas ao algoritmo GCLP nas etapas denominadas **fase local (LP, Local Phase)**. As características levadas em conta dizem respeito a otimizações locais (nós **extremistas**) e a preferências relativas (nós **repelentes**).

Um nó é **extremista em hardware** se ele consome muita área em *hardware* e pouco tempo de execução em *software*. Inversamente, um nó é **extremista em software** se ele consome muito tempo de execução em *software* e pouca área em *hardware*. Desta forma, um nó extremista em *software* reduzirá o valor do *threshold*, sinalizando uma maior possibilidade do nó vir a ser mapeado para *hardware*, dependendo do valor GC e da função objetivo. Já um nó extremista em *hardware*, aumentará o valor do *threshold*, indicando maior possibilidade de vir a ser mapeado para *software*, dependendo do valor GC e da função objetivo.

A **repelência** de nós é uma medida de **comparação** entre **nós similares**, que não sejam extremistas. Esta medida serve para indicar qual entre dois nós representaria um ganho maior caso fosse implementado em *hardware* (menor área de *hardware*) ou em *software* (menor tempo de execução). Esta comparação é feita com base no **valor de repelência (RV)** de cada nó. O valor de repelência de um nó i (RV_i) é computado a partir de uma combinação de valores atribuídos a **propriedades de repelência** para *hardware* e para *software*, em relação ao nó; estas propriedades estão relacionadas ao **subgrafo**, em **nível de instruções**, representante **da tarefa** associada ao nó i . Por exemplo, a porcentagem de instruções em nível de *bit* e o nível de precisão de instruções, são propriedades de repelência para uma implementação em *software*; já a porcentagem de instruções *lookup-table* e a porcentagem de instruções intensivas de memória, são exemplos de propriedades de repelência em *hardware*.

Detalhes do algoritmo GCLP não serão dados aqui. Pelo exposto em [34], o algoritmo parece bem completo, uma vez que leva em consideração tanto características locais quanto características globais das tarefas (nós) que compõem um sistema. Um dos pontos altos do algoritmo é o de **qualificar** o que é denominado como "**experiência do projetista**" na escolha de implementação de tarefas nitidamente direcionadas para *hardware* ou para *software* (nós extremistas); outro ponto alto é a **quantificação** de tarefas com características de implementação muito próximas, através do cálculo do **RV** para nós repelentes, o que possibilita comparar tarefas quanto ao ganho (em área de *hardware* ou em tempo de execução de *software*) que um determinado tipo de implementação produzirá. Em relação às características globais, o algoritmo automatiza a preocupação que um projetista deve ter quanto ao tempo final para encerramento da aplicação.

Outro algoritmo de particionamento é proposto por Bakshi e Gajski [33], o qual visa auxiliar no projeto de sistemas de processamento digital de sinais (DSP), tais como sistemas para processamento de imagens (decodificadores, compactadores). Este algoritmo deverá, dada uma especificação particionada *hardware/software* (**particionamento espacial**) e uma alocação de processadores (número e tipos), realizar as seguintes tarefas:

- **Mapear** cada um dos comportamentos (ou tarefas) de *software* nos processadores alocados;
- **Criar** estágios de *pipeline* e **mapear** cada comportamento a um destes estágios, de forma a explorar a concorrência entre os comportamentos;
- **Escalonar** os comportamentos em cada estágio de *pipeline*, entre os componentes de *hardware* e os processadores alocados, de forma a **satisfazer** uma **restrição** de *throughput* (taxa de processamento) a um **custo mínimo** de *hardware*.

Este conjunto de tarefas realizado pelo algoritmo denomina-se **particionamento temporal**. Inicialmente, vejamos a diferença entre particionamento espacial e temporal.

O **particionamento espacial** refere-se ao particionamento conforme é tratado na maioria das metodologias de projeto em nível de sistema, isto é, trata-se da **associação do tipo de implementação** (*hardware* ou *software*) para cada tarefa (conforme a granulosidade adotada) do sistema. Nesta associação, tarefas críticas são mapeadas para *hardware*, enquanto tarefas menos críticas (ou que exijam programabilidade) são mapeadas para componentes mais baratos ou mais lentos (geralmente, processadores de propósito geral, para rodar tarefas associadas a *software*).

O **particionamento temporal**, por sua vez, refere-se à **exploração de concorrência entre as tarefas**, de forma a atender uma dada **restrição de taxa de processamento** (*throughput*). Esta exploração de concorrência se dá através da **execução de tarefas em estágios de pipeline** distintos, isto é, busca-se analisar quais os ganhos obtidos (em termos de tempo) ao executar-se determinadas **tarefas em paralelo**, distribuindo-as em **diferentes estágios de pipeline**. Desta forma, o resultado do particionamento temporal é o **escalonamento** das tarefas (determinação do tempo inicial e final de execução de cada tarefa), além da **associação (mapeamento)** das tarefas de *software* aos processadores que as executarão. Assim, o particionamento temporal fornece informações bem mais completas do que o particionamento espacial, o qual fornece somente o tipo de implementação para cada tarefa (tipo *software* ou tipo *hardware*).

O grande diferencial do algoritmo de particionamento proposto por Bakshi e Gajski, está na exploração do **pipeline em nível de sistema**, necessário quando o **pipeline em nível de registradores** não é suficiente para atender as restrições de *throughput*.

O **pipeline em nível de registrador** procura explorar a concorrência entre os **subcomponentes** que compõem um componente de **hardware dedicado**. Esta exploração envolve a criação de estágios *pipeline* em vários níveis, tais como, *pipeline* em nível de instrução, *pipeline* em nível funcional (por exemplo, na ULA), entre outros. O *pipeline* em nível de registrador é capaz de gerar *hardware* dedicado com tempo de execução rápido; entretanto, um sistema formado por componentes otimizados de *hardware* dedicado (cada qual associado a uma tarefa) pode não ser capaz de atender as restrições de *throughput*. Neste ponto, a exploração de paralelismo na execução destas tarefas (e também das tarefas associadas a *software*) pode levar ao atendimento do *throughput*. Esta exploração é denominada de **pipeline em nível de sistema**, o qual busca a criação de **estágios de execução concorrente** (estágios de *pipeline*) envolvendo as **tarefas** do sistema.

As informações de entrada do algoritmo de particionamento temporal de Bakshi e Gajski são:

- Um **grafo de fluxo de controle CFG(V,E)**, seção 2.2.2, descrevendo o **particionamento espacial** da especificação da aplicação. Seus **nós** representam **tarefas**, cada qual **associada** a um **tipo de implementação** (ou *hardware* ou *software*). As **arestas** do grafo representam **dependências de controle entre tarefas**;
- Uma **alocação** de processadores **P** selecionada de uma biblioteca de processadores. Esta alocação é formada pela quantidade e tipos de processadores necessários para executar as tarefas associadas a *software*;
- Um **tempo de execução** T_v para cada **nó** v_h associado a *hardware* ($v_h \in V$);
- Um **tempo de execução** T_{vp} para cada **nó** v_s associado a *software* ($v_s \in V$), para cada processador selecionado **p** ($p \in P$);
- A restrição de *throughput* (taxa de processamento) **T**.

As informações geradas pelo algoritmo são:

- O **escalonamento e estágio pipeline** de cada **nó** v , representados através dos pares (x_v, y_v) e (x'_v, y_v) , $v \in V$, tal que $x'_v > x_v$. Os valores x_v e x'_v representam o **tempo de início** e de **término** de execução da tarefa v (este intervalo de tempo é denominado *time-slot*), enquanto o valor y_v representa o **número do estágio pipeline**;
- A **associação** de um **processador p** ($p \in P$) para cada **nó** v_s do tipo *software* ($v_s \in V$), tal que **p** executará v_s ;
- uma **lista de utilização** para cada processador $p \in P$, formada por pares (x_{p1}, x_{p2}) , indicando os **intervalos de tempo** nos quais o **processador é utilizado**.

A associação de comportamentos (tarefas) aos estágios de *pipeline* deve ser feita de forma a atender 2 condições:

- Satisfazer a restrição de *throughput* **T** usando a alocação de processadores **P**; e
- Reduzir o número de estágios de *pipeline* (para reduzir a latência do projeto e a memória exigida para armazenar dados intermediários entre os estágios de *pipeline*).

A seguir, são dadas as etapas do algoritmo de particionamento temporal:

1. Para cada nó em CFG, determine o tempo de término mais longo entre o nó selecionado e qualquer outro nó de saída (sem sucessor). Associe prioridade ao nó.
2. Inicialize a **lista_de_utilização** de todos os processadores.
3. Forme uma nova **lista_de_nós_prontos** (nós não escalonados).
4. **Loop**
5. **nó_atual** = primeiro nó na **lista_de_nós_prontos**
6. **If** (**nó_atual** for do tipo *software*)
7. Encontre um processador e o *time slot* correspondente, que dê o tempo de término mais breve.
8. **If** (nenhum processador disponível)
9. **Exit** /* nenhuma solução viável */
10. **Else**
11. Associe **nó_atual** ao processador e *time-slot*.
12. Atualize a **lista_de_utilização** do processador.
13. **End If**
14. **Else If** (**nó_atual** for do tipo *hardware*)
15. Associe **nó_atual** ao *time-slot* mais breve possível.
16. **End If**
17. Marcar **nó_atual** como escalonado, removê-lo da **lista_de_nós_prontos** e atualizar **lista_de_nós_prontos**.
18. **Until** (todos os nós no CFG estarem escalonados).

O algoritmo começa com a determinação do tempo de término mais longo entre cada nó e qualquer outro nó de saída (sem sucessor), assumindo que o processador mais rápido é utilizado para executar um nó de *software* (passo 1). O tempo calculado dá a prioridade que um nó tem sobre os demais, durante o escalonamento; trata-se de uma indicação direta de sua criticalidade e, assim, quanto maior o tempo de término de um nó, maior será sua prioridade.

Cada processador possui uma lista de utilização, na qual são armazenados os tempos em que o processador estará ocupado, executando alguma tarefa em *software*, em algum estágio de *pipeline*. Um processador poderá executar mais de uma tarefa, desde, é claro, que não haja uma superposição temporal na execução das mesmas. No passo 2, a lista de utilização de cada processador é inicializada para vazio. A lista de nós prontos é formada pelos nós cujos predecessores já tenham sido escalonados. No passo 3, a lista de nós prontos é inicializada em ordem decrescente de prioridade entre os nós.

Na sequência, dentro da parte iterativa do algoritmo (5 a 17), é encontrado o melhor *time-slot* para cada nó na lista de prontos, a partir do primeiro nó. Para nós associados a *software*, o melhor *time-slot* dependerá do processador escolhido para implementá-lo. Após a associação do *time-slot* (e do processador, no caso de nó *software*) ao nó, este será marcado como **escalonado**, sendo removido da lista de nós prontos. Esta lista é atualizada, checando se algum dos sucessores desse nó está pronto para ser escalonado; caso afirmativo, o nó é adicionado à lista. Os passos de 5 a 17 são repetidos até o escalonamento de todos os nós.

A determinação do melhor *time-slot* é feita de acordo com a descrição a seguir.

Seja **max_estágio** o maior número de **estágio de pipeline** associado aos predecessores imediatos de **nó_atual**. Seja **max_pred_delay** o **tempo final** de execução mais alto dos predecessores de **nó_atual** presentes no estágio **max_estágio** (isto é, o tempo final do predecessor que terminou por último, executando no estágio **max_estágio**). Os valores **max_estágio** e **max_pred_delay** correspondem, respectivamente, aos limites inferiores de número do estágio de *pipeline* e de tempo de execução inicial neste estágio, para escalonamento de **nó_atual**.

Após a determinação de **max_estágio** e **max_pred_delay**, verifica-se se **nó_atual** é do tipo *hardware* ou do tipo *software*. Caso seja do tipo *hardware*, **nó_atual** é associado ao estágio **max_estágio** se o seu tempo de execução (iniciado após **max_pred_delay**) não extrapolar o valor do *throughput*, o qual determina o tempo máximo para execução de comportamentos em um estágio de *pipeline*; caso extrapole o valor do *throughput*, **nó_atual** é associado para **max_estágio + 1**.

Caso **nó_atual** seja do tipo *software*, verificam-se as listas de utilização de todos os processadores, para encontrar o tempo de término mais breve para a execução de **nó_atual**, entre as alternativas disponíveis de intervalos de tempo em que os processadores encontram-se ociosos. Neste ponto, há duas possibilidades, sendo escolhida aquela que fornecer o tempo de término mais breve. Estas possibilidades são:

- A execução de **nó_atual** pode começar após o tempo **max_pred_delay** em **max_estágio**;
- A execução de **nó_atual** pode começar no tempo 0 em **max_estágio + 1**.

Para ilustrar o algoritmo de particionamento temporal, considere a atuação do mesmo no seguinte exemplo: o CFG da figura 5.2 representa o particionamento espacial de um sistema. Os nós **A**, **B**, **C**, **D** e **E** devem ser implementados em *software*, enquanto **Hw1** e **Hw2** devem ser implementados em *hardware*. A duração de cada estágio (*throughput*) é $T=12$ ns. A **alocação** é formada por 2 processadores do tipo **P1** e 1 processador do tipo **P2**. Os tempos de execução das tarefas do tipo *software* nos processadores são fornecidos na tabela 5.1. Os tempos de execução dos módulos **Hw1** e **Hw2** são, respectivamente, 9 ns e 5 ns.

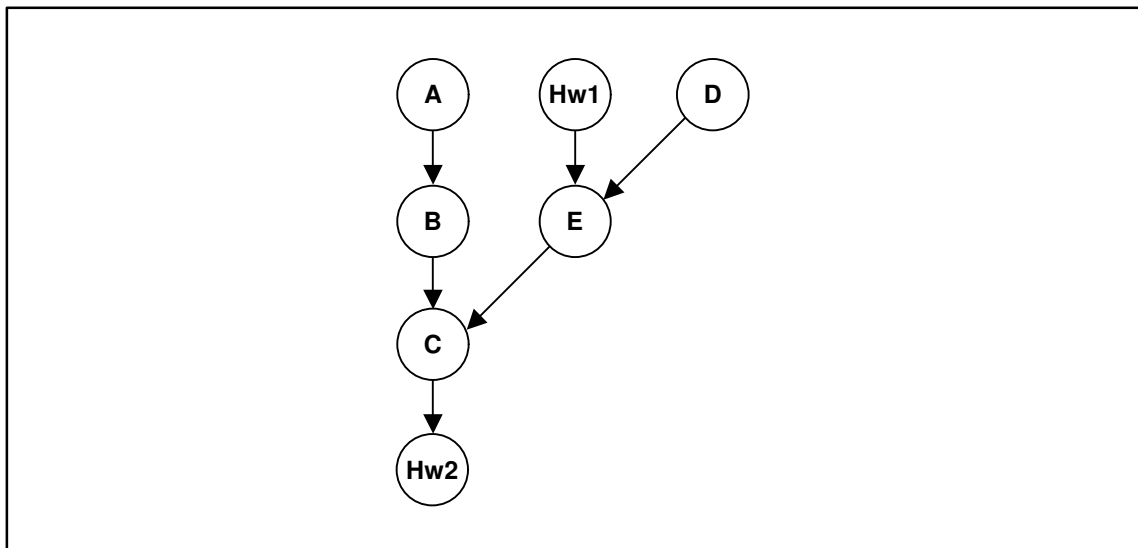


Figura 5.2. CFG do particionamento espacial do sistema.

Tabela 5.1. Tempos de execução nos processadores.

Tarefa	Processador	Tempo de Execução (ns)
A	P1	4
A	P2	6
B	P1	7
B	P2	10
C	P1	9
C	P2	5
D	P1	10
D	P2	7
E	P1	8

Inicialmente, os nós foram ordenados de acordo com o tempo de término mais longo, adotando-se os tempos de execução mais velozes para os comportamentos em *software*. Assim, tem-se os seguintes tempos (em ns):

$$\mathbf{A: 4+7+5+5 = 21}$$

$$\mathbf{B: 7+5+5 = 17}$$

$$\mathbf{C: 5+5 = 10}$$

$$\mathbf{D: 7+8+5+5 = 25}$$

$$\mathbf{E: 8+5+5 = 18}$$

$$\mathbf{Hw1: 9+8+5+5 = 27}$$

$$\mathbf{Hw2: 5}$$

Ordenando os resultados acima, do maior para o menor, formou-se uma lista de prioridades, a qual corresponde à lista inicial de nós prontos. Esta lista não foi alterada ao longo dos passos do algoritmo (exceto, é claro, da retirada do primeiro elemento da lista após cada escalonamento):

lista_de_nós_prontos: (Hw1, D, A, E, B, C, Hw2)

Percorrendo a **lista_de_nós_prontos** acima, o algoritmo gerou os resultados mostrados na figura 5.3. Observa-se, pelas listas de utilização e pelo escalonamento (itens *a* e *b* da figura), que:

- Um dos processadores do tipo **P1** (**P1₁**) executa o comportamento **A** entre os tempos 0 e 4 ns e o comportamento **B** entre os tempos 4 e 11 ns, ambos no estágio 1 do *pipeline*;
- O outro processador do tipo **P1** (**P1₂**) executa o comportamento **E** entre os tempos 0 e 8 ns, no estágio 2 do *pipeline*;
- O processador **P2** executa o comportamento **D** entre os tempos 0 e 7 ns, no estágio 1, e o comportamento **C** entre os tempos 7 e 12 ns, no estágio 3 do *pipeline*;
- O módulo de *hardware* **Hw1** é executado entre os tempos 0 e 9 ns do estágio 1, enquanto **Hw2** foi escalonado para executar entre os tempos 0 e 5 ns do estágio 4.

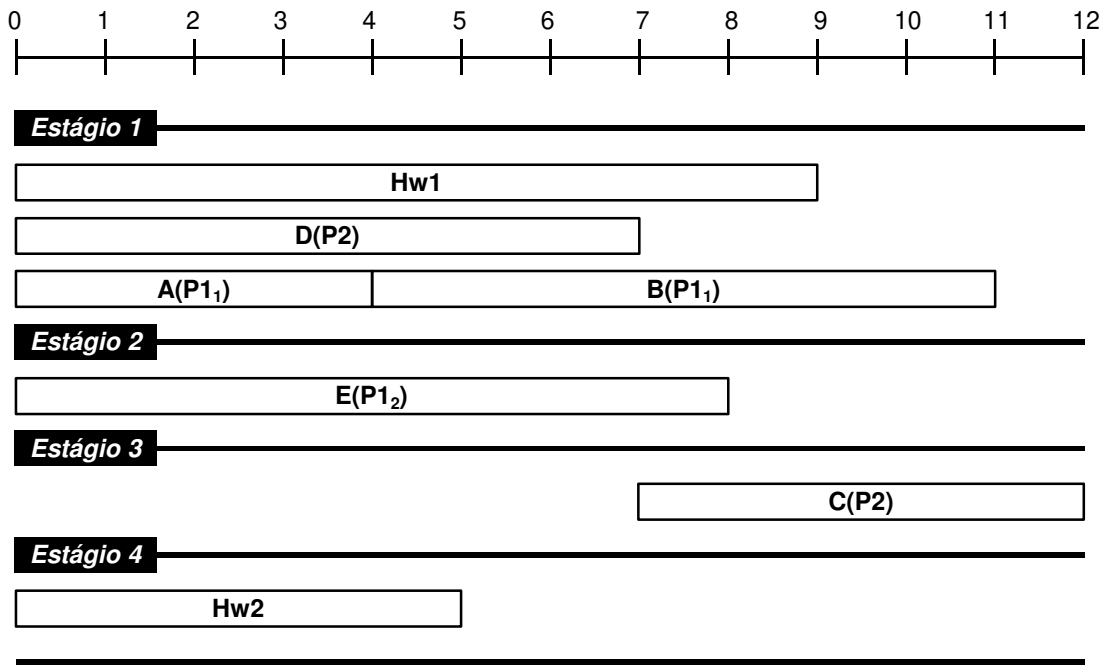
A figura 5.3.c mostra o CFG da figura 5.2 após o particionamento temporal.

Bakshi e Gajski destacam alguns pontos do algoritmo que precisam ser aperfeiçoados, como, por exemplo, a necessidade de inclusão do custo de comunicação entre as tarefas (custo em termos de área e de atraso de comunicação).

a) Listas de utilização dos processadores

$P1_1$: início	0	4	$P1_2$: início	0	$P2$: início	0	7
fim	4	11	fim	8	fim	7	12

b) Escalonamento e estágios *pipeline*



c) Representação gráfica do mapeamento e dos estágios de *pipeline*

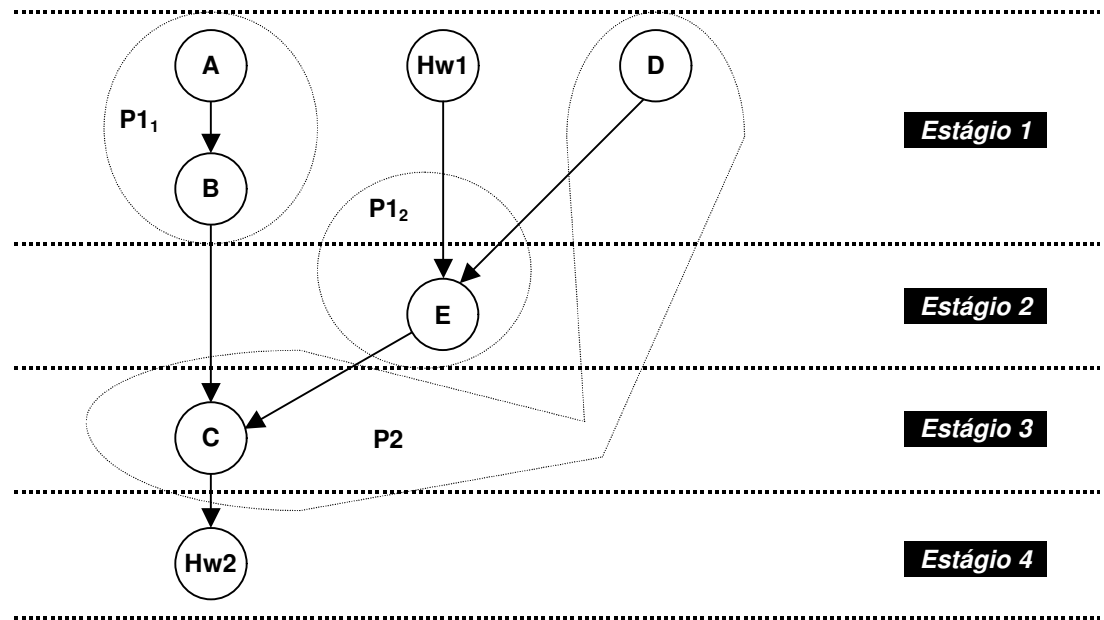


Figura 5.3. Resultados gerados pelo algoritmo de particionamento temporal.

5.5. TÉCNICAS DE VERIFICAÇÃO

A **verificação (validação)** de um sistema embutido fornece uma razoável certeza de que o seu projeto (ou até certo ponto do processo de projeto) está imune de erros ou atende a determinadas restrições de desempenho, podendo ser fabricado (ou prosseguido, no caso de uma etapa intermediária do processo de projeto). Conforme os sistemas tornam-se cada vez mais complexos, a verificação torna-se mais importante, assim como, mais difícil. Os projetistas utilizam uma ou mais das seguintes técnicas para validar um sistema ou parte dele: **simulação, prototipação e verificação formal**.

Simulação é o processo de aplicar estímulos (valores de entrada) a um modelo e produzir respostas (valores de saída) de acordo com este modelo [4].

Projetistas podem usar simuladores RTL para modelos em *hardware* nas linguagens VHDL e Verilog. Similarmente, modelos RTL para processadores de uso genérico podem ser desenvolvidos nestas linguagens. Desta forma, visando propósitos de simulação, pode-se modelar um sistema misto em uma HDL. Porém, simulação RTL é muito lenta na análise de sistemas que executam *software* de razoável complexidade [1].

Visando verificar a eficiência de implementações de sistemas *hardware/software*, Gupta *et al.* [35] utilizam o simulador **Poseidon**, o qual faz a simulação dos elementos de *hardware* e código *assembly (software)* gerados automaticamente a partir de grafos particionados. As entradas para o Poseidon consistem em descrições de modelos, interconexões e interfaces correspondentes. Cada modelo tem um sinal de *clock* associado e um ciclo de *clock* especificado para a sua simulação. A interface é especificada usando componentes como barramentos, filas, registradores, memórias e os protocolos associados.

A dificuldade na simulação de sistemas *hardware/software* está na natureza **heterogênea** dos mesmos. A metodologia Ptolemy (veja seção 5.2) trata do problema da simulação a partir da especificação inicial do sistema. O usuário pode modelar um sistema misto com diferentes paradigmas, incluindo modelos definidos pelo próprio usuário. O Ptolemy fornece uma plataforma de simulação na qual a descrição em diferentes domínios pode ser simulada e interfaceada de forma consistente, provendo meios eficientes para a validação de sistemas *hardware/software*.

A metodologia de coprojeto *hardware/software* em desenvolvimento na **Carnegie Mellon University** [28] permite a simulação tanto das especificações iniciais, como das descrições particionadas e das descrições sintetizadas, permitindo ao projetista verificar os subsistemas de *hardware* e de *software* ao longo do processo de projeto. O Sistema *hardware/software* completo é dividido em dois processos (um de *hardware* e outro de *software*), descritos em linguagens diferentes, que executam em paralelo e interagem entre si através de um mecanismo de comunicação por passagem de mensagens. Neste ponto, uma parte importante do simulador apresentado foi o desenvolvimento de primitivas de comunicação que permitem a troca de mensagens entre as representações dos módulos de *software* (descrito em C) e *hardware* (descrito em Verilog). A simulação do *hardware* e o programa (*software*) são tratados como dois processos **Unix** que se comunicam através de **sockets BSD** (*Berkeley Software Distribution*).

A especificação em alto nível do sistema é uma descrição mista *hardware/software*. Esta descrição inicial é reparticionada de acordo com um determinado conjunto de restrições e

critérios de otimização, para produzir partições *hardware/software* melhores. Assim, parte do que é descrito inicialmente em *hardware* pode terminar sendo executada em *software* e vice-versa. A nova partição de *hardware* é submetida a ferramentas de síntese para obter uma descrição em nível de portas. O simulador é uma parte importante nessa metodologia de coprojeto, uma vez que a simulação conjunta dos processos de *hardware* e de *software* fornece uma verificação funcional dos dois subsistemas executando simultaneamente.

Uma das vantagens propagadas pelos pesquisadores da *Carnegie Mellon University* está na afirmação de que o seu simulador é capaz de manipular vários níveis de abstração e diferentes descrições para os módulos de *hardware* e de *software*. Eles consideram a existência de duas estratégias para descrição de um projeto:

- a) Descrições distintas para *software* e *hardware*;
- b) Descrição única para *software* e *hardware*.

A estratégia com descrições distintas para *software* e *hardware* apresenta como vantagens a possibilidade de uma melhor exploração das características associadas a cada tipo de descrição, além de poder ser aplicada a mais de um nível de abstração. A desvantagem desta estratégia é a necessidade de prover um mecanismo de comunicação adequado e garantir um "casamento semântico" eficiente (entre primitivas e comandos de comunicação, tais como *send*, *receive*, *read*, *write*, *open*).

A estratégia com uma única descrição para *software* e *hardware*, como, por exemplo, o modelo CFSM do ambiente POLIS, apresenta como vantagem possibilitar a homogeneidade na modelagem de todo o sistema, através da generalização dos componentes de *software* e de *hardware*. A desvantagem desta estratégia é que a descrição fica restrita a um determinado nível de abstração, não permitindo maiores detalhamentos posteriores.

A simulação consiste na forma mais tradicional de validar a correção de um sistema [1]. Entretanto, para sistemas complexos, um número limitado de padrões de entrada/saída pode ser examinado, não permitindo assegurar a funcionalidade do sistema. Apesar disso, conforme exposto nos exemplos anteriores, ferramentas de simulação são muito úteis no coprojeto *hardware/software*.

Um **protótipo** é um exemplar ou modelo preliminar de um produto, componente ou sistema. Geralmente abstrato, falta-lhe alguns detalhes da versão final. Protótipos podem ser utilizados para propósitos de teste, demonstração de soluções, exploração de alternativas de implementação e validação de determinadas características de sistemas digitais. Duas classes de protótipos são utilizadas em processos de projeto, quais sejam: **protótipos físicos** e **protótipos virtuais** [4].

Protótipo físico é um exemplar físico de um produto, componente ou sistema. Sua construção geralmente envolve projeto detalhado e fabricação de uma placa ou de um ou mais circuitos integrados, exigindo um gasto de tempo que pode variar de semanas a meses, fora o tempo gasto em eventuais modificações.

Protótipo virtual é um modelo de simulação em computador de um produto final, componente ou sistema. Este termo não se refere a um modelo em particular, mas, ao invés disso, se refere ao papel do modelo dentro de um processo de projeto, papel este relacionado com a exploração de alternativas de projeto, demonstração de conceitos de projeto e teste de requisitos de um projeto. Protótipos virtuais podem ser construídos em qualquer nível de

abstração, podendo incluir uma mistura de diferentes níveis. Em contraste ao protótipo físico, o qual exige um projeto detalhado de *hardware* e de *software*, um protótipo virtual pode ser configurado mais rapidamente e com custo menor.

Apesar de fornecer dados mais precisos sobre confiabilidade e desempenho de um sistema, quando comparada à simulação em nível de sistema, a prototipação é dispendiosa em termos de tempo de projeto e, além disso, não pode ser realizada até que uma grande parte do projeto tenha sido feita.

Verificação formal é um conjunto de técnicas que permitem provar matematicamente se algumas propriedades (como, por exemplo, a ausência de situações de *deadlock*), especificadas formalmente, são verdadeiras para um projeto.

A verificação formal apresenta como principal vantagem, sobre a simulação e a prototipação, a possibilidade de obter, nas fases iniciais do projeto, resultados a respeito da correção do mesmo. Entretanto, técnicas deste tipo exigem modelos formais do comportamento de um sistema, assim como para as propriedades que se deseja verificar [30]. Foi visto, na seção 5.3, que o modelo formal CFSM, utilizado na metodologia POLIS, permite a verificação formal de um sistema misto, através da verificação da congruência da linguagem representativa do comportamento do sistema com a linguagem aceitável para o sistema.

Bawa e Encrenaz [36] classificam os métodos de **verificação formal** para descrições de **hardware síncrono** em três diferentes categorias, quais sejam:

- a) Verificação com **prova de teorema**;
- b) Verificação usando-se **cálculos especiais**;
- c) Verificação de estados finitos através da **enumeração dos estados**.

No primeiro método relacionado acima, o usuário escreve teoremas que devam ser provados através de um assistente de provas. No segundo método, cálculos especiais são utilizados para verificar determinadas propriedades de circuitos descritos num nível de abstração superior ao nível lógico. O terceiro método é baseado na exploração do espaço de estados simbólico a partir do produto de FSMs.

Bawa e Encrenaz propuseram um método formal, baseado na enumeração de estados, para verificar a equivalência de duas FSMs descritas em VHDL. Esta verificação é feita automaticamente com uma ferramenta denominada **PSM** (*Proof States Machines*), através do percorrimento simbólico de uma relação de transição particionada, derivada de duas descrições em VHDL.

O principal problema relacionado à verificação formal é que ela encontra-se restrita às áreas onde a regularidade e abstração podem ser utilizadas. Desta forma, o seu uso na verificação completa de sistemas *hardware/software* complexos ainda está distante.

PARTE II – REDES DE PETRI

CAPÍTULO 6: REDES DE PETRI

6.1. INTRODUÇÃO

Rede de Petri foi criada a partir da tese de doutorado do pesquisador alemão **Carl Adam Petri**, intitulada "*Kommunikation mit Automaten*" ("*Comunicação com Autômatos*"). Desde o princípio, rede de Petri e suas **extensões** (simplificadamente, **redes de Petri – RP**) objetivaram a modelagem de sistemas com componentes concorrentes.

Segundo Heuser [37], as primeiras aplicações de RP aconteceram em 1968, no projeto norte-americano *Information System Theory*, da ADR (*Applied Data Research, Inc.*). Muito da teoria inicial, da notação e da representação de RP foi desenvolvido neste projeto e foi publicado no seu relatório final. Este trabalho ressaltou como RP poderiam ser aplicadas na análise e modelagem de sistemas com **componentes concorrentes**.

A década de 1970 marcou o desenvolvimento da **Teoria de RP** e a expansão do **campo de aplicação de RP**. No início daquela década, o trabalho de Petri chamou a atenção do *Projeto MAC*, do MIT (*Massachusetts Institute of Technology*). O *Grupo de Estruturas Computacionais*, sob a direção de **Jack B. Dennis**, foi a origem de consideráveis pesquisas e publicações sobre RP, envolvendo relatórios e teses de doutorado. Duas importantes conferências foram organizadas pelo grupo: a *Conferência Sobre Sistemas Concorrentes e Computação Paralela*, em Woods Hole (*Projeto MAC*, 1970), e a *Conferência Sobre Redes de Petri e Métodos Relacionados*, no MIT (1975). Os resultados destes e de outros esforços no desenvolvimento da Teoria de RP estão registrados em inúmeros artigos e em dois livros: no de **Reisig** [38], que representa a considerada linha européia deste desenvolvimento (mais teórica), e no de **Peterson** [39], representando a linha norte-americana (mais prática). Em relação às aplicações, RP atingiu áreas como a **modelagem de componentes de hardware**, **controle de processos**, **linguagens de programação**, **sistemas distribuídos** e **protocolos de comunicação**. Ainda na década de 1970, surgiram três tipos de RP capazes de modelar diferentes características **temporais**, quais sejam, as RP Temporizadas de **Ramchandani**, as de **Merlin** e as de **Sifakis**.

As aplicações de RP aumentaram consideravelmente na década de 1980, com o surgimento das chamadas **Redes de Petri de alto nível** (por exemplo, as **Coloridas** e as **Numéricas**). Tais redes acrescentaram uma grande força descritiva ao processo de modelagem, através do uso de **tokens com identidade** e, conseqüentemente, do uso de **conjuntos de tokens**. Desta forma, RP atingiram outras áreas, como **automação de escritórios**, **bancos de dados**, **inteligência artificial** e **sistemas de informação**.

A **consolidação** de RP se deu na década de 1990, fomentada pela segunda versão das redes Coloridas, desenvolvidas por **Kurt Jensen**, da *Århus Universitet* (Dinamarca). Além de trabalharem com **tokens** diferenciáveis, tais redes apresentam tratamento de aspectos temporais. Ainda naquela década, em relação a **sistemas digitais**, vários pesquisadores demonstraram a eficiência e **potencial** de RP na modelagem de **sistemas de eventos discretos**. Este fato deve-se, em grande parte, à existência de um conjunto de **técnicas** para **análise estrutural** e **dinâmica** de RP, que permitem a **validação formal** de importantes **propriedades** de um modelo: vivacidade, segurança, inexistência de conflitos, dentre outras.

Na sequência deste texto, a **composição básica** de RP é dada na seção 6.2. As RP **Elementares** (ou **clássicas**) são apresentadas na seção 6.3. Por se tratarem, matematicamente, de um subconjunto das RP **Lugar/Transição**, sua formalização matemática não é aprofundada, uma vez que tal formalização é dada no capítulo 7, onde as RP Lugar/Transição são descritas. Esta formalização é importante, uma vez que outras extensões de RP utilizam as redes Lugar/Transição como base. Diferentes tipos de RP **Temporizadas** são apresentados no capítulo 8, com ênfase para o modelo proposto por *Merlin*, uma vez que os modelos de *Sifakis* e de *Ramchandani* são considerados como um subconjunto do primeiro. O capítulo 9 encerra esta parte da tese, descrevendo a principal representante das chamadas redes de alto nível, as RP **Coloridas**.

As redes descritas nesta parte da tese são utilizadas como base para outras extensões, tratadas na parte III. Desta forma, não nos preocupamos, neste momento, em direcionar a apresentação das características (e exemplos) destes modelos para a área de sistemas digitais, o que será feito na parte III.

6.2. COMPOSIÇÃO BÁSICA DE REDES DE PETRI

A seguir, é fornecida a composição básica de uma RP sem considerar, por enquanto, qualquer interpretação ou regra de funcionamento dada à mesma.

Uma RP é representada por um grafo (veremos adiante que pode ser representada matematicamente por meio de matrizes), sendo composta de elementos de dois tipos: **lugares**, indicados por nós elípticos ou circulares, e **transições** entre lugares, indicadas por nós retangulares ou em forma de barra. Ramos orientados ligam lugares às transições ou vice-versa, nunca entre elementos do mesmo conjunto. Inexistem nós lugares ou nós transições isolados. Portanto, de acordo com o exposto, uma RP é um grafo orientado bipartido conexo.

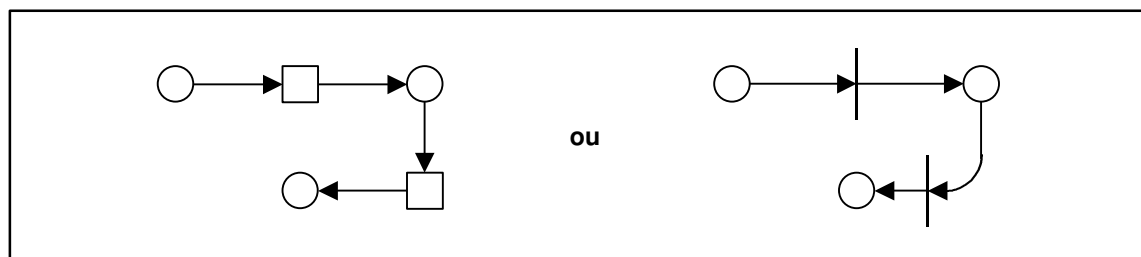


Figura 6.1. Redes de Petri equivalentes: 2 lugares e 2 transições.

Em relação a uma transição, os ramos são classificados em **ramos de entrada**, indicados pela seta entrando no retângulo, e **ramos de saída**, indicados pela seta saindo do retângulo. Analogamente, classificam-se os **lugares** como sendo de **entrada** ou de **saída**. Um lugar não pode estar ligado a uma transição mais que uma vez através de um mesmo tipo de ramo.

De acordo com o tipo da rede, lugares e transições recebem **conotações** diferentes. Na maioria das interpretações de redes, os **lugares** representam elementos do tipo **estado** e as **transições** representam elementos do tipo **ação**.

Esta é a composição básica de uma RP. Na sequência deste texto, serão fornecidos maiores detalhes, de acordo com o tipo de RP estudada.

6.3. REDES DE PETRI ELEMENTARES

Conforme descrito anteriormente, os diversos tipos de RP apresentam **diferenças** em sua **composição** e **regras de funcionamento**. Iniciaremos nossa discussão sobre RP apresentando as características de sua forma clássica, qual seja, a **RP Elementar (RPE)**.

Cada lugar de uma RPE define um estado local ou **condição**, que pode ou não estar vigorando. Gráficamente, um estado local ou condição está vigorando quando o lugar que a define possui uma **marca** (ou *token*, ou ainda, **senha**, indicada por ●), e não está vigorando caso contrário. Assim, dizemos que um lugar está ou não **marcado** (está ou não com um *token*). Denomina-se **marcação da rede** o conjunto de todos os lugares marcados. Enquanto uma marca modela um **estado local** do sistema, uma marcação modela um **estado global**.

Cada transição de uma RPE define um **evento**, que pode ocorrer frente a determinadas marcações e cuja ocorrência tem por efeito o aparecimento e coincidente desaparecimento de marcas. A **ocorrência de um evento** (também conhecida por **disparo de uma transição**) depende somente dos lugares de entrada da respectiva transição e afeta somente os lugares de saída da transição.

Peterson [39] apresenta a seguinte representação matemática para RPE:

Definição 6.1: Representação Matemática de RPE.

Uma RPE pode ser representada pela quintupla $(L, T, \alpha, \beta, M_0)$, onde:

- **L** é o **conjunto de lugares** da rede;
- **T** é o **conjunto de transições** da rede;
- $\alpha: L \times T \rightarrow \{0, 1\}$ é a chamada **função de incidência direta**
 $\alpha(l, t) = 0$, se **l** não é lugar de entrada em relação a **t**;
 $\alpha(l, t) = 1$, se **l** é lugar de entrada em relação a **t**;
- $\beta: L \times T \rightarrow \{0, 1\}$ é a chamada **função de incidência reversa**
 $\beta(l, t) = 0$, se **l** não é lugar de saída em relação a **t**;
 $\beta(l, t) = 1$, se **l** é lugar de saída em relação a **t**;
- $M_0: L \rightarrow \{0, 1\}$ é a chamada **função marcação inicial**
 $M_0(l) = 0$, se o lugar **l** não está inicialmente marcado;
 $M_0(l) = 1$, se o lugar **l** está inicialmente marcado.

Em relação a uma determinada marcação, a possibilidade de ocorrência de um evento segue a regra definida a seguir.

Definição 6.2: Regra de Habilitação.

Uma transição está habilitada em relação a uma dada marcação, quando:

- a) Seus lugares de entrada, caso a transição os tenha, estão marcados; e
- b) Seus lugares de saída, caso a transição os tenha, não estão marcados.

A figura 6.2 apresenta três marcações distintas para uma mesma RP. Na marcação 6.2.a a transição está habilitada, uma vez que todos os seus lugares de entrada estão marcados e seus lugares de saída não estão marcados. A transição não está habilitada na marcação 6.2.b pela falta de um *token* num dos lugares de entrada. A transição também não está habilitada na marcação 6.2.c, devido à presença de um *token* num dos lugares de saída.

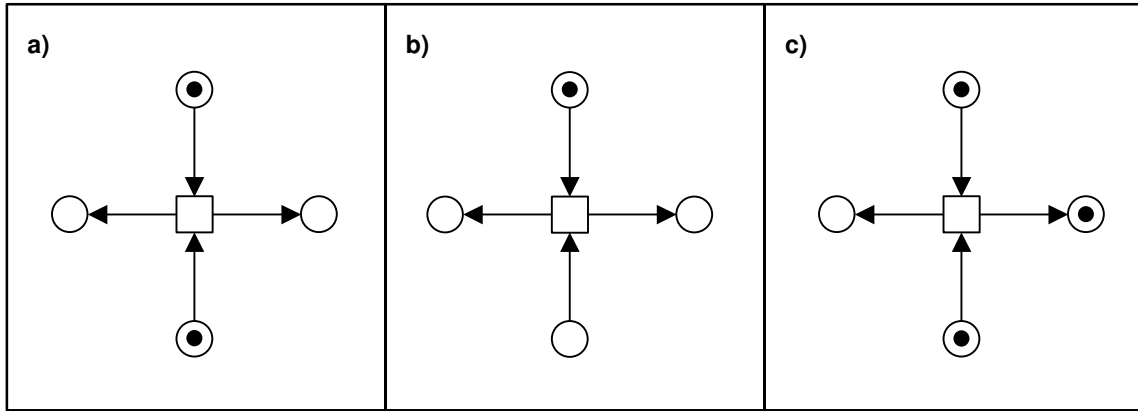


Figura 6.2. Marcações distintas para uma RPE.

Definição 6.3: Efeito do Disparo de uma Transição.

O efeito do disparo de uma transição é a passagem de uma marcação **precursora** para uma marcação **sucessora** de tal forma que:

- a) As marcas dos lugares de entrada da transição disparada desapareçam na marcação precursora; e
- b) É adicionada, na marcação sucessora, uma marca a cada lugar de saída da transição disparada.

A partir desse ponto, é possível compreender o comportamento de uma rede a partir de sua **marcação inicial**, a qual é dada pela função marcação inicial M_0 , representada graficamente pela presença e ausência de *tokens* na rede.

Definição 6.4: Marcação Alcançável (acessível).

Dada uma **marcação inicial** da rede, define-se **marcação alcançável** como sendo qualquer marcação obtida pelo efeito da ocorrência de um ou mais disparos sobre a marcação inicial ou sobre qualquer marcação alcançável.

Para representar graficamente as marcações alcançáveis e os disparos das transições responsáveis pelo surgimento das mesmas, utiliza-se a chamada **máquina de senhas**, definida a seguir.

Definição 6.5: Máquina de Senhas.

Máquina de senhas é um grafo que mostra todas as marcações alcançáveis numa RP, a partir de uma dada marcação inicial. Nas RPE, cada **nó** deste grafo representa uma marcação acessível e cada **arco direcionado** representa o disparo da transição que faz a rede passar de uma marcação (a marcação ligada na **origem** do arco) à outra (a marcação ligada no **destino** do arco).

6.4. EXEMPLO DE MODELAGEM DE UMA BIBLIOTECA EM RPE

Neste, e no próximo capítulo, será utilizado um exemplo padrão com a finalidade de destacar as diferenças existentes entre as RP Elementares e as RP Lugar/Transição. Tal exemplo consiste na modelagem da organização de uma biblioteca quanto ao empréstimo e à devolução de livros. O usuário pode acessar a biblioteca através de 3 balcões de atendimento, quais sejam: o balcão de **pedidos**, o de **retirada** e o de **devolução**. Os livros encontram-se guardados nas prateleiras, e cada livro possui uma **ficha**. Para emprestar um livro, o usuário entrega seu pedido de livro no balcão de pedidos. Se o livro estiver na biblioteca, o seu cartão é atualizado e o livro passa para o balcão de retirada. Quando o usuário devolve um livro, ele o faz utilizando-se do balcão de devolução.

Inicialmente, será modelado o funcionamento da biblioteca para apenas **um usuário** e **um livro**, conforme a rede da figura 6.3. Conforme dito anteriormente, a marcação inicial da rede é a denotada no gráfico da mesma. Assim, a RPE da figura 6.3 tem como marcação inicial os lugares **UsuárioSemLivro** e **LivroNaPrateleira** marcados, representando que o usuário está sem livro e que o livro encontra-se na prateleira.

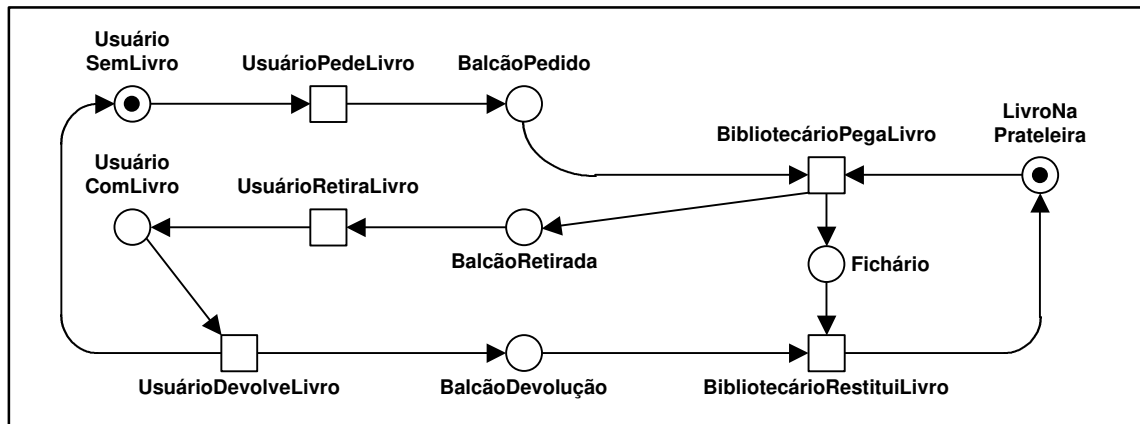


Figura 6.3. Rede de Petri Elementar modelando uma biblioteca com 1 livro.

Inicialmente, a única ação possível é o usuário pedir o livro, o que é feito disparando-se a transição **UsuárioPedeLivro** (observe que todas as demais transições encontram-se desabilitadas). Após o disparo dessa transição, a rede encontra-se na marcação apresentada na figura 6.4.

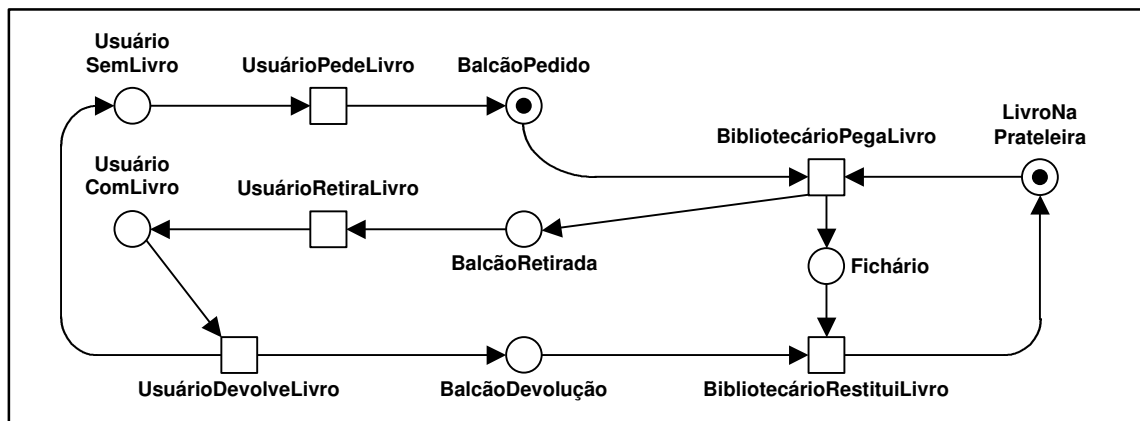


Figura 6.4. Marcação decorrente da inicial após o disparo de UsuárioPedeLivro.

Na marcação presente na rede da figura 6.4, a única transição habilitada é **BibliotecárioPegaLivro**, a qual representa, no sistema modelado, a retirada do livro da prateleira, a atualização de sua ficha e a entrega do mesmo no balcão de retirada.

Pode-se visualizar melhor o comportamento da rede através de sua máquina de senhas, dada na figura 6.5. Nela, vê-se as possíveis marcações da rede e a relação entre elas, ou seja, qual transição cujo disparo leva o modelo de uma marcação à outra.

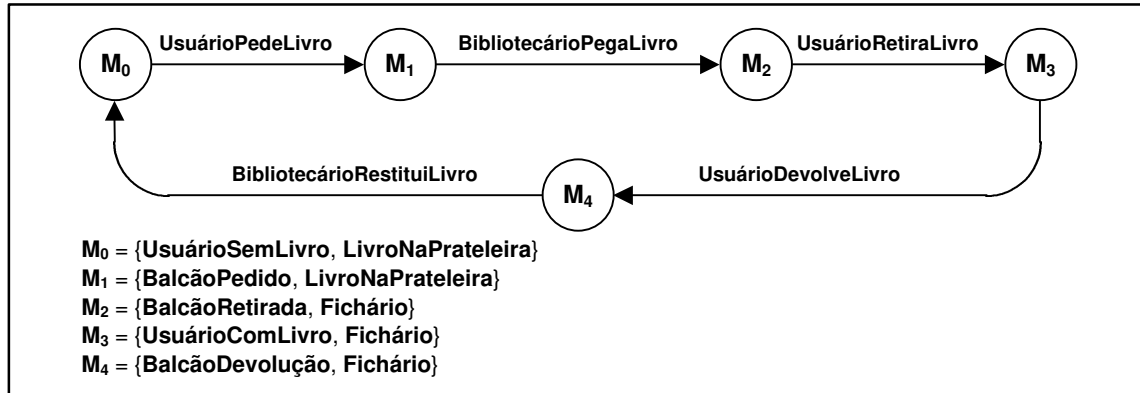


Figura 6.5. Máquina de senhas da RPE da figura 6.3.

Certamente, não se deseja modelar uma biblioteca que possua apenas um livro. A RPE da figura 6.6 modela uma biblioteca com 2 livros. Observe, nesta figura, a necessidade de duplicar-se a rede da figura 6.3. Portanto, RPE mostram-se inadequadas para o tratamento de sistemas onde o aspecto quantitativo seja importante. O próximo capítulo apresenta uma melhoria do modelo representativo da biblioteca, por meio das RP Lugar/Transição.

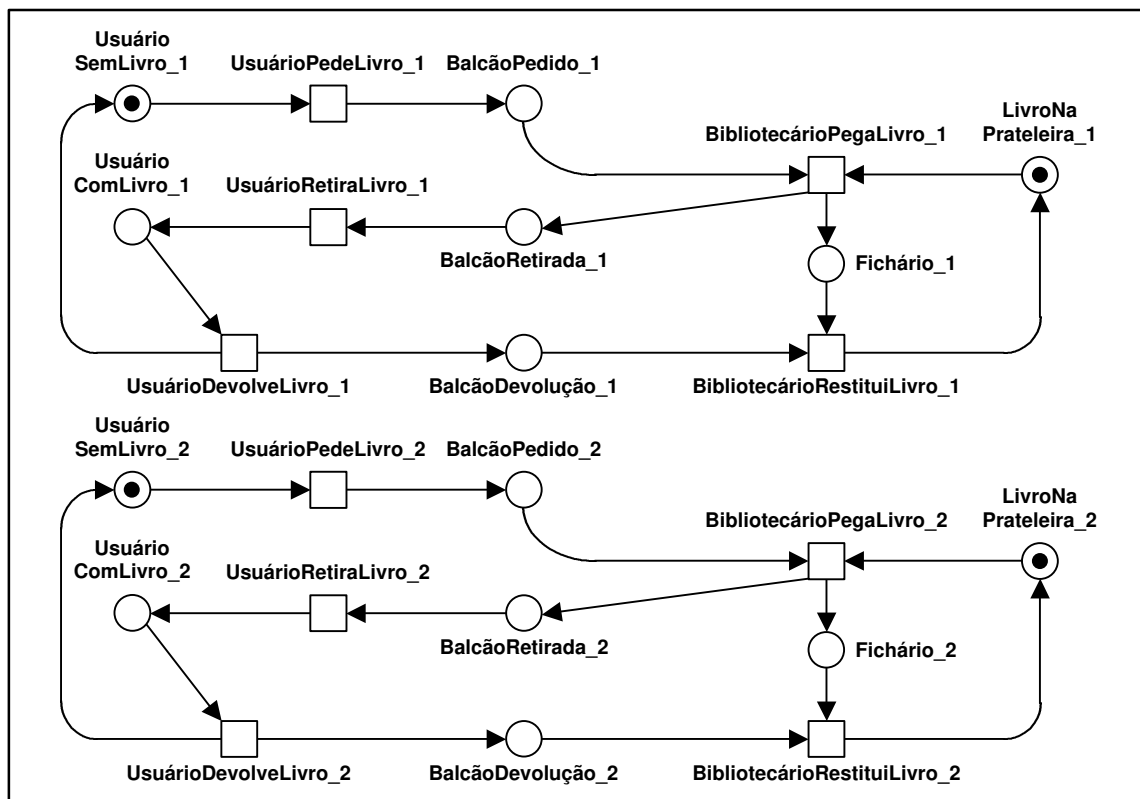


Figura 6.6. Rede de Petri Elementar modelando uma biblioteca com 2 livros.

CAPÍTULO 7: REDES DE PETRI LUGAR/TRANSIÇÃO

7.1. REGRAS DE FUNCIONAMENTO

Os elementos que compõem uma rede Lugar/Transição são os mesmos já apresentados para as RP Elementares, com a adição de **pesos** aos **ramos**. Ao contrário das RPE, um lugar pode estar ocupado por mais de um *token*.

Na literatura, as transições de uma RP L/T são geralmente representadas por **barras**. Chama-se de **lugar realimentado** ao lugar que, em relação a uma mesma transição, é de entrada e de saída.

Os pesos anotados sobre os ramos de entrada são chamados de **pesos de entrada (pe)** e os anotados sobre ramos de saída de **pesos de saída (ps)**. Caso um peso não seja especificado, subentende-se que o mesmo é 1. O peso **pe** representa o número de *tokens* que o lugar de entrada perde, caso haja o disparo da transição associada. Por outro lado, o peso **ps** representa o número de *tokens* que o lugar de saída ganha, caso haja o disparo da transição correspondente.

Segue-se a formalização matemática das RP L/T [40], a qual ajudará no tratamento computacional dos diversos tipos de RP. Nestas definições, **IN** representa o conjunto dos números naturais.

Definição 7.1: Representação Matemática de Redes L/T.

Uma RP L/T pode ser representada pela quintupla $(L, T, \alpha, \beta, M_0)$, onde:

- **L** é o conjunto de lugares da rede;
- **T** é o conjunto de transições da rede;
- $\alpha: L \times T \rightarrow \text{IN}$ é a chamada **função de incidência direta**
 $\alpha(l, t) = 0$, se **l** não é lugar de entrada em relação a **t**;
 $= \text{pe}$, não-nulo, se **l** é lugar de entrada em relação a **t**, com peso de entrada igual a **pe**;
- $\beta: L \times T \rightarrow \text{IN}$ é a chamada **função de incidência reversa**
 $\beta(l, t) = 0$, se **l** não é lugar de saída em relação a **t**;
 $= \text{ps}$, não-nulo, se **l** é lugar de saída em relação a **t**, com peso de saída igual a **ps**;
- $M_0: L \rightarrow \text{IN}$ é a chamada **função marcação inicial**
 $M_0(l) = n$, onde **n** é o número de *tokens* presentes no lugar **l** quando a rede é iniciada.

Definição 7.2: Regra de Habilitação.

Uma transição está habilitada se cada um de seus lugares de entrada contém um número de *tokens* maior ou igual ao peso de entrada (**pe**) do ramo que o une à transição.

Na figura 7.1, a transição **1** não está habilitada, uma vez que o lugar de entrada **1** contém apenas 2 *tokens* e o seu **pe** vale 3. Por outro lado, a transição **2** está habilitada, pois tanto o lugar **2** quanto o lugar **3** possuem mais *tokens* do que seus respectivos **pe**'s.

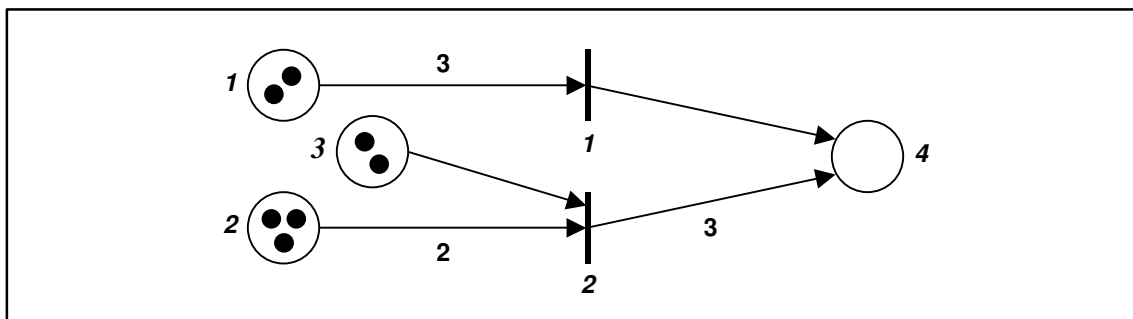


Figura 7.1. Transição 2 habilitada e transição 1 desabilitada.

Definição 7.3: Efeito do Disparo de uma Transição.

O disparo de uma transição t , habilitada, consiste em remover **pe** *tokens* de cada um dos lugares de entrada e adicionar **ps** *tokens* a cada um dos lugares de saída. Diz-se que a rede passou de uma marcação M_i para uma marcação M_j através do disparo da transição t .

Definição 7.4: Conjunto de Marcações Decorrentes de M_0 .

É o conjunto de todas as marcações pelas quais o sistema pode passar, partindo de uma dada marcação inicial M_0 . Notação: $\{M_0\}$.

O disparo da transição 2 da rede da figura 7.1 faz a mesma passar para a marcação denotada na figura 7.2. Temos a seguinte notação para representar esta mudança entre as marcações:

$$M_0 \xrightarrow{2} M_1$$

A notação anterior pode ser estendida para uma seqüência de disparos.

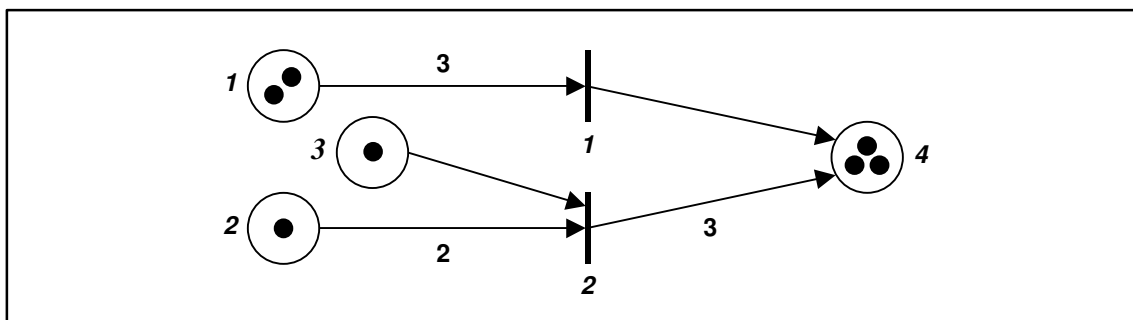


Figura 7.2. Marcação alcançada com o disparo da transição 2.

Definição 7.5: Seqüência de Disparos (ou σ).

É um conjunto de transições disparadas em seqüência, responsável pela passagem da rede de uma marcação M_i a uma marcação M_j , não necessariamente distintas.

Notação: $M_i \xrightarrow{\sigma} M_j$, onde σ é a seqüência de disparos.

Uma marcação pode ser denotada por uma tupla, na qual cada componente representa o número de *tokens* no lugar associado. No exemplo adotado, tem-se:

$$M_0 = (2, 3, 2, 0) \quad M_1 = (2, 1, 1, 3).$$

Desta forma, faz sentido representar a marcação de um lugar l por $M(l)$. Por exemplo, $M_1(l) = 2$.

As funções de incidência podem ser representadas **matricialmente**, para facilitar sua manipulação em computador. São duas as matrizes: a **matriz de incidência direta** (relativa a α) e a **matriz de incidência reversa** (relativa a β).

Definição 7.6: Matriz de Incidência Direta (ou Matriz A).

É uma matriz formada por $\#T$ linhas e $\#L$ colunas, representante da **estrutura da rede** quanto às informações dos lugares de entrada que incidem sobre cada transição e seu respectivo peso de entrada. O elemento da linha i e coluna j ($i = 1, \dots, \#T$; $j = 1, \dots, \#L$), denotado por A_{ij} , contém as seguintes informações:

- a) se $A_{ij} = 0$, o lugar l_j não é lugar de entrada da transição t_i ;
- b) se $A_{ij} \neq 0$, o lugar l_j é um lugar de entrada da transição t_i , com $A_{ij} = pe$.

Definição 7.7: Matriz de Incidência Reversa (ou Matriz B).

É uma matriz formada por $\#T$ linhas e $\#L$ colunas, representante da estrutura da rede quanto às informações dos lugares de saída que são atingidos por cada transição e seu respectivo peso de saída. O elemento da linha i e coluna j ($i = 1, \dots, \#T$; $j = 1, \dots, \#L$), denotado por B_{ij} , contém as seguintes informações:

- a) se $B_{ij} = 0$, o lugar l_j não é lugar de saída da transição t_i ;
- b) se $B_{ij} \neq 0$, o lugar l_j é um lugar de saída da transição t_i , com $B_{ij} = ps$.

A figura 7.3 reapresenta a rede da figura 7.1 com as suas matrizes de incidência direta e reversa.

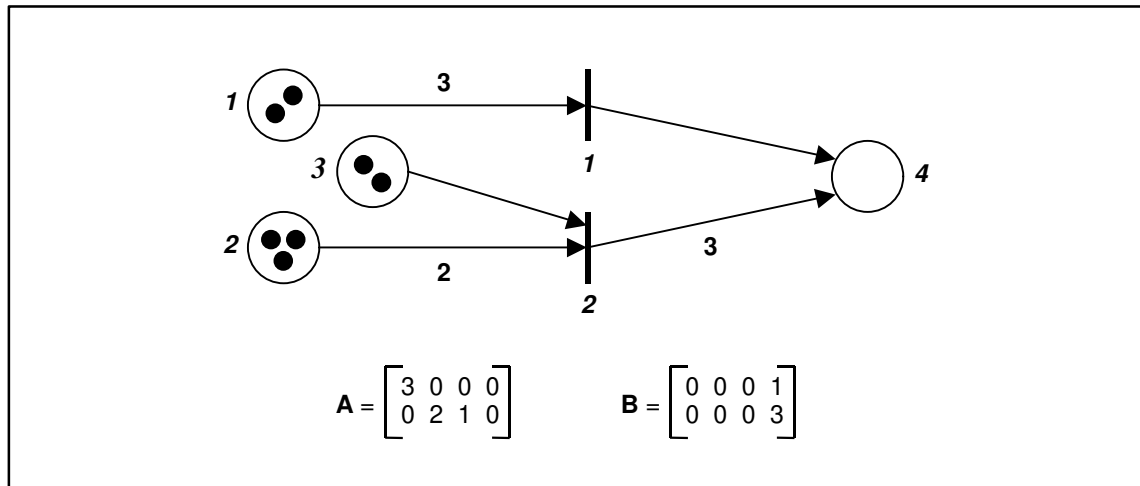


Figura 7.3. Matrizes de incidência direta (A) e reversa (B).

Definidas as matrizes associadas a uma rede L/T, pode-se fazer as definições matemáticas a seguir.

Definição 7.8: Conjunto de Lugares de Entrada e de Saída de uma Transição.

O conjunto de lugares de entrada da transição t_i é definido por:

$$E(t_i) = \{ l_j \in L \mid A_{ij} \neq 0 \} \quad (i = 1, \dots, \#T; j = 1, \dots, \#L).$$

O conjunto de lugares de saída da transição t_i é definido por:

$$S(t_i) = \{ l_j \in L \mid B_{ij} \neq 0 \} \quad (i = 1, \dots, \#T; j = 1, \dots, \#L).$$

Por exemplo, na figura 7.1, temos:

$$E(I) = \{I\}, S(I) = \{4\}, E(2) = \{2,3\}, S(2) = \{4\}.$$

Definição 7.9: Regra de Habilitação (representação matemática).

Uma transição t_i é habilitada ou disparável em uma dada marcação M se, e somente se:

$$\forall l_j \in E(t_i), M(l_j) \geq A_{ij} \quad (i = 1, \dots, \#T; j = 1, \dots, \#L).$$

Na figura 7.3, apenas a transição 2 pode disparar, pois:

$$E(I) = \{I\} \text{ e } M_0(I) = 2 < A_{11} = 3; \text{ e}$$

$$E(2) = \{2,3\} \text{ e } M_0(2) = 3 > A_{22} = 2 \text{ e } M_0(3) = 2 > A_{23} = 1.$$

Definição 7.10: Efeito do Disparo de uma Transição (representação matemática).

O disparo de uma transição habilitada t_i é a passagem da rede de uma marcação M à outra M' , tal que:

$$\forall l_j \in L, M'(l_j) = M(l_j) + B_{ij} - A_{ij} \quad (i = 1, \dots, \#T; j = 1, \dots, \#L).$$

Conforme visto na figura 7.3, para $M_0 = (2,3,2,0)$, o disparo da transição 2 resultará em M_1 , tal que:

$$M_1(I) = M_0(I) + B_{21} - A_{21} = 2 + 0 - 0 = 2;$$

$$M_1(2) = M_0(2) + B_{22} - A_{22} = 3 + 0 - 2 = 1;$$

$$M_1(3) = M_0(3) + B_{23} - A_{23} = 2 + 0 - 1 = 1;$$

$$M_1(4) = M_0(4) + B_{24} - A_{24} = 0 + 3 - 0 = 3;$$

Definição 7.11: RP Lugar/Transição sem Lugares Realimentados.

É uma rede onde não existe nenhum lugar que seja lugar de entrada e de saída ao mesmo tempo, em relação a qualquer transição t_i . Assim:

$$\text{se } A_{ij} \neq 0 \Rightarrow B_{ij} = 0, \text{ ou}$$

$$\text{se } B_{ij} \neq 0 \Rightarrow A_{ij} = 0.$$

Pela definição anterior, conclui-se que, para redes L/T sem lugares realimentados, as matrizes A e B são redundantes. Assim, pode-se utilizar uma única matriz para guardar as informações referentes à topologia da rede.

Definição 7.12: Matriz de Incidência (ou Matriz C).

É a matriz definida pela subtração da matriz B pela A , isto é, $C = B - A$. A matriz C contém as seguintes informações:

- a) se $C_{ij} = 0$, o lugar l_j não é nem lugar de entrada e nem lugar de saída da transição t_i ;
- b) se $C_{ij} > 0$, o lugar l_j é um lugar de entrada da transição t_i , com $C_{ij} = pe$;
- c) se $C_{ij} < 0$, o lugar l_j é um lugar de saída da transição t_i , com $C_{ij} = ps$.

Deixa-se observado que a definição de máquina de senhas para as redes L/T é a mesma que a apresentada para as RP Elementares (definição 6.5). Observa-se, também, que as propriedades a serem descritas a seguir são geralmente semelhantes para as demais extensões de RP.

7.2. PROPRIEDADES GERAIS DE UMA RP LUGAR/TRANSIÇÃO

7.2.1. LIMITABILIDADE

Uma rede L/T é definida como **limitada** a um valor natural n se, para qualquer marcação M decorrente da marcação inicial M_0 , o número de *tokens* em qualquer lugar não ultrapassa n . Matematicamente, tem-se:

$$\forall M \in \{M_0\}, \forall l_j \in L, M(l_j) \leq n \quad (j = 1, \dots, \#L).$$

Recebe o nome de **segura**, a RP L/T limitada a $n = 1$.

Numa rede limitada, o número de marcações é **finito**. O mesmo não ocorre com uma rede **ilimitada**, significando a impossibilidade de implementação do sistema modelado.

A rede da figura 7.4 é ilimitada, pois se uma seqüência de disparos $\sigma = (1, 2, 3)$ for repetida infinitas vezes, implicará em $M_i(I)$ sempre crescente.

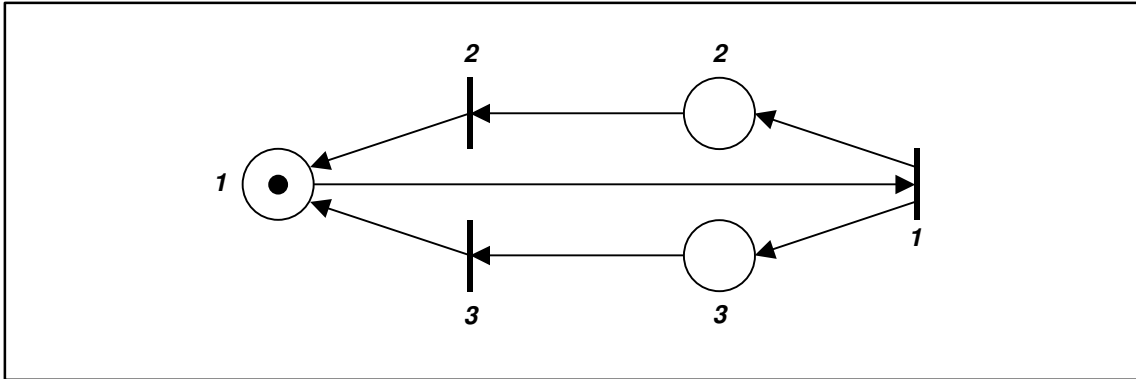


Figura 7.4. Rede L/T não limitada.

7.2.2. VIVACIDADE

Uma rede L/T é **viva** para uma marcação inicial M_0 , se para qualquer marcação decorrente M e para cada transição t considerada, existir uma seqüência de disparos σ , de forma que essa transição seja habilitada. Matematicamente, tem-se:

$$\forall M \in \{M_0\}, \forall t \in T, \exists \sigma \mid (M \xrightarrow{\sigma'} M' \wedge M' \xrightarrow{t} M'')$$

onde $\sigma = (\sigma', t)$ e $M', M'' \in \{M_0\}$.

Um sistema modelado por uma rede L/T viva está livre de *deadlocks*, isto é, não possui situações de impasse.

Considere a rede da figura 7.5.a. Para a marcação inicial $M_0 = (1, 0, 0)$, a máquina de senhas correspondente é mostrada na figura 7.5.b.

A transição **3** nunca irá disparar para a marcação $M_0 = (1, 0, 0)$, pois tem peso de entrada igual a **2**. Observe que a transição **4** também não dispara. Conclusão: **inexiste** uma seqüência de disparos que faça a rede alcançar uma marcação na qual **3** possa disparar. Assim, a rede da figura 7.5.a não é viva para a marcação inicial adotada.

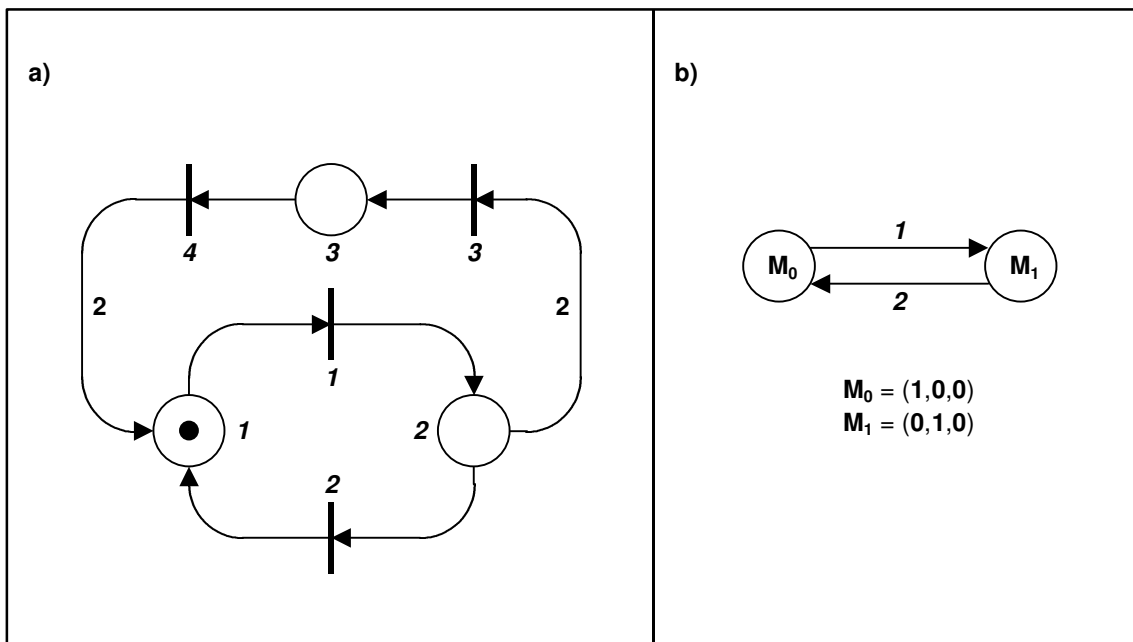


Figura 7.5. Rede L/T não viva e sua máquina de senhas.

Por outro lado, se for adotada a marcação inicial $M'_0 = (2,0,0)$ para a mesma rede, obtém-se a máquina de senhas mostrada na figura 7.6. Nela, observa-se que qualquer transição pode ser disparada a partir de uma marcação qualquer, tornando a rede da figura 7.5.a viva para esta nova marcação inicial. Desse modo, uma rede L/T pode ser viva para determinadas marcações iniciais e não viva para outras marcações iniciais.

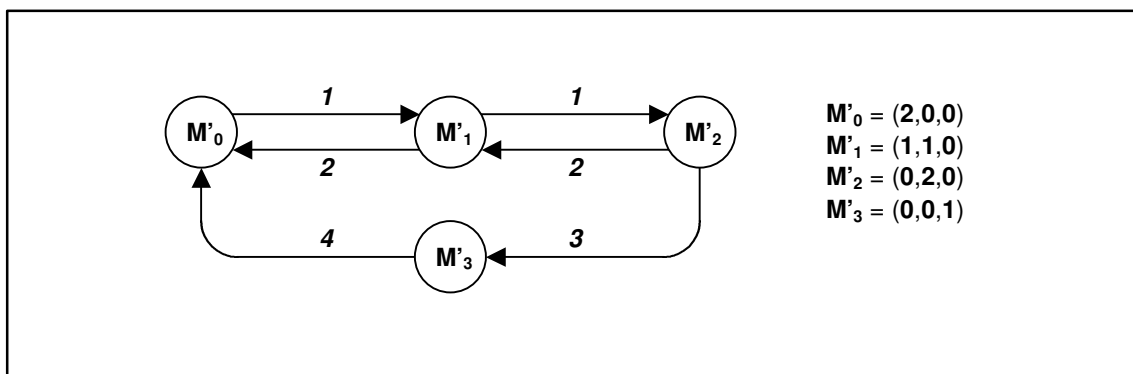


Figura 7.6. Máquina de senhas da rede da figura 7.5.a:
 – rede viva para nova marcação inicial.

7.2.3. REINICIABILIDADE

Uma RP L/T é **reiniciável** para uma dada marcação inicial M_0 , se para qualquer marcação decorrente $M \in \{M_0\}$ existir uma sequência de disparo σ tal que faça a rede voltar à marcação inicial M_0 . Matematicamente, tem-se:

$$\forall M \in \{M_0\}, \exists \sigma \mid (M \xrightarrow{\sigma} M_0).$$

Um sistema físico modelado por uma rede reiniciável **retorna** ao seu estado inicial após a execução de uma ou mais tarefas.

Considere a rede da figura 7.7.a. Para a marcação inicial $M_0 = (1,0,0,1)$, a máquina de senhas correspondente é mostrada na figura 7.7.b.

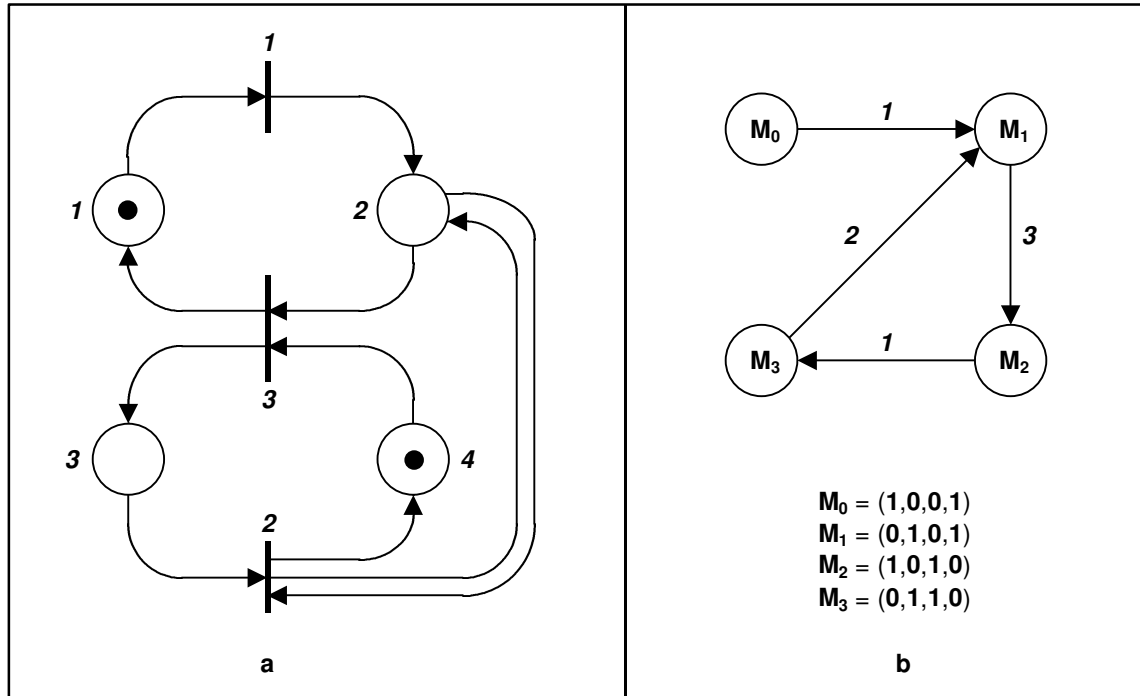


Figura 7.7. Rede L/T limitada, viva e não reiniciável e sua máquina de senhas.

A rede é não reiniciável porque inexistente uma sequência de disparo que partindo da marcação inicial M_0 , faça o sistema retornar a ela. Por outro lado, a rede é limitada e viva.

Adotando a marcação inicial $M'_0 = (1,0,1,0)$ para a mesma rede, obtém-se a máquina de senhas mostrada na figura 7.8. Verifica-se, assim, que para esta marcação inicial a rede se torna reiniciável.

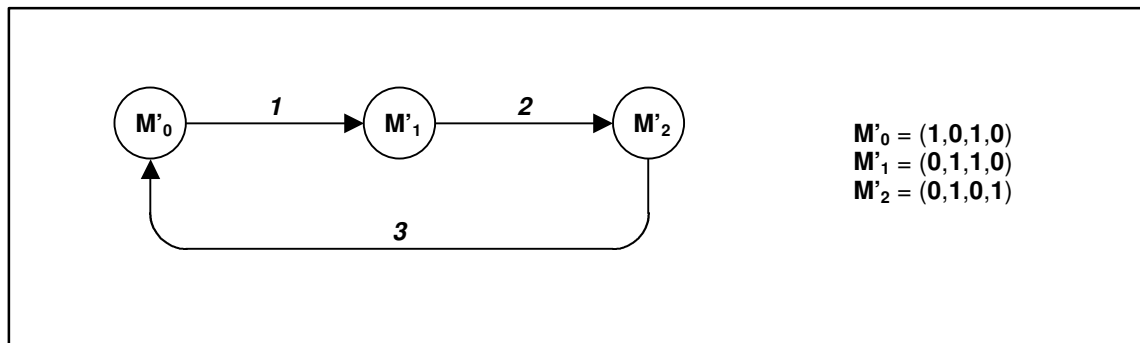


Figura 7.8. Máquina de senhas da rede da figura 7.7.a com nova marcação inicial.

7.3. EXEMPLO DE MODELAGEM DE UMA BIBLIOTECA EM RP L/T

Nesta seção retoma-se o exemplo do sistema descrito na seção 6.4, o qual consiste no atendimento de usuários em uma biblioteca.

No capítulo anterior, pode-se constatar a pouca capacitação das RP Elementares no que diz respeito a sistemas onde o aspecto **quantitativo** predomina. Isto se deve ao fato de uma rede elementar aceitar no máximo um *token* em cada lugar. Entretanto, tal restrição inexistente nas redes lugar/transição.

Nas RPE, a modelagem de **n** elementos acarreta na repetição de uma parte da rede **n** vezes. No caso do sistema da biblioteca, pode-se constatar, na figura 6.6, que a modelagem de 2 livros na prateleira obrigava a **duplicação** da rede inteira. Analogamente, a modelagem de 3 livros obrigaria **triplicar** a rede e, assim, sucessivamente para qualquer quantidade de livros. Esta característica das RPE **amarra** a quantidade de livros à estrutura da rede (ou seja, seus lugares e transições). Isto ocorre em muitos outros sistemas.

As RP L/T permitem a modelagem de qualquer quantidade de livros, bastando inicializar a rede com um número de *tokens* igual ao total de livros, conforme se pode ver na figura 7.9, onde são modelados 2 livros.

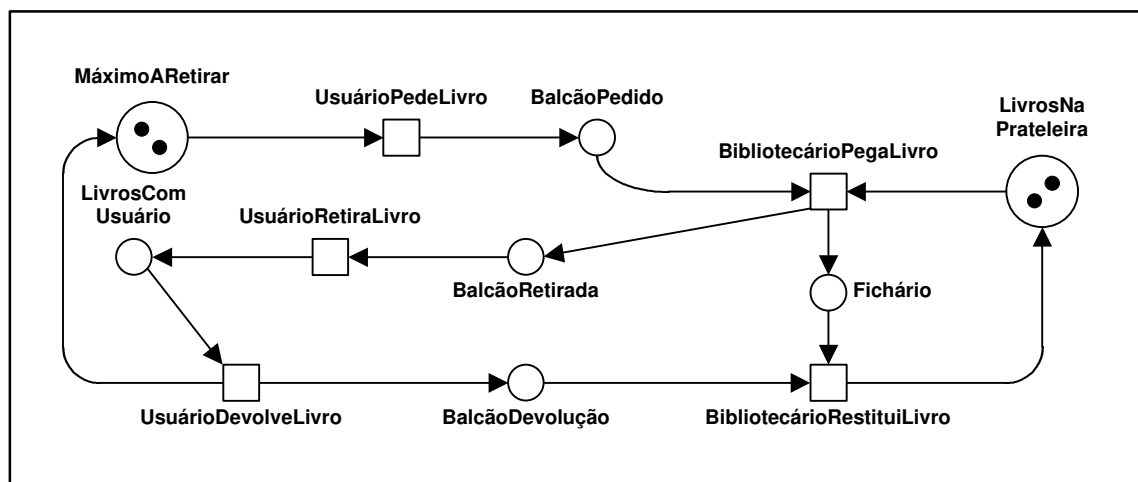


Figura 7.9. RP L/T modelando uma biblioteca com 2 livros.

Agora, tem-se uma situação onde o usuário pode retirar até dois livros. Apesar de inadequada, a RPE da figura 6.6 diferenciava os 2 livros. Assim, RP L/T são adequadas somente quando **não se necessita distinguir** entre a **natureza** dos *tokens* envolvidos, importando-nos **somente a quantidade** dos mesmos. As redes de **alto nível**, cuja principal representante são as Coloridas (capítulo 9), surgiram para resolver este problema.

CAPÍTULO 8: REDES DE PETRI TEMPORIZADAS

8.1. TIPOS DE REDES DE PETRI TEMPORIZADAS

A literatura mostra que RP têm sido estendidas de diferentes formas a fim de incorporar o conceito de **tempo**. Tais extensões surgiram para a modelagem de sistemas temporizados com características diferentes. RP foram estendidas com restrições temporais pela primeira vez na década de 1970, através dos trabalhos de *Ramchandani* [41] e de *Merlin* [42]. A partir deste ponto, vários modelos de RP com tempo foram propostos, motivados por várias aplicações. Estes modelos diferem um do outro em relação a três aspectos [43]:

- a) **O tipo de restrição de tempo adotada;**
- b) **A localização (lugares, transições, arcos) na qual as restrições são anotadas;**
- c) **Quão forte as restrições temporais atuam sobre o controle da rede.**

Restrições de tempo podem ser descritas através de um único valor de tempo (representando **atraso**), por um par de valores de tempo (**intervalo de tempo**) ou por um conjunto de valores de tempo (denominado *time set*). Na literatura recente, a denominação *timed Petri nets* é utilizada para extensões onde as restrições temporais correspondem a atrasos; já a denominação *time Petri nets* é usada para as extensões onde as restrições temporais são intervalos; por fim, a expressão *time(d) Petri nets* é utilizada, por alguns autores, para representar genericamente extensões de RP com restrições temporais, independentemente de como estas restrições se apresentam.

Quanto à **localização**, restrições de tempo podem estar estaticamente associadas com diferentes elementos da rede, quais sejam: **lugares** [44], **transições** [45] ou **arcos** [46]. Há extensões onde as restrições podem ser avaliadas dinamicamente durante o disparo de uma transição, através da computação de uma função associada com a transição [47].

A força com a qual o tempo atua sobre o **controle**, leva a diferenças semânticas. Redes **fracamente temporizadas** são aquelas nas quais as restrições de tempo não podem forçar o disparo de uma transição [46]. Por outro lado, são denominadas **fortemente temporizadas**, as redes cujas restrições temporais podem forçar o disparo de transições [42, 45].

De acordo com a extensão temporizada de RP, o **domínio de tempo** poderá ser constituído por números **naturais**, números **racionais não-negativos** ou números **reais não-negativos**. Entretanto, em geral, estes tipos de domínio poderiam ser utilizados em qualquer modelo. Durações e atrasos são considerados adequados para a análise de estimativas de desempenho, sendo estaticamente associados a lugares ou transições. Intervalos de tempo são considerados mais adequados para propósitos de verificação e especificação de um sistema, sendo estaticamente associados às transições [42, 45] ou arcos [46].

Cerone e Maggiolo-Schettini apresentam em [43] uma plataforma genérica para o estudo da expressividade de várias extensões de RP, as quais diferem em relação à localização das restrições, à força das restrições sobre o disparo de transições e ao domínio de tempo. Eles demonstraram que, para redes fracamente temporizadas, diferentes localizações de restrições não levaram a diferenças na expressividade das extensões; já para redes fortemente temporizadas, a adoção de algumas limitações igualou a força descritiva de modelos, desde

que o domínio de tempo fosse discreto. Os autores afirmam que somente a temporização forte é capaz de aumentar a força descritiva de RP.

Na sequência, é dada uma breve descrição dos tipos básicos de RP temporizadas, criados na década de 1970. Na seção 8.4 são fornecidos maiores detalhes sobre a **RP temporizada de Merlin**, a qual é amplamente utilizada, pelo fato de ser uma extensão fortemente temporizada e por trabalhar com intervalos de tempo, o que possibilita a especificação e verificação de sistemas. Na parte III desta tese, as características da extensão de Merlin serão agregadas à extensão desenvolvida neste trabalho de doutorado, para a especificação de sistemas digitais. Além disso, segundo Marton *et al.* [48], os demais tipos básicos de redes temporizadas são casos particulares do modelo proposto por Merlin.

8.2. RP TEMPORIZADAS COM VALORES DE TEMPO ASSOCIADOS ÀS TRANSIÇÕES

Nesta seção são descritas duas formas de associar tempo às transições de uma RP. Uma dessas formas encontra-se representada na **RP temporizada de Merlin**, enquanto a outra forma é representada pela **RP temporizada de Ramchandani**.

8.2.1. DESCRIÇÃO DAS RP TEMPORIZADAS DE MERLIN

Esta extensão (*Time Petri Nets* – TPN) foi proposta por Merlin em 1974 [42], sendo ampliada em 1976 por Merlin e Faber [45]. Neste modelo, é associado um intervalo de tempo a cada uma das transições da rede.

A TPN é um par $(\mathbf{RP\ L/T}, \mathbf{E})$, onde:

- **RP L/T** é uma RP do tipo Lugar/Transição;
- **E** é uma função que associa um intervalo fechado de números racionais não-negativos $[a_t, b_t]$, $a_t \leq b_t$, para cada transição **t** da RP associada, chamado intervalo de disparo estático de **t**.

As transições são habilitadas da mesma forma que nas RP L/T (definição 7.2). O disparo de uma transição **t** ocorre somente dentro do intervalo $[a_t, b_t]$, contado a partir do instante em que **t** torna-se habilitada. Do mesmo modo que nas RP L/T, o disparo de uma transição é instantâneo e provoca o mesmo efeito (definição 7.3).

8.2.2. DESCRIÇÃO DAS RP TEMPORIZADAS DE RAMCHANDANI

Esta extensão (*Timed Petri Nets* – TdPN) foi proposta por Ramchandani em 1974 [41]. Neste modelo, é associado um tempo para o disparo de cada transição da rede. Diferentemente das RP originais, o disparo da transição não é mais um evento instantâneo.

A TdPN é um par $(\mathbf{RP\ L/T}, \mathbf{D})$, onde:

- **RP L/T** é a Rede de Petri Lugar/Transição $(\mathbf{L}, \mathbf{T}, \alpha, \beta, \mathbf{M}_0)$ sem a consideração dos tempos envolvidos, chamada RP associada;
- **D** é uma função que associa um número relativo à duração do disparo de cada transição da RP associada.

As transições são habilitadas do mesmo modo que nas **RP L/T** (definição 7.2). O disparo de uma transição habilitada **t** causa uma mudança da marcação **M** para a marcação **M'**, através de **duas etapas (não mais uma única etapa instantânea)**:

- 1) Tão logo a transição **t** seja habilitada, ela deve disparar (a menos que outra transição dispare e a desabilite), promovendo a retirada de *tokens* de seus lugares de entrada, isto é, $\forall l \in L, M'(l) = M(l) - \alpha(l, t)$, onde:
 - M(l)** é o número de *tokens* no lugar **l** antes da realização dessa primeira etapa;
 - M'(l)** é o número de *tokens* no lugar **l** após a realização dessa primeira etapa;
 - $\alpha(l, t)$ é o peso do arco que vai do lugar **l** para a transição **t**.
- 2) **D(t)** unidades de tempo após o disparo de **t**, *tokens* são inseridos em seus lugares de saída, isto é, $\forall l \in L, M''(l) = M'(l) + \beta(l, t)$, onde:
 - M''(l)** é o número de *tokens* no lugar **l** após a realização dessa segunda etapa, ou seja, após o disparo completo da transição;
 - $\beta(l, t)$ é o peso do arco que vai da transição **t** para o lugar **l**.

8.3. RP TEMPORIZADAS COM VALORES DE TEMPO ASSOCIADOS AOS LUGARES

A primeira extensão desta família foi proposta por Sifakis em 1977 (*Timed Place/Transition Nets – TdPTN*) [49]. Neste modelo, é associado um tempo de permanência de *tokens* em cada lugar da rede.

A TdPTN é um par **(RP L/T, A)**, onde:

- **RP L/T** é uma RP do tipo Lugar/Transição;
- **A** é uma função que associa um atraso a cada lugar da RP associada.

Sifakis demonstrou, em 1979, a equivalência de seu modelo com o de Ramchandani.

8.4. RP TEMPORIZADAS DE MERLIN (RPTM)

Sendo um dos modelos mais utilizados de redes temporizadas, nesta seção é estudada com mais detalhes a definição matemática de redes de Petri temporizadas de Merlin. Pelo fato de RP temporizadas adotarem siglas muito semelhantes em inglês, vamos adotar neste texto, para propósitos de diferenciação, a sigla em português **RPTM**.

Definição 8.1: Representação Matemática de RPTM.

Uma RPTM pode ser representada pela sêxtupla **(L, T, α , β , **M**₀, E)**, onde:

- **L** é o conjunto de **lugares** da rede;
- **T** é o conjunto de **transições** da rede;
- $\alpha: L \times T \rightarrow \mathbb{IN}$ é a chamada **função de incidência direta**
 - $\alpha(l, t) = 0$, se **l** não é lugar de entrada em relação a **t**
 - = **pe**, não-nulo, se **l** é lugar de entrada em relação a **t**, com peso de entrada igual a **pe**;

- $\beta: L \times T \rightarrow \mathbb{IN}$ é a chamada **função de incidência reversa**
 $\beta(l, t) = 0$, se l não é lugar de saída em relação a t ;
 $= ps$, não-nulo, se l é lugar de saída em relação a t ,
com peso de saída igual a ps .
- $M_0: L \rightarrow \mathbb{IN}$ é a chamada **função marcação inicial**
 $M_0(l) = n$, onde n é o número de *tokens* presentes no lugar l
quando a rede é iniciada;
- $E: T \rightarrow (Q^+ \cup \{0\}) \times (Q^+ \cup \{0\} \cup \infty)$, $Q =$ conjunto dos números racionais,
é a chamada **função de intervalos estáticos**.
 $E(t) = [a_t, b_t]$, onde $0 \leq a_t \leq b_t$

Observações da definição 8.1:

- a) O limite inferior a_t é chamado **tempo de disparo inicial (TDI) estático**, ou simplesmente TDI estático;
- b) O limite superior b_t é chamado **tempo de disparo final (TDF) estático**, ou simplesmente TDF estático;
- c) O intervalo de disparo $[TDI, TDF]$ de uma transição t **delimita** o instante de tempo no qual poderá ocorrer o disparo de uma transição, a partir do momento em que ela torna-se habilitada. Pode-se afirmar que RP L/T é um caso particular de RPTM, onde se associa a cada transição o intervalo $[0, \infty]$.

Definição 8.2: Marcação de uma RPTM.

A marcação de uma RPTM é uma tupla de pares ordenados (l, m) , onde l é um lugar não vazio dentro da marcação e m é o número de *tokens* presentes em l .

Definição 8.3: Função de Intervalos Dinâmicos.

$D: T \rightarrow (Q^+ \cup \{0\}) \times (Q^+ \cup \{0\} \cup \infty)$, $Q =$ conjunto dos números racionais.
 $D(t) = [a_t, b_t]$, onde $0 \leq a_t \leq b_t$

A função de intervalos dinâmicos D associa, a cada transição t habilitada por uma marcação M , um intervalo de tempo no qual é permitido o disparo de t . Este intervalo de tempo é chamado de intervalo de disparo dinâmico, correspondendo ao restante do intervalo no qual t ainda pode disparar.

Definição 8.4: Estado de uma RPTM.

O **estado** de uma RPTM é dado por $S_i = (M_i, D_i)$, onde:

- a) M_i é a **marcação** da RP associada;
- b) D_i é a **função de intervalos dinâmicos** para as transições habilitadas em M_i .

A figura 8.1 mostra uma RPTM com estado inicial $S_0 = (M_0, D_0)$. Nela, tem-se:

- a) $M_0 = ((1, 1), (6, 1))$, significando que o lugar **1** tem 1 *token* e o lugar **6** tem 1 *token*;
- b) $D_0 = (1[0, 2], 6[2, 3])$, significando que a transição **1** está habilitada e tem intervalo de disparo dinâmico $[0, 2]$ e que a transição **6** também está habilitada, com intervalo de disparo dinâmico $[2, 3]$.

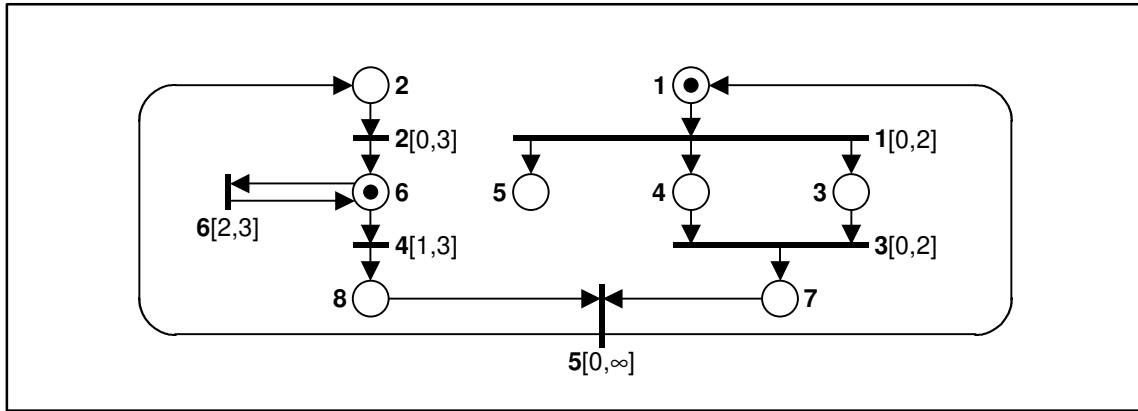


Figura 8.1. Exemplo de RPTM.

A identificação das transições que podem disparar numa RPTM é dada pela definição a seguir.

Definição 8.5: Transição Disparável (em um estado).

Numa RPTM, uma transição t é disparável no instante τ do estado $S = (M, D)$ se as seguintes condições forem satisfeitas:

a) t está habilitada na marcação M , ou seja:

$$\forall l \in L, M(l) \geq \alpha(l, t); e$$

b) τ não é menor que o **TDI** da transição t e não é maior que o menor dos **TDF** de todas as demais transições habilitadas k pela marcação M , ou seja:

$$TDI(t) \leq \tau \leq \min_k (TDF(k)).$$

A condição estabelecida no item **a)** da definição 8.5 é a regra de habilitação de uma transição para as RP L/T, conforme visto na definição 7.9 (definição equivalente).

A condição estabelecida no item **b)** da definição 8.5 representa o fato de uma transição t habilitada **não poder disparar antes** de seu **TDI** dinâmico e **dever disparar até**, inclusive, o instante do menor dos **TDF**'s dinâmicos, a menos que seja desabilitada pelo disparo de outra transição **conflitante** k . Duas transições t e k são conflitantes se possuem lugares de entrada comuns.

O tempo τ é contado a partir do momento da **geração** do estado S . Desta forma, o tempo absoluto, ou seja, aquele contado a partir do estado inicial S_0 , é igual a τ acrescido ao tempo para alcançar o estado S .

Definição 8.6: Efeito do Disparo de uma Transição (em um estado).

O efeito do disparo de uma transição disparável t , no instante τ do estado $S = (M, D)$, é a passagem da rede para um estado $S' = (M', D')$, de tal forma que:

a) A nova marcação M' é gerada como nas RP L/T (equivalente à definição 7.10):

$$\forall l \in L, M'(l) = M(l) - \alpha(l, t) + \beta(l, t);$$

b) O novo domínio D' é gerado em três passos:

b.1) Retira-se da expressão do domínio D os intervalos de disparo relativos às transições k que foram desabilitadas quando da retirada de *tokens* dos lugares de entrada

promovida pelo disparo de t . Estas transições k são tais que k é habilitada em M e k não é habilitada em M^* , onde M^* é a **marcação intermediária** entre M e M' , representante apenas da retirada de *tokens* dos lugares de entrada, isto é:

$$\forall l \in L, M^*(l) = M(l) - \alpha(l, t).$$

A marcação intermediária M^* se deve à possibilidade da rede possuir lugares realimentados. O intervalo de disparo relativo à transição t também é retirado;

- b.2) Os intervalos de disparo restantes em D são deslocados em direção à origem dos tempos em um valor τ e truncados, caso necessário, para não incluir valores negativos;
- b.3) Os intervalos de disparo estáticos das novas transições habilitadas em M' (mas não habilitadas em M^*) são inseridos na nova expressão de domínio D' .

Notação: $S \xrightarrow{(t,\tau)} S'$

No exemplo da figura 8.1, a transição **1** pode disparar em qualquer instante no intervalo $[0,2]$ e a transição **6** pode disparar apenas no instante 2 (o **TDI** da transição **6** é igual ao **TDF** da transição **1**, que é o menor dos **TDF**). Assim, usando a definição 8.6 tem-se:

- a) Disparando a transição **1** num tempo $\tau_1 \in [0,2]$, a rede passa para um estado $S_1 = (M_1, D_1)$, onde:

$$M_1 = ((3,1), (4,1), (5,1), (6,1)) \text{ e}$$

$$D_1 = (3[0,2], 4[1,3], 6[2-\tau_1, 3-\tau_1]), 0 \leq \tau_1 \leq 2$$

Note que a transição **6** é remanescente quando a transição **1** dispara. Assim, seu intervalo foi deslocado em direção à origem dos tempos em um valor τ_1 ;

- b) Disparando a transição **6** no instante 2, a rede passa para um estado $S_2 = (M_2, D_2)$, onde:

$$M_2 = ((1,1), (6,1)) \text{ e}$$

$$D_2 = (1[0,0], 6[2,3])$$

Os disparos das transições **3**, **4** e **6** no estado S_1 e da transição **1** em S_2 conduzem a rede para outros estados. Contudo, não estamos interessados na geração de estados, mas sim no que vamos caracterizar como classes de estados.

Definição 8.7: Seqüência de Disparos Temporizada (σ, θ) .

Uma seqüência de disparos temporizada é o par (σ, θ) , onde σ é uma seqüência de transições disparáveis e θ é uma seqüência de tempos.

$$(\sigma, \theta) = (t_1, \tau_1), (t_2, \tau_2), (t_3, \tau_3), \dots, (t_n, \tau_n), \text{ onde}$$

$$t_i \in T \text{ e } \tau_i \text{ é o instante de disparo de } t_i, 1 \leq i \leq n.$$

A seqüência de disparos temporizada (σ, θ) é responsável pela transição da rede de um estado S a um estado S_n , se as transições t_i na seqüência σ são sucessivamente disparáveis em seus correspondentes instantes de tempo τ_i na seqüência θ .

Notação: $S \xrightarrow{(\sigma, \theta)} S_n$

ou

$$S \xrightarrow{(t_1, \tau_1)} S_1 \xrightarrow{(t_2, \tau_2)} S_2 \cdots \xrightarrow{(t_n, \tau_n)} S_n$$

onde $S_1, S_2, \dots, S_{n-1}$ são estados **intermediários**.

Definição 8.8: Conjunto de Estados Decorrentes de S_0 .

É o conjunto de todos os estados pelos quais o sistema pode passar, partindo de um dado estado inicial S_0 .

Notação: $\{S_0\}$.

O conjunto de estados decorrentes de S_0 , assim como as correspondentes seqüências de disparos temporizadas (σ, θ) **caracterizam** o comportamento da RPTM, tal qual a máquina de senhas para os demais tipos de RP. Entretanto, o fato de τ_i poder assumir uma infinidade de valores dentro do intervalo de disparo $[TDI, TDF]$ de cada transição t_i habilitada numa marcação M , leva a rede a uma infinidade de estados.

Há um método [50, 51] para a solução do problema de explosão de estados, o qual consiste na geração e enumeração de classes de estados, ou seja, agrupa-se o conjunto infinito de estados em um conjunto finito de classes. Este método é descrito na seção seguinte.

8.5. MÉTODO DE ENUMERAÇÃO DE CLASSES DE ESTADOS PARA RPTM

O método de enumeração de classes de estados permite gerar uma representação do comportamento da RPTM. Tal comportamento será representado através de um grafo orientado onde os nós são as classes de estados e os arcos representam a relação de acesso entre as classes, a qual é provocada pelo disparo de uma transição. Este grafo é chamado de grafo de classes de estados.

Ao invés de considerarmos um dado estado S decorrente de S_0 através de uma seqüência temporizada (σ, θ) , consideremos o conjunto dos estados gerados por todas as seqüências temporizadas com a mesma seqüência σ e diferentes seqüências de tempo θ . Este conjunto de estados será chamado de classe de estados associados com a seqüência de disparos σ .

As classes são pares $C = (M, D)$ onde:

- M é a **marcação da classe C** , ou seja, todas as classes em C têm a **mesma** marcação;
- D é o **domínio de disparo da classe**, definido como a **união** dos domínios D_i de todos os estados em C .

A figura 8.2 ilustra a transformação de estados para classes de estados. Nesta figura, S_{ni} representa o estado decorrente do estado inicial S_0 pela seqüência temporizada (σ, θ_i) . A classe C é o conjunto de todos os estados S_{ni} ($i = 1, 2, \dots, \infty$).

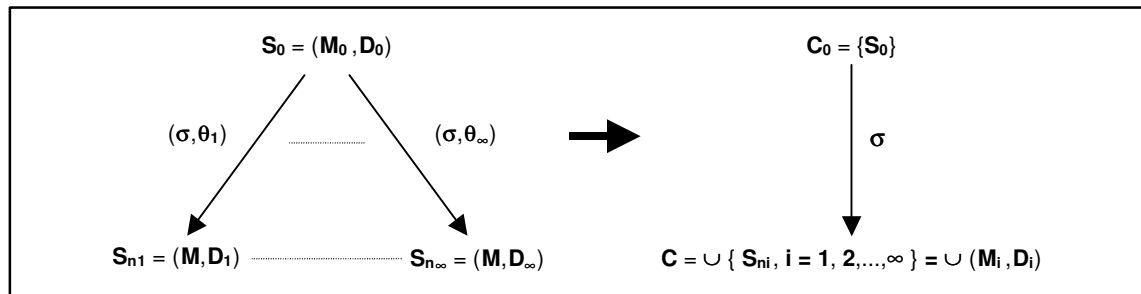


Figura 8.2. Transformação de estados em classes de estados.

Assume-se para a RPTM que a classe de estado atual da rede seja $C = (M, D)$, onde D é o conjunto solução de um certo sistema de inequações lineares no qual a variável de tempo x_i é associada com a transição t_i habilitada na marcação M . Temos:

$$D = \{ x \mid A \cdot x \geq b \}$$

onde A é uma matriz, b é um vetor de constantes e x é um vetor cujas componentes são as variáveis x_i .

As expressões de domínio para as classes de estados permitem introduzir **relações de interdependência** entre os tempos de disparo de diversas transições.

Definição 8.9: Transição Disparável (em uma classe).

Numa RPTM, uma transição t é disparável na classe $C = (M, D)$ se as seguintes condições forem satisfeitas:

a) t está habilitada na marcação M , ou seja:

$$\forall l \in L, M(l) \geq \alpha(l, t); e$$

b) Assumindo que t é a k -ésima transição habilitada na marcação M , então o seguinte sistema de inequações é consistente:

$$\begin{aligned} A \cdot x &\geq b \\ x_k &\leq x_j, \forall j, j \neq k \end{aligned}$$

onde x_k e x_j representam a k -ésima e a j -ésima componentes de x .

Observa-se que a condição estabelecida pelo item 8.9.a é a regra de habilitação de uma transição para as RP L/T, conforme vimos na definição 7.9 (definição equivalente).

A condição estabelecida pelo item 8.9.b significa que no conjunto dos intervalos de tempos de disparos (intervalos denotados pelas variáveis x_i) constituintes do domínio D da classe C , a transição t_k pode ser disparada antes ou no mesmo instante (de simulação) que qualquer outra transição habilitada t_j possa ser disparada.

Definição 8.10: Efeito do Disparo de uma Transição (em uma classe).

O efeito do disparo de uma transição disparável t_j na classe $C = (M, D)$ é a passagem da rede para uma classe $C' = (M', D')$, de tal forma que:

a) A nova marcação M' é gerada como nas RP L/T (equivalente à definição 7.10):

$$\forall l \in L, M'(l) = M(l) - \alpha(l, t) + \beta(l, t);$$

b) Seja o domínio D dado na seguinte forma:

$$TDI(t_i) \leq x_i \leq TDF(t_i)$$

$$TDI(t_j) \leq x_j \leq TDF(t_j)$$

⋮

$$TDI(t_y) \leq x_y \leq TDF(t_y)$$

$$x_k - x_n \leq H_{kn}^*$$

$$x_n - x_k \leq H_{nk}^*$$

⋮

onde:

- $t_z, z = i, j, k, n, y$ são as transições habilitadas em M ;
- $TDI(t_i), \dots, TDI(t_y)$ são os seus **TDI** dinâmicos;
- $TDF(t_i), \dots, TDF(t_y)$ são os seus **TDF** dinâmicos;
- $H_{kn}^* = [(TDF(t_k) \text{ em } D^*) - (TDI(t_n) \text{ em } D^*)]$, onde D^* é o domínio da classe C^* que gerou a classe C , ou seja, C^* é a classe antecessora de C ;
- $H_{nk}^* = [(TDF(t_n) \text{ em } D^*) - (TDI(t_k) \text{ em } D^*)]$;
- Para a transição disparada t_j vale: $TDI(t_j) \leq \min TDF(t_z)$.

O novo domínio D' é gerado em cinco passos [48], quais sejam:

b.1) Para cada transição t_k não desabilitada pelo disparo de t_j , o intervalo de disparo $[TDI, TDF]$ em D' será dado por:

$$TDI(t_k) = \max \{ [0]; [-H_{jk} \text{ se existir em } D]; [(TDI(t_k) \text{ em } D) - (\text{menor } TDF \text{ de todas as variáveis})] \}$$

$$TDF(t_k) = \min \{ [(TDF(t_k) \text{ em } D) - (TDI(t_j) \text{ em } D)]; [H_{kj} \text{ se existir em } D] \}$$

b.2) Para cada par de transições t_k e t_n não desabilitadas pelo disparo de t_j , tem-se:

$$x_k - x_n \leq H_{kn}$$

$$x_n - x_k \leq H_{nk}$$

onde:

$$H_{kn} = \min \{ [(TDF(t_k) \text{ em } D) - (TDI(t_n) \text{ em } D)]; [H_{kn}^* \text{ se existir}] \};$$

$$H_{nk} = \min \{ [(TDF(t_n) \text{ em } D) - (TDI(t_k) \text{ em } D)]; [H_{nk}^* \text{ se existir}] \}.$$

b.3) As variáveis relativas às transições desabilitadas pelo disparo de t_j não aparecerão em D' ;

b.4) Também farão parte do domínio D' as variáveis correspondentes às transições que não eram habilitadas em M mas o são em M' . Cada uma dessas variáveis terá o seu intervalo de tempo determinado pelo **TDI** e **TDF** estáticos da transição correspondente;

b.5) Por fim, os **TDI** e os **TDF** dos intervalos das variáveis são subtraídos de um valor igual ao menor dos **TDI**.

A seguir, as regras dadas pela definição 8.10 serão usadas para gerar as classes de estados para a RPTM da figura 8.1.

A classe inicial (figura 8.1) é $C_0 = (M_0, D_0)$, onde:

$$M_0 = ((1,1), (6,1))$$

$$D_0: \quad 0 \leq x_1 \leq 2$$

$$2 \leq x_6 \leq 3$$

Da classe C_0 , as transições **1** e **6** podem disparar, pois o menor dos **TDF** dinâmicos é 2 (**TDF** de x_1) e o **TDI** dinâmico da transição **6** é 2. Assim:

- a) Disparando a transição **1** é gerada a classe $C_1 = (M_1, D_1)$, tal que:
- A marcação M_1 foi obtida da forma usual, conforme já dito anteriormente (definição 7.10);
 - O domínio D_1 foi obtido usando as seguintes regras da definição 8.10:
 - 8.10.a: Novos valores para **TDI (6)** e **TDF (6)**;
 - 8.10.c: Exclusão de x_1 ;
 - 8.10.d: x_3 e x_4 são as novas variáveis.
- b) Disparando a transição **6** é gerada a classe $C_2 = (M_2, D_2)$, tal que:
- A marcação M_2 foi obtida da forma usual;
 - O domínio D_2 foi obtido usando as seguintes regras:
 - 8.10.a: Novos valores para **TDI (1)** e **TDF (1)**;
 - 8.10.d: x_6 é a nova variável.

Da classe C_1 , as transições **3**, **4** e **6** podem disparar. Lembrando que a marcação de uma classe é gerada de forma usual, tem-se que:

- a) Disparando a transição **3** é gerada a classe $C_3 = (M_3, D_3)$, tal que:
- O domínio D_3 foi obtido usando as seguintes regras:
 - 8.10.a: Novos valores para **TDI (4)**, **TDF (4)**, **TDI (6)** e **TDF (6)**;
 - 8.10.b: Inclusão das relações: $x_4 - x_6 \leq 3$
 $x_6 - x_4 \leq 2$;
 - 8.10.c: Exclusão de x_3 .
- b) Disparando a transição **4** é gerada a classe $C_4 = (M_4, D_4)$, tal que:
- O domínio D_4 foi obtido usando as seguintes regras:
 - 8.10.a: Novos valores para **TDI (3)** e **TDF (3)**;
 - 8.10.c: Exclusão de x_4 e de x_6 .
- c) Disparando a transição **6** é gerada a classe $C_5 = (M_5, D_5)$, tal que:
- O domínio D_5 foi obtido usando as seguintes regras:
 - 8.10.a: Novos valores para **TDI (3)** e **TDF (3)**;
 - 8.10.c: Exclusão de x_4 e x_6 ;
 - 8.10.d: x_4 e x_6 são as novas variáveis.

De forma semelhante, obtém-se as classes de estados restantes. Todas as classes encontram-se descritas na tabela 8.1. O grafo das classes de estados correspondente à RPTM da figura 8.1 é dado na figura 8.3. O grafo das classes para uma RPTM equivale à máquina de senhas de uma RP L/T.

Propriedades de vivacidade, reiniciabilidade e limitabilidade podem ser definidas para RPTM de maneira análoga àquelas definidas para as RP L/T.

Tabela 8.1. Classes de estados para a RPTM da figura 8.1.

Classe	Marcação	Domínio
0	$M_0 = ((1,1),(6,1))$	$D_0: 0 \leq x_1 \leq 2$ $2 \leq x_6 \leq 3$
1	$M_1 = ((3,1),(4,1),(5,1),(6,1))$	$D_1: 0 \leq x_3 \leq 2$ $1 \leq x_4 \leq 3$ $0 \leq x_6 \leq 3$
2	$M_2 = ((1,1),(6,1))$	$D_2: 0 \leq x_1 \leq 0$ $2 \leq x_6 \leq 3$
3	$M_3 = ((5,1),(6,1),(7,1))$	$D_3: 0 \leq x_4 \leq 3$ $0 \leq x_6 \leq 3$ $x_4 - x_6 \leq 3$ $x_6 - x_4 \leq 2$
4	$M_4 = ((3,1),(4,1),(8,1))$	$D_4: 0 \leq x_3 \leq 1$
5	$M_5 = ((3,1),(4,1),(5,1),(6,1))$	$D_5: 0 \leq x_3 \leq 2$ $1 \leq x_4 \leq 3$ $2 \leq x_6 \leq 3$
6	$M_6 = ((7,1),(8,1))$	$D_6: 0 \leq x_5 \leq \infty$
7	$M_7 = ((5,1),(6,1),(7,1))$	$D_7: 1 \leq x_4 \leq 3$ $2 \leq x_6 \leq 3$
8	$M_8 = ((5,1),(6,1),(7,1))$	$D_8: 0 \leq x_4 \leq 3$ $0 \leq x_6 \leq 3$ $x_4 - x_6 \leq 1$ $x_6 - x_4 \leq 2$
9	$M_9 = ((3,1),(4,1),(5,1),(6,1))$	$D_9: 0 \leq x_3 \leq 0$ $1 \leq x_4 \leq 3$ $2 \leq x_6 \leq 3$
10	$M_{10} = ((1,1),(2,1))$	$D_{10}: 0 \leq x_1 \leq 2$ $0 \leq x_2 \leq 3$
11	$M_{11} = (2,1),(3,1),(4,1),(5,1)$	$D_{11}: 0 \leq x_2 \leq 3$ $0 \leq x_3 \leq 2$
12	$M_{12} = ((2,1),(5,1),(7,1))$	$D_{12}: 0 \leq x_2 \leq 3$

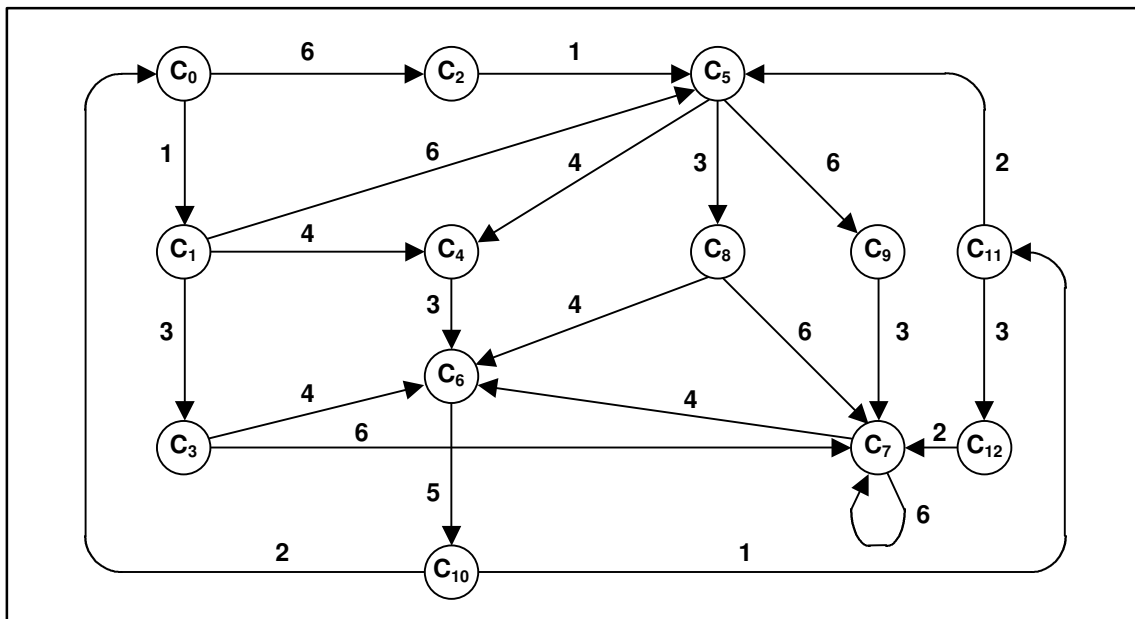


Figura 8.3. Grafo das classes de estados para a RPTM da figura 8.1.

CAPÍTULO 9: REDES DE PETRI COLORIDAS

9.1. INTRODUÇÃO

As redes de Petri **Coloridas (RPC)** constituem uma das chamadas **redes de Petri de alto nível**. As redes de Petri **Predicado/Transição (RPPr/T)** foram o primeiro tipo de rede de alto nível, construídas sem qualquer aplicação particular em mente [52]. RPPr/T são uma generalização das redes Lugar/Transição, relacionando-se às mesmas de uma maneira formal, tornando possível generalizar a maior parte dos conceitos e métodos de análise que foram desenvolvidos para tais redes.

Entretanto, logo se tornaram aparentes alguns problemas técnicos de RPPr/T, quando os métodos de **análise de invariantes de lugar e de transição** são generalizados para esse modelo de rede. É possível calcular invariantes para RPPr/T, mas a interpretação dos mesmos é difícil e deve ser feita com muito cuidado, para evitar resultados errôneos. Na tentativa de superar este problema, **Kurt Jensen** definiu a primeira versão de redes de Petri **Coloridas**, em **1981 (RPC-81)** [53]. As idéias principais desse modelo foram diretamente inspiradas em RPPr/T, com os arcos da rede sendo rotulados com **funções** ao invés de **expressões** (como em RPPr/T), significando que os invariantes passariam a ser interpretados sem problemas.

No entanto, tornou-se patente que as funções presentes nos arcos das RPC-81 são mais difíceis de ler e entender do que as expressões anexadas aos arcos das RPPr/T. O forte relacionamento entre os dois modelos de rede levou à idéia da construção de um modelo melhorado, combinando as qualidades de ambos. Este modelo foi definido por K. Jensen [54], sendo chamado redes de Petri de **Alto Nível (RPAN)**. Entretanto, este nome deu origem a muita confusão, uma vez que o termo "**redes de alto nível**" começou a ser usado na época como um nome genérico para **RPPr/T**, **RPC-81**, **RPAN** e muitos outros modelos de rede capazes de diferenciar qualitativamente seus *tokens*. Para evitar confusão, foi necessário mudar o nome de RPAN para redes de Petri **Coloridas (RPC)**.

RPC têm duas diferentes representações. A **representação com expressões** usa expressões nos arcos e guardas (predicados) nas transições, enquanto a **representação com funções** usa funções lineares com multi-conjuntos. Há translações formais entre as duas representações. A representação com expressões constitui-se num modelo próximo às RPPr/T, enquanto a representação com funções aproxima-se às RPC-81.

Atualmente, a maior parte das aplicações práticas de redes de Petri usa RPPr/T ou RPC. Há pequenas diferenças entre os dois tipos de rede. As principais diferenças encontram-se escondidas nos métodos utilizados para calcular invariantes de lugar e de transição [55].

Muitas outras extensões de rede de Petri de alto nível foram definidas. A maior parte delas é similar às RPC, mas utilizando diferentes linguagens de descrição. Informações importantes sobre redes de alto nível, artigos, métodos de análise e aplicações podem ser encontradas em [56,57,58,59].

A seguir, as RPC serão tratadas em sua representação com funções.

Para introduzir as RPC, será utilizado um exemplo apresentado por K. Jensen [55], no qual é modelado um pequeno **sistema de banco de dados (SBD) distribuído** (figura 9.1).

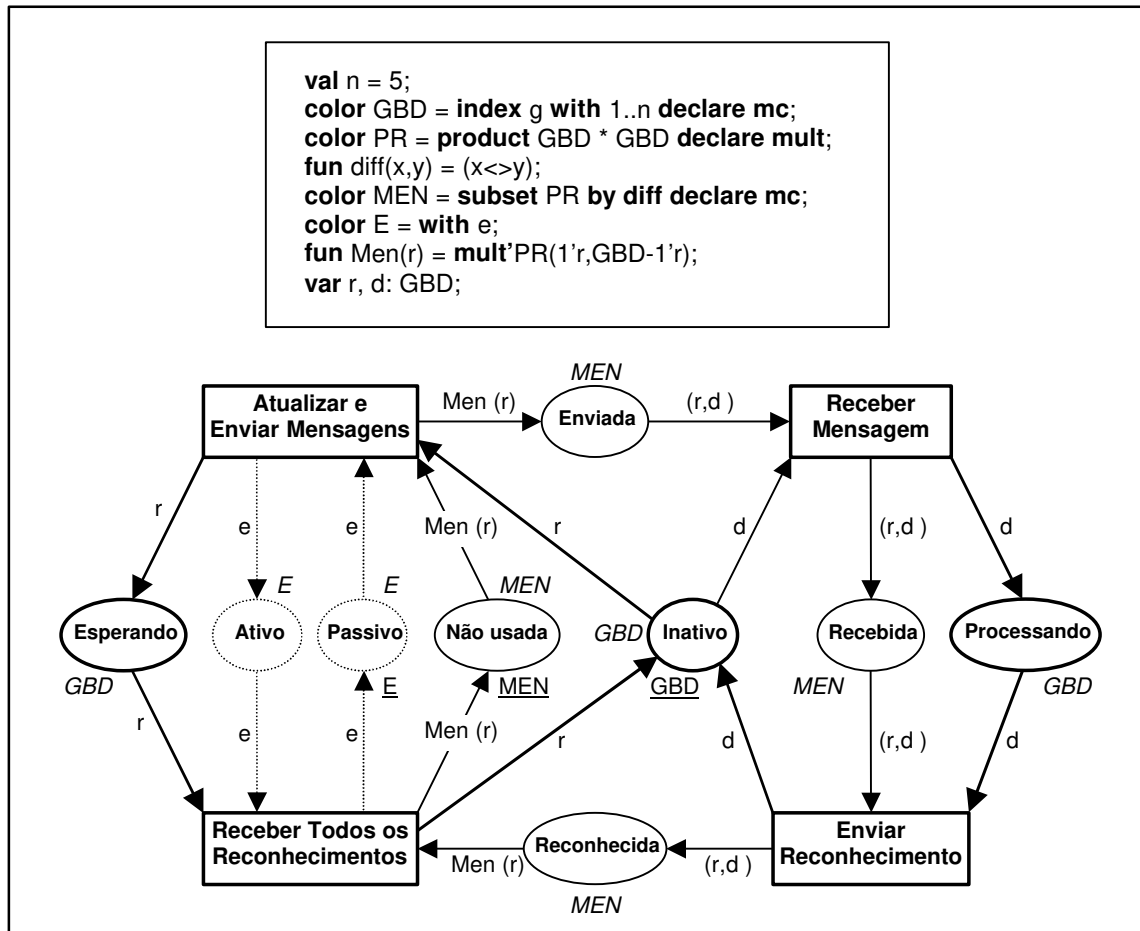


Figura 9.1. Modelo RPC de um sistema de banco de dados distribuído.

O SBD tem **n** diferentes lugares geográficos ($n \geq 3$). Cada lugar contém uma cópia completa da base de dados, a qual é gerenciada por um gerenciador de banco de dados (GBD) local. Quando um GBD g_i faz uma atualização sobre sua cópia da base de dados, ele deve enviar uma mensagem para todos os outros GBD's, a fim de garantir a consistência das **n** cópias da base de dados. Não levando em conta o conteúdo de cada mensagem, cada qual pode ser representada por um par (r,d) , onde **r** representa o remetente e **d** representa o destinatário. Isto significa que o GBD g_i envia as seguintes mensagens:

$$\mathbf{Men}(g_i) = \{(g_i, g_1), (g_i, g_2), \dots, (g_i, g_{i-1}), (g_i, g_{i+1}), \dots, (g_i, g_{n-1}), (g_i, g_n)\}$$

Os estados da RPC são representados por **lugares** (desenhados como elipses). No SBD há nove estados diferentes. Três dos lugares representam três possíveis estados dos GBD's: **Inativo**, **Esperando** e **Processando**. Quatro dos lugares representam os possíveis estados das mensagens: **Não Usada**, **Enviada**, **Recebida** e **Reconhecida**. Os dois lugares remanescentes indicam se uma atualização está sendo feita ou não: **Ativo** e **Passivo**. Por convenção, os nomes são escritos dentro das elipses. Os nomes não têm qualquer significado formal, possuindo, porém, grande importância prática no que diz respeito à compreensão do modelo.

Observação similar aplica-se à aparência dos lugares, isto é, a grossura da linha, cor, tamanho etc.

Cada lugar tem um **tipo** de dados associado (escrito em itálico, próximo ao lugar), determinando quais dados o lugar pode conter. O quadro no topo da figura 9.1 contém declarações. As primeiras seis linhas especificam os possíveis valores dos quatro diferentes tipos: **GBD**, **PR**, **MEN** e **E**. Implicitamente, as declarações também especificam as operações que podem ser efetuadas sobre os valores dos tipos. Para maiores explicações sobre a linguagem de anotação, recomenda-se a referência [57].

Para RPC, utilizam-se os seguintes termos: **tipo**, **valor**, **operação**, **expressão**, **variável**, **amarração** e **cálculo** da mesma maneira como são utilizados em linguagens de programação funcionais. É possível especificar os tipos de muitas outras formas como, por exemplo, através das especificações que são usadas em tipos abstratos de dados. Cada *token* (senha) traz consigo um valor de dados que pertence ao tipo de dados do lugar correspondente. Os lugares **Inativo**, **Esperando** e **Processando** têm o tipo:

$$\mathbf{GBD} = \{g_1, g_2, \dots, g_n\}.$$

Intuitivamente, isto significa que cada *token*, nestes lugares, representa um GBD. Na marcação inicial, todos os GBD's estão inativos, o que é representado pela presença de *n* *tokens* em **Inativo** e nenhum *token* em **Esperando** e **Processando**. Em geral, um lugar pode conter dois ou mais *tokens* com o mesmo valor de dados. Isto significa que se tem que trabalhar com conjuntos de multiplicidade maior de 1, conhecidos por **multi-conjuntos**. Uma marcação de uma RPC é uma função que mapeia cada lugar em um multi-conjunto de *tokens* do tipo correto. A marcação inicial M_0 é especificada por meio de **expressões de inicialização** (sublinhadas próximo aos lugares). A ausência de uma expressão de inicialização implica que a marcação inicial do lugar correspondente é vazia. Para os lugares do tipo **GBD** tem-se a seguinte marcação inicial:

$$\underline{M_0(\mathbf{Inativo})} = \mathbf{GBD}$$

$$\underline{M_0(\mathbf{Esperando})} = \underline{M_0(\mathbf{Processando})} = \emptyset,$$

onde $\emptyset$ denota o multi-conjunto vazio. Por convenção usa-se **GBD** para denotar o tipo, porém pode ser usado também para denotar o conjunto que contém todos os valores de dados do tipo e o multi-conjunto que contém exatamente uma ocorrência de cada valor de dados do tipo. Conseqüentemente, tem-se:

$$\underline{M_0(\mathbf{Inativo})} = 1'g_1 + 1'g_2 + \dots + 1'g_n,$$

onde o coeficiente inteiro (antes de ') indica que **Inativo** tem um *token* para cada valor de dados g_i no tipo **GBD**.

Não Usada, **Enviada**, **Recebida** e **Reconhecida** têm o tipo **MEN**, o qual é um subtipo do produto cartesiano $\mathbf{GBD} \times \mathbf{GBD}$. Intuitivamente, isto significa que cada *token* nestes lugares representam uma mensagem (*r,d*). Na marcação inicial, todas as mensagens são não usadas. Conseqüentemente, tem-se:

$$\underline{M_0(\mathbf{Não Usada})} = \mathbf{MEN} = \{(r,d) \in \mathbf{GBD} \times \mathbf{GBD} \mid r \neq d\}$$

$$\underline{M_0(\mathbf{Enviada})} = \underline{M_0(\mathbf{Recebida})} = \underline{M_0(\mathbf{Reconhecida})} = \emptyset,$$

onde a exigência $r \neq d$ indica que o remetente e o destinatário de uma mensagem são distintos.

Ativo e Passivo têm o tipo $E = \{e\}$. Sua marcação inicial é:

$$M_0(\text{Passivo}) = E = 1'e$$

$$M_0(\text{Ativo}) = \emptyset.$$

Por razões históricas, a teoria de RPC refere-se aos valores de *tokens* como **cores** de *tokens* e aos tipos de dados como **conjuntos de cores**. Os tipos de uma RPC podem ser arbitrariamente complexos como, por exemplo, um registro no qual um campo é um real, outro uma *string* de texto e um terceiro uma lista de inteiros. Assim, é muito mais adequado imaginar um *continuum* de cores ao invés de um conjunto discreto de cores.

As ações de uma RPC são representadas por meio de **transições** (representadas por retângulos). No SBD da figura 9.1, há quatro transições diferentes. Um arco com origem em um lugar e término em uma transição indica que a transição pode remover *tokens* do respectivo lugar, enquanto um arco saindo de uma transição para um lugar indica que a transição pode adicionar *tokens*. O número exato de *tokens* e seus valores de dados são determinados pelas **expressões de arcos** (junto aos arcos). A transição **Atualizar e Enviar Mensagens (EM)** tem seis arcos com três diferentes expressões de arcos: **e**, **r** e **Men(r)**. A primeira delas é constante. As duas outras possuem uma variável livre **r** do tipo **GBD**, declarada na última linha das declarações. Para falar sobre o disparo de uma transição **EM** necessita-se amarrar (associar) **r** a um valor de **GBD**. Caso contrário, não se pode calcular as expressões **r** e **Men(r)**, sendo que esta última mapeia cada **GBD r** às mensagens que **r** envia.

Assumindo que a variável **r** é amarrada ao valor **g₂**, para a transição **EM**, tem-se a seguinte notação: **(EM, <r = g₂>)**, a qual é nomeada de **elemento de amarração**. Para este elemento de amarração, calculam-se as expressões de arcos como segue (daqui por diante será assumido que **n = 5 GBD's**):

$$e \rightarrow e$$

$$r \rightarrow g_2$$

$$\text{Men}(r) \rightarrow 1'(g_2, g_1) + 1'(g_2, g_3) + 1'(g_2, g_4) + 1'(g_2, g_5).$$

Isto significa que o disparo da transição **EM** com a amarração **<r = g₂>**, removerá um *token* com um valor **e** de **Passivo** e adicionará um *token* com valor **e** em **Ativo**. O disparo removerá, também, um *token* com valor **g₂** de **Inativo** e adicionará um *token* de igual valor em **Esperando**. Finalmente, removerá quatro *tokens* de **Não Usada** (com os valores especificados pelo cálculo de **Men(r)**) e adicionará quatro *tokens* com estes valores em **Enviada**. Intuitivamente, isto significa que o **GBD g₂** muda do estado passivo para ativo e, simultaneamente, envia uma mensagem para cada um dos outros **GBD's**.

O disparo do elemento de amarração **(EM, <r = g₂>)** é possível se os seis *tokens* a serem removidos existem, isto é, se **Passivo** tem um *token e*, **Inativo** tem um *token g₂* e **Não Usada** tem os quatro *tokens* especificados por **Men(g₂)**. Este é o caso da marcação inicial **e**, assim, diz-se que o elemento de amarração **(EM, <r = g₂>)** está **habilitado** em **M₀**. É fácil verificar que todas as cinco possíveis amarrações para **EM** geram elementos de amarração habilitados em **M₀**.

As outras três transições trabalham de forma similar. As duas transições à direita usam duas variáveis diferentes, **r** e **d**, tomadas de **GBD**. Na marcação inicial, estão desabilitadas todas as possíveis amarrações para as transições **Receber Mensagem (RM)**, **Receber Todos os Reconhecimentos (RT)** e **Enviar Reconhecimento (ER)**. Para **RM**, não há *token* em **Enviada**. Para **ER** e **RT**, não há *token* em qualquer de seus lugares de entrada. Assim,

conclui-se que a única coisa que pode acontecer em M_0 é o disparo de um dos cinco elementos de amarração de **EM**. Cada um deles está habilitado e a escolha entre qual deles deva disparar é não-determinística. Tão logo um elemento de amarração dispare, os outros se tornam desabilitados devido à ausência de um *token* em **Passivo**. Neste caso, diz-se que os elementos de amarração estão em **conflito** entre si.

Suponha que o elemento de amarração ($EM, \langle r = g_2 \rangle$) dispare, passando a rede para uma marcação M_1 , com os seguintes *tokens* (lista-se apenas os lugares não vazios):

$$M_1(\text{Inativo}) = GBD - 1'g_2 = 1'g_1 + 1'g_3 + 1'g_4 + 1'g_5$$

$$M_1(\text{Esperando}) = 1'g_2$$

$$M_1(\text{Enviada}) = Men(g_2) = 1'(g_2, g_1) + 1'(g_2, g_3) + 1'(g_2, g_4) + 1'(g_2, g_5)$$

$$M_1(\text{Não Usada}) = MEN - Men(g_2)$$

$$M_1(\text{Ativo}) = 1'e.$$

Na marcação M_1 a transição **RM** está habilitada para todas as quatro possíveis amarrações na qual r está limitada a g_2 enquanto d está restrito aos outros quatro valores de **GBD** distintos de g_2 . Isto significa que todos os outros gerenciadores estão prontos para receber as mensagens enviadas por g_2 . O elemento de amarração ($RM, \langle r = g_2, d = g_3 \rangle$) move um *token* com valor (g_2, g_3) de **Enviada** para **Recebida** e move um *token* com valor g_3 de **Inativo** para **Processando**. Isto significa que o GBD g_3 muda do estado inativo para o de processando e, simultaneamente, a mensagem (g_2, g_3) muda do estado enviada para o de recebida. Analogamente o elemento de amarração ($RM, \langle r = g_2, d = g_4 \rangle$) move um *token* com valor (g_2, g_4) de **Enviada** para **Recebida** e move um *token* com valor g_4 de **Inativo** para **Processando**. Os dois elementos de amarração tratam de *tokens* diferentes e, conseqüentemente, não influem um no outro. Isto significa que eles podem disparar **concorrentemente** um com outro, isto é, no mesmo **passo**. É fácil verificar que todos os quatro elementos de amarração de **RM** são concorrentes entre si. Isto significa que o próximo passo pode conter todos eles ou qualquer subconjunto não-vazio deles. Novamente, a escolha é não-determinística.

As relações de conflito e de concorrência entre elementos de amarração são dependentes de marcação. Assim, uma situação de conflito pode tornar-se numa de concorrência com a simples adição de *tokens*. Por exemplo, pode-se mudar a marcação inicial de **Passivo** de $1'e$ para $3'e$, permitindo, desta forma, passos onde até três elementos de amarração de **RM** disparem concorrentemente. É também permitido que um elemento de amarração possa ser concorrente com ele próprio, uma ou mais vezes. Isto significa que, em geral, um passo é um multi-conjunto de elementos de amarração.

Suponha que o passo:

$$1'(RM, \langle r = g_2, d = g_3 \rangle) + 1'(RM, \langle r = g_2, d = g_4 \rangle)$$

dispare na marcação M_1 , gerando a marcação M_2 :

$$M_2(\text{Inativo}) = M_1(\text{Inativo}) - (1'g_3 + 1'g_4) = 1'g_1 + 1'g_5$$

$$M_2(\text{Esperando}) = 1'g_2$$

$$M_2(\text{Processando}) = 1'g_3 + 1'g_4$$

$$M_2(\text{Enviada}) = M_1(\text{Enviada}) - (1'(g_2, g_3) + 1'(g_2, g_4)) = 1'(g_2, g_1) + 1'(g_2, g_5)$$

$$M_2(\text{Recebida}) = 1'(g_2, g_3) + 1'(g_2, g_4)$$

$$M_2(\text{Não Usada}) = MEN - Men(g_2)$$

$$M_2(\text{Ativo}) = 1'e.$$

Em M_2 , há quatro elementos de amarração habilitados:

$$(RM, \langle r = g_2, d = g_1 \rangle)$$

$$(RM, \langle r = g_2, d = g_5 \rangle)$$

$$(ER, \langle r = g_2, d = g_3 \rangle)$$

$$(ER, \langle r = g_2, d = g_4 \rangle).$$

Os quatro elementos de amarração são concorrentes entre si, uma vez que utilizam *tokens* diferentes. Suponha que os três primeiros disparem tendo, assim, o passo:

$$1'(RM, \langle r = g_2, d = g_1 \rangle) + 1'(RM, \langle r = g_2, d = g_5 \rangle) + 1'(ER, \langle r = g_2, d = g_3 \rangle)$$

A RPC passaria para uma marcação M_3 :

$$M_3(\text{Inativo}) = (M_2(\text{Inativo}) - (1'g_1 + 1'g_5)) + 1'g_3 = 1'g_3$$

$$M_3(\text{Esperando}) = 1'g_2$$

$$M_3(\text{Processando}) = (M_2(\text{Processando}) - 1'g_3) + (1'g_1 + 1'g_5) \\ = 1'g_1 + 1'g_4 + 1'g_5$$

$$M_3(\text{Enviada}) = M_2(\text{Enviada}) - (1'(g_2, g_1) + 1'(g_2, g_5)) = \emptyset$$

$$M_3(\text{Recebida}) = (M_2(\text{Recebida}) - 1'(g_2, g_3)) + (1'(g_2, g_1) + 1'(g_2, g_5)) \\ = 1'(g_2, g_1) + 1'(g_2, g_4) + 1'(g_2, g_5)$$

$$M_3(\text{Reconhecida}) = 1'(g_2, g_3)$$

$$M_3(\text{Não Usada}) = MEN - Men(g_2)$$

$$M_3(\text{Ativo}) = 1'e.$$

Em M_3 , há três elementos de amarração habilitados:

$$(ER, \langle r = g_2, d = g_1 \rangle)$$

$$(ER, \langle r = g_2, d = g_4 \rangle)$$

$$(ER, \langle r = g_2, d = g_5 \rangle).$$

O disparo concorrente desses três elementos de amarração leva a uma marcação M_4 :

$$M_4(\text{Inativo}) = 1'g_1 + 1'g_3 + 1'g_4 + 1'g_5$$

$$M_4(\text{Esperando}) = 1'g_2$$

$$M_4(\text{Reconhecida}) = 1'(g_2, g_1) + 1'(g_2, g_3) + 1'(g_2, g_4) + 1'(g_2, g_5) = Men(g_2)$$

$$M_4(\text{Não Usada}) = MEN - Men(g_2)$$

$$M_4(\text{Ativo}) = 1'e.$$

Em M_4 , há apenas um elemento de amarração habilitado:

$$(RT, \langle r = g_2 \rangle).$$

O disparo deste elemento de amarração leva à marcação inicial M_0 . As marcações e passos considerados anteriormente formam uma **seqüência de disparos**:

$$M_0 [P_0 > M_1 [P_1 > M_2 [P_2 > M_3 [P_3 > M_4 [P_4 > M_0.$$

A seqüência de disparos adotada é apenas uma dentre muitas outras.

Para garantir a consistência entre as diferentes cópias da base de dados, o modelo simplesmente permite apenas uma atualização por vez. Quando um GBD inicializa uma atualização, a mesma tem que ser realizada por todos os outros GBD's antes que qualquer

outra atualização possa ser iniciada. A exclusão mútua é garantida pelo lugar **Passivo**. A descrição do SBD é muito alto nível (e fora da realidade), no sentido que a exclusão mútua é descrita através de um mecanismo global (o lugar **Passivo**). Para implementar o SBD em *hardware* distribuído, a exclusão mútua pode ser tratada através de um "mecanismo distribuído". Tal implementação poderia ser descrita por uma RPC mais detalhada.

A RPC para o SBD tem vários lugares redundantes, os quais poderiam ser omitidos sem mudar o comportamento do sistema. Como exemplo, poder-se-ia omitir o lugar **Não Usada**. Assim, haveria somente a representação explícita daquelas mensagens que estão correntemente em uso (lugares **Enviada**, **Recebida** e **Reconhecida**). Da mesma forma, poder-se-ia omitir os lugares **Ativo** e **Processando**. A redundância de lugares pode ser justificada pela melhor clareza do modelo, ficando, portanto, a cargo do responsável pela modelagem do sistema.

A atribuição de valores de dados a cada *token* permite o uso de muito menos lugares do que aqueles necessários por uma RP de baixo nível. Para o SBD, uma RP clássica teria n lugares para cada um dos estados **Inativo**, **Esperando** e **Processando**, uma vez que esta seria a única maneira da RP clássica distinguir entre os n diferentes GBD's. Analogamente, seriam necessários $|MEN| = n^2 - n$ lugares diferentes para os estados referentes às mensagens (**Não Usada**, **Enviada**, **Recebida** e **Reconhecida**). Desta forma, uma RP clássica necessitaria de 97 lugares para representar um SBD com 5 GBD's, enquanto uma RPC necessita apenas de 9 lugares.

O uso de variáveis nas expressões dos arcos significa que cada transição de uma RPC pode disparar de várias maneiras. Assim, pode-se utilizar uma única transição para descrever uma classe de atividades relacionadas, enquanto numa RP clássica seria necessária uma transição para cada uma dessas atividades. No SBD, há somente uma transição **EM** e somente uma transição **RT**. Na RP clássica haveria uma transição de cada uma dessas para cada GBD. Analogamente, haveria uma transição **RM** e uma transição **ER** para cada elemento de **MEN**. Assim, com 5 GBD's, a RP clássica teria 50 transições, enquanto a RPC tem apenas 4.

Em adição às expressões de arcos, é possível anexar uma expressão booleana (baseada nas variáveis) a cada transição. Tal expressão é chamada **guarda**. Esta especifica que se deve aceitar elementos de amarração para os quais a expressão booleana seja verdadeira. No SBD, poder-se-ia adicionar a guarda:

$$[r \neq g_3]$$

à transição **EM**. Tal guarda significaria a exclusão de $(EM, \langle r = g_2 \rangle)$ do conjunto de elementos de amarração e, conseqüentemente, não seria possível o GBD g_3 iniciar uma atualização.

Após esta apresentação informal, prossegue-se a definição formal de RPC não hierárquicas.

9.2. DEFINIÇÃO FORMAL

RPC são matematicamente definidas por uma tupla. Em princípio, é fácil transladar um diagrama RPC em uma tupla RPC e vice-versa.

Inicialmente, será apresentada a definição de multi-conjuntos, onde **IN** denota o conjunto dos inteiros não-negativos.

Definição 9.1: Multi-Conjunto.

Um **multi-conjunto** $\mathbf{m}$, sobre um conjunto não-nulo $\mathbf{S}$, é uma função $\mathbf{m}:\mathbf{S}\rightarrow\mathbf{IN}$ representando a soma formal:

$$\sum_{s\in\mathbf{S}} \mathbf{m}(s)s.$$

Denota-se $\mathbf{S}_{\mathbf{MC}}$ o conjunto de todos os multi-conjuntos sobre $\mathbf{S}$.

Os inteiros não-negativos $\{\mathbf{m}(s): s\in\mathbf{S}\}$ são os coeficientes do multi-conjunto.

Se, e somente se, $\mathbf{m}(s) \neq 0, s\in\mathbf{m}$.

Há uma correspondência 1-1 entre conjuntos sobre $\mathbf{S}$ e os multi-conjuntos para os quais todos os coeficientes são zero ou um. Será utilizado $\mathbf{A} \subseteq \mathbf{S}$ para denotar o multi-conjunto que consiste de uma única ocorrência para cada elemento em $\mathbf{A}$. Analogamente, será usado um elemento $s\in\mathbf{S}$ para denotar o multi-conjunto $1's$.

Existe um conjunto de operações padrão para multi-conjuntos (com exceção de $|\mathbf{m}|$, todas as demais são derivadas das operações padrão para funções).

Definição 9.2: Operações sobre Multi-Conjuntos.

Comparação, adição, multiplicação escalar, e tamanho de multi-conjuntos são definidos como segue, para todo $\mathbf{m}, \mathbf{m}_1, \mathbf{m}_2 \in \mathbf{S}_{\mathbf{MC}}$ e todo $\mathbf{n} \in \mathbf{IN}$:

- i) $\mathbf{m}_1 \neq \mathbf{m}_2 = \exists s \in \mathbf{S}: \mathbf{m}_1(s) \neq \mathbf{m}_2(s)$ (comparação, onde R
 $\mathbf{m}_1 R \mathbf{m}_2 = \forall s \in \mathbf{S}: \mathbf{m}_1(s) R \mathbf{m}_2(s)$ pode ser $\leq, \geq$ ou $=$);
- ii) $\mathbf{m}_1 + \mathbf{m}_2 = \sum_{s \in \mathbf{S}} (\mathbf{m}_1(s) + \mathbf{m}_2(s))s$ (adição)
 $\mathbf{m}_1 - \mathbf{m}_2 = \sum_{s \in \mathbf{S}} (\mathbf{m}_1(s) - \mathbf{m}_2(s))s$ (subtração: $\mathbf{m}_1 \geq \mathbf{m}_2$);
- iii) $\mathbf{n} * \mathbf{m} = \sum_{s \in \mathbf{S}} (\mathbf{n} * \mathbf{m}(s))s$ (multiplicação escalar);
- iv) $|\mathbf{m}| = \sum_{s \in \mathbf{S}} \mathbf{m}(s)$ (tamanho).

Obs.: Quando $|\mathbf{m}| = \infty$ diz-se que $\mathbf{m}$ é infinito. Caso contrário, $\mathbf{m}$ é finito.

As operações com multi-conjuntos têm um grande número de propriedades algébricas padrão. Maiores detalhes podem ser encontrados em [57].

Para dar uma definição abstrata de RPC, não é necessário fixar a sintaxe na qual o modelador especifica as expressões da rede. A referência [55] assume que tal sintaxe existe juntamente com uma semântica bem definida, tornando possível falar, de uma forma não ambígua, sobre:

- a) Os elementos de um tipo $\mathbf{T}$. O conjunto de todos os elementos em $\mathbf{T}$ é denotado pelo próprio nome $\mathbf{T}$;
- b) O tipo de uma variável $\mathbf{v}$ – denotado por **Tipo**($\mathbf{v}$);
- c) O tipo de uma expressão **expr** – denotado por **Tipo**(**expr**);
- d) O conjunto de variáveis em uma expressão **expr** – denotado por **Var**(**expr**);

- e) Uma amarração de um conjunto de variáveis V – associa-se a cada variável $v \in V$ um elemento $b(v) \in \text{Tipo}(v)$;
- f) O valor obtido pelo cálculo de uma expressão expr em uma amarração b – denotado por $\text{expr} \langle b \rangle$. Exige-se que $\text{Var}(\text{expr})$ seja um subconjunto de variáveis de b , sendo o cálculo executado através da substituição de cada variável $v \in \text{Var}(\text{expr})$ pelo valor $b(v) \in \text{Tipo}(v)$, determinado pela amarração.

Uma expressão sem variáveis é dita uma **expressão fechada**. Ela pode ser calculada para todas as amarrações e cada cálculo fornece sempre o mesmo valor. Neste caso, o valor obtido pelo cálculo de expr em uma amarração b será denotado apenas por expr .

Será utilizado B para denotar o tipo booleano ($B = \{\text{falso}, \text{verdadeiro}\}$), deste ponto em diante. Além disso, a notação $\text{Tipo}(v)$ será estendida para $\text{Tipo}(A) = \{\text{Tipo}(v) \mid v \in A\}$, onde A é um conjunto de variáveis.

Segue-se a definição de RPC. Algumas motivações e explanações de partes individuais da definição são dadas imediatamente após a definição.

Definição 9.3: Representação Matemática de RPC.

Uma **RPC** é uma tupla $RPC = (\Sigma, L, T, A, N, C, G, E, I)$, onde:

- i) Σ é um conjunto finito não-nulo de **tipos**, também chamados conjuntos de cores;
- ii) L é um conjunto finito de **lugares**;
- iii) T é um conjunto finito de **transições**;
- iv) A é um conjunto finito de **arcos**, de forma que:

$$L \cap T = L \cap A = T \cap A = \emptyset$$

- v) N é uma função **nó**, definida de A em $L \times T \cup T \times L$;
- vi) C é uma função **cor**, definida de L em Σ ;
- vii) G é uma função **guarda**, definida de T em expressões, de forma que:

$$\forall t \in T: [\text{Tipo}(G(t)) = B \wedge \text{Tipo}(\text{Var}(G(t))) \subseteq \Sigma];$$

- viii) E é uma função de **expressão de arco**, definida de A em expressões, de forma que:

$$\forall a \in A: [\text{Tipo}(E(a)) = C(I)_{MC} \wedge \text{Tipo}(\text{Var}(E(a))) \subseteq \Sigma]$$

onde I é o lugar de $N(a)$;

- ix) I é uma função de **inicialização**, definida de L em expressões fechadas, de forma que:

$$\forall l \in L: [\text{Tipo}(I(l)) = C(l)_{MC}].$$

Observações:

- i) O conjunto de **tipos** determina os valores de dados e as operações e funções que podem ser usadas nas expressões da rede, isto é, expressões de arcos, guardas e expressões de inicialização. Assume-se que cada tipo possui pelo menos um elemento;
- ii) + iii) + iv) Os **lugares**, **transições** e **arcos** são descritos pelos conjuntos L , T e A , finitos e disjuntos dois a dois. A exigência de estes conjuntos serem finitos evita um grande número de problemas técnicos, tais como a possibilidade de existir um número infinito de arcos entre dois nós;

- v) A função **nó** mapeia cada arco em um par onde o primeiro elemento é o nó origem e o segundo elemento é o nó destino. Os dois nós devem ser de tipos diferentes, isto é, um deve ser um lugar e o outro uma transição. RPC permitem a existência de vários arcos entre o mesmo par ordenado de nós;
- vi) A função **cor** C mapeia cada lugar l a um tipo $C(l)$. Intuitivamente, isto significa que cada *token* em l deve ter um valor de dados que pertença a $C(l)$;
- vii) A função **guarda** G mapeia cada transição t em uma expressão booleana onde todas as variáveis têm tipos que pertençam a Σ . Expressões booleanas sempre verdadeiras são omitidas;
- viii) A função de **expressão de arco** E mapeia cada arco a em uma expressão do tipo $C(l)_{MC}$. Isto significa que cada expressão de arco deve ser calculada sobre os multi-conjuntos do tipo do lugar adjacente l ;
- ix) A função de **inicialização** I mapeia cada lugar l em uma expressão fechada que deve ser do tipo $C(l)_{MC}$. Funções de inicialização que resultam em vazio são omitidas.

Após a definição da estrutura das RPC, passa-se às considerações sobre seu comportamento. Para tal, inicialmente, será introduzida a seguinte notação para todo $t \in T$ e todo par de nós $(x_1, x_2) \in (L \times T \cup T \times L)$:

- a) $A(t) = \{a \in A \mid N(a) \in L \times \{t\} \cup \{t\} \times L\}$;
- b) $Var(t) = \{v \mid v \in Var(G(t)) \vee \exists a \in A(t): v \in Var(E(a))\}$;
- c) $A(x_1, x_2) = \{a \in A \mid N(a) = (x_1, x_2)\}$;
- d) $E(x_1, x_2) = \Sigma_{a \in A(x_1, x_2)} E(a)$.

A soma em d) indica adição de expressões. A partir dos argumentos, está sempre claro se trata-se da função $E \in [A \rightarrow Expr]$ ou da função $E \in [(L \times T \cup T \times L) \rightarrow Expr]$. Uma observação similar aplica-se à A , $A(t)$ e $A(x_1, x_2)$.

A seguir, definem-se amarrações. Deve-se notar que ii) implica que todas as amarrações satisfazem a guarda correspondente ($G(t) \langle b \rangle$ denota o cálculo da expressão guarda $G(t)$ na amarração b).

Definição 9.4: Amarração.

Uma **amarração** de uma transição t é uma função b definida sobre $Var(t)$, tal que:

- i) $\forall v \in Var(t): b(v) \in Tipo(v)$;
- ii) $G(t) \langle b \rangle$.

Denota-se $B(t)$ o conjunto de todas as amarrações para t .

Conforme mostrado na seção 9.1, amarrações são escritas geralmente na forma $\langle v_1 = c_1, v_2 = c_2, \dots, v_n = c_n \rangle$, onde $Var(t) = \{v_1, v_2, \dots, v_n\}$. A ordem das variáveis não tem importância.

Definição 9.5: Um **elemento de token** é um par (l, c) onde $l \in L$ e $c \in C(l)$, enquanto um **elemento de amarração** é um par (t, b) onde $t \in T$ e $b \in B(t)$. O conjunto de todos os elementos de *token* é denotado por ET enquanto o conjunto de todos os elementos de amarração é denotado por EA .

Uma **marcação** é um multi-conjunto sobre **ET** enquanto um **passo** é um multi-conjunto não-nulo e finito sobre **EA**. A marcação inicial **M₀** é a marcação obtida do cálculo das expressões de inicilização:

$$\forall (l,c) \in ET: M_0(l,c) = (I(l))(c).$$

Os conjuntos de todas as marcações e passos são denotados por **M** e **P**, respectivamente.

Cada marcação **M** ∈ **ET_{MC}** determina uma única função **M*** definida sobre **L**, tal que **M*(l) ∈ C(l)_{MC}**:

$$\forall l \in L, \forall c \in C(l): (M^*(l))(c) = M(l,c).$$

Por outro lado, cada função **M*** definida sobre **L**, tal que **M*(l) ∈ C(l)_{MC}** para todo **l** ∈ **L**, determina uma única marcação **M**:

$$\forall (l,c) \in ET: M(l,c) = (M^*(l))(c).$$

Desta forma, marcações são geralmente representadas por funções definidas sobre **L**.

A seguir, é dada a definição formal de habilitação e disparo.

Definição 9.6: Um passo **P** está habilitado em uma marcação **M** se, e somente se, a seguinte propriedade for satisfeita:

$$\forall l \in L: \sum_{(t,b) \in P} E(l,t) \langle b \rangle \leq M(l).$$

Então, diz-se que **(t,b)** está habilitado e que **t** está habilitada. Se **P** tiver mais de um elemento, diz-se que os mesmos estão habilitados concorrentemente. Quando um passo **P** está habilitado em uma marcação **M₁** ele pode disparar, mudando a marcação da rede de **M₁** para **M₂**, a qual é definida por:

$$\forall l \in L: M_2(l) = (M_1(l) - \sum_{(t,b) \in P} E(l,t) \langle b \rangle) + \sum_{(t,b) \in P} E(t,l) \langle b \rangle.$$

Diz-se que **M₂** é diretamente alcançável de **M₁**, o que é denotado por:

$$M_1 [P > M_2.$$

O cálculo da expressão **E(l,t) ** fornece os *tokens* que são removidos de **l** quando **t** dispara com a amarração **b**. Somando-se todos os elementos de amarração **(t,b) ∈ P**, obtém-se todos os *tokens* que são removidos de **l** quando **P** dispara. Exige-se que este multi-conjunto seja menor ou igual à marcação de **l**. Isto significa que cada elemento de amarração **(t,b) ∈ P** deve ser capaz de obter os *tokens* especificados por **E(l,t) **, sem ter que compartilhar estes *tokens* como outros elementos de amarração de **P**.

O disparo de um passo é um evento indivisível. Embora a fórmula anterior exija que a subtração seja realizada antes da adição, não deve ser reconhecida a existência de uma marcação intermediária, onde os *tokens* da primeira soma teriam sido removidos, enquanto aqueles da segunda soma não tivessem sido ainda adicionados. Deve ser observado que, quando elementos de amarração estão habilitados concorrentemente, é possível o disparo de um passo que contenha apenas alguns deles.

Definição 9.7: Uma **seqüência de disparos finita** é uma seqüência de marcações e passos:

$$M_1 [P_1 > M_2 [P_2 > M_3 \dots M_n [P_n > M_{n+1}$$

onde $n \in \mathbb{N}$ e $M_1 [P_i > M_{i+1}$ para todo $i \in \{1, 2, \dots, n\}$, sendo M_i a **marcação inicial**, M_{n+1} a **marcação final** e n o **comprimento**.

Analogamente, uma **seqüência de disparos infinita** é uma seqüência de marcações e passos:

$$M_1 [P_1 > M_2 [P_2 > M_3 \dots$$

onde $M_1 [P_i > M_{i+1}$ para todo $i \geq 1$.

Uma marcação M'' é **alcançável** de uma marcação M' se, e somente se, existe uma **seqüência de disparos finita** começando em M' e terminando em M'' . O conjunto de marcações que são alcançáveis de M' é denotado por $[M' >$. Uma marcação é **alcançável** se, e somente se, pertence a $[M_0 >$.

Seqüências de disparo de tamanho zero são permitidas. Isto significa que $M \in [M >$ para toda marcação M .

9.3. PROPRIEDADES DINÂMICAS

As propriedades dinâmicas caracterizam o comportamento de uma dada RPC como, por exemplo, se é possível alcançar uma marcação na qual nenhum passo esteja habilitado. A seguir serão introduzidas algumas das mais importantes propriedades dinâmicas.

A propriedade de **limitabilidade** diz qual a quantidade máxima de *tokens* que um determinado lugar pode ter:

Definição 9.8: Dados um lugar $l \in L$, um inteiro não-negativo n e um multi-conjunto $m \in C(I)_{MC}$:

i) n é limite inteiro para l se, e somente se:

$$\forall M \in [M_0 >: |M(l)| \leq n;$$

ii) m é um limite multi-conjunto para l se, e somente se:

$$\forall M \in [M_0 >: M(l) \leq m.$$

Para o SBD (figura 9.1) com n GBD's, os limites dos lugares encontram-se na tabela 9.1.

Tabela 9.1. Limites inteiro e multi-conjunto do sistema de banco de dados.

Lugar	Limite Multi-Conjunto	Limite Inteiro
Inativo	GBD	n
Esperando	GBD	1
Processando	GBD	$n - 1$
Não Usada	MEN	$n^2 - n$
Enviada	MEN	$n - 1$
Recebida	MEN	$n - 1$
Reconhecida	MEN	$n - 1$
Passivo	E	1
Ativo	E	1

Observe que os limites inteiro e multi-conjunto são suplementares entre si. De um deles é geralmente possível deduzir informação que não pode ser deduzida do outro e vice-versa. Este é, por exemplo, o caso dos lugares **Inativo** e **Esperando**. Para **Esperando**, o limite inteiro fornece uma informação muito mais precisa do que o limite multi-conjunto. No caso de **Inativo** ocorre o contrário.

As propriedades **caseiras** dizem para quais marcações (ou conjuntos de marcações) é sempre possível de se retornar.

Definição 9.9: Marcação e Espaço Caseiros.

Dados uma marcação $M \in \mathbf{M}$ e um conjunto de marcações $X \subseteq \mathbf{M}$:

i) M é uma **marcação caseira** se, e somente se:

$$\forall M' \in [M_0>: M \in [M'>;$$

ii) X é um **espaço caseiro** se, e somente se:

$$\forall M' \in [M_0>: X \cap [M'> \neq \emptyset.$$

É fácil ver que M é uma marcação caseira se, e somente se, $\{M\}$ é um espaço caseiro. Note que a presença de uma marcação caseira afirma que é possível alcançar um espaço caseiro, entretanto, não garante que se alcance, uma vez que possam existir infinitas seqüências de disparos que não contenham tal espaço caseiro. Uma observação similar aplica-se a espaços caseiros.

Para o SBD é possível mostrar que M_0 é uma marcação caseira. Disto, conclui-se que toda marcação alcançável (decorrente) a partir de M_0 é também uma marcação caseira e, portanto, o conjunto das possíveis marcações da rede é um espaço caseiro.

A propriedade de **vivacidade** diz se um conjunto X de elementos de amarração permanece ativo. Se X permanecer ativo, significa que para toda marcação alcançável M' a partir de M_0 , existe uma seqüência de disparos iniciando em M' que contenha um elemento de X .

Definição 9.10: Dados uma marcação $M \in \mathbf{M}$ e um conjunto de elementos de amarração $X \subseteq \mathbf{EA}$:

i) M é **morta** se, e somente se, nenhum elemento de amarração estiver habilitado, isto é, se, e somente se:

$$\forall x \in \mathbf{EA}: \neg M[x>;$$

ii) X é **morto** em M se, e somente se, nenhum elemento de X pode tornar-se habilitado, isto é, se, e somente se:

$$\forall M' \in [M>, \forall x \in X: \neg M'[x>;$$

iii) X é **vivo** se, e somente se, não existe marcação alcançável na qual X é morto, isto é, se, e somente se:

$$\forall M' \in [M_0>, \exists M'' \in [M'> \text{ e } \exists x \in X: M''[x>.$$

A propriedade de vivacidade apenas exige que os elementos de X possam tornar-se habilitados. Assim, podem existir infinitas seqüências de disparos começando em M' e que não contenham elementos de X . Deve-se observar que **vivo** não é negação de **morto**. Cada conjunto vivo de elementos de amarração é não-morto, mas o oposto não é verdade. Para a transição $t \in \mathbf{T}$, usa-se $\mathbf{EA}(t) \subseteq \mathbf{EA}$ para denotar o conjunto de todos os elementos de

amarração que contenham t . Diz-se que t é viva ou morta se, e somente se, $EA(t)$ possui a respectiva propriedade. Diz-se, também, que t é **estritamente viva** se, e somente se, $\{x\}$ é vivo para todo $x \in EA(t)$.

As quatro transições do SBD são estritamente vivas. Isto pode ser verificado provando que a marcação inicial é uma marcação caseira e que existe uma seqüência de disparos iniciada em M_0 e que contém todos os elementos de amarração de EA .

As propriedades de **equitatividade** dizem como geralmente disparam os diferentes elementos de amarração. Para um conjunto de elementos de amarração $X \subseteq EA$ e uma seqüência infinita de disparos σ da forma:

$$\sigma = M_1 [P_1 > M_2 [P_2 > M_3 \dots$$

Usa-se $H_{X,i}(\sigma)$ para denotar o número de elementos de X que estão habilitados na marcação M_i . Analogamente, usa-se $D_{X,i}(\sigma)$ para denotar o número de elementos de X que disparam no passo P_i (quando um elemento dispara concorrentemente com ele mesmo isto é refletido na contagem).

Usa-se $H_X(\sigma)$ e $D_X(\sigma)$ para denotar, respectivamente, o número total de habilitações e disparos em σ , ou seja:

$$H_X(\sigma) = \sum_{i=1}^{\infty} H_{X,i}(\sigma) \quad D_X(\sigma) = \sum_{i=1}^{\infty} D_{X,i}(\sigma)$$

Como todos os elementos nos dois somatórios são inteiros não-negativos, é fácil ver que os somatórios devem ser convergentes – ou para um elemento de $\mathbf{IN}$ ou para ∞ .

Definição 9.11: Seja $X \subseteq EA$ um conjunto de elementos de amarração e σ uma seqüência de disparos infinita:

i) X é **imparcial** de σ se, e somente se, ele tiver infinitos disparos, isto é, se, e somente se:

$$D_X(\sigma) = \infty;$$

ii) X é **equitativo** à σ se, e somente se, um número infinito de habilitações implica um número infinito de disparos, isto é, se, e somente se:

$$H_X(\sigma) = \infty \Rightarrow D_X(\sigma) = \infty;$$

iii) X é **ajustado** por σ se, e somente se, uma habilitação persistente implica um disparo, isto é, se, e somente se:

$$\forall i \geq 1: [H_{X,i}(\sigma) \neq 0 \Rightarrow \exists k \geq i: [H_{X,k}(\sigma) = 0 \vee D_{X,k}(\sigma) \neq 0]].$$

Diz-se que uma transição $t \in T$ é **imparcial**, **equitativa** ou **ajustada** se, e somente se, $EA(t)$ possuir a respectiva propriedade. Diz-se que t é **estritamente imparcial** (**equitativa** / **ajustada**) se, e somente se, $\{x\}$ é imparcial (equitativo / ajustado) para todo $x \in EA(t)$.

Todas as transições do SBD da figura 9.1 são imparciais. EM é estritamente ajustada, ao passo que as outras três são estritamente equitativas.

9.4. GRAFO DE DISPARO

Esta seção trata de **grafos de disparo**, também conhecidos por **estado de espaços**, **grafos de alcançabilidade** ou **máquina de senhas**. A idéia básica por trás de grafos de disparo é construir um grafo com um nó para cada marcação alcançável e um arco para cada elemento de amarração. Obviamente, tal grafo pode tornar-se extremamente grande, mesmo para pequenas redes.

A figura 9.2 mostra um grafo de disparo para o SBD com 3 GBD's. Para economizar espaço, as marcações são representadas pela listagem dos GBD's que têm uma mensagem endereçadas para eles – em **Enviada**, **Recebida** ou **Reconhecida**, respectivamente. Isto significa que (2,3,-) denota uma marcação na qual **g₂** está **Inativo**, **g₃** está **Processando** e **g₁** está **Esperando**. Analogamente, (23,-,-) denota uma marcação na qual **g₂** e **g₃** estão em **Inativo**, enquanto **g₁** está **Esperando**. A marcação inicial é representada por (-,-,-).

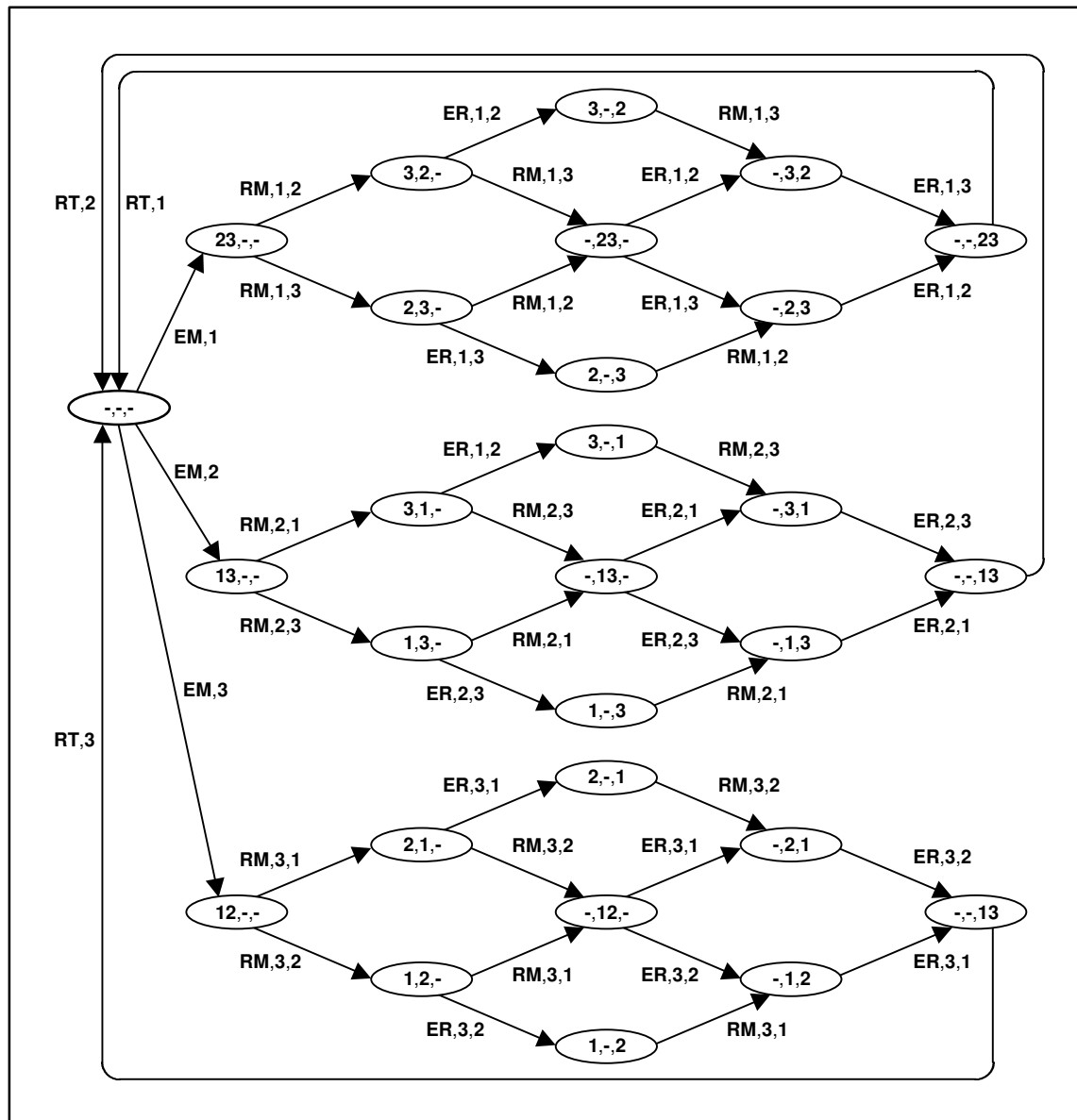


Figura 9.2. Grafo de disparo para o sistema de banco de dados com 3 gerenciadores.

Cada arco representa um disparo $M_1 [b > M_2]$ onde M_1 e M_2 são marcações alcançáveis, enquanto b é um elemento de amarração em M_1 . Adotou-se EM, i e RM, i, k ao invés de $(EM, <r = g_i >)$ e $(RM, <r = g_i, d = g_k >)$. Representação análoga foi adotada para RT e ER .

Deve-se observar que, em contraste com a teoria dos grafos, é permitido que um grafo de disparo tenha vários arcos entre o mesmo par ordenado de nós (por esta razão A é definido como um conjunto separado e não como um subconjunto de $V \times V$).

Diz-se que um arco a com $N(a) = (v_1, v_2)$ vai do **nó origem** v_1 ao **nó destino** v_2 , o que define duas funções: $o, d \in [A \rightarrow V]$. Para cada arco, a primeira função mapeia para o seu nó origem, enquanto a segunda mapeia para o seu nó destino.

Definição 9.12: Grafo de Disparo Integral.

Um **grafo de disparo integral** de uma RPC é um grafo orientado $GD = (V, A, N)$, onde:

- i) $V = [M_0 >$ é o conjunto de vértices;
- ii) $A = \{(M_1, b, M_2) \in V \times EA \times V \mid M_1 [b > M_2]\}$ é o conjunto de arestas;
- iii) $\forall a = (M_1, b, M_2) \in A: N(a) = (M_1, M_2)$ é o conjunto de nós.

Quando todas as variáveis de uma RPC têm tipos finitos, é direto provar que o GD é finito se, e somente se, todos os lugares são limitados.

Quando se desenha um GD como o da figura 9.2, usualmente escreve-se dentro de cada nó um texto descrevendo a marcação que o mesmo representa. Analogamente, escreve-se próximo a cada arco o elemento de amarração que ele representa.

Definem-se caminhos orientados finitos e componentes fortemente conexos de forma usual. Um **componente fortemente conexo (CFC)** é um subgrafo no qual existe um caminho orientado entre dois nós quaisquer (ver apêndice I). Usa-se v^c para denotar o **CFC** que contém v . Uma notação similar é usada para arcos.

Definição 9.13: O grafo orientado $G^* = (V^*, A^*, N^*)$ é o **grafo de componentes fortemente conexos** do grafo orientado $GCFC = (V^*, A^*, N^*)$ se, e somente se, as seguintes propriedades forem satisfeitas:

- i) $V^* = CFC$ é o conjunto dos componentes fortemente conexos;
- ii) $A^* = \{a \in A \mid o(a)^c \neq d(a)^c\}$;
- iii) $\forall a \in A^*: N^*(a) = (o(a)^c, d(a)^c)$.

O $GCFC$ contém um nó para cada **CFC** de GD . O $GCFC$ contém os arcos do GD que unem dois diferentes **CFC**'s. Intuitivamente, pode-se obter o $GCFC$ do grafo original. Para tal, junta-se todos os nós de cada **CFC** em um único nó, que contém todos os arcos do grafo original, exceto os arcos que unem dois nós dentro do mesmo **CFC**.

Através do GD da figura 9.2 e do $GCFC$ correspondente, é possível investigar as propriedades dinâmicas do SBD da figura 9.1. Isto é feito usando-se um conjunto de proposições denominadas **regras de prova**. As regras de prova relacionam propriedades de GD e de $GCFC$ às propriedades dinâmicas da RPC. Na proposição 9.1, usa-se CFC_T para denotar os **CFC**'s terminais, ou seja, os nós do $GCFC$ que não têm arcos de saída. Usa-se $(t, b) \in A$ para denotar que um elemento de amarração (t, b) é rótulo de um dos arcos de GD .

Proposição 9.1: Para grafos de disparo, são válidas as seguintes regras de prova para todo $\mathbf{l} \in \mathbf{L}$ e todo $\mathbf{t} \in \mathbf{T}$:

- i) Maior limite inteiro($\mathbf{l}$) = $\max\{|\mathbf{M}(\mathbf{l})| : \mathbf{M} \in \mathbf{V}\}$;
- ii) Maior limite multi-conjunto($\mathbf{l}$) = $\sum_{\mathbf{c} \in \mathbf{C}(\mathbf{l})} \max\{|\mathbf{M}(\mathbf{l}, \mathbf{c})| : \mathbf{M} \in \mathbf{V}\}$;
- iii) $|\mathbf{CFC}| = 1 \Rightarrow \mathbf{M}_0$ é uma marcação caseira;
- iv) $|\mathbf{CFC}| = 1 \wedge \forall \mathbf{b} \in \mathbf{B}(\mathbf{t}) : (\mathbf{t}, \mathbf{b}) \in \mathbf{A} \Rightarrow \mathbf{t}$ é estritamente viva.

As regras de prova na proposição 9.1 permitem verificar as propriedades de limitabilidade, vivacidade e caseira para o SBD da figura 9.1.

Mesmo para pequenos **GD's**, como o da figura 9.2, a construção e análise são entediante e sujeitas a erros. Neste contexto, técnicas estão sendo desenvolvidas no sentido de reduzir **GD's**, sem perder muita informação [60]. A seguir é esboçada uma forma de redução.

Muitos sistemas contêm algum tipo de simetria. No SBD, todos os **GBD's** são tratados de forma similar. Assim, pode-se fazer uma permutação entre os mesmos, sem modificar o comportamento do sistema. Como exemplo, há muita similaridade entre as marcações (1,-,3) e (2,-,3). Estas marcações são simétricas, no sentido de que uma pode ser obtida a partir da outra através da permutação de $\mathbf{g}_1$ com $\mathbf{g}_2$.

Usa-se $\Phi_{\mathbf{GBD}}$ para denotar todas as permutações dos elementos do tipo **GBD**. Diz-se que $\phi \in \Phi_{\mathbf{GBD}}$ é uma **simetria** e que ela mapeia a marcação $\mathbf{M}$ em uma marcação simétrica $\phi(\mathbf{M})$, obtida de $\mathbf{M}$ pela troca de cada $\mathbf{g} \in \mathbf{GBD}$ por $\phi(\mathbf{g})$. Diz-se, também, que $\mathbf{M}$ e $\phi(\mathbf{M})$ são **equivalentes**, o que é escrito como $\mathbf{M} \approx_{\mathbf{M}} \phi(\mathbf{M})$. $\Phi_{\mathbf{GBD}}$ é um grupo algébrico. Isto implica que $\approx_{\mathbf{M}}$ é uma relação de equivalência. Notação e terminologia similares são usadas por elementos de amarração e para multi-conjuntos de valores de *tokens*. Todas as expressões de arcos do SBD (figura 9.1) satisfazem a seguinte propriedade para todo $\phi \in \Phi_{\mathbf{GBD}}$ e todo $\mathbf{b} \in \mathbf{B}(\mathbf{t})$:

$$\mathbf{E}(\mathbf{a}) \langle \phi(\mathbf{b}) \rangle = \phi(\mathbf{E}(\mathbf{a}) \langle \mathbf{b} \rangle). \quad (1)$$

Intuitivamente, isto significa que amarrações simétricas têm efeitos simétricos. Pode-se obter os *tokens* usados pela amarração $\phi(\mathbf{b})$ aplicando-se ϕ aos *tokens* usados por $\mathbf{b}$. A propriedade (1) é local e estática, podendo ser determinada de transições individuais sem se considerar o conjunto de todas as marcações alcançáveis. De (1) pode-se provar a propriedade dinâmica a seguir, satisfeita para todas as marcações $\mathbf{M}', \mathbf{M}'' \in [\mathbf{M}_0]$, todos os elementos de amarração $\mathbf{b} \in \mathbf{EA}$ e toda $\phi \in \Phi_{\mathbf{GBD}}$:

$$\mathbf{M}'[\mathbf{b}] \mathbf{M}'' \Leftrightarrow \phi(\mathbf{M}')[\phi(\mathbf{b})] \phi(\mathbf{M}''). \quad (2)$$

Intuitivamente, isto significa que marcações simétricas têm elementos de amarração habilitados simétricos, os quais levam a marcações sucessoras simétricas. Quando (2) é satisfeita, faz sentido construir-se grafos de disparo onde se tem somente um nó para cada classe de equivalência de marcações e um arco para cada classe de equivalência de elementos de amarração disparados. Tal grafo é denominado **grafo de disparo com simetria (GDS)**. O **GDS** para o SBD com três **GBD's** é mostrado na figura 9.3, onde cada classe de equivalência é representada por meio de um de seus membros.

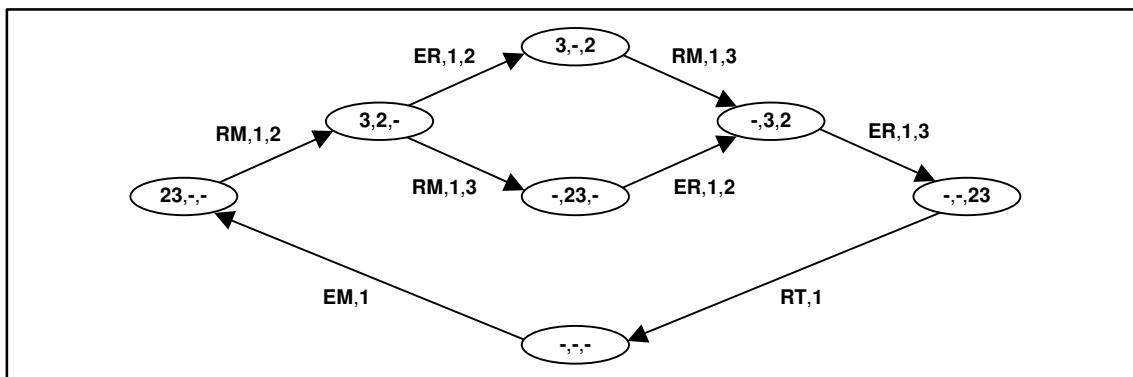


Figura 9.3. Grafo de disparo com simetria para o SBD com 3 gerenciadores.

Geralmente, um **GDS** é muito menor do que o **GD** correspondente. Para o SBD, o **GD** cresce com velocidade exponencial enquanto o **GDS** cresce quadraticamente [81]. A tabela 9.2 ilustra a complexidade de **GD** e **GDS**. Entretanto, é igualmente importante o tempo gasto para calcular ambos os grafos. A complexidade do tempo de construção de **GD** é da ordem $O(n^2 * 3^n)$, enquanto a complexidade do tempo de construção de **GDS** é da ordem $O(n^3)$.

Embora os **GDS**'s sejam geralmente muito menores do que os correspondentes **GD**'s, eles são igualmente poderosos no que diz respeito a verificação de propriedades dinâmicas. As regras de prova na proposição 9.2 são intimamente relacionadas àquelas da proposição 9.1. Usa-se * para denotar a multiplicação de um multi-conjunto **GBD** por um inteiro. Como antes, usa-se $(t,b) \in A$ para denotar que um elemento de amarração (t,b) encontra-se em um dos arcos do grafo. Entretanto, deve-se observar que cada arco agora representa uma classe de equivalência de elementos de amarração. Assim, exige-se que (t,b) pertença a uma dessas classes de equivalência.

Tabela 9.2. Tamanho dos grafos de disparo integral e com simetria para o SBD.

GBD	Grafo de Disparo Integral		Grafo de Disparo com Simetria	
	Nós – $O(n * 3^n)$	Arcos – $O(n^2 * 3^n)$	Nós – $O(n^2)$	Arcos – $O(n^2)$
2	7	8	4	4
3	28	42	7	8
4	109	224	11	14
5	406	1090	16	22
6	1459	4872	22	32
7	5104	20426	29	44
8	17497	81664	37	58
9	59050	314946	46	74
10	196831	1181000	56	92
15	71744536	669615690	121	212
20	23245229341	294439571680	211	382

Proposição 2.2: Para grafos de disparo com simetria, são válidas as seguintes regras de prova, para todo $l \in L$ e todo $t \in T$:

- Maior limite inteiro(l) = $\max\{|M(l)| : M \in V\}$;
- Maior limite multi-conjunto(l) = $\max\{M(l,c) : M \in V \wedge c \in C(l)\} * C(l)$;

- iii) $|\mathbf{CFC}| = 1 \Rightarrow \mathbf{M}_0$ é uma marcação caseira;
- iv) $|\mathbf{CFC}| = 1 \wedge \forall \mathbf{b} \in \mathbf{B}(\mathbf{t}): (\mathbf{t}, \mathbf{b}) \in \mathbf{A} \Rightarrow \mathbf{t}$ é estritamente viva.

A regra ii) somente é válida quando todos os valores do tipo $\mathbf{C(I)}$ podem ser mapeados entre si através do conjunto de permutações permitidas. Caso contrário, dever-se-ia ter uma regra levemente mais complexa, porque se teria que fazer uma investigação separada para cada classe de equivalência de $\mathbf{C(I)}$.

O uso de permutações é somente uma das maneiras de se obter simetria. Em geral, não é necessário impor quaisquer restrições sobre a forma pela qual uma simetria mapeia marcações em marcações e elementos de amarração em elementos de amarração, bastando-se satisfazer (2).

Há muitas outras formas de reduzir grafos de disparo. **Conjuntos inflexíveis** descartam alguns dos muitos elementos de amarração. **Marcações cobertas** cuidam das situações nas quais alguns lugares tornam-se ilimitados. Acredita-se que seja possível combinar o uso de simetria, conjuntos inflexíveis e marcações cobertas [58], o que seria melhor.

A análise através de grafos de disparo tem várias propriedades atrativas. Primeiro, é extremamente fácil usar grafos de disparo. Suas construção e análise podem ser completamente computadorizadas. Isto significa que o modelador não necessita ter qualquer conhecimento da matemática subjacente. Por exemplo, não é necessário saber como checar se duas marcações são simétricas ou não, ou ainda, como trabalham as regras de prova. Segundo, o grafo de disparo contém todos os detalhes a respeito do comportamento da RPC, uma vez que ele representa todas as possíveis seqüências de disparos. Assim, é possível investigar todos os tipos de propriedades dinâmicas por meio de grafos de disparo.

Um problema com grafos de disparo está no fato de ser necessário fixar todos os parâmetros do sistema (como, por exemplo, o número de GBD's no SBD) antes de sua construção. Isto significa que sempre serão encontradas propriedades que são específicas aos valores escolhidos para os parâmetros do sistema. Na prática, o problema não é tão grande. O entendimento de como um SBD trabalha com poucos GBD's leva ao entendimento de como o sistema trabalha para muitos GBD's. Naturalmente, isto somente é verdade quando da discussão sobre a validação do modelo e não quando se fala sobre o seu desempenho.

PARTE III – REDE DE PETRI PARA SISTEMAS DIGITAIS

CAPÍTULO 10: REDES DE PETRI E SISTEMAS DIGITAIS

10.1. INTRODUÇÃO

Na parte II desta tese, foi visto que rede de Petri e suas extensões (**redes de Petri**, genericamente) constituem uma classe de modelos conceituais bastante poderosa para a modelagem de **diversos tipos de sistemas**. Por exemplo, tem-se usado redes de Petri para modelar protocolos de comunicação, processos de manufatura, arquiteturas de computadores etc. Tais modelos possibilitam a representação de sistemas paralelos, os quais podem então ser simulados, por exemplo, nos níveis funcional e lógico, incluindo ou não aspectos de temporização, com um grau razoável de simplicidade.

Mais recentemente, redes de Petri têm demonstrado potencialidade na modelagem de **sistemas de eventos discretos**. Este fato deve-se, em grande parte, à existência de um conjunto de técnicas para **análise estrutural** e **dinâmica** de redes de Petri, permitindo a **validação formal** de importantes **propriedades** de um modelo: vivacidade, segurança, inexistência de conflitos, dentre outras.

Exemplo dessa utilização de redes de Petri pode ser encontrado na **Universidade do Minho** (Portugal) [61], onde foi desenvolvida uma aplicação computacional que aceita como entrada uma especificação textual baseada em **RP-Shobi** (rede de Petri **interpretada, orientada a objetos, hierárquica e síncrona**), valida as propriedades do sistema modelado e converte a especificação para um dado subconjunto de VHDL, visando posterior utilização em ferramentas de simulação e síntese. Desta forma, de um lado se aproveita a maior facilidade de utilização de redes de Petri na especificação em alto nível de sistemas digitais, sua simulação e a validação de algumas de suas propriedades e, de outro lado, aproveitam-se as maiores robustez e capacidade de síntese da linguagem VHDL.

Pode-se perguntar o porquê da utilização de redes de Petri no projeto de sistemas digitais. Além das técnicas de análise estrutural e dinâmica descritas anteriormente, as extensões de rede de Petri apresentam, não simultaneamente, a maior parte das **características desejáveis** a uma boa metodologia de modelagem de sistemas digitais, as quais encontram-se distribuídas por vários modelos conceituais de especificação (capítulo 2). Estas características, elencadas por Gajski [3], são:

- a) **Expressão de concorrência;**
- b) **Expressão do sistema em termos de transições de estados;**
- c) **Representação de hierarquia comportamental;**
- d) **Estruturas de programação;**
- e) **Término comportamental;**
- f) **Tratamento de exceções;**
- g) **Representação de tempo;**
- h) **Comunicação;**
- i) **Sincronização de processos.**

A seguir, verifica-se a relação das características supracitadas com redes de Petri.

- a) **Concorrência:** Redes de Petri são inerentemente concorrentes, de tal forma que nenhuma consideração faz-se necessária nesse sentido;
- b) **Transições de estados:** Redes de Petri são inteiramente construídas sobre o conceito de estados e transições entre eles. Desta forma, não há necessidade de incorporar quaisquer adições especiais sobre transições de estados em extensões baseadas em redes de Petri;
- c) **Hierarquia comportamental:** Existem várias abordagens na representação de hierarquia em redes de Petri, especialmente nas extensões de alto nível. Basicamente, um **lugar** ou uma **transição** pode representar uma subrede de nível hierárquico inferior, de maneira que tal subrede possa, sem quaisquer transformações da rede principal, substituir o lugar ou a transição correspondente;
- d) **Estruturas de programação:** A maior parte das extensões de redes de Petri não apresenta esta característica. Algumas extensões apresentam estruturas de programação associadas aos **lugares** da rede, enquanto outras extensões apresentam-nas associadas às **transições**. Por exemplo, **PNES** (*Petri Net for Embedded Systems – Rede de Petri para Sistemas Embutidos* – descrita na seção 10.4) apresentam estruturas de programação relacionadas aos lugares, enquanto **RPC** (**Redes de Petri Coloridas**, capítulo 9) as apresentam relacionadas às transições;
- e) **Término comportamental:** Esta característica relaciona-se com a representação adequada das **ações** de um sistema e de seu **término**. As transições de redes de Petri expressam de forma precisa o início e o término de cada ação do sistema;
- f) **Tratamento de exceções:** A princípio, redes de Petri não tratam de exceções. Entretanto, atualmente, existem esboços de extensões de rede de Petri que apresentam esta característica em parte, como **PNES** e **PCN** (*Place Chart Nets* – descrita na seção 10.3);
- g) **Representação de tempo:** A associação de tempo à rede depende da extensão de rede de Petri adotada (capítulo 8). Algumas extensões associam tempo às **transições**, enquanto outras, aos **lugares**. É possível provar que ambas representações são equivalentes [48];
- h) **Comunicação:** Em geral, os sistemas consistem de vários comportamentos que interagem entre si, de forma cooperada. É importante para uma linguagem de descrição de sistemas fornecer um mecanismo genérico de comunicação, que a destaque das demais partes do sistema. Neste ponto, rede de Petri e suas extensões não fazem esta distinção explicitamente;
- i) **Sincronização de processos:** Apesar de serem assíncronas por definição, redes de Petri não apresentam problemas para a introdução de um sinal de sincronismo.

Das extensões de redes de Petri, **PCN** e **PNES** apresentam a **maior parte** das **características** supracitadas. De um lado, **PCN** é uma extensão de **baixo nível**, **hierárquica**, possuidora de mecanismos de **preempção**. Entretanto, **não apresenta estruturas de programação** e nem representação de **tempo**. Por outro lado, **PNES** apresenta **estruturas de programação**, é **temporizada**, **hierárquica**, possui mecanismos de **preempção**, mas **não pode ser classificada como rede de alto nível**, pois trabalha com **tokens** booleanos. Além disso, a **temporização** em **PNES** leva em conta **somente tempo de computação**, não possuindo mecanismos para a representação de **tempo de comunicação**. Embora possuam mecanismos de preempção, tanto **PCN** quanto **PNES** **não possuem** mecanismos explícitos de **interrupção**.

Exemplos da utilização de redes de Petri no projeto de **sistemas digitais** encontram-se na seção 10.2. A seção 10.3 apresenta o modelo **PCN**, enquanto a seção 10.4 descreve **PNES**. Baseando-se nas **características desejáveis para representação de sistemas digitais**, nos modelos **PCN** e **PNES**, nas **experiências** do uso de redes de Petri por outros

pesquisadores e na **extensão de rede de Petri** desenvolvida pelo autor durante o trabalho de **mestrado** [62], é apresentada, no próximo capítulo, a extensão de rede de Petri desenvolvida neste trabalho de doutorado, voltada para a **representação de sistemas digitais**, a qual atende as características destacadas nesta seção. Desta forma, o capítulo atual apresenta os **fundamentos** que direcionaram nosso trabalho, visando à criação de uma abordagem baseada em redes de Petri para a **representação, simulação, análise e síntese em alto nível de sistemas digitais**.

10.2. REDES DE PETRI NO PROJETO DE SISTEMAS DIGITAIS

Esta seção faz um levantamento da **aplicação** de redes de Petri no projeto de sistemas digitais. Parte das considerações apresentadas aqui são fornecidas por Marranghello [63].

A metodologia de síntese em alto nível **CAMAD** [64] especifica um sistema através de um **subconjunto** da linguagem **VHDL**, denominado **S’VHDL**. A especificação **S’VHDL** de um sistema é mapeada para uma **rede de Petri temporizada estendida** (**ETPN – Extended Timed Petri Net**), composta por duas partes separadas e relacionadas: uma **estrutura de controle** e um **datapath**. O **datapath** é representado como um **grafo orientado** onde os **nós** representam **unidades de manipulação de dados** (**unidades funcionais** e de **armazenamento**) e os **arcos** representam **relações de dependência**. A **estrutura de controle**, representada como uma **rede de Petri temporizada de Peng** [65], comunica-se com o **datapath** através do **envio de sinais de controle** e do **recebimento de sinais condicionais**. O **grafo do datapath** constitui-se numa **descrição RTL** (*Register Transfer Level*) **extraída** diretamente da **especificação S’VHDL** e, como tal, pode ser sintetizada por **ferramentas de síntese para VHDL** (visto que **S’VHDL** é um subconjunto de **VHDL**). A **porção da ETPN referente ao controle**, assim como os **sinais de comunicação correspondentes**, são usualmente implementados em *hardware* através da **síntese** de uma ou mais **máquinas de estados finitos** (FSM), de acordo com o estilo escolhido pelo projetista do sistema.

Outra metodologia de síntese em alto nível para sistemas embutidos foi desenvolvida por Rajesh K. Bawa, Emmanuelle Encrenaz e Ludovic Jacomme (*Université Pierre et Marie Curie*, Paris, França), baseada na **descrição VHDL** de um sistema [36], a qual é **convertida** para um modelo descrito em **rede de Petri temporizada e interpretada** (**ITPN – Interpreted and Timed Petri Nets**) [66], visando a **verificação formal** de suas **propriedades**. A partir da descrição ITPN, é **derivado** um **sistema de equações booleanas**. Determinadas propriedades do sistema são verificadas usando **análise simbólica** destas equações [67]. Após a validação do sistema, o mesmo é **sintetizado em código VHDL**, no nível de transferência entre registradores (**RTL**), para propósitos de implementação através de ferramentas de síntese para circuitos VLSI [68].

Um *framework* (modelo baseado em componentes computacionais) para o projeto de sistemas **reativos** complexos, tais como controladores programáveis, está em desenvolvimento por um grupo liderado por Luís Gomes, na **Universidade Nova de Lisboa**, Portugal. O sistema é definido como uma **rede de Petri Reativa** [69], a qual é uma classe baseada em **RPC** (capítulo 9), **SIPN** (*Synchronous Interpreted Petri Nets – Redes de Petri Interpretadas Síncronas*) [70] e *Statecharts* [7] (seção 4.5). O modelo inclui esquemas de prioridade para transições e construções para hierarquia estrutural. A **análise** de certas **propriedades** do modelo (como ausência de *deadlocks*) é feita usando-se a **máquina de**

senhas (espaço de estados, grafo de alcançabilidade) resultante de sua simulação. Um dos objetivos do *framework* (em desenvolvimento) é **suportar a implementação** de *hardware* em **componentes lógicos programáveis**, tais como **FPGAs**, a partir do modelo em redes de Petri, dentro de um ambiente de coprojeito *hardware/software*.

Conforme abordado na introdução deste capítulo, na **Universidade do Minho** (Portugal), foi desenvolvida uma aplicação computacional que aceita como entrada uma especificação textual baseada em **RP-Shobi** (rede de Petri **interpretada, orientada a objetos, hierárquica e síncrona**) [71], **valida as propriedades** do sistema modelado e **converte** a especificação para um dado **subconjunto de VHDL**, visando posterior utilização em **ferramentas de simulação e síntese**. A RP-Shobi (um aperfeiçoamento do modelo **SIPN**, incluindo conceitos de orientação a objeto e hierarquia) possibilita a **representação comportamental** tanto de **controladores paralelos** quanto do *datapath*. O ambiente **SOFHIA** suporta especificações RP-Shobi: a **especificação RP-Shobi** de um **controlador paralelo** é descrita através de uma **notação intermediária** denominada **CONPAR**, sendo **posteriormente compilada em código VHDL** [72]. Assim, de um lado aproveita-se a maior facilidade de utilização de redes de Petri na especificação em alto nível destes controladores, sua simulação e a validação de algumas de suas propriedades e, de outro lado, aproveita-se a capacidade de simulação e síntese da linguagem VHDL.

Um outro exemplo do uso de **redes de Petri de alto nível**, estendidas com mecanismos de **orientação a objeto**, é o trabalho desenvolvido por Robert Esser, Jörn W. Janneck e Martin Naedele, no *Swiss Federal Institute of Technology*, em Zurique (Suíça) [73,74]. O modelo de redes de Petri utilizado é uma **composição** de redes de Petri de **alto nível** com redes de Petri **temporizadas**. Este modelo exerce o papel de uma **linguagem kernel**, representando diferentes **modelos de computação**. A adição de estruturas orientadas a objeto, para as redes de Petri, permite a modelagem de um **comportamento concorrente complexo**, junto com os **dados** que são manipulados por este comportamento. A abordagem permite que não somente os *tokens* sejam orientados a objeto, mas também a própria rede. Desta forma, **sistemas com múltiplos níveis** podem ser modelados, tais como **sistemas operacionais** rodando *software*.

Outra metodologia para a síntese em alto nível é apresentada em [75], voltada para **controladores paralelos sincronizados**. O processo de projeto inicia-se com uma **especificação formal em redes de Petri**, a qual é **representada** por um **diagrama de decisão binário (BDD – Binary Decision Diagram)**. Através da **manipulação booleana** deste diagrama é possível fazer a **análise comportamental** do sistema modelado, assim como a **verificação** de determinadas **propriedades**. Uma **descrição RTL** do sistema é **gerada**, a partir do uso de técnicas orientadas a sintaxe e de percorrimto simbólico de redes de Petri. O **circuito lógico** obtido na sequência pode ser **otimizado e sintetizado** em um **ASIC** ou em um **FPGA**.

A ferramenta de CAD **PETRIFY** [76] foi implementada por Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor e Alexandre Yakovlev, na *Universitat Politècnica de Catalunya*, em Barcelona (Espanha), para a **síntese** de circuitos **independentes de velocidade**. Nesta ferramenta, o **comportamento** do sistema pode ser descrito como uma **rede de Petri**, um **STG (Signal Transition Graph – Grafo de Transição de Sinais)** ou por um **TS (Transition System)**. STG é um tipo especial de rede de Petri **rotulada**, enquanto TS é um tipo especial de **grafo de estados**, onde os arcos são rotulados com símbolos de um alfabeto de eventos. A **descrição inicial** de um sistema é analisada e uma descrição mais simples é **produzida** pela ferramenta **PETRIFY**, em uma **notação de redes**

de Petri ou em uma **notação STG** [77]. O sistema digital é analisado, utilizando-se modelos baseados em estados. A **especificação** (que pode ser **otimizada**) é **sintetizada**, através da manipulação da função booleana de informações obtidas pelo espaço de estados. Como produto final da ferramenta PETRIFY, é gerada uma **net list** do circuito em nível lógico, independente de velocidade, correspondente à especificação comportamental inicial.

Exemplo de outra ferramenta baseada em redes de Petri é a **ADEPT** (*Advanced Design Environment Prototyping Tool*) [78]. Desenvolvida na *University of Virginia* (EUA), sob a supervisão de James H. Aylor (atual *Web Editor* da *IEEE Computer*), esta ferramenta é baseada sobre um **modelo misto**, composto de especificações em **VHDL** e em **RPC** (capítulo 9). A descrição completa de um sistema, **especificada inicialmente** por **ferramentas esquemáticas**, é transformada nos chamados **módulos ADEPT**, os quais são dados tanto em VHDL quanto em RPC. Esta duplicidade de especificação permite a **análise de confiabilidade** (com a **transformação** da **rede de Petri** em um modelo de **Markov**) e a **simulação comportamental** (com base em modelos VHDL).

Os trabalhos desenvolvidos na *Universidade de Pernambuco* [79,80,81,82], sob a supervisão de Edna Barros e Paulo Maciel, concentram-se sobre o **particionamento hardware/software**. O problema do particionamento é tratado com uma abordagem baseada em **clusters** (cada *cluster* corresponde a um **agrupamento** formado por um **arquivo de registradores** e um **conjunto de unidades funcionais**). A representação interna utilizada para este propósito constitui-se num **grupo** de redes de Petri (**hierárquicas, interpretadas e temporizadas**), transformadas de uma para outra ao longo do processo de projeto.

A despeito do preconceito da comunidade de coprojeto *hardware/software*, pode-se constatar, pelo exposto anteriormente, que redes de Petri têm sido empregadas neste domínio por vários grupos de pesquisa pelo mundo. Grande parte do preconceito no uso de redes de Petri pode ser atribuída ao desconhecimento de trabalhos como os destacados aqui: **muitos pesquisadores resumem redes de Petri à classe das redes Elementares**, as quais, nitidamente, são pouco adequadas para a representação de sistemas digitais.

10.3. PLACE CHART NETS

O modelo *Place Chart Nets* (PCN) foi proposto por *Kishinevsky et al.* em 1997 [83], na época, como a única extensão de rede de Petri agregando as noções de **preempção** e **hierarquia**.

PCNs combinam as características de **redes de Petri** e de *Statecharts* [7] (seção 4.5). Ao contrário de *Statecharts* e de outros modelos reativos, os quais são inerentemente síncronos, PCNs especificam o comportamento de um sistema através do uso de **ordens parciais**. Ao contrário de modelos hierárquicos baseados em redes de Petri, a **hierarquia** em PCN é **determinada por preempção**.

Para tratar de preempções, os autores definiram **lugares especiais**, denominados *place charts*, aos quais uma **subrede** pode ser **associada**. A **operação normal** da rede é feita através da **subrede associada** a cada *place chart*. **Transições preemptivas** têm *place charts* como **lugares de entrada**. Estas transições servem para modelar situações de **erro, exceções**. Uma transição preemptiva, quando do seu disparo, remove **todos os tokens** das **subredes associadas aos place charts de entrada** da transição.

A figura 10.1 mostra um exemplo de PCN, retirado da referência [83]. Após o **recebimento de uma mensagem (newmsg)**, o sistema **computa o CRC (computeCRC)**; CRC é a sigla de *Cyclic Redundancy Check* – verificação por redundância cíclica) e continua **processando a mensagem recebida (getHDR, readCAM, incrHOP, newPATH)**. Se, durante este processamento, um **erro de CRC** for detectado, uma condição de erro surge e faz a **preempção** (cancelamento) do **processamento da mensagem**, através do disparo da transição **Error**. O resultado deste disparo é a remoção de **todos** os *tokens* presentes na subrede do *place chart* destacado na figura. Além disso, um novo *token* é inserido no **lugar de entrada** da transição **newmsg**.

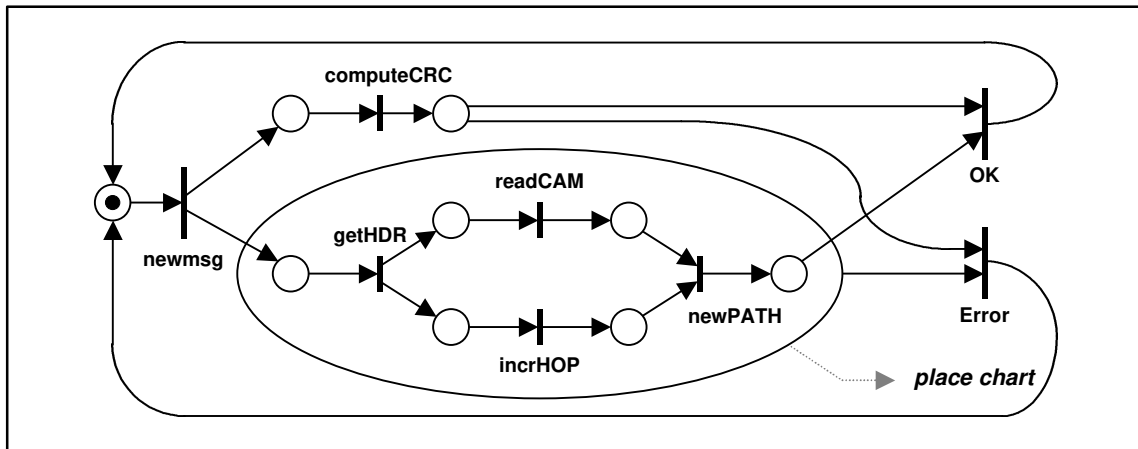


Figura 10.1. Parte de uma PCN modelando um protocolo de comunicação.

PCNs são uma generalização não trivial de redes de Petri Elementares, no sentido que:

- i) A limitabilidade de PCNs é decidível (isto é, tem solução algorítmica);
- ii) Para o caso finito (limitado), modelar uma PCN pode exigir uma rede de Petri Elementar exponencialmente maior;
- iii) Para o caso infinito (ilimitado), a classe de linguagens PCN inclui a classe de linguagens de redes de Petri Elementares.

Para ilustrar o ganho na força descritiva resultante do uso de PCN (item *i* acima), a figura 10.2 [83] apresenta uma rede de Petri Elementar com comportamento equivalente à PCN da figura 10.1. A rede Elementar foi obtida após a realização de dois procedimentos, quais sejam:

- 1º) Construção da máquina de senhas (grafo de alcançabilidade) para a PCN da figura 10.1;
- 2º) Síntese de uma rede de Petri Elementar **segura** (limitada a, no máximo, 1 *token* por lugar), utilizando-se a ferramenta PETRIFY (veja seção 10.2).

A rede plana (não hierárquica) da figura 10.2 possui seis cópias da transição denominada **Error**, uma para cada lugar da subrede do *place chart* da figura 10.1.

Pode-se resumir PCN como uma rede portadora de **hierarquia comportamental**, com **mecanismos de preempção** e de **baixo nível** (*tokens* booleanos). **Falta** à PCN, representação de **tempo**, **estruturas de programação** e **mecanismos explícitos** para o tratamento de **interrupções**. Sem esta última característica, não pode-se classificar PCN como portadora de mecanismos para o tratamento de exceções, uma vez que ela possibilita apenas a representação explícita de preempção. Esta seção não entra em detalhes de PCN, uma vez que tal extensão pode ser vista como um subconjunto de PNES, explicada na seqüência.

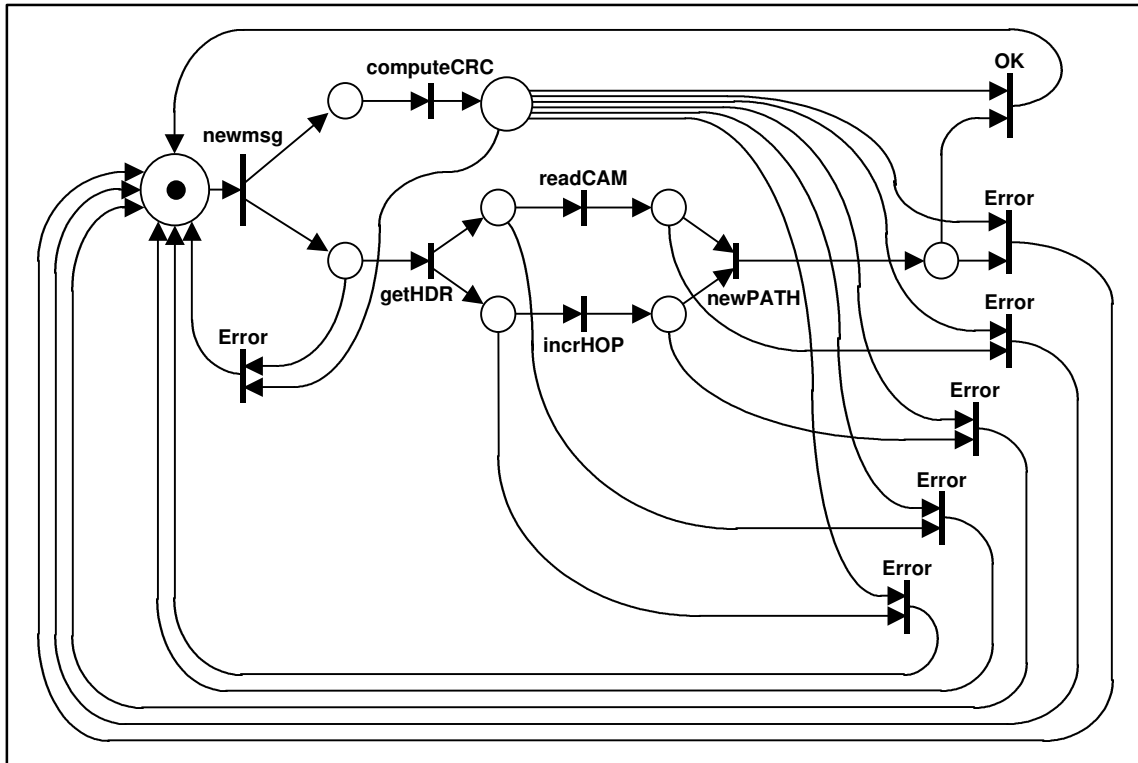


Figura 10.2. Rede de Petri elementar equivalente à PCN da figura 10.1.

10.4. REDE DE PETRI PARA SISTEMAS EMBUTIDOS

A extensão **PNES** (*Petri Net for Embedded Systems* – Rede de Petri para Sistemas Embutidos) foi proposta por *Mirkowski* e *Yakovlev* [84,85], para tentar superar as deficiências de PCN.

A principal diferença entre PNES e PCN é que PNES permite a associação de **estruturas de programação** aos **lugares** da rede. Estas estruturas são utilizadas para representar o *datapath* na rede.

Define-se **modo** da PNES como uma **quádrupla** composta por um **lugar** da rede, uma **função** que **associa** algumas **sentenças de programação** ao **lugar**, uma **marcação inicial para o lugar** e um **rótulo de tempo de execução** (*time-stamp*) para o conjunto de sentenças de programação do lugar. O *time-stamp* começa a ser contado quando um *token* é inserido no lugar associado ao modo. Este *token* somente estará **disponível para consumo**, por transições comuns que tenham o modo como entrada, quando o *time-stamp* do modo **expirar**.

PNES define **hierarquia comportamental** ao permitir que um modo seja descrito por uma subrede. Neste caso, o modo passa a ser chamado **supermodo**.

Para tratamento de **preempção**, PNES define a chamada **transição de exceção**. Representada por um **trapezóide**, uma transição de exceção é habilitada por **qualquer marcação** de seus modos de entrada, possibilitando o cancelamento imediato de todas as operações (sentenças de programação) em curso nestes modos. Uma **restrição** é que **preempções** podem ser modeladas **somente no nível hierárquico mais alto**.

Pode-se resumir PNES como uma rede portadora de **hierarquia comportamental**, **mecanismos de preempção**, **estruturas de programação** e mecanismos para representação de **tempo de computação**. Assim como PCN, **falta à PNES mecanismos explícitos para tratamento de interrupções**. Além disso, não possui representação para tratamento de **tempo de comunicação**. PNES **não** pode ser considerada uma extensão de **alto nível**, pois representa somente *tokens* booleanos.

Na seqüência, a definição formal de PNES, esboçada nas referências [84,85]. Esta referência traz também um exemplo, apresentado na seção 10.4.2.

10.4.1. DESCRIÇÃO FORMAL

Para manter o modelo simples, os criadores de PNES separaram **exceções** (leia-se **preempção**) da **hierarquia**, de forma que exceções só podem ser definidas no **nível hierárquico mais alto**. Subredes de nível inferior podem estar relacionadas a exceções de nível superior, mas não podem introduzir novas exceções.

Definição 10.1: HOPN.

Uma **HOPN** (*Hardware-Oriented Petri Net – Rede de Petri Orientada a Hardware*) é uma tupla

$$\text{HOPN} = (\mathbf{M}, \mathbf{T}, \mathbf{A}, \mathbf{L}, \mathbf{C}, \mu_0)$$

onde:

- **M** é um conjunto de **modos**;
- **T** é um conjunto de **transições**;
- **A** é um conjunto de **arcos**;
- **L** é um conjunto de **sentenças de programação**;
- **C** é um conjunto de **condições booleanas**;
- μ_0 é uma **marcação inicial**,

tal que:

- i) Um **modo** M_i é uma quádrupla: $M_i = (P_i, \lambda_i, \mu_i, \tau_i)$, onde
 - P_i é um **lugar** da rede de Petri;
 - $\lambda_i: P_i \rightarrow L \cup \emptyset$ é uma função que associa um conjunto (possivelmente vazio) de **sentenças de programação** (especificando comportamento) ao lugar. Diz-se que um lugar representa o modo e as sentenças que são associadas a ele;
 - $\mu_i \in \{0,1\}$ é a **marcação do lugar**. Um lugar pode conter um *token* (marca) ou não. A presença de um *token* em um lugar significa a execução das sentenças representadas pelo lugar;
 - $\tau_i \in \mathbf{IN}$ é um tempo finito (possivelmente ausente) associado ao lugar. O tempo representa o **tempo de execução** das sentenças comportamentais representadas pelo lugar;
- ii) Um modo pode ser hierárquico. Neste caso, P_i pode ser substituído por uma **HOPN**, conforme a definição 10.3 adiante;
- iii) Uma transição pode ser associada a uma condição booleana, chamada **guarda** desta transição, isto é, existe uma função $\chi_{Ti}: T_i \rightarrow C \cup \emptyset$;

- iv) Arcos conectam modos a transições: $A \subseteq (M \times T) \cup (T \times M)$;
- v) Uma **marcação da rede** é um conjunto de marcações de todos os modos da rede, isto é,

$$\mu_i = \{ \mu_1, \mu_2, \dots, \mu_n \} \text{ para } M_1, M_2, \dots, M_n \in M.$$

Definição 10.2: Regras Semânticas de HOPN.

- i) Uma marcação de um modo M_i está **disponível** para as transições de saída se o modo estiver marcado durante, pelo menos, o tempo a ele associado, isto é, $\theta(\mu_i) \geq \tau_i$, onde $\theta(x)$ é o tempo que x permanece inalterado;
- ii) Os **resultados** da execução das sentenças de programação associadas a um lugar estão disponíveis somente quando a marcação do lugar estiver disponível;
- iii) Uma **transição** está **habilitada** quando todas os seus modos de entrada estiverem marcados e suas respectivas marcações estiverem disponíveis;
- iv) Uma **transição dispara** quando ela estiver habilitada e sua guarda for verdadeira. O disparo de uma transição remove os *tokens* de todos os seus modos de entrada e insere um *token* em cada um dos seus modos de saída.

Definição 10.3: Modo Hierárquico (Supermodo).

Um modo M_j de uma HOPN é chamado **hierárquico** se ele representar outra HOPN. Neste caso, ele é designado pelo termo **supermodo**, e a rede é chamada **subrede**, sendo denotada por $\infty(M_j)$. Esta subrede atende as seguintes condições:

- i) Ela tem um **modo de entrada** (*entry mode*), denotado por $m_{in}(M_j)$, o qual recebe um *token* quando o supermodo correspondente recebe um *token*. O modo de entrada tem o mesmo conjunto de transições de entrada do supermodo;
- ii) A **marcação** de um **supermodo** representa qualquer marcação válida da subrede,
- iii) Ela tem um **modo de término** (*termination mode*), denotado por $m_{out}(M_j)$, o qual corresponde ao único modo marcado na subrede, ao final do processamento (comportamento) representado por ela. O modo de saída tem o mesmo conjunto de transições de saída do supermodo.

Definição 10.4: PNES.

Uma **PNES** (*Petri Net for Embedded Systems – Rede de Petri para Sistemas Embutidos*) é uma tupla

$$PNES = (HOPN, E, EA)$$

onde:

- **HOPN** é uma rede de Petri orientada a *hardware*, conforme definição 10.1;
- **E** é um conjunto de **exceções**;
- **EA** é um conjunto de **arcos estendidos**,

tal que:

- i) **Exceção** é uma forma de transição, com uma guarda não trivial, isto é, para toda exceção E_i existe uma função $\chi_{E_i} : E_i \rightarrow C$;
- ii) Arcos estendidos são uma extensão dos arcos da HOPN, de forma a conectar modos com transições e exceções: $EA \subseteq A \cup (M \times E) \cup (E \times M)$;

- iii) Os conceitos de modos de entrada e de modos de saída de transições são estendidos para exceções. Desta forma, $\bullet E_i$ representa o conjunto dos modos de entrada da exceção E_i , enquanto $E_i \bullet$ representa o conjunto de modos de saída de E_i ;
- iv) Se um supermodo M_j é entrada de uma exceção E_k , então todos os modos de sua subrede estão implicitamente conectados à exceção E_k , isto é,

$$\forall m_i \in \alpha(M_j), M_j \in \bullet E_k \Rightarrow m_i \in \bullet E_k$$

As regras de disparo de exceções diferem daquelas para transições. Em particular, a existência de ao menos um modo de entrada marcado (com *token*) é condição suficiente para habilitar uma exceção (dependendo da guarda, é claro). Além disso, se o modo tiver um tempo associado a ele, sua marcação estará sempre disponível para uma exceção.

Definição 10.5: Regras Semânticas de PNES.

- i) Uma **exceção** está **habilitada** quando pelo menos um de seus modos de entrada estiver marcado;
- ii) Uma **exceção dispara** quando está habilitada e sua guarda for verdadeira. O disparo de uma exceção remove os *tokens* de todos os seus modos de entrada marcados (mesmo daqueles com marcação indisponível) e insere um *token* em cada um dos seus modos de saída.

10.4.2. EXEMPLO DE MODELAGEM

Para apresentar os conceitos de modelagem descritos nesta seção, é usado um exemplo tomado do livro de Gajski *et al.* [3]. Trata-se do procedimento de atendimento de chamadas de uma secretária eletrônica, cujo comportamento é altamente dependente de ocorrências externas. Exemplos dessas ocorrências:

- a) O usuário atender o telefone que toca;
- b) O usuário não atender o telefone em um determinado espaço de tempo, obrigando a secretária a fazê-lo. Isto ocasionaria, por exemplo, a secretária tocar uma mensagem pré-gravada para que o interlocutor deixasse o seu recado;
- c) O usuário atender o telefone enquanto a secretária estiver tocando a mensagem pré-gravada;
- d) O usuário atender o telefone enquanto a secretária estiver gravando uma mensagem da pessoa que telefonou;
- e) O usuário teclar o botão 1 enquanto a secretária estiver tocando a mensagem pré-gravada ou gravando a mensagem da pessoa que telefonou, fazendo com que a secretária entrasse imediatamente no modo de operação remoto.

Considerando-se o nível de granulosidade de tarefas, este procedimento de atendimento de chamadas pode ser dividido nas seguintes partes:

- a) **Announcement:** Toca uma mensagem pré-gravada, verifica se a chamada continua ativa e prepara a gravação da mensagem do interlocutor;
- b) **RecordMessage:** Grava a mensagem do interlocutor. Também controla a gravação, limitando-a em tempo, e indicando o início e o término da mesma;
- c) **HangUp:** Encerra a ligação;
- d) **RemoteOperation:** Verifica o número de identificação do usuário. Também encerra a ligação se o usuário não é autorizado, e fornece instruções ao usuário, caso o mesmo seja autorizado.

Todas as tarefas descritas anteriormente, exceto **HangUp**, são compostas, compreendendo várias operações. Tarefas compostas em PNES são representadas como supermodos (representados graficamente por elipses, ao contrário dos modos, representados por círculos).

Os dois tipos de fluxo de controle entre os modos, quais sejam, operação "normal" e operação "interrompida" (quando o fluxo normal é interrompido por eventos que exigem ação imediata), são representados, respectivamente, por transições comuns (retângulos) e por transições de exceção (trapezóides).

A figura 10.3 apresenta um modelo PNES [84] para representação do procedimento de atendimento de chamadas. Observe o uso de transições comuns e de exceção. Devido à semântica de habilitação de uma transição de exceção, as duas exceções que levam de **Announcement** e **RecordMessage** para **RemoteOperation** foram unificadas.

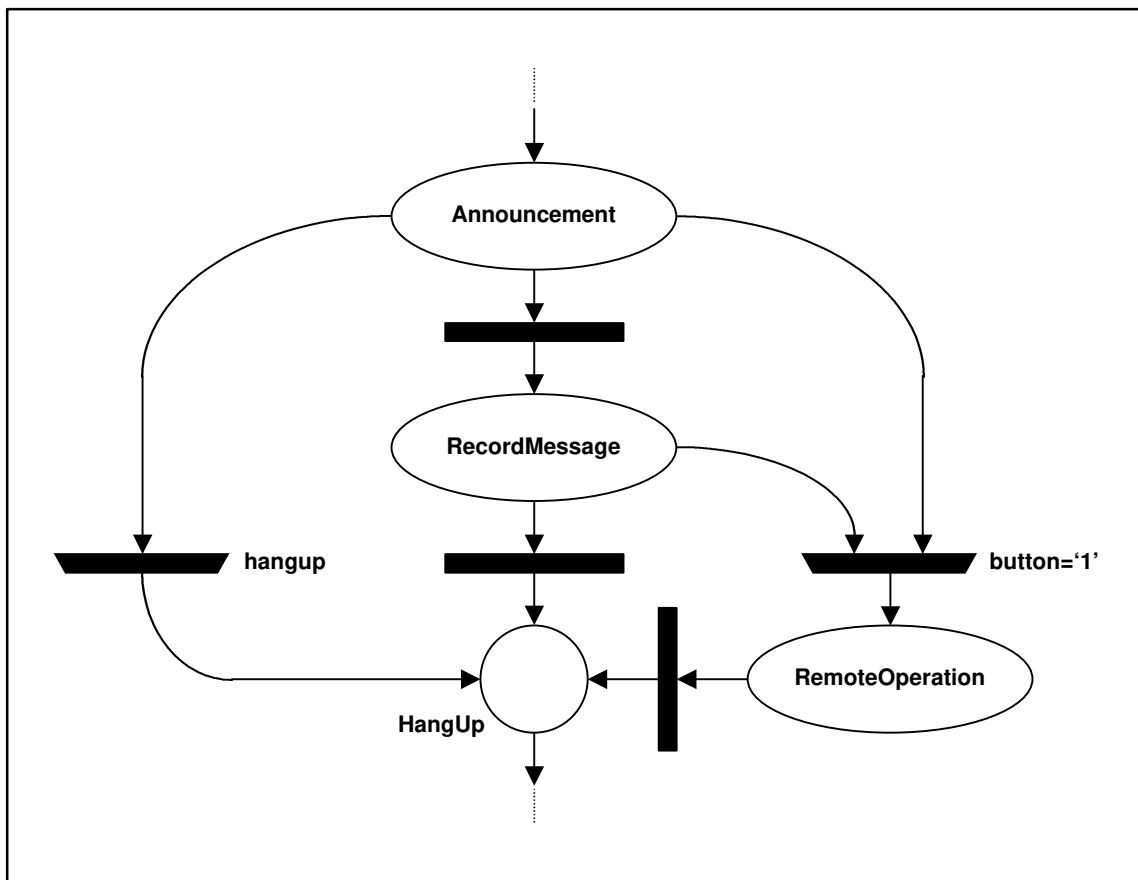


Figura 10.3. Modelo PNES para o procedimento de atendimento de chamadas.

Cada um dos três modos pode ser refinado para um nível inferior de hierarquia, onde seu comportamento pode ser especificado de uma maneira mais detalhada, transformando-se, desta forma, num supermodo. Por exemplo, conforme dito anteriormente, o modo **Announcement** compreende três operações, quais sejam:

- Tocar uma mensagem pré-gravada;
- Verificar se a chamada continua ativa;
- Preparar a gravação da mensagem do interlocutor.

A figura 10.4 apresenta o refinamento do supermodo **Announcement** da figura 10.3. Enquanto o sinal **ann_play** for igual a 1, a mensagem pré-gravada estará sendo tocada.

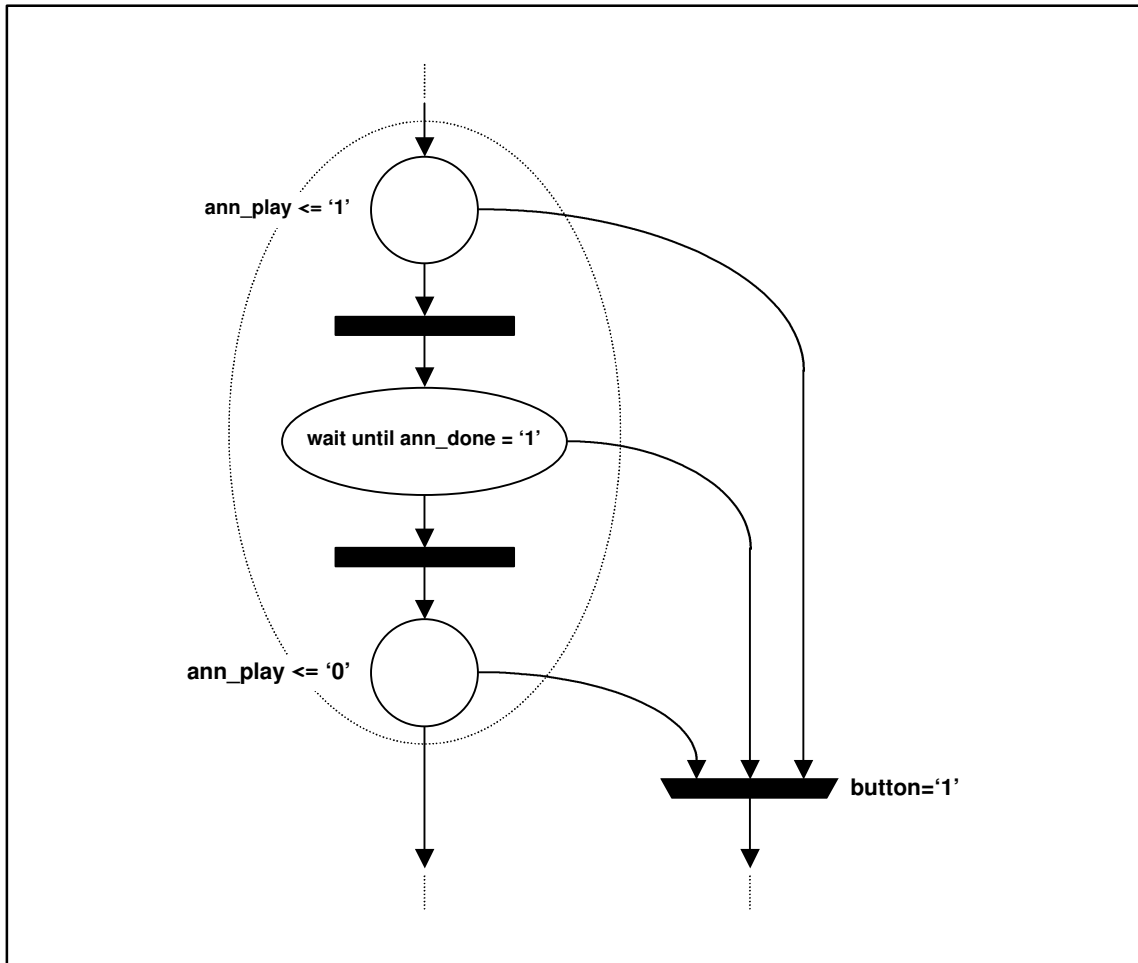


Figura 10.4. Refinamento do supermodo Announcement da figura 10.3.

CAPÍTULO 11: REDE DE PETRI PARA SISTEMAS DIGITAIS (RPSD)

11.1. INTRODUÇÃO

No capítulo anterior vimos que, embora apresentem características potenciais para a representação de sistemas digitais, as extensões de rede de Petri formalmente definidas para esta finalidade não apresentam os requisitos necessários em sua totalidade. Esta constatação levou-nos à criação de uma extensão que visa atender a tais características, possibilitando a modelagem de sistemas digitais em alto nível. Esta extensão constitui o modelo conceitual base de uma metodologia para a modelagem, simulação, análise e síntese em alto nível de tais sistemas. Neste capítulo, são apresentados componentes, características de modelagem e exemplos da extensão proposta. Nos próximos capítulos, são destacadas sua equivalência com outras extensões de rede de Petri, suas técnicas de análise e sua inclusão numa metodologia de coprojeto *hardware-software*.

Em nossa opinião, falta, às extensões de rede de Petri voltadas para sistemas digitais, contemplar as seguintes características:

- a) **Alto nível de representação:** Ao trabalhar com *tokens* booleanos, pode-se associá-los com o vigorar ou não de um determinado sinal (ou comportamento) em um circuito, estando próximo à síntese do circuito correspondente ao modelo do sistema. Entretanto, uma linguagem deve facilitar o processo de modelagem de sistemas. Para tal, há a necessidade de que uma extensão de rede de Petri deva apresentar *tokens* que carreguem tipos de dados mais abstratos do que simplesmente booleanos, aumentando, dessa forma, sua capacidade representacional;
- b) **Definição de tempos para computação e comunicação:** A maioria dos modelos de sistemas digitais não possui representação adequada de tempo. O ideal seria a associação de tempo à execução de comportamentos (como ocorre nas PNES), além da associação de intervalos de tempo para a possível ocorrência de ações (para efeito de modelagem de mecanismos de comunicação, tais como protocolos). Desta forma, redes de Petri deveriam trabalhar com duas estruturas de tempo: uma que associe intervalos de tempo às transições, denotando os tempos mínimo e máximo para que uma transição, uma vez habilitada, possa disparar, e outra que associe tempos para a execução dos comportamentos (representados pelos lugares). Ambas abordagens de temporização podem ser encontradas em diferentes extensões de redes de Petri (veja capítulo 8), entretanto, no que pudemos verificar até este ponto, não simultaneamente;
- c) **Tratamento de exceções completo (preempção e interrupção):** As duas extensões de rede de Petri que tratam de exceções são PCN e PNES. Nesta última, existe o chamado *supermodo*, lugar correspondente a uma subrede, representante de operações que possam ser abortadas quando necessário, através do disparo de uma transição especial denominada *exceção*. Conceito semelhante é encontrado em PCN. Tais redes caracterizam-se, portanto, como portadoras de mecanismos de preempção. Assim, falta-lhes um mecanismo para a representação explícita de interrupções.

Destarte, tem-se que as extensões de rede de Petri não são completas para a modelagem de sistemas digitais, o que nos levou à concepção de uma nova extensão, denominada **Rede de Petri para Sistemas Digitais (RPSD)**. As principais **características** de **RPSD** podem ser assim resumidas:

a) **Alto nível de representação:**

- Três tipos de **lugares** permitem representar **componentes de armazenamento**, **estados locais** de componentes ou **componentes funcionais (comportamentos)** do sistema modelado;
- **Tokens** permitem representar (de acordo com o lugar no qual estão inseridos), o **conteúdo** de uma **célula** pertencente a um **componente de armazenamento** (em um lugar de armazenamento), o **status (estado)** de um **componente do sistema** (em um lugar descritor de estado local) ou um **item de dado** pertencente a uma lista ordenada (descritora de um **sample de dados**), a qual deve ser processada pelo **comportamento** associado a um **componente funcional** (num lugar funcional; neste caso, o item de dado, ou *token*, pode representar um **argumento de entrada**, um **valor de dado** que está **sendo processado** ou, ainda, um **valor de dado resultante** da execução do comportamento);
- **Transições** permitem não só **alterar** entre **modos de computação** (através da remoção/adição de *tokens* de/a **lugares associados a estados locais** ou **componentes funcionais**), mas também realizar **operações de leitura/escrita** envolvendo **valores de dados** (descritos por *tokens*) mantidos por **componentes de armazenamento** (representados por lugares de armazenamento);
- **Ramos** especiais são adotados de acordo com os lugares aos quais estão conectados. Estes ramos são **rotulados** com **expressões** que definem **quais ou quantos tokens** devem ser **inseridos, removidos, lidos** ou **escritos** nos lugares associados.

b) **Hierarquia comportamental:**

- Modelos tratam de vários processos em uma mesma estrutura, permitindo a **parametrização** de processos;
- Existe uma **instância hierárquica (cópia do comportamento** associado a um lugar descritor de componente funcional) para cada **sample de dados** (lista ordenada de *tokens*) presente em um **componente funcional do sistema**.

c) **Estruturas de programação:**

- Estruturas relacionadas aos lugares são utilizadas para **manipulação de dados** inerentes ao **sistema** modelado;
- Estruturas relacionadas às transições são utilizadas para **manipulação do fluxo de controle** da rede.

d) **Tratamento completo de exceções:**

- **Mecanismos preemptivos** permitem abortar **comportamentos / estados locais** associados a um **componente do sistema**;
- **Mecanismos interruptivos** permitem **suspender temporariamente** as **instâncias de comportamento** associadas a um **componente funcional do sistema**, para **posterior retomada**.

e) **Representação completa de tempo:**

- **Mecanismo de tempo de comunicação** permite a associação de um **intervalo de tempo de disparo** para cada transição, possibilitando a representação tanto de **eventos aleatórios** quanto de **eventos escalonados**;
- **Mecanismo de tempo de computação** permite a associação indireta de *time-stamps* a componentes funcionais do sistema modelado.

As próximas seções descrevem a **definição, características e exemplos de RPSD**.

Como decorrência deste trabalho de doutorado, pretendemos, no futuro próximo, implementar uma plataforma em *software* capaz de capturar descrições em RPSD de sistemas digitais embutidos, para as suas simulação, análise e síntese em alto nível (veja descrição da metodologia baseada em RPSD no capítulo 15).

As próximas seções descrevem a **definição, características e exemplos de RPSD**. Comparações entre RPSD e linguagens de especificação utilizadas no projeto de sistemas digitais são dadas. Esta comparação, entre um modelo conceitual (RPSD) e linguagens de especificação, é válida, uma vez que, conforme exposto anteriormente, é pretendida a implementação de uma plataforma em *software* que utilizará RPSD como modelo para a **captura** de especificações de sistemas digitais embutidos.

11.2. EXTENSÃO RPSD

Esta seção apresenta a extensão RPSD, em duas partes: na primeira, é dada a **descrição funcional** de RPSD, enquanto na segunda parte é dada a **definição formal** desta extensão.

11.2.1. DESCRIÇÃO FUNCIONAL

RPSD pode ser representada por um **grafo** que modela os **componentes** de um **sistema digital** e os **eventos** que podem ocorrer nele, promovendo mudanças nestes componentes (mudança no estado do sistema). A estrutura desse grafo é composta por elementos do tipo **lugar** e do tipo **transição**, os quais estão relacionados por setas orientadas denominadas **ramos**. A figura 11.1 apresenta os elementos constituintes de RPSD.

Os **lugares** da rede representam **componentes de armazenamento, componentes funcionais** (comportamentos) ou **estados locais** (modos) de componentes do sistema modelado. Um lugar pode ser ocupado por um ou mais *tokens*, cada qual semanticamente consistente com o tipo do lugar ao qual está associado, isto é, um *token* pode representar o **conteúdo** de uma **posição de armazenamento** (quando o lugar descreve um **componente de armazenamento**), um **item de dado** pertencente a uma lista ordenada, descritora de um **sample de dados** (presente na **entrada**, na **saída** ou **em processamento** numa **instância** de um lugar descritor de um **componente funcional**) ou um **componente** do sistema que encontra-se em **determinado estado** (quando o lugar descreve um estado local).

As **transições** representam **eventos** que podem ocorrer no sistema modelado, de acordo com um conjunto de **condições**, inclusive, **temporais**. Elas sinalizam **alterações** nos **componentes** (de armazenamento e/ou funcionais) ou mudanças de estados locais de

componentes do sistema, através da **leitura**, **escrita**, **remoção** e/ou **adição** de *tokens* a lugares.

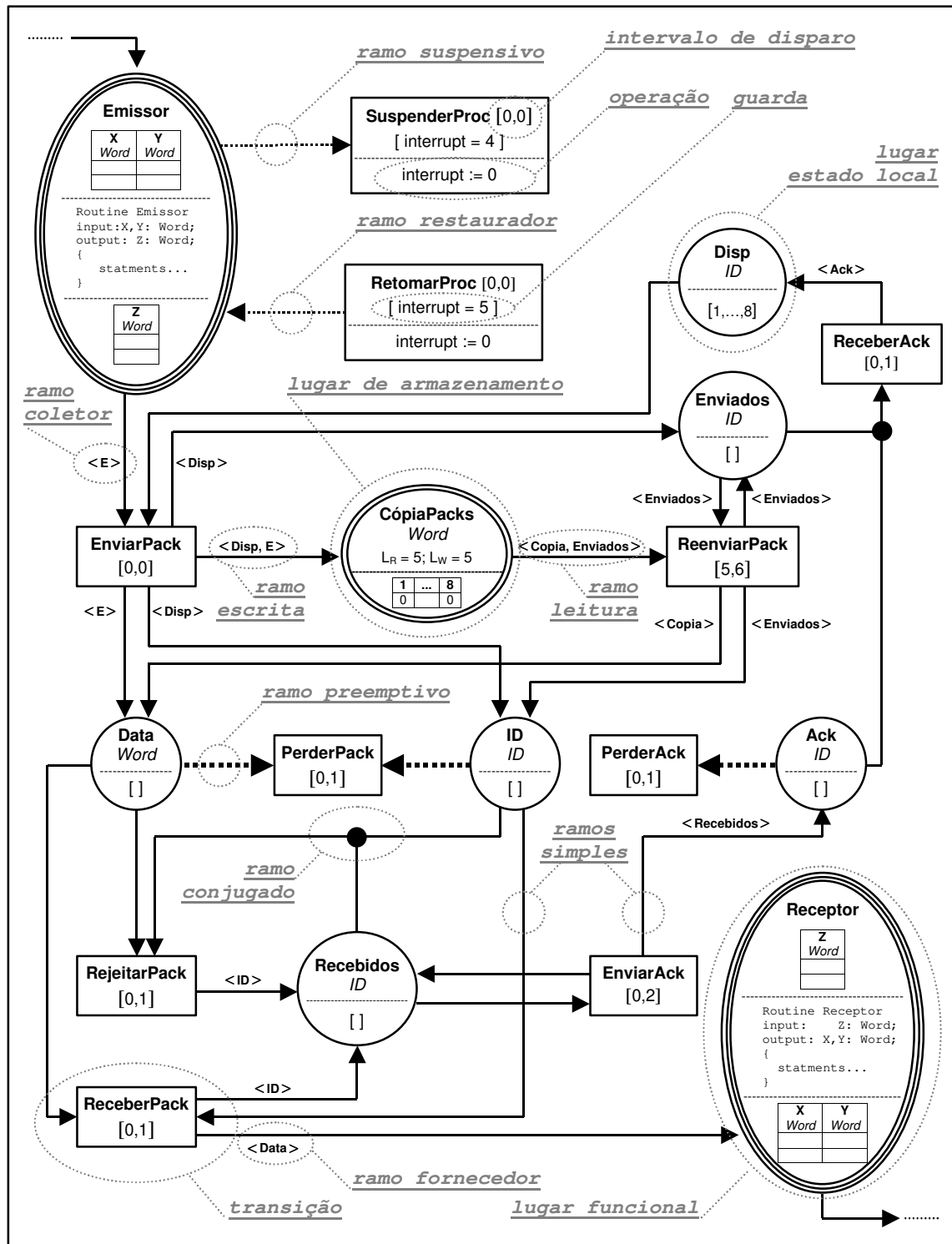


Figura 11.1. Elementos constituintes de RPSD.

As características de **lugares**, **ramos** e **transições**, além da **forma de funcionamento** de RPSD, encontram-se descritos na sequência.

11.2.1.1. LUGARES

RPSD possui 3 tipos de lugares, adequados para diferentes finalidades no processo de modelagem de um sistema. Tais tipos – denominados **estado local**, **armazenamento** e **funcional** – são descritos na sequência. Nesta descrição, **anotações** (A_n) refere-se à classe composta por conjuntos utilizados na definição de um ou mais lugares de um modelo RPSD. Maiores detalhes sobre a classe **anotações** são encontrados na seção 11.2.1.4.

11.2.1.1.1. TIPO I: ESTADO LOCAL

Lugares do tipo I são utilizados para indicar que um ou mais **componentes do sistema** (representados por *tokens*) de um **mesmo tipo** (por exemplo, **processos**, **unidades funcionais**, **de armazenamento** ou **de comunicação**), encontram-se num dado **estado local**.

A figura 11.2 mostra um lugar funcional capaz de modelar o estado de espera de até 4 **processos**, representados pelos *tokens* **p1**, **p2**, **p3** e **p4**. Este lugar faz parte de uma RPSD maior, a qual poderia modelar, por exemplo, **processos numa fila de espera**, aguardando para entrar em execução. Na marcação (estado) da figura, estão aguardando para entrar em execução, pela ordem de inserção no lugar (veja adiante), os processos **p3**, **p1** e **p2**.

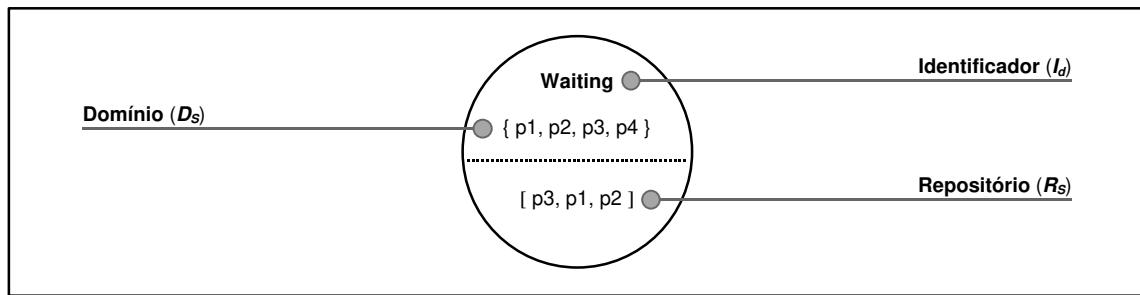


Figura 11.2. Componentes de um lugar tipo I – Estado Local.

Três **parâmetros** compõem um lugar do tipo I (figura 11.2):

- P1. Identificador (I_d):** Seqüência de caracteres finita e única que denomina o lugar;
- P2. Domínio (D_S):** Conjunto finito não-nulo, delimitando os valores de *tokens* que podem ser inseridos no lugar. D_S pertence à classe **anotações** (A_n – descrita adiante) da RPSD portadora do lugar; e
- P3. Repositório (R_S):** Subconjunto **ordenado** (pela ordem de inserção de *tokens* no lugar) do **domínio** D_S , indicando **quais** *tokens* estão presentes no lugar em um dado momento da rede. Corresponde à **marcação do lugar**. Não são permitidas múltiplas cópias de um *token* de mesmo valor. O conteúdo inicialmente atribuído a R_S (na definição do lugar) corresponde à marcação inicial M_0 do lugar.

Dos parâmetros acima descritos, apenas R_S é **dinâmico**, isto é, pode ser alterado ao longo da execução da rede.

Observe a forma de representação de um lugar do tipo I na figura 11.2. Nela, pode-se ver que lugares deste tipo são divididos em duas partes, por uma linha tracejada. Na parte superior do lugar encontram-se descritos seus **identificador** e **domínio** (D_S). Na parte inferior,

o **repositório** (R_S) do lugar é descrito por uma lista entre colchetes, a qual é cronologicamente ordenada pela ordem de inserção de *tokens* no lugar.

Duas **operações** são permitidas para lugares do tipo I:

- O1. Inserção de *tokens*:** Realizada com o auxílio de um *ramo de saída comum*, com origem em uma transição e término no lugar. Este ramo é rotulado com uma *função de inserção*, a qual determina qual conjunto de *tokens* (subconjunto de D_S) deve ser inserido no lugar pelo disparo da transição relacionada. Ramos de saída comuns são tratados adiante;
- O2. Remoção de *tokens*:** Realizada com o auxílio de um *ramo de entrada comum ou conjugado*, com origem no lugar e término em uma transição. *Ramos de entrada comuns* permitem a remoção de um conjunto de *tokens* de cardinalidade variável (dentro de limites **mínimo** e **máximo** permitidos). Pode-se determinar se os *tokens* deste conjunto serão removidos **sem qualquer critério** ou em **ordem cronológica**, seguindo-se o princípio FIFO (*First in First Out*), razão pela qual o **repositório** R_S é descrito, na representação gráfica do lugar, por uma lista entre colchetes (**fila**), ordenada cronologicamente. *Ramos de entrada conjugados* removem, **aleatoriamente**, um mesmo conjunto de *tokens* de cada um dos lugares ligados às suas origens, também dentro de limites mínimo e máximo permitidos. Ambos tipos de ramos são discutidos mais adiante.

O conjunto de *tokens* T a ser inserido num lugar p do tipo I, conectado a uma transição por um *ramo de saída comum*, deve ser compatível com o **domínio** desse lugar, além de ser nula a intersecção do mesmo com o conjunto de *tokens* presentes neste lugar, isto é,

$$T \subseteq D_S(p) \wedge T \cap R_S(p) = \emptyset.$$

A figura 11.3 mostra um trecho de modelo RPSD possuidor de dois lugares do tipo I (**Waiting** e **Processing**) e de uma transição (**process**), conectados por ramos de entrada e de saída comuns. O disparo da transição **process** corresponde ao evento de submissão de um processo para sua execução. A figura mostra duas **seqüências de disparo** da transição **process** (veja na seção 11.2.1.3 as **regras de habilitação** e **disparo** de uma transição). Na parte **a**), a ordem de retirada dos *tokens* é **cronológica** (veja na seção 11.2.1.2 a descrição do **ramo de entrada comum**). Assim, o primeiro *token* removido é **p3** (mais antigo), seguido da remoção de **p1** e de **p2**. Por outro lado, na parte **b**), não há rótulos nos ramos de entrada, denotando que a ordem de retirada de *tokens* é aleatória, razão pela qual os mesmos foram removidos de forma **não-determinística**. Esta opção na remoção de *tokens* permite que RPSD represente tanto comportamentos **determinísticos** como **não-determinísticos**.

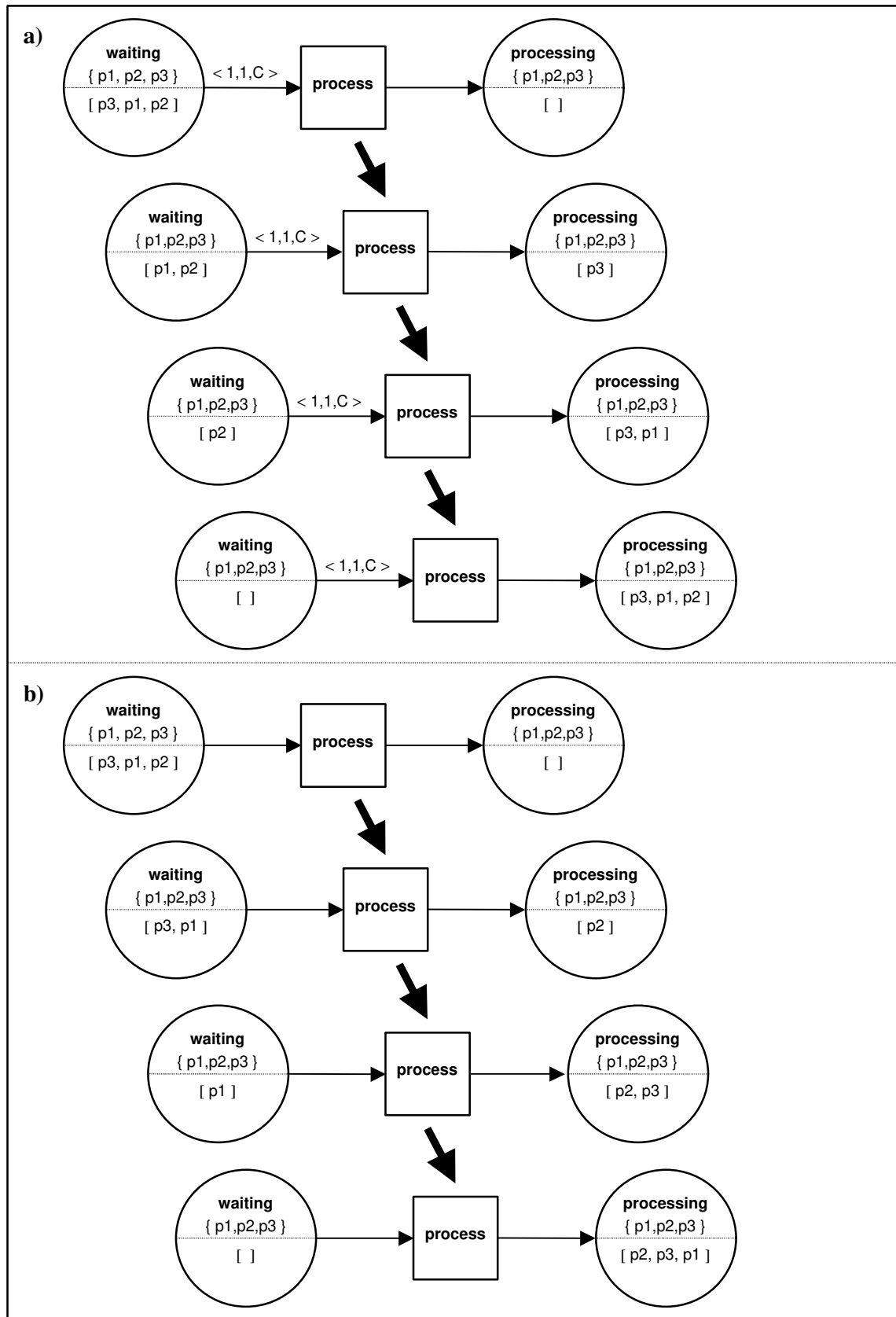


Figura 11.3. Sequências de disparo de process:
a) Cronologicamente; b) Aleatoriamente.

11.2.1.1.2. TIPO II: ARMAZENAMENTO

Lugares deste tipo são utilizados para representar **componentes de armazenamento** do sistema (tais como registradores e memórias), nos quais operações de *leitura/escrita* de dados (*tokens*) podem ser feitas.

A figura 11.4 mostra um lugar de armazenamento representando um banco de 4 registradores (**Reg. File**), quais sejam, **A**, **B**, **C** e **D**. Cada registrador armazena um valor (*token*) hexadecimal de 2 dígitos. Tanto operações de leitura quanto de escrita, em qualquer um dos registradores, são concretizadas 5 unidades de tempo após sua solicitação.

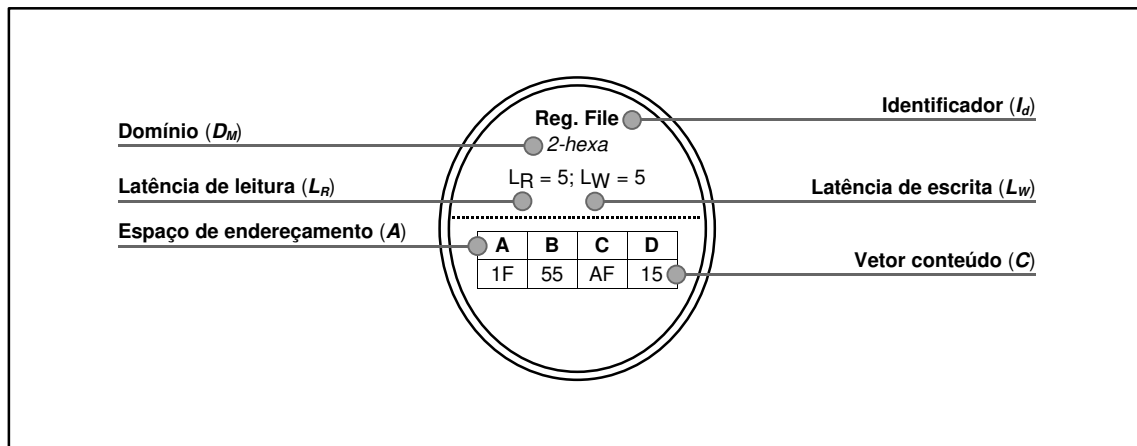


Figura 11.4. Componentes de um lugar tipo II – Armazenamento.

Um lugar do tipo II é definido por 6 parâmetros (figura 11.4):

- P1. Identificador (I_d):** Seqüência de caracteres finita e única que denomina o lugar;
- P2. Domínio (D_M):** Conjunto finito não-nulo, delimitando os valores de *tokens* que podem ser armazenados no lugar. D_M pertence à classe **anotações** (A_n) da RPSD portadora do lugar;
- P3. Espaço de endereçamento (A):** Conjunto ordenado finito delimitando a **faixa** de endereçamento. Pertencente à classe **anotações** (A_n) da RPSD portadora do lugar, A pode ser omitido na representação gráfica do lugar, ficando implícita a existência de uma única posição de armazenamento. Rótulos alfanuméricos podem ser utilizados para designar um endereço, ao invés de números;
- P4. Vetor conteúdo (C):** Vetor com número de entradas (posições) igual à cardinalidade do **espaço de endereçamento** (A) do lugar. Os valores inicialmente atribuídos às suas entradas correspondem à **marcação inicial** M_0 do lugar (na **classe inicial** C_0). Entradas não inicializadas correspondem ao valor *default* do **domínio** D_M do lugar. A atribuição de **nulo** ($\emptyset$) indica valores inicialmente **indeterminados** (**aleatórios**).
- P5. Latência de escrita (L_W):** Número natural (eventualmente nulo) representando quantas **unidades de tempo** são necessárias para que um dado (*token*) seja atualizado no lugar, isto é, o tempo gasto numa **operação de escrita**;
- P6. Latência de leitura (L_R):** Número natural (eventualmente nulo) representando quantas **unidades de tempo** são necessárias para que um dado (*token*) possa ser utilizado pela transição que o solicitou, isto é, o tempo gasto numa **operação de leitura**.

Observe as notações utilizadas na figura 11.4, para a descrição de lugares de armazenamento. Um lugar deste tipo é representado por um conjunto de duas elipses concêntricas, dividido em duas partes. Na parte superior encontram-se o **identificador** do lugar, seu **domínio** (*2-hexa*, denotando números de 2 dígitos, em notação hexadecimal) e suas **latências de leitura** (L_W) e de **escrita** (L_R). Na parte inferior do lugar, uma **tabela** descreve o **espaço de endereçamento** (cabecalho da tabela) e o **conteúdo** de cada endereço (**vetor conteúdo**).

Duas **operações** envolvendo lugares do tipo II são permitidas:

- O1. Escrita:** Realizada com o auxílio de um *ramo de escrita*, com origem em uma transição e término no lugar. Este ramo é rotulado com uma expressão que denota os *tokens* a serem escritos em determinados *endereços* (células de armazenamento) do lugar. *Ramos de escrita* são tratados adiante;
- O2. Leitura:** Realizada com o auxílio de um *ramo de leitura*, com origem no lugar e término numa transição. Este ramo é rotulado com uma expressão que descreve em quais *tokens* devem ser armazenados os valores lidos de determinados *endereços* do lugar. *Ramos de leitura* são detalhados adiante.

O tipo do dado (*token*) a ser escrito/lido deve ser compatível com o *domínio do lugar* (D_M) e o endereço fornecido deve ser válido (pertencente ao *espaço de endereçamento A*).

A figura 11.5 apresenta o uso de lugares do tipo II. Nesta rede, a transição **Transfer** transfere dados entre dois bancos de registradores (lugares **Reg. File I** e **Reg. File II**). A cada disparo da transição, o conteúdo de uma posição de armazenamento (**addr**) de **Reg. File I** é lido e armazenado num *token t*, o qual é escrito no endereço correspondente em **Reg. File II**. Após realizar a operação de escrita, a transição executa suas **operações** associadas, descritas na parte inferior da mesma, as quais são responsáveis pela atualização da próximo endereço a ser acessado. Quando não explicitado o tipo de uma variável da rede (como **addr**), assume-se o tipo **inteiro**, com valor *default* igual a zero. Apesar dos endereços serem denotados por nomes, eles correspondem a **tipos enumerados**, contados a partir de zero.

O número de *tokens* em lugares de armazenamento é estático, existindo um para cada endereço. O disparo de transições não interfere nesta quantidade, pois *tokens* são apenas **atualizados** (em operações de escrita) ou **lidos** (em operações de leitura). Assim, estes lugares não interferem no funcionamento da rede, a menos que endereços incorretos de leitura/escrita sejam fornecidos pelos ramos associados, o que impossibilitará a habilitação das transições relacionadas.

A permissão de fazer a leitura/escrita de vários dados ao mesmo tempo pode facilitar o trabalho de modelagem, desde que não se altere o comportamento esperado para o sistema. Assim, pode-se representar a rede da figura 11.5 pela rede da figura 11.6, mais sucinta. O resultado esperado será o mesmo (transferência de dados), embora o mecanismo adotado seja diferente. Uma observação importante é que, na rede da figura 11.5, após ser feita uma operação de leitura no lugar **Reg. File I**, deverá ter transcorrido um tempo igual ou superior à sua **latência de leitura** (5 unidades de tempo), para que outra operação (leitura ou escrita) possa ser feita no mesmo lugar. Desta forma, caso deseje-se que endereços de **Reg. File I** já lidos possam ser atualizados (por outra transição da rede, não mostrada nas figuras), a rede da figura 11.6 implicaria na necessidade de ter que esperar um tempo maior (20 unidades de tempo, igual à **latência de escrita multiplicada pelo número de dados envolvidos**), para que a operação de atualização (escrita) possa ser efetuada sobre o lugar.

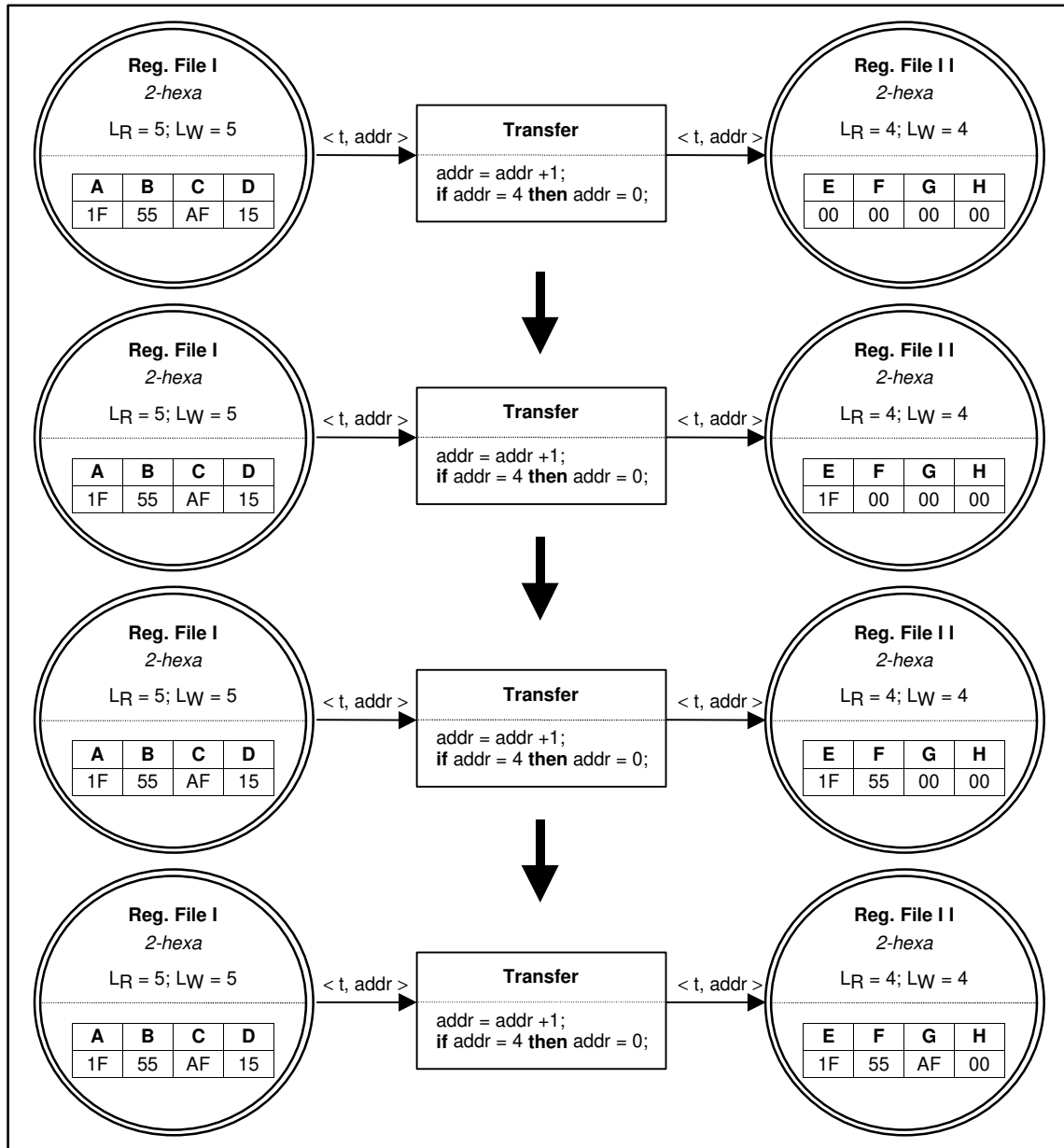


Figura 11.5. Transferência de dados entre lugares de armazenamento.

11.2.1.1.3. TIPO III: FUNCIONAL

Um lugar funcional é utilizado para indicar que um ou mais ***samples de dados*** (representados por ***listas ordenadas de tokens***) encontram-se em processamento por um ou mais ***componentes funcionais de mesmo tipo*** (cuja funcionalidade é descrita pelo parâmetro ***comportamento*** associado ao lugar).

A figura 11.7 mostra dois exemplos de lugares funcionais. Na parte **a)**, o lugar **Comparador** representa um **comportamento** (descrito por uma **rotina de programação**), responsável pela **seleção** do maior número inteiro, dentre dois valores submetidos. É possível a **comparação simultânea** de até 3 pares de inteiros. Isto equivale à modelagem de 3 comparadores (que podem ser implementados como **circuitos** ou como **rotinas de software**).

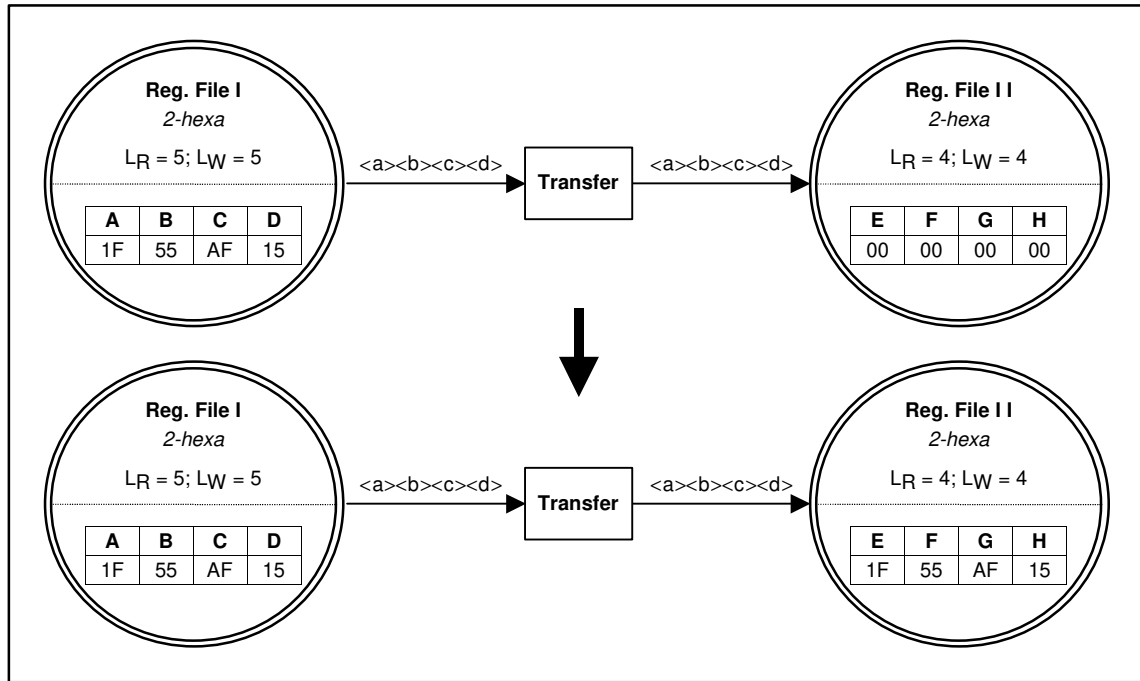


Figura 11.6. RPSD com comportamento similar à rede da figura 11.5.

Na parte **b)** da figura 11.7, o lugar **Clock Divider** representa dois divisores de *clock* por 4: a inserção de 4 *tokens* do tipo **clk** neste lugar é responsável pela geração de um *token clk* em uma de suas saídas. A figura não mostra como estas inserções são feitas: uma possibilidade é que, a transição responsável pela inserção de cada *token clk* no lugar, seja temporizada de acordo com o tempo de duração de um **ciclo** de *clock* do sistema modelado. O comportamento do lugar **Clock Divider** é dado por uma **RPSD-Transformacional (RPSD-T)**, descrita adiante), a qual constitui uma RPSD com algumas características especiais, de forma a garantir **interfaceamento adequado**. Como o lugar **Clock Divider** comporta até dois **samples de entrada**, há duas **instâncias** (cópias) da RPSD-T.

O funcionamento dos lugares funcionais da figura 11.7 ficará mais claro, à medida que compreendermos os componentes e regras de funcionamento desse tipo de lugar, descritos na sequência.

Sete parâmetros definem um lugar funcional (figura 11.7):

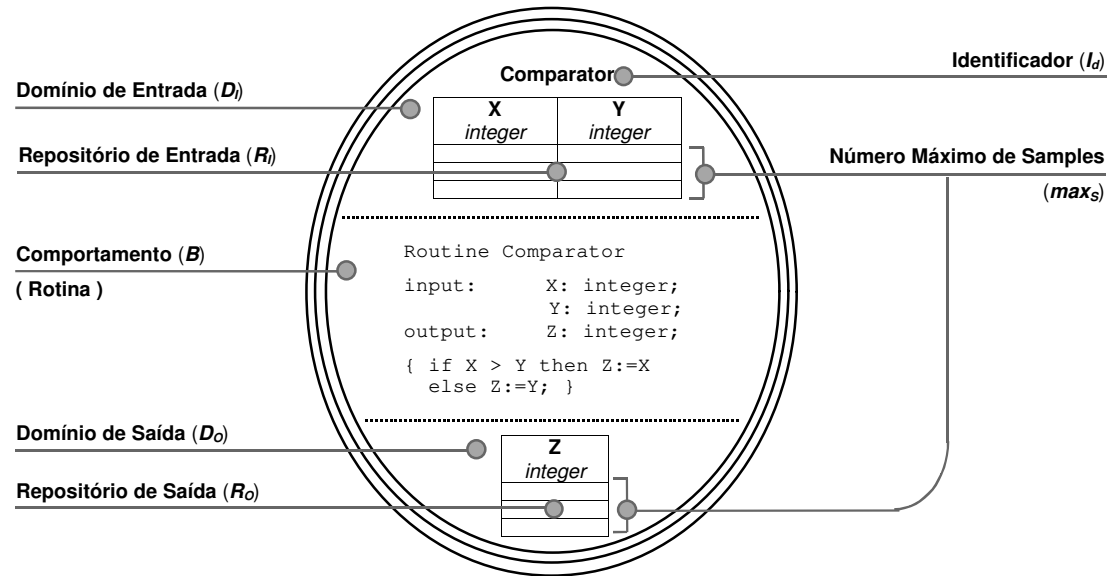
P1. Identificador (I_d): Seqüência de caracteres finita e única que denomina o lugar;

P2. Domínio de entrada (D_I): Lista ordenada de n **parâmetros de entrada**,

$$D_I(p) = ((l_{I1}, type_{I1}), (l_{I2}, type_{I2}), \dots, (l_{In}, type_{In})) \mid \\ l_{I1}, l_{I2}, \dots, l_{In} \in \text{Rótulos} \wedge type_{I1}, type_{I2}, \dots, type_{In} \in A_n \wedge n \geq 1,$$

onde A_n é a classe **anotações** da RPSD portadora do lugar e **Rótulos** é um dos conjuntos que compõem A_n . D_I indica **quantos** e de quais **tipos** devem ser os *tokens necessários* à execução do **comportamento B** associado ao lugar, conforme descrição adiante. O **tipo** de um *token* (indicado por **type**) é o conjunto que contém o valor associado ao *token*, de acordo com a definição da classe A_n . Um *token* pode pertencer a mais de um **tipo** (conjunto);

a) Lugar tipo III com comportamento descrito por rotina



b) Lugar tipo III com comportamento descrito por subrede

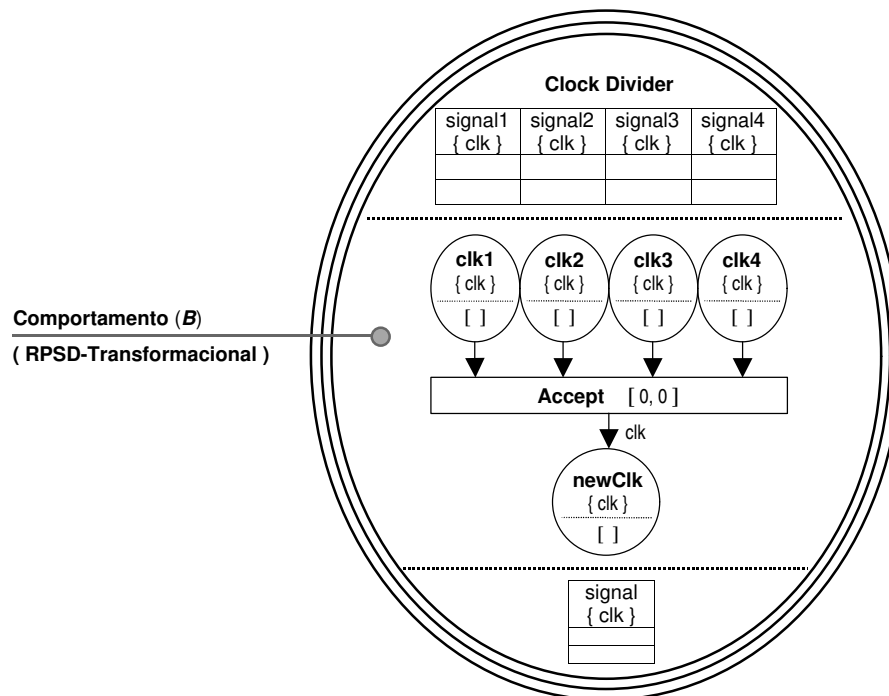


Figura 11.7. Componentes de um lugar tipo III – Funcional.

P3. Número máximo de samples (max_s): Número natural não-nulo, indicando a quantidade máxima de *samples* de dados que podem ocupar o lugar concomitantemente. Um *sample* de dados encontra-se em um de quatro estados possíveis, quais sejam:

- **sample inexistente:** *sample* não possui qualquer dado, isto é, não possui *token* algum pertencente aos *tipos* que compõem o **domínio de entrada D_I** do lugar. *Samples* neste estado não são contabilizados (para verificação de max_s);
- **sample de entrada em formação:** *sample* não encontra-se completo (falta *token* para um ou mais dos *tipos* listados no **domínio de entrada D_I**), uma vez que os seus *tokens* (argumentos) não precisam ser inseridos ao mesmo tempo;
- **sample em processamento:** *sample* entra neste estado após estar completo, ativando uma **instância** do **comportamento** associado ao lugar. Esta instância do **comportamento** é responsável pela transformação de um *sample de entrada completo* (argumentos para os **parâmetros** do **comportamento**) em um *sample de saída transformado*;
- **sample de saída:** *sample* resultante da transformação promovida pela instância do **comportamento** associado ao lugar. Neste estado, os *tipos* de *tokens* presentes no *sample* são designados pela lista D_O do **domínio de saída**, conforme definição adiante;

P4. Repositório de entrada (R_I): Estrutura armazenadora de *samples* de entrada, representada por uma matriz R_I de max_s linhas e n colunas, onde max_s é o **número máximo de samples**, e n a cardinalidade do **domínio de entrada D_I** do lugar. Cada linha de R_I é **dinamicamente** associada a uma **instância** (cópia) do **comportamento**, ou seja, cada linha armazena um *sample de entrada* (**conjunto de tokens representando argumentos**), necessário à ativação de uma determinada instância do **comportamento**. Por construção, a j -ésima coluna de R_I (indicada por R_{Ij}), $1 \leq j \leq n$, está associada ao j -ésimo *tipo* de D_I (indicado por $type_{Ij}$), isto é, cada coluna de R_I admite apenas um único *tipo* de *token*. Graficamente representado por uma tabela, R_I tem os índices de linhas crescentes de baixo para cima (primeira linha de R_I associada à última linha da tabela; última linha de R_I associada à primeira linha da tabela), e os índices de colunas crescentes da esquerda para a direita. Não é necessário que o **preenchimento** de uma linha de R_I ocorra de uma só vez. Cada vez que um *token* é **inserido** no lugar, ele é armazenado na **coluna menos ocupada** equivalente a seu *tipo*. Caso haja mais de uma coluna nesta condição, o *token* ocupará aquela de **menor índice**;

P5. Domínio de saída (D_O): Lista ordenada de m **resultados de saída**,

$$D_O(p) = ((l_{01}, type_{01}), (l_{02}, type_{02}), \dots, (l_{0m}, type_{0m})) \mid \\ l_{01}, l_{02}, \dots, l_{0m} \in \text{Rótulos} \wedge type_{01}, type_{02}, \dots, type_{0m} \in A_n \wedge m \geq 1,$$

onde A_n é a classe **anotações** da RPSD portadora do lugar e **Rótulos** é um dos conjuntos que compõem A_n . D_O indica os *tipos* dos *tokens* **resultantes** da execução do **comportamento** associado ao lugar;

P6. Repositório de saída (R_O): Estrutura armazenadora de *samples de saída*, representada por uma matriz R_O de max_s linhas e m colunas, onde max_s é o **número máximo de samples**, e m a cardinalidade do **domínio de saída D_O** do lugar. Cada linha de R_O é **dinamicamente** associada a uma **instância** do **comportamento**, ou seja, cada linha armazena um *sample de saída* (**conjunto de tokens representando resultados**), gerado por uma determinada **instância** do **comportamento** (subrede ou **rotina**). Por construção,

a k -ésima coluna de R_O (indicada por R_{Ok}), $1 \leq k \leq m$, está associada ao k -ésimo **tipo** de D_O (indicado por $type_{Ok}$), isto é, cada coluna de R_O admite apenas um único **tipo** de *token*. Podem ser armazenados, no máximo, max_S *samples de saída*. A **remoção** de um *sample de saída* ocorre através do **disparo** de uma transição associada ao lugar através de um **ramo coletor**. Este ramo é rotulado com uma expressão $\langle t_1, t_2, \dots, t_m \rangle$, onde $t_1, t_2, \dots, t_m$ são *tokens* com **tipos implicitamente arbitrados** pelo **domínio de saída** D_O do lugar, isto é,

$$t_1 \in type_{O1}, t_2 \in type_{O2}, \dots, t_m \in type_{Om};$$

P7. Comportamento (B): Estrutura responsável pela descrição da **funcionalidade** associada ao lugar. Uma **instância (cópia)** do **comportamento** é **ativada** quando um *sample* de *tokens* no **repositório de entrada** R_I encontra-se **completo**. Assim, pode existir **mais de uma** instância do **comportamento** ativa por vez, sendo o número de instâncias **limitado** a max_S . **Comportamento** é descrito em uma de duas formas, quais sejam:

- **RPSD-Transformacional (RPSD-T)**: Trata-se de uma RPSD cuja construção exige a presença de um **conjunto de lugares tipo I somente de entrada** (sem transições que insiram *tokens*) e de um **conjunto de lugares tipo I somente de saída** (sem transições que consumam *tokens*), associados, respectivamente, aos *tokens* que entram e que saem do **comportamento**. Mais especificamente, deve existir um **lugar de entrada do tipo I** (estado local) **para cada token do sample de entrada** e um **lugar de saída do tipo I para cada token do sample de saída**. Formalmente, para um lugar p do tipo III, tem-se a seguinte definição, onde o símbolo # representa cardinalidade e, P_I , o conjunto de lugares do tipo I de uma RPSD-T:

$RPSD-T(p)$ = RPSD com as seguintes restrições:

- i) $\forall type_{Ij} \in D_I(p) \Rightarrow \exists input_j \in P_I(RPSD-T(p)) \mid D_S(input_j) = type_{Ij}, j = 1, \dots, \#D_I(p)$; e
- ii) $\forall type_{Ok} \in D_O(p) \Rightarrow \exists output_k \in P_I(RPSD-T(p)) \mid D_S(output_k) = type_{Ok}, k = 1, \dots, \#D_O(p)$.

Uma instância da RPSD-T é criada para cada *sample* de *tokens* completo que se formar no **repositório de entrada** do lugar pai ($R_I(p)$). A operação de uma instância da RPSD-T cessa quando todos os seus lugares de saída ($output_k, k = 1, \dots, \#D_O(p)$) estiverem com um *token*, independentemente de existirem transições habilitadas na instância. Neste momento, os *tokens* presentes nestes lugares são disponibilizados no **repositório de saída** do lugar pai ($R_O(p)$). Após a remoção destes *tokens*, a instância deixará de existir.

RPSD permite **hierarquia estrutural**, uma vez que o comportamento **B**, de qualquer lugar funcional (tipo III) da RPSD-T, também pode ser descrito por outra RPSD-T e, assim, sucessivamente;

- **Rotina**: Trata-se de uma **listagem** de sentenças de programação (**statements**), descrevendo computações que devem ser executadas **seqüencialmente**. Estas sentenças podem usar os valores dos *tokens* contidos em um *sample* retirado do **repositório de entrada** do lugar, transformando-os em valores de *tokens* de um *sample* de saída. Os **parâmetros** de entrada e de saída são fixos, de acordo com a definição dos **domínios de entrada** (D_I) e de **saída** (D_O) do lugar. **Variáveis locais** podem ser declaradas. Para garantir um interfaceamento adequado, a **listagem** da rotina associada a um lugar p do tipo III, deverá seguir a formatação apresentada na figura 11.8.

```

Routine  $p$ 

input:  input1 : typeI1 , // tipos definidos em  $D_I(p)$ 
         input2 : typeI2 ,
         ...,
         inputn : typeIn ;

output: output1 : typeO1 , // tipos definidos em  $D_O(p)$ 
         output2 : typeO2 ,
         ...,
         outputm : typeOm ;

[var: ... ]           // declaração opcional
                        // de variáveis locais

{

    statements

}

```

Figura 11.8. Formatação para rotina associada a um lugar do tipo III.

Uma instância da rotina é criada para cada *sample* de *tokens* completo (linha completa) que se formar no *repositório de entrada* do lugar pai ($R_I(p)$). A operação da instância termina quando for executado o último *statement* ou um *statement return*.

A área de sistemas digitais embutidos é bastante ampla. Neste contexto, seguindo a orientação de **não limitarmos** o uso de **RPSD** a uma única classe específica de aplicações dentro desta área, nós **não definimos** e **nem fixamos** uma **linguagem formal** para a **descrição de rotinas** associadas a lugares funcionais, uma vez que diferentes **domínios de projeto** apresentam **especificidades** melhor atendidas por **diferentes linguagens**. Como um modelo descrito em RPSD pode representar, por exemplo, um sistema multiprocessador, seria interessante deixar a cargo do projetista a escolha da **linguagem** e de seu **compilador** correspondente, melhor adequados aos processos em *software* executados por um determinado processador. O que realmente importa, de fato, é exigir que a linguagem adotada para descrever a rotina associada a um lugar do tipo III permita **interfaceamento adequado**, seguindo a **formatação** dada na figura 11.8. No entanto, **para demonstrar** a forma de **funcionamento** de um **lugar funcional**, adotamos uma **pseudo-linguagem**, de sintaxe semelhante a C, com os seguintes grupos de *statements*:

- **atribuição**: **:=** (com operadores aritméticos comuns);
- **iteração**: **for...next** (com índices fixos);
- **seleção**: **if-then-else,**
 switch-case (com operadores relacionais comuns).

Nas partes **a)** e **b)** da figura 11.7, pode-se observar a notação gráfica de um lugar funcional. O conjunto de **três elipses concêntricas** é dividido em **três regiões**. Na região **superior**, encontram-se representados o **identificador (Comparador/Clock Divider)**, o **domínio de entrada (cabeçalho da tabela**, indicando o rótulo e tipo de cada **parâmetro de entrada** do **comportamento** associado) e o **repositório de entrada (corpo da tabela)** do lugar – as três linhas da tabela, na parte **a)**, inferem que o número máximo de *samples* (max_S) no lugar **Comparador** é 3 ($max_S(\text{Clock Divider}) = 2$). Na região **intermediária** do lugar funcional, encontra-se a descrição do **comportamento** associado. Na figura 11.7.a, o comportamento é dado por uma **rotina (Routine Comparador)**, devido à sua simplicidade. Comportamentos mais complexos, envolvendo a execução de atividades em paralelo (dentro do comportamento), exigem sua descrição através de uma RPSD-T, a qual já foi citada neste texto; para tornar sua compreensão mais direta, o comportamento do lugar **Clock Divider** é descrito por uma RPSD-T. Finalmente, na parte **inferior** do lugar, encontram-se representados o **domínio de saída (cabeçalho da tabela**, indicando o rótulo e o tipo de cada **resultado** gerado pela execução do **comportamento** associado) e o **repositório de saída (corpo da tabela)** do lugar. Este último, possui o mesmo número de linhas do repositório de entrada.

Lugares funcionais permitem dois tipos de **operações**:

O1. Inserção de tokens: Realizada com o auxílio de um *ramo fornecedor*, com origem em uma transição t e término num lugar funcional p . Tal ramo **insere** um conjunto de *tokens* no **repositório de entrada** do lugar quando do disparo da transição, de acordo com uma **função de inserção** $I(p,t)$, a qual segue a seguinte formatação:

$$I(p,t) = \langle \text{boolean}_1, \text{set}_1 \rangle | \langle \text{boolean}_2, \text{set}_2 \rangle | \dots | \langle \text{boolean}_n, \text{set}_n \rangle, n \geq 0.$$

Nesta função, a primeira condição booleana (boolean_i) satisfeita é responsável pela inserção do conjunto correspondente (set_i), desde que os *tokens* deste conjunto sejam compatíveis com os **parâmetros de entrada** do **comportamento** (definidos no **domínio de entrada** do lugar) e a capacidade do lugar não seja estourada. Caso ausente, a **função de inserção** corresponde ao conjunto de *tokens* retirados dos lugares ligados à transição considerada por **ramos de entrada comuns, conjugados, coletores ou de leitura** (ramos são tratados adiante).

Adotando-se a representação de matriz (dada anteriormente) para os **repositórios de entrada** (R_I) e **de saída** (R_O), toda vez que uma linha ℓ de R_I tiver sido preenchida, tem-se um *sample de entrada completo*, fazendo com que o mecanismo de controle da rede **ative** uma **instância** do **comportamento** B associado ao lugar.

A função $L(B_i)$, $1 \leq i \leq max_S$, associa **dinamicamente** um número natural l , $0 \leq l \leq max_S$, a uma instância B_i de B , onde l indica a linha de R_I onde B_i captura argumentos, assim como a linha de R_O onde B_i armazena resultados; $L(B_i) = 0$ indica que a instância B_i encontra-se **desativada**. RPSD exige que, cada *sample de entrada completo* seja processado pela **instância desativada** de **menor índice**. Por exemplo, supondo-se que B possui 5 instâncias – B_1 , B_2 , B_3 , B_4 e B_5 , e que a função L fornece as seguintes associações:

$$L(B_1) = 1; L(B_2) = 0; L(B_3) = 2; L(B_4) = 0; L(B_5) = 3,$$

somente B_2 e B_4 estão **desativadas**, pois a instância B_1 está associada à linha 1 dos **repositórios** R_I e R_O , a instância B_3 à linha 2, e a instância B_5 à linha 3. Neste caso, o próximo *sample de entrada* se formará na linha 4 de R_I , e será processado pela instância B_2 de B . Além disso, a função dinâmica L passará a indicar tal mudança, isto é, $L(B_2) = 4$.

Imediatamente após a linha ℓ de R_I ser totalmente preenchida, o *sample* é removido do **repositório de entrada** R_I , para processamento por uma instância B_i . Contudo, a linha $L(B_i)$ de R_I , inicialmente ℓ , não poderá ser ocupada (está **indisponível**) até que o *sample de saída* correspondente seja gerado e removido do lugar (linha $L(B_i)$ do **repositório de saída** R_O). Quando outro *sample* estiver completo, outra instância de B será **ativada**. De forma geral, após a **execução** de uma instância B_k , $1 \leq k \leq \max_S$, um *sample de saída* é **depositado** na $L(B_k)$ -ésima linha do **repositório de saída** R_O . A linha $L(B_k)$ de R_O será preenchida de uma só vez, após o término de execução da instância B_k . A instância B_k será **desativada** (atribuição de 0 a $L(B_k)$) somente quando o *sample de saída* correspondente for retirado de R_O ;

O2. Remoção de tokens: Realizada com o auxílio de um *ramo coletor*, com origem no lugar e término em uma transição. Quando do disparo da transição relacionada, este ramo remove o *sample de saída* que ocupa a linha de **menor índice** (mais baixa, na representação gráfica da tabela), dentre as linhas do **repositório de saída** que armazenam *samples de saída*. Isto garante a remoção dos resultados referentes às entradas mais antigas, dentre os resultados disponíveis. Após a **remoção** dos *tokens* de uma linha k de R_O , $1 \leq k \leq \max_S$, a linha k de R_I (antes **indisponível**) torna-se **disponível**, e todos os *tokens* do **repositório de entrada** situados entre as linhas $k + 1$ e $\max_S$, são deslocados uma linha para baixo (sentido decrescente de índices de linha), possibilitando a inserção de mais um *sample de tokens* no **repositório de entrada**. Além disso, todas as instâncias ativas B_i , $1 \leq i \leq \max_S$, tal que $L(B_i) > k$, têm a associação com linhas dos **repositórios** decrementada em 1, isto é,

$$L(B_i) := L(B_i) - 1.$$

Este controle é feito para que o número de *samples* em processamento e nos **repositórios de entrada** e **de saída** não ultrapasse o **número máximo de samples** permitido ($\max_S$). Como *samples de entrada* distintos podem apresentar tempos de execução diferentes, pode acontecer de uma instância B_i , associada ao *sample de entrada* mais antigo, estar em execução no momento em que, uma outra instância B_j , ativada posteriormente, gera um *sample de saída*. Assim, o *sample de saída* mais antigo no **repositório de saída** (linha $L(B_j)$) **não** corresponderá ao *sample de entrada* mais antigo (linha $L(B_i)$). Desta forma, para controlar o **fluxo** de *samples* processado por um lugar funcional, recomenda-se que cada *sample* carregue um **número de controle**, que possa ser verificado após a remoção do *sample*, já transformado, do lugar. Para tal, em relação ao lugar funcional, basta adicionar mais um **tipo** às definições dos **repositórios** R_I e R_O .

Cada *token* do conjunto T a ser inserido num lugar funcional p , conectado a uma transição por um *ramo fornecedor*, deve pertencer a um dos **tipos** presentes na definição do **domínio de entrada** desse lugar. Além disso, T não pode estourar a capacidade de p . Formalmente, tem-se:

- a) $\forall t \in T \Rightarrow \exists \text{type}_x \in D_I(p) \mid t \in \text{type}_x$; e
- b) $\forall t, w \in T \Rightarrow \exists m, n \in R_I \mid \begin{array}{l} \text{indices}(m) \neq \text{indices}(n) \wedge \\ \text{status}(m) = \text{disponível} \wedge \\ \text{status}(n) = \text{disponível} \wedge \\ \text{type}(t) = \text{type}(m) \wedge \\ \text{type}(w) = \text{type}(n) \end{array}$

onde *indices*, *status* e *type* são funções que retornam, respectivamente, os *índices* de um elemento da matriz, o *estado* (*disponível* ou *indisponível*) de uma linha da matriz e o *tipo* de um *token*.

Ao contrário de lugares do tipo I, lugares funcionais permitem a presença de mais de um *token* de mesmo tipo, pois o *token* ocupa um elemento de armazenamento único, o qual não pode ser sobrescrito pelo disparo de outra transição, até que os resultados derivados do *sample* do qual faz parte sejam removidos do lugar. O número de *tokens* nos **repositórios** de um lugar funcional é variável, porém limitado ao valor:

$$(\max\{\#D_I, \#D_O\}) \times \max_S.$$

Os *tokens* são deslocados dentro de um lugar funcional através de mecanismos bem definidos, os quais podem ser abstraídos, sem qualquer perda de compreensão do que um lugar desse tipo representa dentro do sistema modelado. Basicamente, um lugar funcional representa um **comportamento**, com **parâmetros de entrada** e de **saída** bem definidos. **Argumentos** para os parâmetros de entrada podem ser inseridos a qualquer instante, pouco a pouco ou de uma só vez. Quando se tem argumentos necessários para execução do comportamento, esta ocorre e seus resultados são armazenados para retirada no momento que se fizer necessária. O principal **conceito** embutido na especificação de um **lugar funcional** é o da possibilidade de **parametrizar funções** que podem ser **executadas em paralelo**.

Outro importante conceito é o da possibilidade de representação de comportamentos complexos, uma vez que um comportamento pode ser representado por um **subrede** (RPSD-T) cujos comportamentos podem também ser descritos por subredes e, assim, sucessivamente. No processo de modelagem de um sistema, **comportamentos concorrentes** devem ser descritos por meio de **vários lugares funcionais**, enquanto **comportamentos sequenciais** podem ser representados por uma **rotina** associada a um lugar funcional.

Todas as instâncias do **comportamento B** associado a um lugar funcional podem ser **abortadas**, **suspensas** ou **retomadas**, de acordo com o disparo de transições ligadas ao lugar através de **ramos** voltados para o **tratamento de exceções**, descritos mais adiante.

As figuras 11.9 a 11.11 ilustram o funcionamento de um lugar do tipo III, no qual o comportamento é descrito por uma rotina.

A figura 11.9 apresenta um comparador entre 2 números inteiros, os quais são obtidos de um componente de armazenamento do sistema modelado. O maior dentre os dois números é escolhido, sendo inserido em outro componente de armazenamento. O comparador é implementado por um lugar funcional (**Comparador**), enquanto os componentes de armazenamento são representados por lugares do tipo II (**Argumentos** e **Resultados**). Há duas transições responsáveis pelo fornecimento de **argumentos** (*tokens*) para o **Comparador**, quais sejam, **Inserir1Argumento** e **Inserir2Argumentos**. A primeira destas transições insere 1 argumento por vez, enquanto a segunda insere 2 argumentos de cada vez. A **habilitação** das transições depende da variável **controle**, utilizada em suas **guardas** (conforme será visto mais adiante, a guarda é uma expressão booleana, anotada entre colchetes, que impede a habilitação da transição relacionada quando seu valor é **FALSE**): quando **controle** é igual a 0 ou a 1, a transição **Inserir1Argumento** está habilitada; para **controle** igual a 2, a transição habilitada passa a ser **Inserir2Argumentos**. Complementando o modelo, o maior número inteiro de cada *sample* de dados (par de números inteiros) é armazenado no lugar **Resultados**, através do disparo da transição **TransferirResultado**. O primeiro resultado gerado será guardado no endereço **R1**, o segundo em **R2** e o terceiro em **R3**. O conteúdo do lugar **Argumentos** é fornecido por outras transições, não mostradas no modelo. Também não são mostradas as transições responsáveis pela leitura dos dados presentes no lugar **Resultados**. Este processo é repetido enquanto a variável booleana **cont** for verdadeira (**TRUE**).

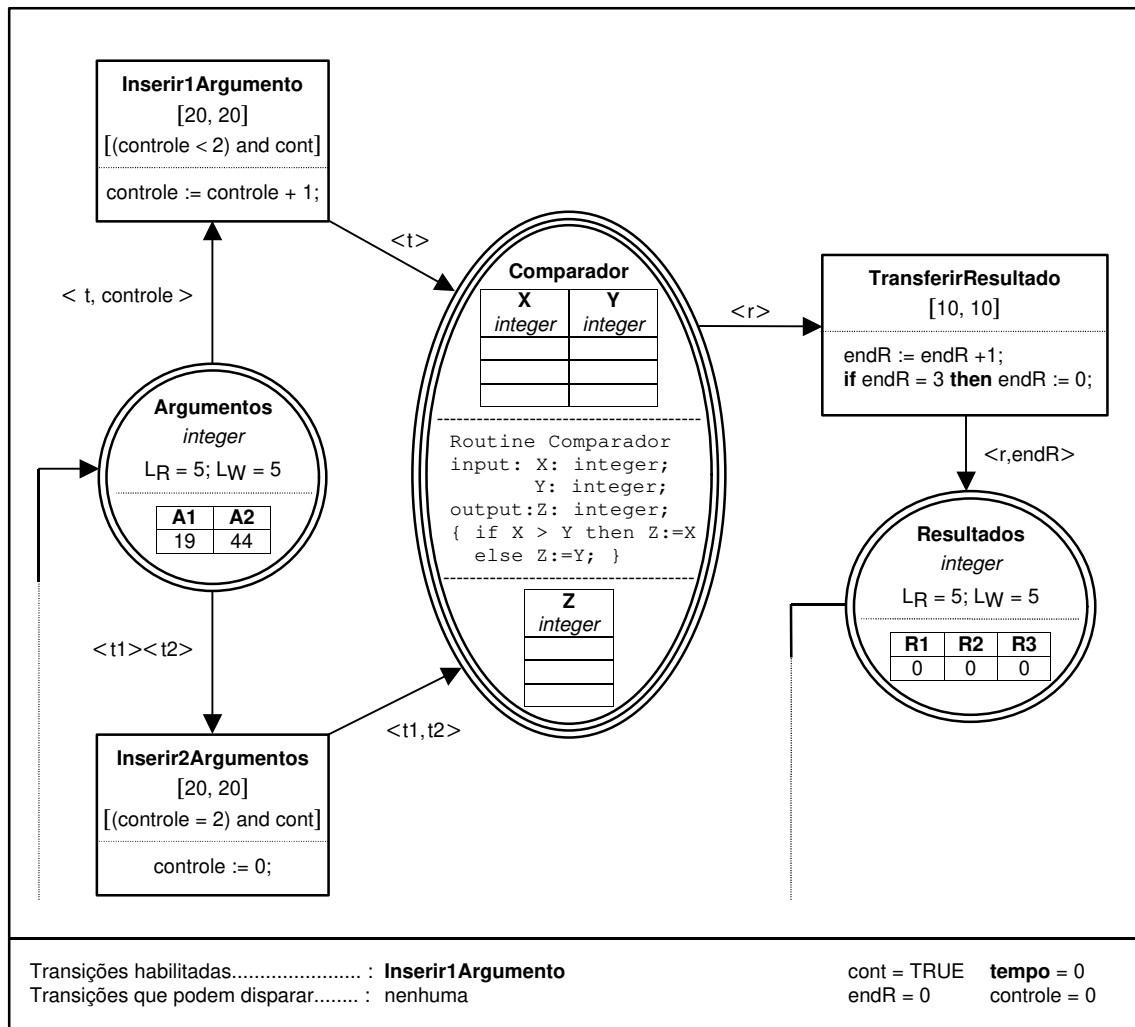


Figura 11.9. Utilização de lugar funcional para comparação entre 2 números inteiros.

Supondo que a transição **Inserir1Argumento** tenha acabado de ficar habilitada no **tempo 0** da rede mostrada na figura 11.9, a mesma não poderá disparar, uma vez que não atende suas **regras de disparo**, as quais constituem restrições temporais. Conforme será visto na seção 11.2.1.3, o intervalo de disparo [20,20] denota que a transição terá que disparar 20 *u.t.*'s (**unidades de tempo**) após a sua habilitação. Este não é o caso ilustrado na figura, pois não transcorreu sequer 1 *u.t.* (**tempo** = 0). Por outro lado, na rede da figura 11.10, já foram transcorridas 25 *u.t.*'s, o que possibilitou o **disparo** da transição **Inserir1Argumento** (o que ocorreu no **tempo 20**) e a **inserção** do **argumento** no **repositório de entrada** do **Comparador** (o que ocorreu no **tempo 25**, devido à latência de leitura do lugar **Argumentos**). Como o lugar **Argumentos** ficou liberado para nova operação (escrita/leitura) somente no tempo 25, a transição **Inserir1Argumento** só poderá disparar novamente quando o **tempo** for igual a 45 (20 *u.t.*'s após 25). No **tempo 50**, o segundo disparo de **Inserir1Argumento** terá acarretado a inserção do **segundo argumento** necessário à **ativação** de uma **instância** do comportamento **Comparador**. Supondo que a rotina gaste no mínimo 100 *u.t.*'s, quando a transição **Inserir2Argumentos** disparar no **tempo 70**, a primeira instância gerada para o comportamento ainda estará em execução e, como o disparo de tal transição promove a ativação de uma nova instância (no **tempo 75**), teremos uma situação na qual duas instâncias estão ativas ao mesmo tempo.

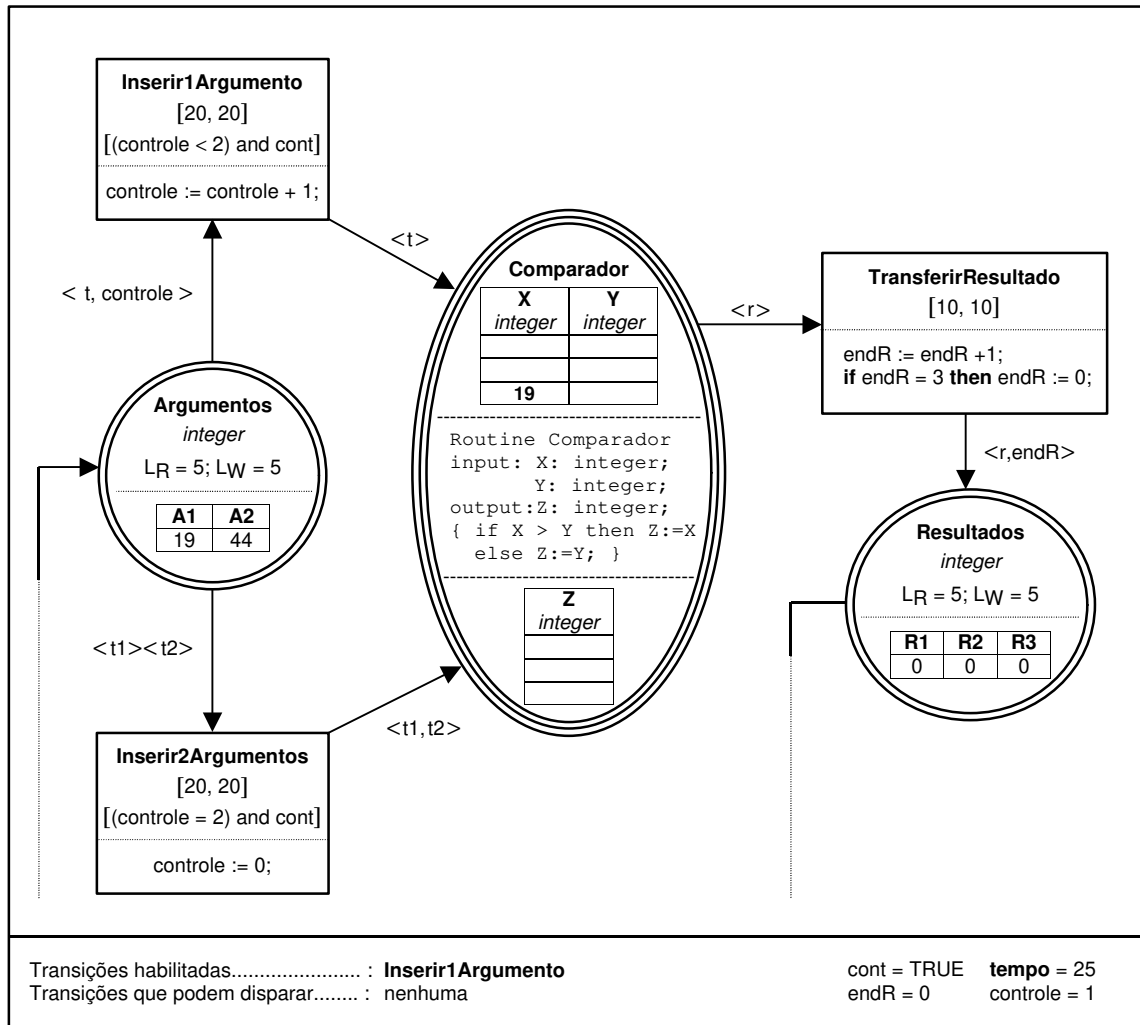


Figura 11.10. Rede da figura 11.9 após o disparo de Inserir1Argumento.

Para manter o número total de *samples* no lugar limitado a *max_s* (considerados os *samples em formação* no repositório de entrada, os *samples em processamento* por alguma instância e os *samples resultantes* no repositório de saída), as linhas do repositório de entrada que foram ocupadas por algum *sample em processamento* ou *já processado*, não poderão ser ocupadas até que o *sample* resultante correspondente seja **totalmente** removido do repositório de saída. A figura 11.11 mostra a representação dessa **indisponibilidade** de linhas do repositório de saída, tomando-se uma seqüência de marcações pelas quais o lugar **Comparador** pode passar, ao longo de uma seqüência de disparo de transições na rede adotada como exemplo. Observe que, enquanto o valor 44, correspondente ao primeiro *sample* processado (o qual ocupava a primeira linha do repositório de entrada, de baixo para cima) não tiver sido removido do repositório de saída, sua linha no repositório de entrada não poderá ser ocupada. Esta situação perdura até a 5ª marcação mostrada na figura 11.11. Na 6ª marcação, o valor 44 já foi removido do lugar, o que ocasionou no deslocamento, em uma posição para baixo, dos conteúdos presentes nos *repositórios de entrada e de saída* do lugar.

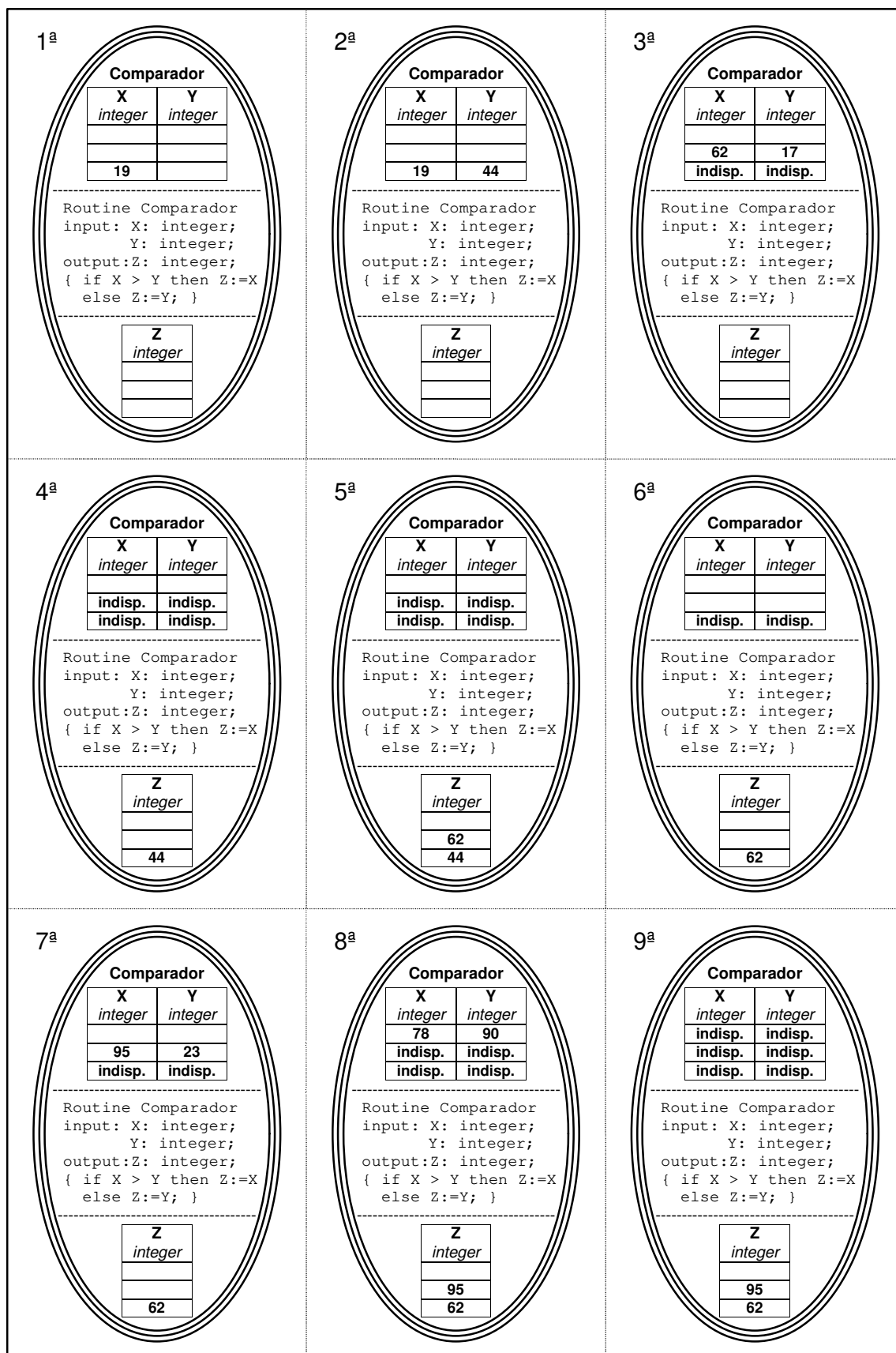


Figura 11.11. Sequência de marcações para o lugar funcional da rede da figura 11.9.

11.2.1.2. RAMOS

Lugares e transições são conectados por *ramos*, os quais são classificados, grosso modo, em 2 tipos: de *entrada* (com origem num lugar e término numa transição) e de *saída* (com origem numa transição e término num lugar). Um lugar conectado a uma transição por meio de um ramo de entrada é denominado *lugar de entrada* da transição. Analogamente, define-se *lugar de saída*. Há 6 tipos de ramos de entrada e 4 tipos de ramos de saída, denotando a forma pela qual lugares e transições estão relacionados.

Os tipos de *ramos de entrada* são:

- a) **Comum:** Permite que uma transição t remova um conjunto de dados (*tokens*) de um lugar p do tipo I. Este ramo é rotulado com uma tripla, denominada *expressão de remoção*, dada por $E(p,t) = \langle \min, \max, \text{ordem} \rangle$, onde $\min$ e $\max$ denotam, respectivamente, os números **mínimo** e **máximo** de *tokens* a retirar do lugar p , e $\text{ordem} \in \{R, C\}$ denota se os *tokens* serão retirados seguindo-se um critério **não-determinístico** (representado por R , de ordem randômica) ou **determinístico** (de acordo com o ordem de inserção no lugar – princípio FIFO, representado por C , de ordem cronológica). Caso o rótulo seja omitido, assume-se $E(p,t) = \langle 1, 1, R \rangle$. Representado por linha contínua (veja figura 11.3);
- b) **Conjugado:** Permite que uma transição t remova um mesmo conjunto de dados (*tokens*) de um grupo de lugares p_i , $i = 1, 2, \dots, n$ ($n \geq 2$), do tipo I. Um ramo desse tipo retira, aleatoriamente, um mesmo conjunto de *tokens* de cada um dos lugares ligados às suas origens, razão pela qual é rotulado por uma *expressão de remoção* $E(p_i,t) = \langle \min, \max \rangle$, para algum i entre 1 e n , denotando, respectivamente, o número mínimo e máximo de *tokens* a retirar de cada um dos lugares relacionados. Caso o rótulo seja omitido, assume-se $E(p_i,t) = \langle 1, 1 \rangle$. Representado por linha contínua (veja figura 11.1);
- c) **Leitura:** Permite que uma transição t faça a leitura de um ou mais dados (*tokens*) presentes em um lugar p do tipo II. Este tipo de ramo é rotulado com uma *expressão de leitura* $R(p,t)$, no seguinte formato:

$$R(p,t) = \langle t_1, a_1 \rangle \langle t_2, a_2 \rangle \dots \langle t_n, a_n \rangle$$

onde $t_1, t_2, \dots, t_n$ representam nomes de *tokens* que armazenarão, cada qual, o conteúdo de um dos endereços $a_1, a_2, \dots, a_n$ do lugar relacionado. Caso os endereços sejam omitidos, $t_1, t_2, \dots, t_n$ assumirão o conteúdo das n primeiras posições de armazenamento do lugar. Representado por linha contínua (veja figuras 11.5, 11.6 e 11.9);

- d) **Coletor:** Permite que uma transição t remova um *sample* de dados (*tokens*) completo do repositório de saída R_o de um lugar p do tipo III. Um ramo coletor deve ser rotulado com uma *expressão coletora*

$$C(p,t) = \langle t_1, t_2, \dots, t_m \rangle,$$

onde $t_1, t_2, \dots, t_m$ são *tokens* com *tipos* implicitamente equivalentes àqueles definidos no *domínio de saída* do lugar, isto é, sendo o domínio de saída de p dado por

$$D_o(p) = (\text{type}_{o1}, \text{type}_{o2}, \dots, \text{type}_{om}), m \geq 1,$$

então

$$t_1 \in \text{type}_{o1}, t_2 \in \text{type}_{o2}, \dots, t_m \in \text{type}_{om}.$$

Somente *samples* completos podem ser retirados. Representado por linha contínua (veja figura 11.9);

- e) **Suspensivo:** Este é um ramo voltado para o **tratamento de exceções**, que permite uma transição **suspender temporariamente** as **instâncias ativas** do comportamento associado a um lugar do tipo III. Caso o comportamento do lugar seja descrito por uma **RPSD-T**, as instâncias em execução serão suspensas através da **impossibilidade de habilitação** de suas transições (há uma **variável de controle** *ENABLE* implicitamente associada a cada transição, cujo valor passa para **FALSE** quando da atuação de um ramo suspensivo). No caso do comportamento ser descrito por uma **rotina**, cada instância termina de executar o *statement* em processamento e fica **suspensa**. Representado por linha tracejada (veja figura 11.1);
- f) **Preemptivo:** Este também é um ramo voltado para o **tratamento de exceções**, permitindo que uma transição **remova todos** os *tokens* presentes em um lugar do tipo I ou III, independentemente de quaisquer condições. Serve também para **resetar** as células de armazenamento de lugares do tipo II. Representado por linha tracejada grossa (veja figura 11.1).

Os tipos de *ramos de saída* são:

- a) **Comum:** Permite que uma transição *t* **adicione** um conjunto de dados (*tokens*) a um lugar *p* do tipo I. Um ramo desse tipo insere um conjunto de *tokens* no lugar quando do disparo da transição, de acordo com uma **função de inserção** $I(p,t)$, a qual é composta por um conjunto finito de **alternativas de pares**, cada qual formado por uma **condição booleana** e um **conjunto de tokens**, seguindo a formatação:

$$I(p,t) = \langle \text{boolean}_1, \text{set}_1 \rangle | \langle \text{boolean}_2, \text{set}_2 \rangle | \dots | \langle \text{boolean}_n, \text{set}_n \rangle, n \geq 0.$$

A primeira condição booleana satisfeita é responsável pela inserção do conjunto correspondente, desde que não existam *tokens* em comum entre o conjunto a ser inserido e o repositório (*R_s*) do lugar em foco. Caso ausente, a função de inserção corresponde ao conjunto de *tokens* retirados dos lugares ligados à transição considerada por **ramos de entrada comuns, conjugados, coletores ou de leitura**. Também neste caso, uma das condições de habilitação da transição relacionada é que não haja *tokens* em comum entre o conjunto a ser inserido e o repositório do lugar considerado. Representado por linha contínua (veja figura 11.3);

- b) **Escrita:** Permite a **escrita** de um ou mais dados (*tokens*) em um lugar do tipo II. Estes ramos são rotulados com uma **expressão de escrita**, no seguinte formato:

$$\langle t_1, a_1 \rangle \langle t_2, a_2 \rangle \dots \langle t_n, a_n \rangle$$

onde *t₁, t₂, ..., t_n* representam constantes ou nomes de *tokens*, compatíveis com o *domínio do lugar* (*D_M*), que deverão ser armazenados nos endereços *a₁, a₂, ..., a_n*, válidos de acordo com a definição do **espaço de endereçamento** (*A*) do lugar relacionado. Caso os endereços sejam omitidos, *t₁, t₂, ..., t_n* serão escritos nas *n* primeiras posições de armazenamento do lugar. Representado por linha contínua (veja figuras 11.5 e 11.6);

- c) **Fornecedores:** Permite que uma transição *t* **adicione** um conjunto de argumentos (*tokens*) a um lugar *p* do tipo III. Tal ramo insere um conjunto de *tokens* no *repositório de entrada* do lugar quando do disparo da transição, de acordo com uma **função de inserção** $I(p,t)$, a qual segue a mesma formatação dada para ramos de saída comuns, ou seja:

$$I(p,t) = \langle \text{boolean}_1, \text{set}_1 \rangle | \langle \text{boolean}_2, \text{set}_2 \rangle | \dots | \langle \text{boolean}_n, \text{set}_n \rangle, n \geq 0.$$

A primeira condição booleana satisfeita é responsável pela inserção do conjunto correspondente, desde que os *tokens* deste conjunto sejam compatíveis com os parâmetros de entrada do comportamento (definidos no *domínio de entrada* do lugar) e a capacidade do lugar não seja estourada. Caso ausente, a função de inserção corresponde ao conjunto de *tokens* retirados dos lugares ligados à transição considerada por **ramos de entrada comuns, conjugados, coletores ou de leitura**. Representado por linha contínua (veja figura 11.9);

- d) **Restauradores**: Este é outro tipo de ramo envolvido no tratamento de exceções, o qual permite que uma transição **retome** um **comportamento** descrito por um lugar do tipo III, que tenha sido suspenso pelo disparo de outra transição, ligada ao lugar por um ramo *suspensivo*. Caso o comportamento do lugar seja descrito por uma **RPSD-T**, as **instâncias suspensas voltam a executar**, através da **não interferência** nas regras de habilitação de suas transições (valor da **variável de controle ENABLE** passa para **TRUE**). Caso o comportamento seja descrito por uma **rotina**, cada instância retoma sua execução no **statement seguinte** àquele em que o processamento havia sido suspenso. Representado por linha tracejada (veja figura 11.1).

11.2.1.3. TRANSIÇÕES

Transições representam *eventos* que podem ocorrer no **sistema** modelado. Diz-se que uma transição está **habilitada** quando o *evento* que ela representa **pode ocorrer** no instante considerado, o que é constatado através de certas **regras e condições**. O **disparo** de uma transição habilitada corresponde à **ocorrência** do evento que ela representa. De acordo com os lugares envolvidos, o evento pode promover:

- A **inserção** ou a **remoção** de **tokens** representantes de componentes do sistema, em lugares do **tipo I**;
- A **leitura** ou a **atualização** (escrita) de **células de armazenamento**, em lugares do **tipo II**;
- A **inserção** de **argumentos** para os **parâmetros de entrada** necessários à ativação de **comportamentos** associados a lugares do **tipo III**, ou a **retirada** de **resultados** do processamento desses **comportamentos**.

Transições podem ser utilizadas na modelagem de **exceções**, nas quais os eventos podem ter como conseqüências:

- A **remoção** de **todos** os **tokens** representantes de componentes do sistema, em lugares do **tipo I**;
- A **limpeza** (**reset**) de **todas** as **células de armazenamento**, em lugares do **tipo II**;
- A **suspensão temporária** ou a **retomada** de **comportamentos** associados a lugares do **tipo III**;
- A **suspensão definitiva** (**preempção**) de **comportamentos** associados a lugares do **tipo III**, com a **remoção** de **todos** os **dados** (**tokens**) do lugar, inclusive das subredes existentes.

Quatro parâmetros definem uma transição (figura 11.12):

- P1. Identificador (I_d)**: Seqüência de caracteres finita e única que denomina a transição;
- P2. Guarda da transição (G)**: **Expressão booleana** para **habilitação** de uma transição. Pode envolver **subconjuntos de tokens** pertencentes ao **domínio** de cada lugar relacionado à transição, **variáveis da rede** (**memória da rede**) e **funções pré-definidas**. Sua ausência corresponde ao valor lógico **TRUE**. Representada entre colchetes;

- P3. Operações da transição (T_o):** Conjunto formado por *operações de atribuição* (*variável := expressão*) e *comandos condicionais* (*If-Then-Else*), capazes de alterar a *memória da rede*, a qual é representada por um *conjunto de variáveis*. Anotado dentro da transição, é executado concomitantemente com o disparo desta;
- P4. Intervalo de disparo estático (T_s):** Intervalo de limites inteiros não-negativos, indicando os tempos de atraso *mínimo* e *máximo* para o disparo de uma transição após a sua habilitação. $T_s = [LDI, LDS]$, onde **LDI** é o *limite de disparo inferior* e **LDS** é o *limite de disparo superior*. Duas observações importantes:
- O intervalo $[0,0]$ denota *disparo imediato*, significando que, uma vez habilitada, a transição irá disparar *imediatamente*;
 - A *ausência de intervalo* denota que a transição representa um *evento ocasional* do sistema, isto é, um evento que poderá ocorrer ou não, como, por exemplo, a ativação de um sinal de *reset*.

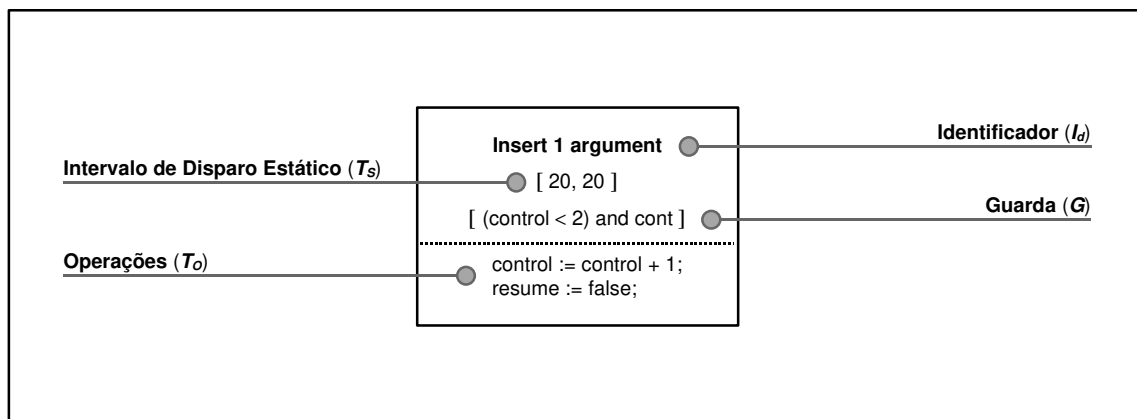


Figura 11.12. Componentes de uma transição.

As *regras de habilitação* e *disparo* de uma transição RPSD dependem dos *ramos* envolvidos e da *guarda da transição*.

Um lugar conectado a uma transição por meio de um ramo de entrada é denominado *lugar de entrada* da transição. Analogamente, define-se *lugar de saída*. Como uma transição pode estar ligada a seus lugares de entrada e de saída por **diferentes tipos de ramos**, cada um desses ramos define uma *regra de habilitação referente ao ramo*, a qual é uma das condições que devem ser atendidas para que a transição torne-se habilitada.

A regra de habilitação referente a um **ramo de entrada comum**, rotulado com uma tripla $\langle min, max, ordem \rangle$, é que o lugar correspondente possua ao menos um subconjunto de *tokens* de cardinalidade entre *min* e *max*. Pode ocorrer de mais de um subconjunto satisfazer tal exigência, independentemente da ordem ser *aleatória* ou *cronológica*. Neste caso, somente um destes subconjuntos será removido do lugar quando do disparo da transição, o que dependerá de outras condições, relativas à guarda da transição e aos *tokens* presentes nos seus lugares de saída.

Um **ramo de entrada conjugado** estende a regra de habilitação de um ramo de entrada comum ao grupo de lugares ao qual está ligado.

A regra de habilitação de um **ramo de saída comum** é que o lugar ligado a ele não possua *tokens* pertencentes ao conjunto resultante da *função de inserção de tokens*, além desses *tokens* serem de tipos **compatíveis** com o *domínio do lugar* (D_S).

A regra de habilitação referente a um **ramo de leitura** possui duas condições. A primeira é que a *expressão de leitura* do ramo seja **compatível** com o lugar, de maneira que os **endereços**, caso fornecidos, sejam **válidos** (isto é, pertençam ao *espaço de endereçamento* A). A segunda condição é relativa às *latências* de *leitura* e *escrita*: para que uma operação de leitura possa ser feita sobre o lugar relacionado p , é necessário que tenha transcorrido um intervalo de tempo mínimo T , desde a última operação O (de leitura ou de escrita) envolvendo p , tal que:

se O = operação de leitura $\Rightarrow T = L_R(p) \times n$, caso contrário

se O = operação de escrita $\Rightarrow T = L_W(p) \times n$,

onde n denota a quantidade de *tokens* envolvidos na operação O . Por exemplo, se a última operação envolvendo o lugar p foi a escrita de 5 *tokens*, então é necessário que tenha transcorrido um intervalo de $L_W \times 5$ unidades de tempo para que qualquer outra operação (escrita ou leitura) possa ser feita sobre o lugar.

Um **ramo de escrita** possui regra de habilitação semelhante à de um ramo de leitura. A primeira condição é que a *expressão de escrita* seja **compatível** com as definições do lugar, de forma que os **tipos** de seus *tokens* pertençam ao *domínio do lugar* (D_M) e os **endereços**, caso presentes, sejam **válidos** (pertencentes a A). A segunda condição componente da regra é que tenha transcorrido, desde a última operação envolvendo o lugar, um intervalo de tempo igual ou superior à *latência* correspondente (L_W para escrita e L_R para leitura) multiplicada pelo número de *tokens* envolvidos em tal operação.

Um **ramo fornecedor** tem por regra de habilitação que a **cardinalidade** do conjunto de *tokens* resultante da *função de inserção de tokens* seja **menor ou igual** à quantidade de **células livres** (*disponíveis*) no *repositório de entrada* (R_I) do lugar associado, além dos tipos destes *tokens* serem compatíveis com o *domínio de entrada* (D_I) do lugar.

A regra de habilitação de um **ramo coletor** possui duas condições: primeira, deve existir pelo menos um *sample* de *tokens* no *repositório de saída* (R_O) do lugar relacionado; segunda, a quantidade de *tokens* da *expressão de remoção* deve ser igual à cardinalidade do *domínio de saída* (D_O) do lugar.

RPSD tem 3 tipos de ramos relacionados ao **tratamento de exceções**: os ramos de entrada *preemptivos* e *suspensivos*, e os ramos de saída *restauradores*.

Para lugares do tipo I ou III, a regra de habilitação referente a um **ramo preemptivo** é que o respectivo lugar possua ao menos um *token*. Caso a transição relacionada dispare, **todos** os *tokens* do lugar serão **removidos**. Em relação a lugares do tipo I, isto significa que estados locais mantidos por diferentes componentes do sistema podem ser extintos de uma só vez. Para lugares do tipo III, instâncias de computação associadas a um lugar do tipo III poderão ser abortadas a qualquer momento, mesmo antes de sua execução, quando argumentos ainda estão sendo recebidos. Ramos preemptivos associados a lugares do tipo II não apresentam restrições quanto à habilitação da transição relacionada. Caso esta dispare, o conteúdo das células de armazenamento do lugar serão *resetados*, assumindo o valor *default* correspondente ao tipo de *token* definido no domínio lugar (D_M).

A regra de habilitação de um **ramo suspensivo** é que o lugar funcional possua ao menos uma **instância** em execução.

Complementando o ramo suspensivo, tem-se o **ramo restaurador**. Este **não interfere** na habilitação da transição à qual está ligado na RPSD, servindo apenas para **retomar** a execução de **instâncias** do **comportamento** de um lugar funcional, que possam eventualmente ter sido suspensas pelo disparo de uma transição ligada ao lugar por um ramo suspensivo.

Para que uma transição da RPSD torne-se **habilitada**, deve-se satisfazer:

- a) a **guarda da transição** (caso presente) deve ser **verdadeira (TRUE)** para os conjuntos de **tokens** que se queira retirar dos **lugares de entrada** da transição; e
- b) as **regras de habilitação** referentes aos **ramos** ligados à transição devem ser **satisfeitas**.

Para que possa disparar, uma transição deve estar **habilitada** e respeitar suas **regras de disparo**, que constituem uma condição referente ao **tempo**. O mecanismo adotado em RPSD é similar àquele das **redes de Petri Temporizadas de Merlin** [42,45] (seção 8.4): utilização de **intervalos de disparo [LDI, LDS]** associados a cada transição, indicando os tempos **mínimo (LDI)** e **máximo (LDS)** para o **disparo** de uma transição **após** a sua **habilitação**. Este mecanismo é estendido em RPSD, sendo útil nos casos em que deseja-se modelar intervalos de tempo nos quais possa ocorrer **comunicação**. As transições **sem intervalo de disparo** representam **eventos ocasionais** no sistema modelado, os quais **poderão** ou **não** ocorrer. Um caso especial é o **intervalo de disparo** [0,0], o qual implica no **disparo imediato** da transição, tão logo ela se torne habilitada.

São as seguintes as **regras de disparo** de uma transição:

- a) O **limite superior do intervalo de disparo dinâmico** deve ser igual a 0; ou
- b) O **limite inferior do intervalo de disparo dinâmico** precisa ser menor ou igual ao mínimo dos limites superiores dos intervalos de disparo dinâmicos das transições habilitadas.

O **efeito do disparo de uma transição t** , promove a mudança da **classe atual C_i** para uma **nova classe C_{i+1}** . Este efeito é detalhado na seção 11.2.2, pois algumas definições se fazem necessárias.

11.2.1.4. INICIALIZAÇÃO DE LUGARES

Na modelagem de um sistema, em muitos casos, faz-se necessária a **inicialização** de determinados **componentes** do mesmo.

A figura 11.13 apresenta a **definição textual** e a **representação gráfica** de um lugar do tipo I. A definição do **repositório R_S** (figura 11.13.a) corresponde à inicialização do lugar. A representação gráfica do lugar na **classe inicial C_0** corresponde à definição textual do mesmo (observe a disposição dos **tokens** no **repositório** do lugar na figura 11.13.b). O disparo de transições conectadas a **Waiting** possibilitam a alteração do conteúdo inicial de **R_S** (figura 11.13.c). O **repositório R_S** é um subconjunto ordenado do **domínio D_S** .

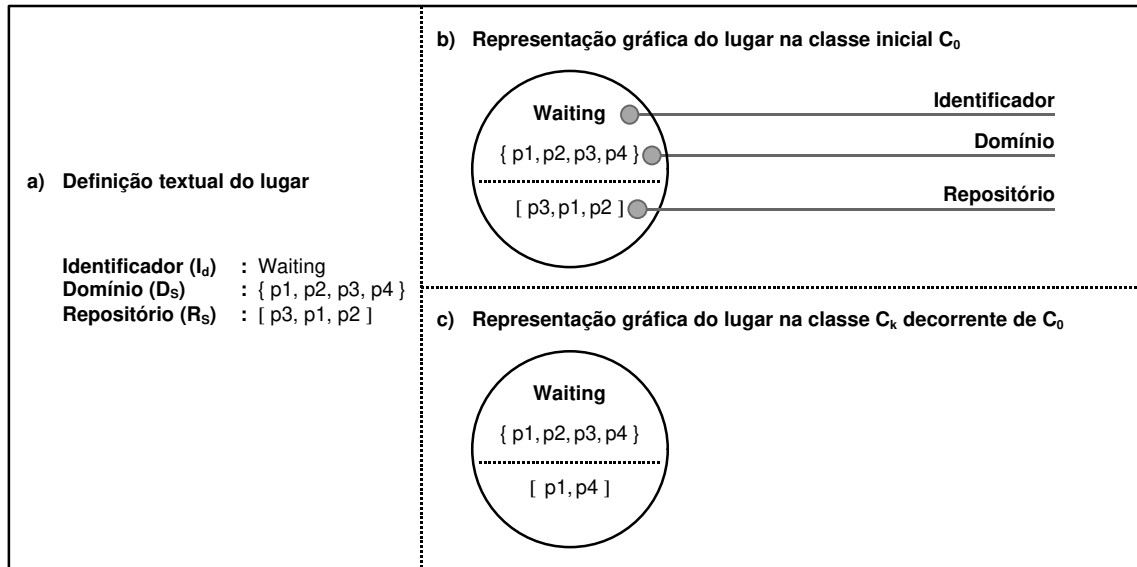


Figura 11.13. Diferentes representações para um lugar tipo I – Estado Local.

Lugares do tipo II também necessitam de mecanismo de inicialização direta. Como tais lugares representam **células de armazenamento**, as quais podem ter um valor **pré-determinado** quando da inicialização de um sistema (por exemplo, memórias ROM), faz-se necessário atribuir valores iniciais a tais células. Conforme visto na seção 11.2.1.1.2, as células de armazenamento são indicadas pelo chamado **vetor conteúdo C** de cardinalidade $\#A$, onde A é o **espaço de endereçamento** do lugar. As entradas do vetor C na definição de um lugar do tipo II correspondem à **marcação inicial** do lugar (na **classe inicial C_0**). As entradas não inicializadas do vetor C correspondem a valores *default*, de acordo com a definição do **domínio D_M** do lugar. Um caso especial é o da atribuição de **nulo** ($\emptyset$) ao vetor C , indicando valores inicialmente **indeterminados (aleatórios)**. A figura 11.14 mostra três inicializações distintas para um mesmo lugar: na figura 11.14.a, **Register** possui valores pré-determinados (observe que valores ausentes correspondem ao valor *default* 0); já nas figuras 11.14.b e 11.14.c, a definição de C como nulo faz com que valores quaisquer sejam aleatoriamente atribuídos às células de armazenamento de **Register**.

A partir da constatação de que, muitas vezes, a **enumeração** dos elementos que compõem os conjuntos descritores de lugares sobrecarrega a representação gráfica de modelos em RPSD, a definição da extensão possibilita o uso de uma classe formada pelos conjuntos utilizados na definição dos diversos lugares de uma rede. Esta classe de conjuntos é denominada **anotações (A_n)**. A classe **anotações** contém a descrição de todos os conjuntos que podem ser utilizados na definição do **espaço de endereçamento (A)** de lugares do tipo II, e na definição dos conjuntos **domínio** (conjuntos D_s , D_M , D_I e D_O , detalhados adiante).

Cada conjunto em **anotações** é definido através da enumeração finita de seus elementos ou através da associação com um tipo pré-definido, pertencente a alguma biblioteca que possa ser utilizada por ferramentas que, futuramente, trabalhem com representações em RPSD. Pode-se, por exemplo, definir o conjunto de números reais **Float** através da associação

$$\text{Float} := \text{IEEE\_754\_32},$$

onde **IEEE_754_32** corresponde ao conjunto de números de ponto flutuante da norma IEEE 754 com formato simples (32 bits).

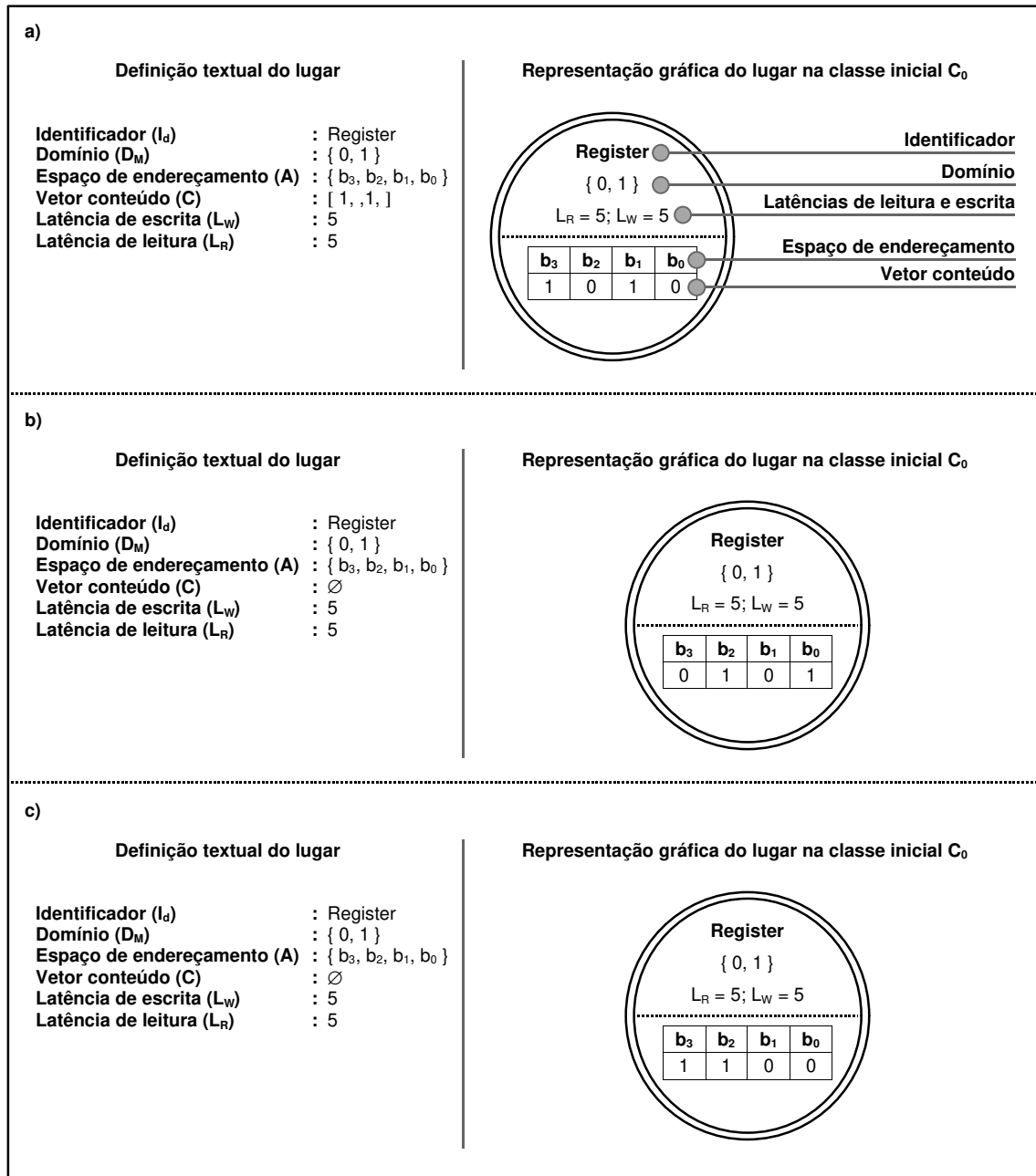


Figura 11.14. Diferentes inicializações para um lugar tipo II – Armazenamento.

Além da descrição de cada conjunto (por enumeração ou por associação), **anotações** contém também o valor *default* de cada conjunto. Quando omitido, o valor *default* de um conjunto corresponde ao primeiro elemento enumerado na definição do conjunto. Valores *default* são utilizados para **inicializar** (conforme exposto anteriormente) ou promover o **reset** de lugares de armazenamento (seção 11.2.1.1.2).

A representação gráfica de RPSD admite tanto a **descrição** (enumeração de todos os elementos) de um dado conjunto em cada lugar em que aparece (figuras 11.13 e 11.14), quanto a **referência** a tal conjunto (figura 11.15). Entretanto, ao descrever uma única vez os conjuntos na classe **anotações**, facilita-se alterações na definição dos lugares, uma vez que o gráfico da rede permanece inalterado. Por exemplo, suponha que o conjunto

$$Processes = \{p1, p2, p3\}$$

seja utilizado como o *domínio* D_S de 15 lugares do tipo I. Caso deseje-se acrescentar novos elementos a este conjunto, dois procedimentos são possíveis:

- Alterar as 15 ocorrências de **Processes** (quando o conjunto encontra-se descrito em cada um dos lugares que o utiliza); ou
- Alterar a definição de **Processes** em **anotações** (quando o conjunto é apenas referenciado em cada um dos lugares que o utiliza).

Independentemente da representação adotada, a classe **anotações** sempre existirá, uma vez que faz parte da tupla definidora de qualquer RPSD. Ressalta-se, também, que os conjuntos descritos nesta classe podem ser utilizados na representação textual dos lugares.

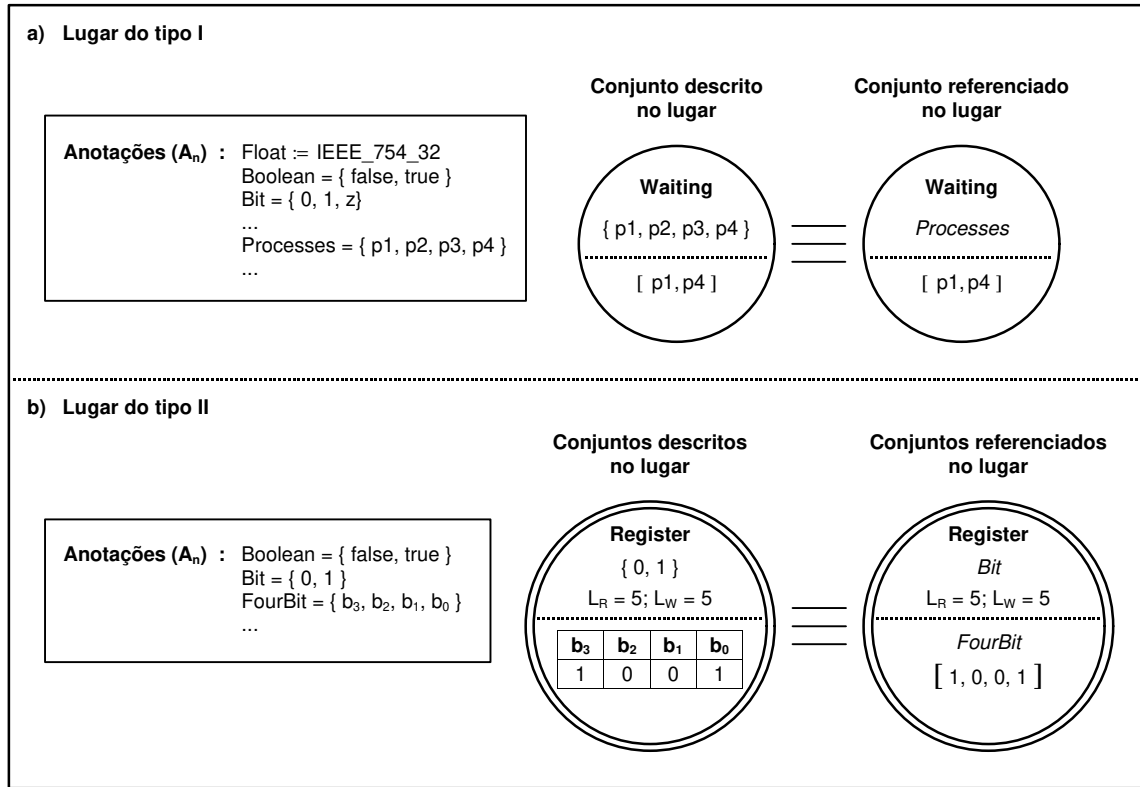


Figura 11.15. Referência a conjuntos descritos na classe anotações.

Devido à sua semântica, lugares do tipo III não necessitam de um mecanismo de inicialização direta.

11.2.2. DESCRIÇÃO FORMAL

Rede de Petri para Sistemas Digitais é representada pela tupla

$$RPSD = (A_n, Mem, P, T, I_U, I_M, I_R, I_C, I_S, I_P, O_U, O_W, O_S, O_R, C_0)$$

onde:

- A_n é a classe **anotações** da rede, a qual contém a descrição de **todos** os **conjuntos** que podem ser utilizados na definição dos lugares da rede (elementos do conjunto P). Cada conjunto em **anotações** é definido através da enumeração finita de seus elementos ou através da associação com um tipo pré-definido, pertencente a alguma biblioteca que possa ser agregada à RPSD. Pode-se designar o valor *default* de cada conjunto. Quando omitido, o valor *default* de um conjunto corresponde ao primeiro elemento enumerado na definição do conjunto.
- **Mem** é o conjunto finito de **variáveis da rede** e seus respectivos **tipos**, denominado **memória da rede**. Este conjunto é formado por pares ordenados $\langle \text{var}_i, \text{type}_i \rangle$, onde var_i é o **identificador** único de uma variável e, type_i , o seu **tipo**. Tipos **finitos** são descritos na classe A_n , enquanto tipos **não finitos** são apenas nominados.
- $P = P_I \cup P_{II} \cup P_{III}$ é o conjunto finito não-nulo de **lugares** da rede, tal que:
 - P_I = conjunto de lugares do **tipo I**: $\forall p \in P_I, p = (I_d, D_S, R_S)$;
 - P_{II} = conjunto de lugares do **tipo II**: $\forall p \in P_{II}, p = (I_d, D_M, A, C, L_W, L_R)$; e
 - P_{III} = conjunto de lugares do **tipo III**: $\forall p \in P_{III}, p = (I_d, D_I, \text{max}_S, R_I, D_O, R_O, B)$,
 onde $I_d, D_S, R_S, D_M, A, C, L_W, L_R, D_I, \text{max}_S, R_I, D_O, R_O$ e B seguem as definições dadas na seção 11.2.1.1.
- T é o conjunto finito não-nulo de **transições** da rede: $\forall t \in T, t = (I_d, G, T_O, T_S)$, tal que:
 - I_d é uma seqüência de caracteres finita e única que identifica a transição;
 - G é uma **expressão booleana** denominada **guarda da transição**;
 - T_O é o conjunto de **operações da transição**; e
 - T_S é o **intervalo de disparo estático da transição**.
- I_U é o mapeamento dos **ramos de entrada comuns** da rede;
- I_M é o mapeamento dos **ramos de entrada conjugados** da rede;
- I_R é o mapeamento dos **ramos de entrada de leitura** da rede;
- I_C é o mapeamento dos **ramos de entrada coletores** da rede;
- I_S é o mapeamento dos **ramos de entrada suspensivos** da rede;
- I_P é o mapeamento dos **ramos de entrada preemptivos** da rede;
- O_U é o mapeamento dos **ramos de saída comuns** da rede;
- O_W é o mapeamento dos **ramos de saída de escrita** da rede;
- O_S é o mapeamento dos **ramos de saída fornecedores** da rede;
- O_R é o mapeamento dos **ramos de saída restauradores** da rede;
- C_0 é a **classe inicial** da RPSD, denotada por $C_0 = (M_0, T_0, V_0)$, onde:
 - M_0 é a **marcação inicial**;
 - T_0 é o **domínio de tempo inicial**;
 - V_0 é o **conjunto de valores iniciais para as variáveis da rede**.

A seguir, algumas informações sucintas, sobre componentes, e complementares, sobre o funcionamento, de RPSD.

a) Lugar

Elemento da rede com significado determinado por seu tipo, o qual pode ser:

- **Tipo I:** representando um **estado local** (modo) de um ou mais componentes do sistema modelado. Cada *token* presente num lugar deste tipo denota um componente do sistema;
- **Tipo II:** representando um **componente de armazenamento** do sistema modelado. À este componente associa-se um conjunto de **células de armazenamento**, cada qual acessada através de um **endereço**. Um *token* corresponde ao **conteúdo** de uma dessas células de armazenamento; ou
- **Tipo III:** representando um **componente funcional** ou um **conjunto parametrizado** de componentes funcionais do sistema modelado. O **comportamento** (funcionalidade) de um lugar deste tipo pode ser descrito por uma **rotina** (seqüência de sentenças de programação) ou por uma subrede do tipo **RPSD-T**, a qual transforma um conjunto de entradas em um conjunto de saídas. **Hierarquia comportamental** é permitida, seja através da **decomposição seqüencial** de um comportamento em sentenças de programação (rotina), ou seja através da **decomposição concorrente** de um comportamento em lugares pertencentes a uma RPSD-T, subrede cujos lugares funcionais também podem ser decompostos (em rotinas ou em outras RPSD-T). Cada *token* num lugar deste tipo corresponde ao valor de um **item de dado** pertencente a um **sample de dados**, o qual pode estar presente na **entrada**, na **saída** ou **em processamento** por alguma **instância** do comportamento associado ao lugar. *Samples* na entrada do lugar correspondem à parâmetros de entrada do comportamento, enquanto *samples* na saída do lugar correspondem aos resultados gerados pela ativação do comportamento.

Lugares do tipo I são graficamente representados por uma elipse; lugares do tipo II, por duas elipses concêntricas; lugares do tipo III, por três elipses concêntricas. Todo lugar deve possuir um identificador único.

b) Transição

Elemento da rede correspondente a um **evento** no sistema modelado. De acordo com os lugares envolvidos, o evento pode promover:

- A **inserção** ou a **remoção** de **tokens representantes de componentes do sistema**, em lugares do **tipo I**;
- A **leitura** ou a **atualização** (escrita) de **células de armazenamento**, em lugares do **tipo II**;
- A **inserção** de **argumentos** para os **parâmetros de entrada** necessários à ativação de **comportamentos** associados a lugares do **tipo III**, ou a **retirada** de **resultados** do processamento desses **comportamentos**.

Transições podem ser utilizadas na modelagem **exceções**, nas quais os eventos podem ter como conseqüências:

- A **remoção** de **todos** os **tokens representantes de componentes do sistema**, em lugares do **tipo I**;
- A **limpeza** (*reset*) de **todas** as **células de armazenamento**, em lugares do **tipo II**;
- A **suspensão temporária** ou a **retomada** de **comportamentos** associados a lugares do **tipo III**;
- A **suspensão definitiva** (*preempção*) de **comportamentos** associados a lugares do **tipo III**, com a **remoção** de **todos** os **dados** (*tokens*) do lugar, inclusive das subredes existentes.

Quatro componentes definem uma transição, quais sejam:

- b.1) **Identificador (I_d)**: Sequência de caracteres finita e única que denomina a transição;
- b.2) **Guarda da transição (G)**: Expressão booleana para **habilitação** de uma transição. Pode envolver **subconjuntos de tokens** pertencentes ao **domínio** de cada lugar relacionado à transição, **variáveis da rede (memória da rede)** e **funções pré-definidas**. Sua ausência corresponde ao valor lógico **TRUE**. Representada entre colchetes;
- b.3) **Operações da transição (T_o)**: Conjunto formado por **operações de atribuição** (*variável := expressão*) e **comandos condicionais** (*If-Then-Else*), capazes de alterar a **memória da rede**, a qual é representada por um **conjunto de variáveis**. Anotado dentro da transição, é executado concomitantemente com o disparo desta;
- b.4) **Intervalo de disparo estático (T_s)**: Intervalo de limites inteiros não-negativos, indicando os tempos de atraso **mínimo** e **máximo** para o disparo de uma transição após a sua habilitação. $T_s = [LDI, LDS]$, onde **LDI** é o **limite de disparo inferior** e **LDS** é o **limite de disparo superior**. Duas observações importantes:
 - O intervalo $[0,0]$ denota **disparo imediato**, significando que, uma vez habilitada, a transição irá disparar **imediatamente**;
 - A **ausência de intervalo** denota que a transição representa um **evento ocasional** do sistema, isto é, um evento que poderá ocorrer ou não, como, por exemplo, a ativação de um sinal de *reset*.

Transições são representadas por retângulos.

c) Classe

Nome dado a um **estado** da RPSD. A classe é descrita por uma tripla $C_i = (M_i, D_i, V_i)$, $i \in \mathbb{IN}$, onde M_i é a **marcação** da rede, D_i o seu **domínio de tempo** e V_i a **valoração das variáveis** da rede.

- c.1) **Marcação**: É um conjunto de **pares ordenados**, onde o primeiro elemento do par é o **identificador** (nome) de um lugar e o segundo elemento é o **conjunto de tokens** presentes no respectivo lugar, em determinado momento. Lugares vazios não são representados. O conjunto de *tokens* tem representações diferentes, de acordo com o tipo do lugar, podendo assumir uma das seguintes configurações:
 - c.1.1) Para lugares do tipo I: representado por uma **lista de tokens, ordenada** de acordo com o **tempo de ocupação** do lugar pelos *tokens* (dos mais antigos para os mais recentes);
 - c.1.2) Para lugares do tipo II: representado por uma **lista de pares ordenados** $\langle \text{endereço}, \text{valor} \rangle$, na qual endereços com valores *default* não são apresentados;
 - c.1.3) Para lugares do tipo III: dividido em **três subconjuntos**, quais sejam:
 - c.1.3.1) **Repositório de entrada**: representado por uma matriz $R_I(p)$ de $\max_S(p)$ linhas e $\#D_I(p)$ colunas (uma coluna para cada **tipo** de *token* necessário ao acionamento do **comportamento** associado ao lugar p);
 - c.1.3.2) **Repositório de saída**: representado por uma $R_O(p)$, com $\max_S(p)$ linhas e $\#D_O(p)$ colunas (uma coluna para cada tipo de *token* resultante do processamento do **comportamento** associado ao lugar p);
 - c.1.3.3) **Comportamento**: duas representações, de acordo com a forma como o comportamento associado ao lugar é descrito:

- **Rotina:** descrita por uma **lista ordenada de conjuntos de valoração de instâncias**. Cada conjunto destes corresponde a uma lista de pares ordenados $\langle \text{identificador}, \text{valor} \rangle$, onde **identificador** é o nome de uma entrada (**input**), saída (**output**) ou variável utilizada na rotina (**routine**). A ordenação entre estes conjuntos se dá por um critério cronológico, qual seja, o da criação (ativação) da instância correspondente. Instâncias que concluíram seu processamento (inexistentes, portanto) não são representadas;
- **RPSD-T:** descrita pela marcação da subrede associada a cada instância em execução (independente de estar temporariamente suspensa ou não). Como uma instância da RPSD-T é uma RPSD, sua marcação segue os critérios aqui estabelecidos.

c.2) **Domínio de Tempo:** É um **conjunto de triplas**, onde o primeiro elemento de cada tripla é o **identificador** de uma transição habilitada e os demais elementos representam os **limites inferior e superior** do intervalo de disparo dinâmico da respectiva transição numa dada marcação da rede. Chama-se **intervalo de disparo estático** àquele anotado junto com a transição, entre colchetes. Este intervalo é adotado tão logo a transição torne-se habilitada. A partir deste instante, o intervalo passa a ser chamado de **intervalo de disparo dinâmico**. O intervalo estático corresponde ao valor *default* do intervalo de disparo, sendo denotado por $[\text{LDI}, \text{LDS}]$, onde **LDI** é o **limite de disparo inferior** e **LDS** é o **limite de disparo superior**. O intervalo dinâmico é decorrente do decréscimo de tempo sobre o estático no período de tempo em que a transição continua habilitada sem disparar;

c.3) **Valoração das variáveis da rede:** É um conjunto de **pares ordenados** $\langle \text{var}_i, \text{valor}_i \rangle$, onde **var_i** é o **identificador** único de uma variável da rede e, **valor_i**, o seu **valor**. As variáveis da rede são definidas em **Mem** e seus valores podem ser acessados pelos rótulos dos ramos ou pelas guardas e operações associadas às transições. Somente as operações associadas às transições podem alterar o valor de uma variável da rede.

d) Funcionamento da Rede

As **regras de habilitação e disparo** de uma transição são aquelas mencionadas na seção 11.2.1.3, não necessitando de maiores detalhes. Já o **efeito do disparo** de uma transição deve ser fornecido.

O **efeito do disparo de uma transição t**, promove a mudança da **classe atual C_i** para uma **nova classe C_{i+1}**, o que ocorre em seis etapas:

- Desativação das instâncias de comportamento e remoção de todos os tokens presentes nos repositórios de entrada (R_I) e de saída (R_O) dos lugares de tipo III, ligados a t por ramos preemptivos;*
- Suspensão dos instâncias de comportamento dos lugares de tipo III, ligados a t por ramos suspensivos;*
- Reativação das instâncias de comportamento dos lugares de tipo III, ligados a t por ramos restauradores;*
- Leitura de tokens dos lugares de entrada do tipo II, remoção de tokens dos lugares de entrada dos tipos I e III, escrita de tokens nos lugares de saída do tipo II e adição de tokens aos lugares de saída dos tipos I e III, de acordo com os ramos que relacionam tais lugares a t;*

v) Realização das **operações da transição**, as quais podem afetar o conjunto de variáveis da rede;

vi) Atualização do **tempo** da rede, denotado pelos intervalos de disparo das transições presentes no **Domínio de Tempo**. Analisando-se o estado futuro da rede, são válidas as seguintes regras:

vi.1) Se **t** permanecer habilitada em C_{i+1} , então seu **intervalo de disparo dinâmico** $T_D(t)$ assumirá os valores do **intervalo estático** $T_S(t)$;

vi.2) As transições que se tornaram habilitadas na classe alcançada C_{i+1} têm seus intervalos de disparo atualizados para seu valor *default* (**estático**);

vi.3) Para cada transição **s** habilitada em C_i que permanecer habilitada em C_{i+1} , o seu intervalo de disparo será calculado da seguinte forma:

vi.3.1) Caso $i > 0$ e as transições **t** e **s** estavam habilitadas em C_{i-1} :

$$LDI(s)_{i+1} = \max \{0; (LDI(s)_i - \text{menor } LDS_i); (LDI(s)_{i-1} - LDS(t)_{i-1})\}$$

$$LDS(s)_{i+1} = \min \{(LDS(s)_i - LDI(t)_i); (LDS(s)_{i-1} - LDI(t)_{i-1})\}$$

onde:

$LDI(s)_i$ é o **limite de disparo inferior** de **s** em C_i ;

$LDS(s)_i$ é o **limite de disparo superior** de **s** em C_i ;

LDS_i é o **conjunto** dos limites de disparo superiores das transições habilitadas em C_i .

vi.3.2) Caso contrário:

$$LDI(s)_{i+1} = \max \{0; (LDI(s)_i - \text{menor } LDS_i)\}$$

$$LDS(s)_{i+1} = LDS(s)_i - LDI(t)_i.$$

O **Grafo de Classes** de uma RPSD é um grafo que mostra todas as classes acessíveis na rede, a partir de uma dada classe inicial. Nas RPSD, cada **nó** desse grafo representa uma **classe acessível** e cada **arco** direcionado é rotulado pela **amarração** de transição cujo disparo faz a rede passar da classe ligada na origem do arco à classe ligada no destino do arco. A amarração de transição é designada por um par ordenado, composto pelo nome da transição e por um conjunto de *tokens* que foi removido/lido de cada lugar de entrada quando do disparo da transição. A seção 13.2 mostra um algoritmo para a geração do grafo de classes de uma RPSD.

11.3. RPSD E CARACTERÍSTICAS DE MODELAGEM DE SISTEMAS DIGITAIS

Nesta seção é mostrado como RPSD atende às características necessárias para uma boa metodologia de modelagem de sistemas digitais, de acordo com Gajski *et al.* [3], descritas na seção 10.1. Há modelos conceituais de especificação (capítulo 2) que satisfazem uma ou mais destas características, entretanto, no que pudemos constatar, não há um modelo que satisfaça a todas estas características conjuntamente. Além disso, a principal vantagem de um modelo baseado em redes de Petri está na aplicação de técnicas de validação de sistemas (as técnicas para análise de RPSD encontram-se no capítulo 13). Cada item seguinte traz, ainda, uma comparação sucinta entre a expressividade de RPSD e as linguagens de especificação abordadas no capítulo 4.

a) Concorrência

Modelos RPSD podem representar tanto concorrência **orientada a dados** (figura 11.16) quanto concorrência **orientada a controle** (figura 11.17).

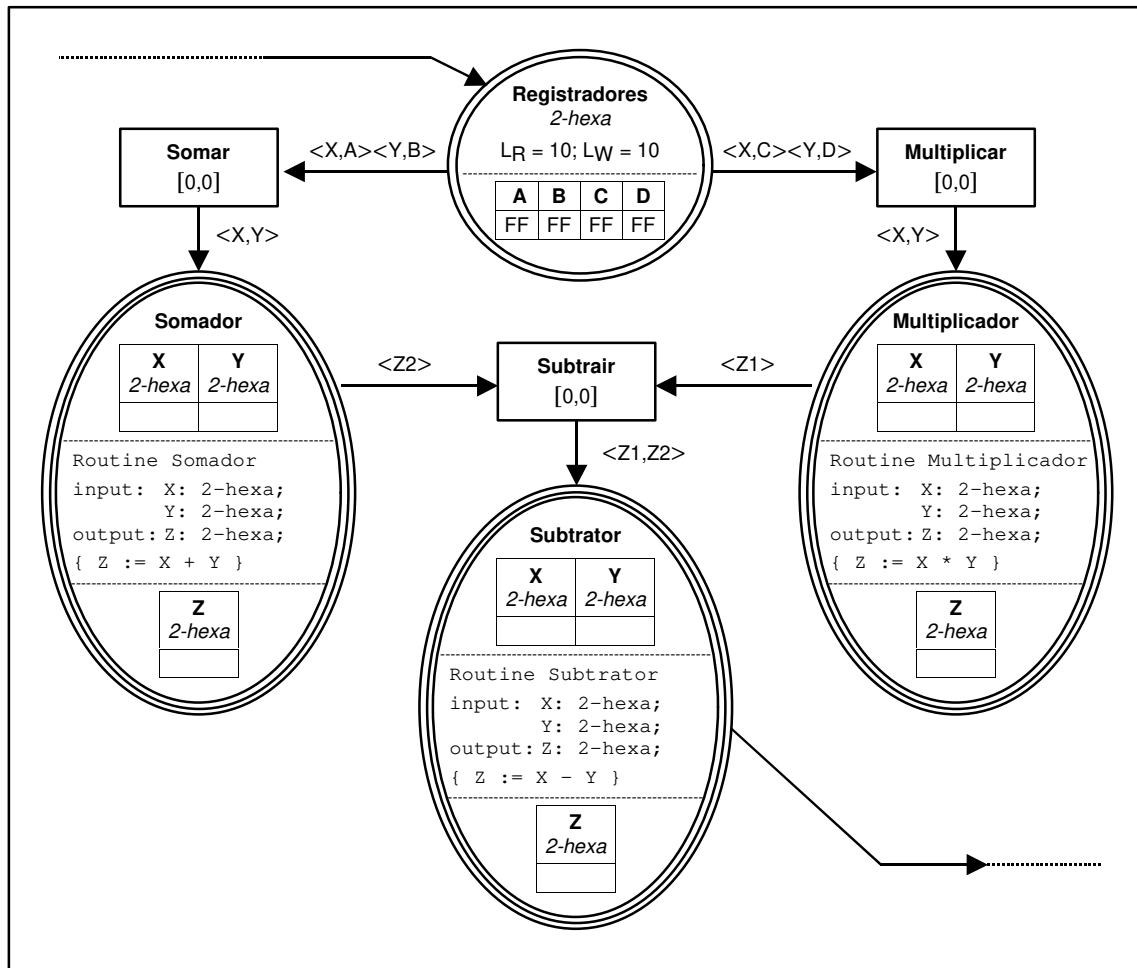


Figura 11.16. Concorrência orientada a dados.

Na **concorrência orientada a dados**, não há uma ordem de execução explícita das tarefas envolvidas. A ordem de execução é determinada pela **dependência de dados** entre

as tarefas. Na rede da figura 11.16, o lugar **Registradores** possui 4 registradores: **A**, **B**, **C** e **D**; os registradores **A** e **B** são utilizados numa operação de soma (acionada pelo disparo da transição **Somar** e executada pelo lugar funcional **Somador**), enquanto os registradores **C** e **D** são empregados numa operação de multiplicação (acionada por **Multiplicar** e executada por **Multiplicador**). Estas operações podem ser acionadas em qualquer ordem (observe que as transições são conflitantes – o disparo de uma posterga o disparo da outra, devido à latência de leitura do lugar de armazenamento), e seus resultados são disponibilizados nos *repositórios de saída* dos lugares **Somador** e **Multiplicador**, após a execução das rotinas envolvidas. Entretanto, para que a operação acionada pela transição **Subtrair** possa ocorrer, é necessária a presença dos dados em ambos *repositórios de saída* mencionados, isto é, é necessária a presença de um *token* no repositório de saída do lugar **Somador**, assim como a presença de um *token* no repositório de saída de **Multiplicador**. Desta forma, embora as operações de soma e de multiplicação possam ocorrer ao mesmo tempo, a operação de subtração dependerá da disponibilidade dos dados resultantes da execução das duas outras operações.

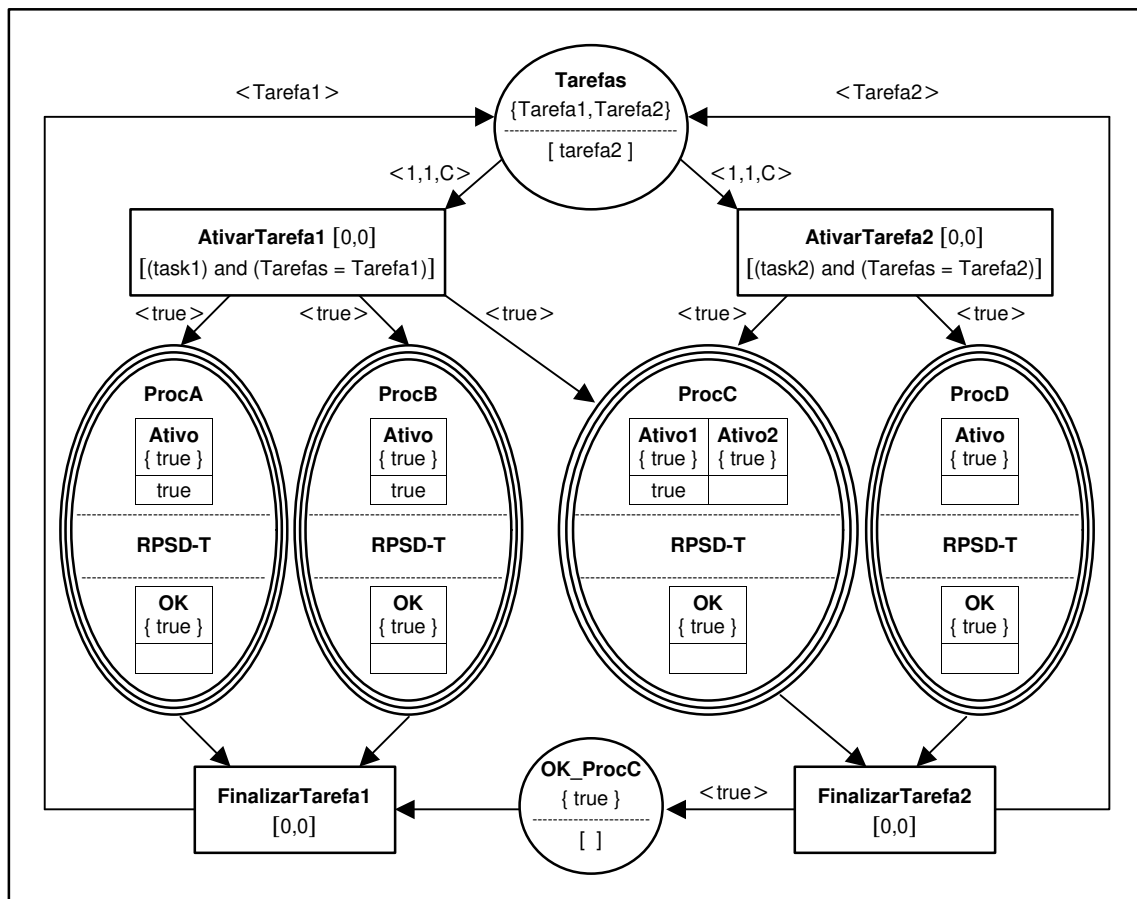


Figura 11.17. Concorrência orientada a controle.

Na **concorrência orientada a controle**, há **segmentos de controle** determinando a **ordem de execução** das tarefas envolvidas. Cada segmento de controle agrupa um conjunto de tarefas executadas sequencialmente. A concorrência caracteriza-se pela execução simultânea destes agrupamentos sequenciais (segmentos de controle). Na figura 11.17, as tarefas **tarefa1** e **tarefa2** podem executar concorrentemente (execução simultânea de seus processos constituintes, **ProcA**, **ProcB**, **ProcC** e **ProcD**). No caso

específico do estado em que encontra-se a rede da figura, temos que os processos **ProcA** e **ProcB** acabam de ser ativados (presença do *token true* em ambos *repositórios de entrada*); já os processos **ProcC** e **ProcD** não estão ativos, uma vez que não possuem ainda o *token true* em seus *repositórios de entrada*, necessário à ativação da instância do comportamento representante de cada processo (as RPSD-T descritoras dos quatro processos não encontram-se representadas na figura). Mesmo que os processos **ProcA** e **ProcB** sejam encerrados antes de **ProcC** e **ProcD**, a transição **FinalizarTarefa1** não poderá disparar, o que ocorrerá somente quando o processamento de **ProcC** estiver encerrado (na figura 11.17 ele não começou, devido à falta do *token true* no repositório de entrada de **ProcC**, o que depende do disparo de **AtivarTarefa2**). Desta forma, a tarefa **Tarefa1** só poderá ser encerrada após o término do processo **ProcC** da tarefa **Tarefa2**, explicitando uma dependência de controle de execução, embora os processos destas tarefas possam executar em paralelo. Para melhor compreensão do exemplo na figura, duas considerações se fazem necessárias, quanto às guardas das transições **AtivarTarefa1** e **AtivarTarefa2**:

- As variáveis booleanas **task1** e **task2** têm seus valores determinados por operações associadas a outras transições, não mostradas na figura; e
- O identificador de lugar **Tarefas** é utilizado para representar o resultado da expressão **<1,1,C>**, associada aos **ramos de entrada** de ambas transições com o lugar de mesmo nome. Assim, **AtivarTarefa1** estará habilitada sempre que **task1** for **TRUE** e **<1,1,C>** resultar em **Tarefa1** (o que ocorre quando o *token Tarefa1* ocupa a primeira posição do repositório do lugar **Tarefas**, isto é, quando for o *token* há mais tempo no lugar). A transição **AtivarTarefa2** tem comportamento análogo.

Nas linguagens de especificação abordadas no capítulo 4, foi visto que algumas delas (**SDL**, **Statecharts**, **CSP**) suportam **concorrência em nível de tarefas**, enquanto outras (**VHDL**, **Verilog**) suportam **concorrência em nível de tarefas** e em **nível de statements** (sentenças). **Ambos tipos de concorrência são suportados por RPSD**. A concorrência em nível de tarefas é obtida ao permitir que vários processos (comportamentos) executem ao mesmo tempo, seja em lugares funcionais diferentes, ou seja dentro do mesmo lugar funcional (várias instâncias de um mesmo comportamento). Apesar de não possuir os mesmos construtores de linguagens como VHDL, RPSD permite representações equivalentes para cláusulas **wait** e **after**, através da modelagem de mecanismos de escalonamento. Comparada com **Silage**, RPSD também permite concorrência orientada a dados, conforme visto no presente item.

b) *Transição de estados*

Tal como qualquer outra extensão de rede de Petri, RPSD é construída sobre o conceito de **estados** e de **transições** entre estados. Seus **lugares** representam **componentes de armazenamento**, **componentes funcionais (comportamentos)** ou **estados locais (modos)** de componentes do sistema modelado. As mudanças sofridas por componentes, modos e comportamentos ocorrem através do disparo de transições. O disparo de uma transição representa a ocorrência de um **evento** no sistema modelado sob certas condições, inclusive temporais, conforme visto anteriormente.

Das linguagens vistas no capítulo 4, somente **Statecharts** possui esta característica. Desta forma, neste quesito, RPSD supera todas as demais linguagens de especificação abordadas.

c) *Hierarquia*

Algumas extensões de rede de Petri incluem **estruturas hierárquicas**, ao permitir que um ou **lugar** ou uma **transição** represente uma subrede de nível hierárquico inferior. RPSD utiliza estruturas hierárquicas associadas a **lugares funcionais**, devido às mesmas razões para a escolha de lugares para a representação de hierarquia em PNES (seção 10.4), quais sejam:

- i) Os lugares funcionais representam comportamentos, os quais podem ser executados por um ou mais componentes funcionais do sistema modelado. Assim, se desejar-se **decompor** o comportamento complexo de um componente funcional em um conjunto de **subcomportamentos sequenciais ou paralelos**, faz-se natural a escolha por mecanismos associados a tais tipos de lugares; e
- ii) Na Teoria de redes de Petri, transições são atômicas, isto é, não devem ser interrompidas após terem iniciado. Assim, na ocorrência de uma **exceção**, se estruturas hierárquicas fossem associadas a uma transição, todas as operações descritas pela transição deveriam ser encerradas antes que o fluxo de controle da rede fosse transferido. Entretanto, uma exceção exige a imediata interrupção de um comportamento.

RPSD permite tanto **hierarquia comportamental** como **estrutural**. A **hierarquia comportamental** é obtida ao permitir que o comportamento de um lugar funcional possa ser **decomposto paralela** (através da representação por uma subrede, denominada RPSD-T) ou **sequencialmente** (através de uma rotina de programação). A descrição por meio de uma RPSD-T (subrede no mesmo estilo de RPSD) permite a obtenção de outra forma hierárquica, a **hierarquia estrutural**, uma vez que o comportamento **B**, de qualquer lugar funcional de uma RPSD-T, também pode ser descrito por outra RPSD-T e, assim, sucessivamente.

Em um dado momento, como o comportamento descrito por um lugar funcional pode estar sendo executado concorrentemente por uma ou mais unidades funcionais, há uma instância hierárquica do comportamento para cada *sample* de dados completo presente no lugar. A figura 11.18 mostra as três instâncias relativas à subrede do lugar **Comparador** da figura 11.9, na 4ª marcação mostrada na figura 11.11. Observe que há duas instâncias ativas e uma não ativa. A instância 1 já terminou o seu processamento, aguardando a retirada do resultado. A instância 2 acaba de ser ativada (a primeira sentença de programação ainda não foi executada, razão pela qual a variável **z** está com seu valor *default*). Caso novo par de valores seja fornecido ao repositório de entrada de **Comparador**, a instância 3 será ativada.

Ao permitir as duas formas de hierarquia, RPSD se iguala a linguagens como **VHDL** e **Verilog**, superando linguagens como **CSP**, **Statecharts** e **Silage**, nas quais o suporte à hierarquia é limitado.

d) *Estruturas de programação*

Poucas extensões de rede de Petri possuem esta característica. Algumas extensões apresentam estruturas agregadas a lugares enquanto outras apresentam-nas associadas a transições.

Em RPSD, **estruturas de programação** são associadas tanto a **lugares** (do tipo III) quanto a **transições**. As estruturas presentes nas rotinas descritoras do comportamento de lugares funcionais, são utilizadas na **manipulação de dados inerentes ao sistema modelado**; as estruturas de programação relacionadas às transições permitem a manipulação das **variáveis** da rede, gerando uma grande capacidade representacional e tornando o **fluxo de controle de execução** de modelos RPSD mais fácil. Os dados que alimentam as estruturas relacionadas a um lugar funcional são **tokens** provenientes do disparo de transições conectadas ao lugar por ramos de saída. As estruturas de programação relacionadas às transições permitem a manipulação de **variáveis da rede**, as quais são definidas na **memória da rede** – componente **Mem** de uma RPSD, conforme visto na seção 11.2.2.

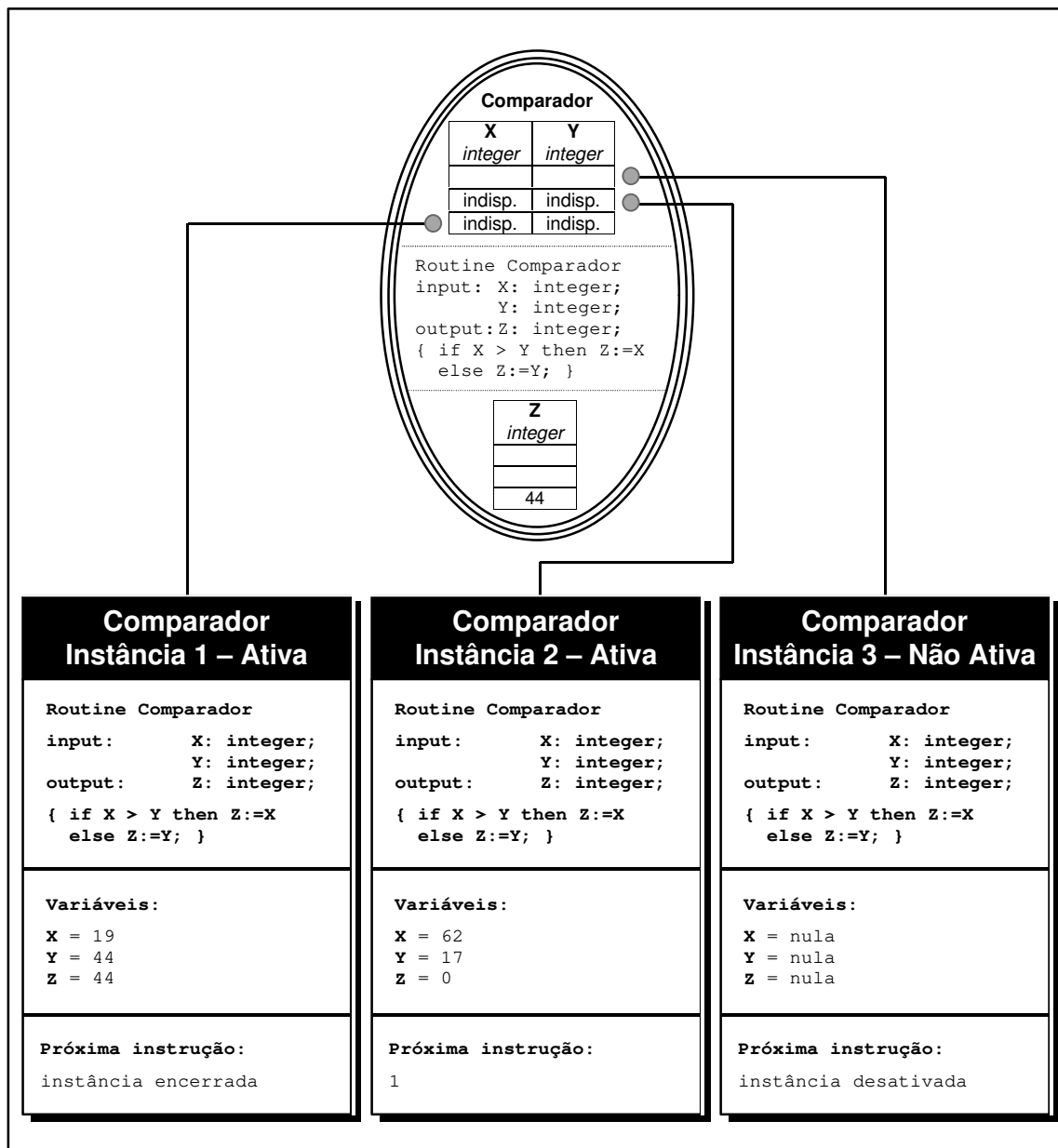


Figura 11.18. Instâncias hierárquicas do lugar Comparador da figura 11.9 (4ª marcação da figura 11.11).

e) *Término comportamental*

Término comportamental refere-se à habilidade de um **comportamento indicar** a sua **finalização** (todas as computações internas do comportamento executadas), assim como à habilidade de outras partes do sistema **detectarem** este fato.

Comportamentos em RPSD são representados por **lugares funcionais**, cada qual descrito ou por uma **rotina de programação** ou por uma **RPSD-Transformacional (RPSD-T)**. Quando o comportamento é descrito por uma **rotina**, o término de qualquer uma de suas instâncias (cópias) ocorre **após a execução do último statement** (sentença de programação) da mesma. No caso do comportamento ser descrito por uma **RPSD-T**, o término de uma instância dessa subrede ocorrerá quando da presença de *tokens* em **todos** os seus **lugares somente de saída** (veja a seção 11.2.1.1.3 para maiores detalhes). Independentemente da forma de descrição (rotina ou RPSD-T), o **término** de qualquer **instância** de um comportamento é indiretamente **sinalizado** para os demais lugares da rede, através da **disponibilização** dos **resultados** gerados no **repositório de saída** do lugar correspondente. Esta disponibilização de resultados tem influência na habilitação das transições relacionadas ao lugar.

A maioria das linguagens de especificação comentadas no capítulo 4 apresenta esta característica. **Statecharts** pode especificar comportamento **não-determinístico**, o que também pode ser modelado em RPSD, através de **operações de retirada de tokens**, envolvendo **lugares do tipo I** (estado local), nas quais um **conjunto aleatório** de *tokens* pode ser retirado do lugar (quando assim especificado).

f) *Tratamento de exceções*

Em um sistema modelado, a ocorrência de um dado **evento** pode exigir que um **modo de computação** (comportamento ou estado local) seja imediatamente **interrompido** ou **abortado**, transferindo o **fluxo de controle de execução** da rede para um outro modo de computação. Um evento desse tipo é chamado de **exceção**.

Basicamente, há dois tipos de exceções: **preempção** (quando o **modo atual é abortado**) e **interrupção** (quando o **modo atual é temporariamente suspenso**, podendo ser **retomado** após decorrido um certo tempo, mediante dadas condições).

Conforme visto nas seções 10.3 e 10.4, as únicas extensões de rede de Petri que tratam de exceções são a PCN e a PNES. Em PNES, um **supermodo** é definido como um lugar relacionado a uma subrede representativa de operações que possam ser abortadas pelo disparo de uma transição especial, denominada **exceção**. Um conceito similar é encontrado em PCN.

Para modelar **preempção**, RPSD faz uso do chamado **ramo de entrada preemptivo**. Este tipo de ramo permite que uma transição **remova todos** os *tokens* de um lugar do tipo I ou III, independentemente de quaisquer condições. Serve também para **resetar** as células de armazenamento de lugares do tipo II. A figura 11.19 apresenta um trecho de RPSD com ramos de entrada preemptivos.

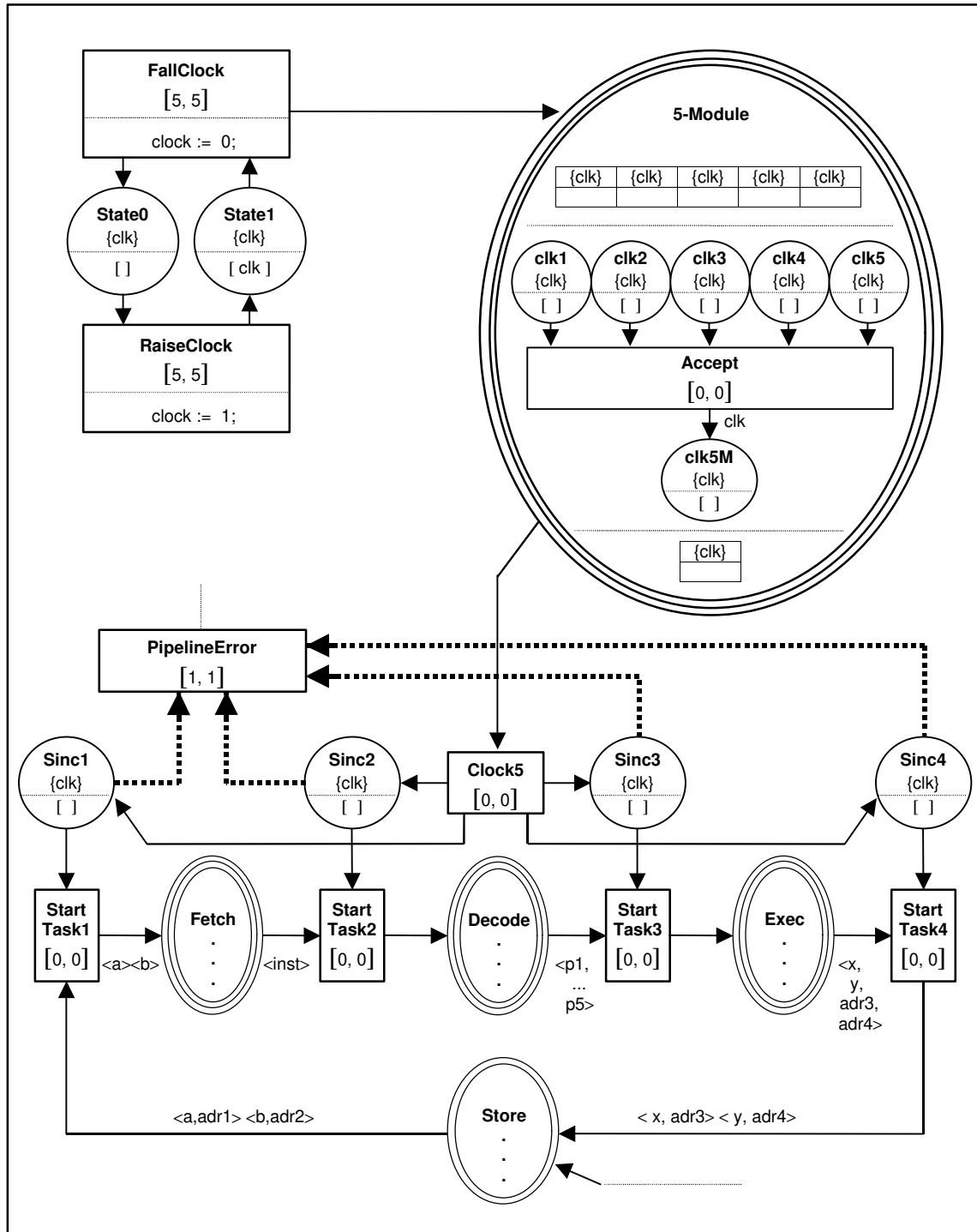


Figura 11.19. Modelo RPSD de um *pipeline* de 4 estágios, com transição preemptiva.

A rede da figura 11.19 modela o funcionamento de um *pipeline* de instrução com 4 estágios (descritos pelos lugares **Fetch**, **Decode**, **Exec** e **Store**). Parte da rede representa a alternância de um sinal de **clock**; outra parte descreve um divisor de **clock** módulo 5 (lugar **5-Module**). Caso ocorra um **erro de processamento**, em algum dos lugares funcionais representantes dos estágios do *pipeline*, a transição correspondente (**StartTask1**, **StartTask2**, **StartTask3** ou **StartTask4**) ficará **desabilitada** logo após o disparo de

Clock5. Isto fará com que um ou mais dos lugares relacionados à sincronização dos estágios (**Sinc1**, **Sinc2**, **Sinc3** ou **Sinc4**) permaneça com um *token* presente, **habilitando**, desta forma, a transição **PipelineError** (a presença de ao menos um *token* num dos lugares de entrada conectados a uma transição por ramos preemptivos, satisfaz a regra de habilitação referente ao ramo). Se, após decorrida **uma unidade de tempo** (observe o intervalo de disparo de **PipelineError**), esta situação perdurar, **PipelineError** irá disparar, removendo todos os *tokens* presentes nos seus lugares de entrada. Outra parte da rede (não ilustrada na figura 11.19) deverá dar o tratamento adequado para tal situação. O importante, porém, é que, a adoção desta transição preemptiva não permite que erros no *pipeline* interfiram no mecanismo de **clock**, o que poderia causar transtornos em outros pontos da rede.

RPSD utiliza dois tipos de ramos para representar **interrupção: ramo de entrada suspensivo** e **ramo de saída restaurador**. O primeiro permite uma transição **suspender temporariamente** as **instâncias ativas** do comportamento associado a um lugar funcional. Caso o comportamento do lugar seja descrito por uma **RPSD-T**, as instâncias em execução serão **suspensas** através da **impossibilidade de habilitação** de suas transições (há uma **variável de controle ENABLE** implicitamente associada a cada transição, cujo valor passa para **FALSE** quando da atuação de um ramo suspensivo). No caso do comportamento ser descrito por uma **rotina**, cada instância termina de executar o *statement* (sentença de programação) em processamento e fica **suspensa**. O ramo restaurador tem ação oposta, permitindo que uma transição **retome** as instâncias do **comportamento** descrito por um lugar funcional, que tenham sido suspensas pelo disparo de outra transição, ligada ao lugar por um ramo suspensivo. Caso o comportamento do lugar seja descrito por uma **RPSD-T**, as **instâncias suspensas voltam a executar**, através da **não interferência** nas regras de habilitação de suas transições (valor da **variável de controle ENABLE** passa para **TRUE**). Caso o comportamento seja descrito por uma **rotina**, cada instância retoma sua execução no *statement* **seguinte** àquele em que o processamento havia sido suspenso.

Na figura 11.20, as transições **Suspender** e **Retomar** utilizam ramos para o tratamento de interrupções. A transição **Suspender** é responsável pela **suspensão temporária** das **instâncias ativas** do **comportamento** associado ao lugar **FiatLux**, uma vez que conecta-se com tal lugar através de um **ramo de entrada suspensivo**. Por outro lado, a transição **Retomar** tem efeito contrário, qual seja, a **retomada de execução** das **instâncias** anteriormente **suspensas** pelo disparo da transição **Suspender**, uma vez que conecta-se ao lugar **FiatLux** por meio de um **ramo de saída restaurador**. Na figura, pode-se observar que há duas instâncias em execução, pois as duas últimas linhas da tabela representante do **repositório de entrada** do lugar estão **indisponíveis**.

Suponhamos que as instâncias em execução do comportamento associado ao lugar **FiatLux** apresentem a configuração ilustrada na figura 11.21 e que, no momento considerado em tal figura, o valor da variável **interrupt** passe para 4, o que habilitará a transição **Suspender**. Como esta transição dispara imediatamente após a sua habilitação (devido ao seu intervalo de disparo [0,0]), as instâncias da rotina **FiatLux** executarão o *statement* atual e ficarão suspensas, até que a transição **Retomar** dispare. Para que isto ocorra, é necessário que seja atribuído o valor 0 (zero) à variável **interrupt**, habilitando a transição, e que este valor se mantenha por, pelo menos, 100 unidades de tempo sem sofrer alteração (o que desabilitaria a transição).

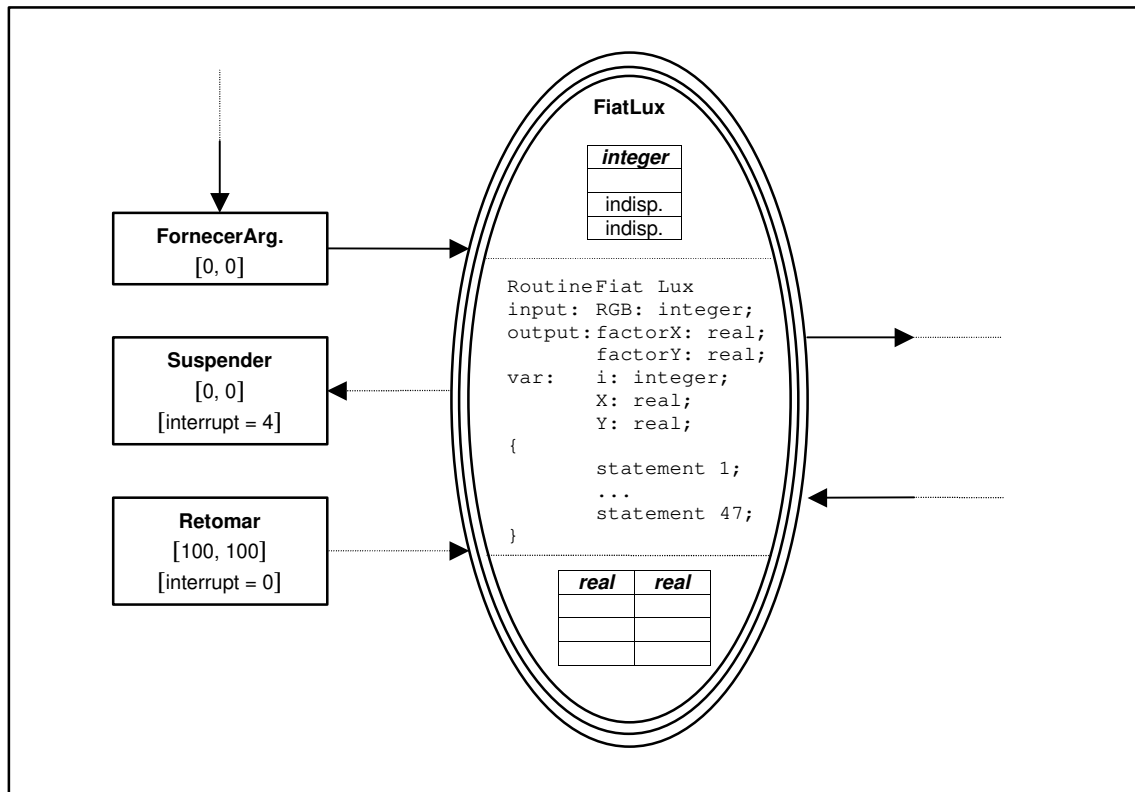


Figura 11.20. Modelagem de interrupção em RPSD.

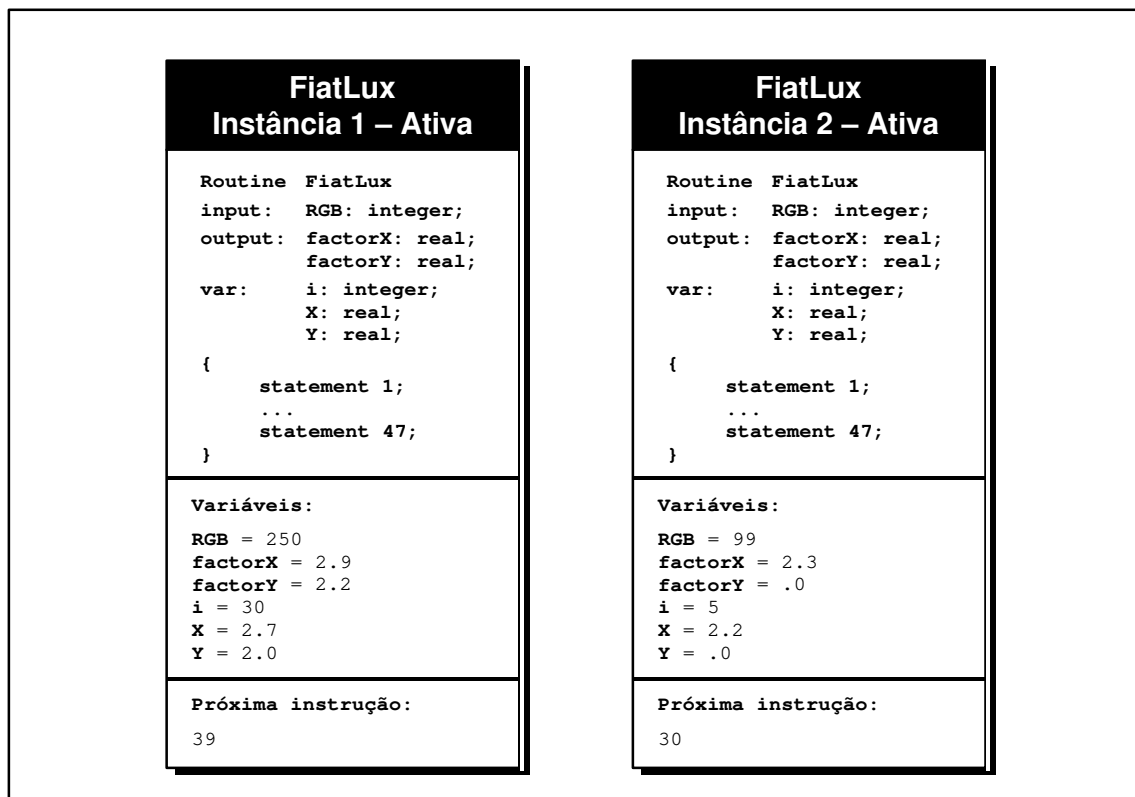


Figura 11.21. Instâncias ativas do lugar FiatLux da RPSD da figura 11.20.

O tratamento de exceções em RPSD supera, em muito, os poucos mecanismos encontrados em algumas das linguagens tratadas no capítulo 4. Naquele capítulo, foi visto que somente **Verilog** e **Statecharts** apresentam mecanismos **explícitos** para o tratamento de preempções.

g) *Representação de tempo*

A maior parte das linguagens utilizadas na modelagem de sistemas digitais não possui representação de tempo adequada. No caso de redes de Petri, há diferentes mecanismos para tratamento de tempo, de acordo com a extensão adotada (veja capítulo 8). Basicamente, as extensões associam mecanismos de tempo a lugares ou a transições. Entretanto, não há uma extensão de rede de Petri que utilize ambas abordagens simultaneamente.

As formas de representação de tempo em RPSD permitem a especificação tanto de **temporização funcional** (que pode interferir no comportamento da rede) quanto de **restrições temporais** (limitando tempos de execução ou de permanência num dado estado). Além de associar **intervalos de tempo** para a possibilidade de **ocorrência de transições entre comportamentos e/ou estados locais** (para modelagem de **mecanismos de comunicação**), foi verificada a necessidade de associar-se **tempos** para as **operações de escrita e leitura** de dados presentes em lugares do tipo II, que modelam componentes de armazenamento do sistema. Desta forma, RPSD trabalha com **duas estruturas** de tempo: uma que associa **intervalos de tempo para as transições** (denotando o tempo mínimo e o máximo para o disparo de uma transição habilitada) e outra que associa **latências de leitura e de escrita** para operações envolvendo componentes de armazenamento. *Time-stamps* para modos (estados locais ou comportamentos), **restringindo o tempo de execução** de um comportamento ou o **tempo de permanência** em um estado a um valor *MaxTime*, podem ser indiretamente especificados através do uso de **ramos preemptivos** conectados a transições com intervalo de disparo

[*MaxTime*, *MaxTime*].

RPSD possui uma representação de tempo mais completa do que a encontrada em linguagens como **VHDL**, **Verilog**, **Statecharts** e **SDL**, nas quais somente temporização funcional pode ser especificada, e **CSP**, que não possui mecanismos de especificação de tempo.

h) *Comunicação*

Geralmente, um sistema digital é composto por vários comportamentos que interagem entre si, de forma cooperada. É importante para uma linguagem de descrição de sistemas fornecer um mecanismo de comunicação genérico, o qual faça a **distinção** entre os **comportamentos** relativos à **comunicação** daqueles relativos à **computação**.

As **transições RPSD com intervalo de tempo de disparo** satisfazem as necessidades de **comunicação** explicitamente, uma vez que elas permitem que **comunicações** possam ser realizadas dentro de **restrições de tempo** específicas.

Comunicação por **memória compartilhada** também é possível em RPSD, com o uso das **variáveis da rede**, para um dado modelo.

i) *Sincronização de processos*

Redes de Petri são assíncronas por definição. No entanto, a introdução de construções estruturais para sincronização não apresentam qualquer problema.

RPSD permite tanto o uso de técnicas de **sincronização dependente de dados** quanto de técnicas de **sincronização dependente de controle**.

Sincronização dependente de dados pode ser obtida através do uso de memória compartilhada (variáveis da rede, utilizadas pelas transições, ramos e comportamentos, podendo sofrer atribuições somente através de transições). Na figura 11.9, a variável de rede **controle** é utilizada para sincronizar a habilitação (e, conseqüentemente o disparo) das transições **Inserir1Argumento** e **Inserir2Argumentos**.

Pode-se obter sincronização dependente de controle através do uso de transições como pontos de sincronização para modos computacionais, tal como acontece com as transições **FinalizarTarefa1** e **FinalizarTarefa2** da figura 11.17. Estas transições retornam as tarefas **Tarefa1** e **Tarefa2** ao lugar **Tarefas**, após a conclusão dos processos relacionados (**ProcA**, **ProcB** e **ProcC** para a tarefa **Tarefa1**, **ProcC** e **ProcD** para **Tarefa2**).

RPSD permite a representação dos mecanismos de comunicação e de sincronização de processos encontrados nas linguagens de especificação mostradas no capítulo 4.

11.4. EXEMPLO DE MODELAGEM

A fim de ilustrar a força descritiva da extensão RPSD, nós concebemos um processador hipotético e desenvolvemos seu modelo equivalente em RPSD (figura 11.22). Tal processador tem 13 registradores (tabela 11.1), 31 instruções (tabela 11.2) e 4 modos de endereçamento (figura 11.23). Na sequência, discutiremos algumas características do processador proposto, antes de examinarmos aspectos de seu modelo RPSD.

Parte I: Sistema de memória e conjunto de registradores

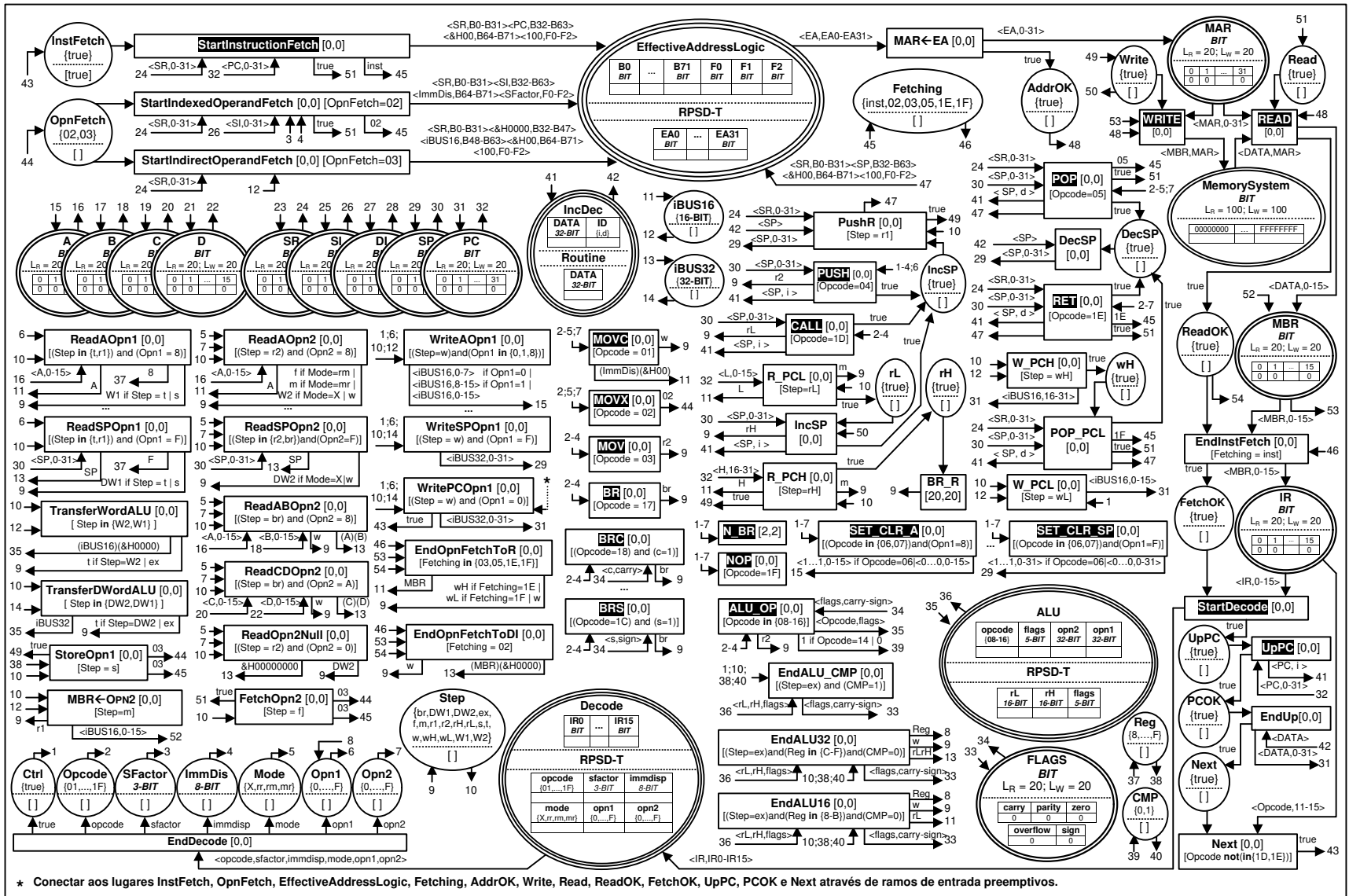
Para nosso processador hipotético, considerou-se que o **sistema de memória** tem 2^{32} posições, de 8 bits cada, com endereços numerados de **&H00000000** a **&HFFFFFFF** (**&H** = hexadecimal). Além disso, cada acesso à memória transfere uma palavra de 16 bits. O sistema de memória armazena tanto dados quanto programas. Sua organização não é especificada – entretanto, o modelo da figura 11.22 pode adotar diferentes organizações de memória simplesmente alterando as latências de leitura e de escrita associadas ao lugar do tipo II **MemorySystem**. Para operar, um processador necessita armazenar dados temporários, o que é realizado através do uso de **registradores**. Nós definimos 13 registradores, descritos na tabela 11.1. **Registradores visíveis ao usuário** (aqueles diretamente acessíveis pelos programas do usuário) são identificados por um dígito hexadecimal. Há quatro **registradores de propósito geral**, de 16 bits (**A**, **B**, **C** e **D**), cada qual podendo ser utilizado como dois

registradores de 8 bits (AH, AL, BH, BL, CH, CL, DH e DL). Além disso, há nove **registradores especializados**, quatro deles **visíveis ao usuário**: SR, SI, DI, e SP. O **registrador de segmento SR** armazena o endereço base do segmento atual (o conjunto de posições de memória no qual o próximo operando/instrução será buscado). **Acessos à memória** são feitos com **base** no valor armazenado em SR. Deste ponto em diante, sempre que for mencionado que um registrador **R** armazena uma posição de memória, deve-se entender que a posição **efetiva** de memória é derivada do conteúdo dos registradores **R** e **SR**. Os registradores **SI** (*source index*) e **DI** (*destination index*) são usados no **modo de endereçamento indireto** (*indexed addressing mode* – veja figura 11.23), útil no processamento de vetores. O registrador **SP** (*stack pointer*) armazena o **topo da pilha** mantida na memória: esta pilha é atualizada pelas instruções **PUSH, POP, CALL e RET** (veja adiante). Os registradores **MAR, MBR, PC, IR e FLAGS** **não são visíveis ao usuário**. Acessos à memória utilizam os registradores **MAR** e **MBR**, respectivamente, para armazenar a **posição de memória** (endereço de 32 bits) e os **dados trocados** (para ou de uma posição de memória). O registrador **PC** (*program counter*) armazena a posição de memória a partir da qual a **próxima instrução** será buscada (*fetch*). A instrução **mais recentemente trazida** (*buscada*) da memória é armazenada no registrador **IR** (*instruction register*). Finalmente, o registrador **FLAGS** armazena **bits de condição** (conhecidos como *condition bits* ou *flags*), relacionados à operação mais recentemente executada pela **ALU** (*Arithmetic and Logic Unit* – **Unidade Lógico-Aritmética**): após uma operação na **ALU**, os *flags* **carry, parity, zero, overflow** e **sign** são *setados* (recebem o bit **0** ou **1**), de acordo com o **resultado** gerado pela operação. Tais *flags* são usados por **instruções de desvio condicional** (*conditional branch*) para determinar se um **desvio** (*branch*) será adotado no **fluxo de execução** do programa corrente.

Tabela 11.1. Conjunto de registradores para o processador proposto.

Registrador	Código (hexadecimal)	Tamanho (bits)	Descrição
AH/AL	0/1	8	registradores de propósito geral
BH/BL	2/3	8	
CH/CL	4/5	8	
DH/DL	6/7	8	
A	8	16	
B	9	16	
C	A	16	
D	B	16	
SR	C	32	<i>segment register</i>
SI	D	32	<i>source index register</i>
DI	E	32	<i>destination index register</i>
SP	F	32	<i>stack pointer</i>
MAR	-	32	<i>memory address register</i>
MBR	-	16	<i>memory buffer register</i>
PC	-	32	<i>program counter</i>
IR	-	16	<i>instruction register</i>
FLAGS	-	5	<i>flag register</i>

Figura 11.22. Modelo RPSD do processador pro posto.



Parte II: Conjunto de instruções (*instruction set*)

Foi definido um pequeno, embora ilustrativo, **conjunto de instruções (*instruction set*)** para o processador proposto, conforme mostra a tabela 11.2. Antes de examinar tal conjunto, vejamos como uma instrução é composta.

Tabela 11.2. Conjunto de instruções para o processador proposto.

Tipo	Opcode (hexa)	Instrução	Operandos	Descrição
Trasferência de Dados (<i>Data Transfer</i>)	01	MOVC	R _{8-bit} , Const	$R_{8-bit} \leftarrow \text{Const}$
	02	MOVX	Const	$DI \leftarrow (\text{Const})$
	03	MOV	R1, R2	$R1 \leftarrow (R2) \mid R1 \leftarrow R2 \mid (R1) \leftarrow R2$
	04	PUSH	R	topo da pilha $\leftarrow R$
	05	POP	R	$R \leftarrow \text{topo da pilha}$
	06	SET	R	$R \leftarrow 111\dots 1$
	07	CLR	R	$R \leftarrow 000\dots 0$
Aritmética (<i>Arithmetic</i>)	08	ADD	R1, R2	$R1 \leftarrow R1 + R2$
	09	SUB	R1, R2	$R1 \leftarrow R1 - R2$
	0A	MUL	R1, R2	$R1 \leftarrow R1 * R2$
	0B	DIV	R1, R2	$R1 \leftarrow R1 / R2$
	0C	NEG	R	$R \leftarrow -R$
	0D	ABS	R	$R \leftarrow R $
	0E	INC	R	$R \leftarrow R + 1$
	0F	DEC	R	$R \leftarrow R - 1$
Lógica (<i>Logical</i>)	10	AND	R1, R2	$R1 \leftarrow R1 \text{ AND } R2$
	11	OR	R1, R2	$R1 \leftarrow R1 \text{ OR } R2$
	12	XOR	R1, R2	$R1 \leftarrow R1 \text{ XOR } R2$
	13	NOT	R	$R \leftarrow \text{NOT } R$
	14	CMP	R1, R2	compara R1 e R2, configurando <i>flags</i>
	15	LRS	R	deslocamento lógico para a direita em R
	16	LLS	R	deslocamento lógico para a esquerda em R
Transferência de Controle (<i>Control Transfer</i>)	17	BR	R	desvio incondicional para o endereço (R)
	18-1C	BRC/P/Z/O/S	R	desvio para o endereço (R) se o <i>flag carry (parity/zero/overflow/sign)</i> for 1
	1D	CALL	R	chamar subrotina no endereço (R)
	1E	RET	nenhum	retornar de subrotina
	1F	NOP	nenhum	nenhuma operação

Cada instrução possui 16 bits, ocupando **duas posições consecutivas** na memória: o **endereço da instrução** na memória corresponde ao **endereço da primeira posição**

ocupada pela instrução. Os **campos** de uma instrução incluem seu **opcode** (código da operação) e seus **operandos** (*nenhum operandolum/duois*), com seus respectivos **modos de endereçamento** (*addressing modes*). Nós definimos três **formatos de instrução** (*instruction formats*) e quatro **modos de endereçamento**, conforme mostra a figura 11.23.

Modo de endereçamento imediato (<i>immediate addressing mode</i>)					
operando = constante = immediate value					
Campo	Opcode (somente 01)		Destination Register (8 bits)		ImmediateValue
Tamanho	5		3		8
Faixa Bits	15	11	10	8	7
					0
Modo de endereçamento indexado (<i>indexed addressing mode</i>)					
operando = conteúdo da posição de memória especificada por (segment register) + (source index register) * (scaling factor) + displacement					
Campo	Opcode (somente 02)		ScalingFactor		Displacement
Tamanho	5		3		8
Faixa Bits	15	11	10	8	7
					0
Modo de endereçamento direto por registrador (<i>register direct addressing mode</i>)					
operando = conteúdo do registrador					
Modo de endereçamento indireto por registrador (<i>register indirect addressing mode</i>)					
operando = conteúdo da posição de memória especificada por (segment register) + (referenced register)					
Campo	Opcode	n/a	AddrMode (00 01 10)*	DestinationRegister	SourceRegister
Tamanho	5	1	2	4	4
Faixa Bits	15	11	10	9	8
				7	4
					3
					0
* bit 9 = modo de endereçamento do registrador destino (DestinationRegister) (0=direto; 1=indireto) bit 8 = modo de endereçamento do registrador fonte (SourceRegister) (0=direto; 1=indireto)					

Figura 11.23. Formatos de instrução e modos de endereçamento do processador.

O **modo de endereçamento imediato** (*immediate addressing mode*) é utilizado somente pela instrução **MOVC** (opcode **01**), cujos operandos são uma constante (campo *ImmediateValue*) e um especificador de registrador de 8 bits (campo *DestinationRegister*, o qual armazena um código hexadecimal entre **0** e **7**, denotando um dos registradores de 8 bits).

Somente a instrução **MOVX** usa o **modo de endereçamento indexado** (*indexed addressing mode*). Neste modo, o operando corresponde à posição de memória especificada pela soma

$$(SR) + (SI) * (ScalingFactor) + Displacement,$$

onde *(SR)* e *(SI)* são, respectivamente, o conteúdo dos registradores **SR** e **SI**; *ScalingFactor* e *Displacement* são campos da instrução.

O último formato na figura 11.23 é utilizado pelo restante das instruções, permitindo dois modos de endereçamento. Um operando no **modo de endereçamento direto por registrador** (*register direct addressing mode*) corresponde ao conteúdo de um registrador. Por outro lado, um operando no **modo de endereçamento indireto por registrador** (*register indirect addressing mode*) corresponde a uma posição de memória obtida da soma do conteúdo do registrador especificado com o conteúdo de **SR**. O campo *AddrMode* tem dois bits: o primeiro define o modo de endereçamento (**direto/indireto**) do operando especificado pelo campo *DestinationRegister*, enquanto o segundo é relacionado ao operando referenciado pelo campo *SourceRegister*.

O conjunto de instruções é dividido em **quatro grupos**, conforme mostrado na tabela 11.2. Instruções de **transferência de dados** (*data transfer*) são responsáveis por copiar dados para dentro de um registrador ou posição de memória, a partir de um operando fonte (*source operand*). Tal operando pode ser uma constante (no caso das instruções **MOVC**, **MOVX**, **SET** e **CLR**) ou um outro registrador (no caso da instrução **MOV**). A instrução **PUSH R** incrementa o conteúdo do registrador **SP** (isto é, atualiza o topo da pilha) e copia o conteúdo do registrador **R** para dentro da posição de memória referenciada por **SP**. A instrução **POP R** tem efeito contrário, ao copiar para o registrador **R** o conteúdo da posição de memória referenciada por **SP**, decrementando, em seguida, tal ponteiro.

Instruções **lógicas e aritméticas** (*arithmetic and logical*) denotam operações envolvendo um ou dois operandos, especificados por registradores: o resultado de uma operação é computado pela **ALU**, sendo armazenado no registrador especificado pelo primeiro operando (campo *DestinationRegister*). Em seguida, o registrador **FLAGS** é atualizado. Exceção é a instrução **CMP**, a qual apenas compara o conteúdo de dois registradores e atualiza o registrador **FLAGS**, não alterando, portanto, o primeiro operando (*DestinationRegister*).

Instruções de **transferência de controle** (*control transfer*) permitem alterar o fluxo normal de execução das instruções (uma após a outra, numa sequência linear de endereços). A instrução de desvio **BR R** altera o fluxo de execução para a instrução presente na posição de memória referenciada pelo registrador **R**. Similarmente a **BR**, as instruções condicionais de desvio **BRC/BRP/BRZ/BRO/BRS** levam em conta (para fazer o desvio), respectivamente, os flags **carry**, **parity**, **zero**, **overflow** e **sign**, os quais foram *setados* (para 0 ou 1) de acordo com a operação lógica/aritmética mais recentemente executada pela **ALU**. As instruções **CALL** e **RET** são usadas com sub-rotinas. **CALL R** salva o conteúdo do registrador **PC** na pilha (cujo topo é apontado por **SP**) e desvia o fluxo de execução para a posição de memória referenciada pelo registrador **R**; **RET** recupera o conteúdo de **PC** anteriormente salvo por uma instrução **CALL**, retomando o fluxo de execução que havia sido interrompido pela chamada da subrotina correspondente. O uso da pilha permite o aninhamento de sub-rotinas (uma sub-rotina ser chamada a partir de outra subrotina). Finalmente, a instrução **NOP** não executa qualquer operação, apenas deixando o processador ocioso.

Parte III: Descrição do modelo RPSD

Exposta a especificação do processador proposto, examinemos como o modelo RPSD da figura 11.22 reflete as características de tal especificação.

Cada **registrador** na tabela 11.1 é descrito por um lugar do tipo II no modelo RPSD, com latências de leitura e de escrita iguais a **20**, representando que um acesso ao registrador consome **20** unidades de tempo (por exemplo, **20** ns). O tamanho de um registrador é representado pela quantidade de **células de armazenamento** no lugar relacionado. Por exemplo, o lugar **FLAGS** (correspondente ao registrador **FLAGS**) possui cinco **células de armazenamento**, denominadas **carry**, **parity**, **zero**, **overflow** e **sign**. Cada **célula de armazenamento** pode ser ocupada por apenas um *token* (representando um valor), pertencente ao **tipo BIT** (*nota: conforme visto na seção 11.2.2, os tipos são definidos na classe anotações, da tupla definidora de um modelo RPSD, não sendo exibidos na representação gráfica do modelo*). Todos os lugares relacionados a registradores são inicializados com **0's** (bits zero).

O **sistema de memória** é representado pelo lugar de armazenamento **MemorySystem**, cujas **células de armazenamento** são numeradas de **&H00000000** a **&HFFFFFFF** (notação hexadecimal). Cada **célula de armazenamento** armazena um *token* pertencente ao **tipo BYTE** (conjunto {**00000000**,...,**11111111**}). As duas primeiras células de armazenamento devem armazenar a primeira instrução a ser executada pelo processador.

Um **acesso de escrita** (*write access*) é acionado pelo disparo da transição **Write**, o que depende da presença de um *token true* nos lugares de tipo I **AddrOK** e **Write**: a presença de um *token true* em **AddrOK** representa que o registrador **MAR** armazena um endereço atualizado; já a presença de um *token true* em **Write** representa o vigorar de um sinal de controle, responsável pelo acionamento da operação de escrita. O disparo da transição **WRITE** envolve três etapas:

- remoção do *token true* de ambos lugares **AddrOK** e **Write**;
- atribuição (cópia) dos conteúdos dos lugares **MAR** e **MBR** aos **especificadores (contêineres)** de *token* de mesmo nome, utilizados nas **expressões dos ramos** conectados à transição **WRITE**, isto é, o especificador de *token MAR* (**MBR**) recebe o conteúdo do lugar **MAR** (**MBR**) – observe as expressões de ramos **<MAR,0-31>** e **<MBR,0-15>**; e
- atualização da célula de armazenamento de **MemorySystem**, correspondente ao conteúdo do especificador de *token MAR*, com o valor representado pelo especificador de *token MBR* – observe a expressão de ramo **<MBR,MAR>**.

Das etapas acima descritas, a primeira não possui atraso. A segunda consome **20** unidades de tempo, enquanto a terceira consome **100** unidades de tempo (observe as latências associadas aos lugares).

Um **acesso de leitura** (*read access*) apresenta efeito contrário a um de escrita, ao atualizar o lugar **MBR** com o conteúdo de uma célula de armazenamento em **MemorySystem** (a célula referenciada pelo conteúdo do lugar **MAR**).

Há quatro lugares do tipo III na figura 11.22, representando **componentes funcionais** do processador proposto. Os comportamentos associados a tais lugares não são mostrados na figura. O lugar **EffectiveAddressLogic** é responsável por calcular o **endereço efetivo (real)** de um **operando** ou **instrução**. Ao calcular um endereço efetivo, o conteúdo de um registrador de 32 bits é multiplicado por um **fator** de **1**, **2** ou **4** (chamado *scaling factor*) e, então, adicionado à soma de um deslocamento de 8 bits (chamado *displacement*) com o conteúdo do registrador de segmento (**SR**). Matematicamente, temos:

$$(SR) + (Reg_{32}) * (ScalingFactor) + Displacement.$$

Argumentos para os parâmetros *Reg₃₂*, *ScalingFactor* e *Displacement* são fornecidos ao lugar **EffectiveAddressLogic** de acordo com a transição disparada. Por exemplo, ao disparar a transição **StarInstructionFetch**, tais argumentos serão, respectivamente, o conteúdo do registrador **PC**, o valor binário **100** (representando o fator **1** – o bit menos significativo fica à esquerda, por convenção), e o valor hexadecimal **&H00** (representando um deslocamento nulo).

O lugar funcional **IncDec** incrementa/decrementa o número de 32 bits associado a seu parâmetro **DATA**, de acordo com seu parâmetro **ID** (**i** = incrementar; **d** = decrementar). Este lugar é usado no incremento de **PC** (após o ciclo de busca de instrução, descrito adiante) e na atualização de **SP** (incrementando **SP** nas instruções **PUSH** e **CALL**; decrementando **SP** nas instruções **POP** e **RET**).

O lugar funcional **ALU** é usado para executar operações aritméticas e lógicas. Este lugar possui quatro parâmetros de entrada: **opcode** (entre os hexadecimais **08** e **16** – veja a tabela 11.2), **flags** (os cinco bits de condição, a serem adquiridos do lugar **FLAGS**), **opn2** (o conteúdo de 32 bits do registrador especificado como segundo operando da operação) e **opn1** (o valor do primeiro operando da operação). O resultado gerado é dividido em duas partes: **rL** (os 16 bits menos significativos do resultado) e **rH** (os 16 bits mais significativos do resultado). A saída **flags** contém os cinco bits de condição a serem armazenados no lugar **FLAGS**.

Finalmente, o lugar funcional **Decode** é responsável pela decomposição dos 16 bits que representam a instrução corrente (adquirida do lugar **IR**), em seis parâmetros a serem utilizados pelas transições relacionadas ao ciclo de execução de instrução (veja adiante).

O ciclo de instrução (*instruction cycle*) é composto de **três estágios**:

- I) Ciclo de busca de instrução (*instruction fetch cycle*). Neste ciclo, a instrução referenciada por **PC** é buscada (trazida – *fetched*) de **MemorySystem** e, então, armazenada no lugar **IR**. Este ciclo é executado através da seguinte sequência de disparo de transições:

StartInstructionFetch – MAR ← EA – READ – EndInstFetch;

- II) Ciclo de decodificação de instrução (*instruction decoding cycle*). Este ciclo **decompõe** a instrução buscada no estágio anterior, **disponibilizando** seus **campos** para o próximo estágio, através dos lugares de tipo I **Opcode**, **Sfactor**, **ImmDis**, **Mode**, **Opn1** e **Opn2** (veja figura 11.23). Este ciclo é iniciado com o disparo da transição **StartDecode** e finalizado com o disparo da transição **EndDecode**;

- III) Ciclo de execução de instrução (*instruction execute cycle*). Uma vez decomposta, a instrução deve ser executada, de acordo com seu **opcode**. Alguns exemplos de execução de instruções são dados na sequência. Durante este estágio, o lugar **Ctrl** (tipo I) armazenará um **token true** (inserido no lugar pelo disparo anterior de **EndDecode**), o que impedirá nova habilitação da transição **EndDecode**. Consequentemente, mesmo se a próxima instrução buscada já tiver sido decodificada, ela não poderá entrar em execução até que a instrução correntemente em execução termine (observe as transições responsáveis pela remoção do **token true** de **Ctrl**).

Os estágios do ciclo de instrução podem ser executados em paralelo. Uma vez que ciclo de execução de instrução pode acessar a **memória** (mais especificamente, nas instruções com **opcode 02, 03, 05, 1E e 1F**), um **conflito** surgiria se o ciclo de busca de instrução acessasse a memória ao mesmo tempo. Para evitar tal conflito, o lugar **Fetch**ing (tipo I) **controla os acessos à memória**. Observe as transições que podem inserir um *token* em **Fetch**ing: a primeira destas transições a disparar determinará a **prioridade de acesso** ao lugar **MemorySystem** (através das transições **MAR←EA**, **READ** e **WRITE**) ou para o ciclo de busca de instrução ou para ciclo de execução de instrução.

Seja *I* a instrução que acaba de ser buscada. Quando o ciclo de decodificação de instrução começa, o conteúdo de **PC** será incrementado, para que a próxima instrução (*I+I*) possa ser buscada. Entretanto, isto somente deverá ocorrer (com a inserção do *token true* no lugar **InstFetch**) se, e somente se, o **opcode** da instrução *I* (representado pelos bits compreendidos entre as células **11 e 15** do lugar **IR**) for diferente de **1D** (instrução **CALL**) e **1E** (instrução **RET**). Isto se deve ao fato de tais instruções promover um desvio no fluxo de execução, tornando inútil a busca da instrução *I+I*. **Instruções de desvio** (**opcode** entre **18 e 1C**) também podem alterar o fluxo de execução. Durante a execução de tais instruções, a busca e a decodificação das instruções subsequentes não serão interrompidas de imediato (como nos casos de **CALL** e **RET**). Isto se deve à possibilidade do desvio não ser adotado (o que provoca o disparo da transição **N_BR**). No caso do desvio ser adotado, o disparo da transição **WritePCOpn1** (a **última transição no ciclo de execução** de qualquer instrução de desvio) removerá os *tokens* de todos os lugares que possam ter sido alterados pela busca e pela decodificação das instruções subsequentes (observe o uso de ramos de entrada **preemptivos** no modelo do processador).

Para finalizar a discussão sobre o modelo da figura 11.22, vamos inspecionar o ciclo de execução de algumas **instruções**, quais sejam: **MOV**, **BRC**, **ADD**, **CMP**, **CALL** e **RET**. Este ciclo começa após o disparo de **EndDecode**. Em todos os casos, o lugar **Step** (tipo I) é usado para ordenar o fluxo de execução, isto é, o *token* presente em **Step** determinará a próxima transição a ser habilitada e, conseqüentemente, disparada.

A tabela 11.3 sintetiza o comportamento do modelo RPSD, durante o ciclo de execução, para as instruções consideradas. Por exemplo, suponha que a instrução

MOV B,(A)

acaba de ser decodificada. Neste caso, o conteúdo do lugar **Mode** (tipo I) será **rm** (representando que o primeiro operando corresponde a um registrador, enquanto o segundo corresponde a uma posição de memória referenciada por um registrador). Ocorrerá, então, a seguinte seqüência de disparo de transições, a fim de copiar o conteúdo da posição de memória referenciada pelo registrador **A** para o registrador **B** (acompanhe na figura 11.22):

**MOV – ReadAOpn2 – FetchOpn2 – StartIndirectOperandFetch –
MAR←EA – READ – EndOpnFetchtoR – WriteBOpn1.**

Apesar da transição **WriteBOpn1** não estar presente na figura 11.22, ela é uma transição similar a **WriteAOpn1**, substituindo-se o rótulo de saída **15** por **17** (representando a atualização do registrador **B**).

Tabela 11.3. Ciclo de execução de instrução para alguns exemplos.

Exemplo	Seqüência de ações	Lugares do tipo I após o ciclo de decodificação			Seqüência de disparo de transições
MOV B,(A)	• Copia o conteúdo da posição de memória referenciada pelo registrador A para o registrador B.	Opcode 03	Sfactor 000	ImmDis 00000000	MOV – ReadAOpn2 – FetchOpn2 – StartIndirectOperandFetch – MAR←EA – READ – EndOpnFetchToR – WriteBOpn1
		Mode rm	Opn1 9	Opn2 8	
BRC A	• Copia o conteúdo dos registradores A e B para o registrador PC ($PC_{low} \leftarrow A$, $PC_{high} \leftarrow B$), se o bit de condição Carry (no registrador FLAGS) for 1; • Limpa o <i>pipeline</i>; • Prepara um novo ciclo de busca.	Opcode 18	Sfactor 000	ImmDis 00000000	BRC – ReadABOpn2 – WritePCOpn1
		Mode Rr	Opn1 0	Opn2 8	
ADD SI,A	• Insere no registrador SI a soma dos conteúdos registradores A e SI.	Opcode 08	Sfactor 000	ImmDis 00000000	ALU_OP – ReadAOpn2 – TransferWordALU – ReadSIOpn1 – TransferDWordALU – EndALU32 – WriteSIOpn1
		Mode X	Opn1 D	Opn2 8	
CMP A,B	• Compara o conteúdo dos registradores A e B; • Configura os bits de condição em FLAGS.	Opcode 14	Sfactor 000	ImmDis 00000000	ALU_OP – ReadBOpn2 – TransferWordALU – ReadAOpn1 – TransferWordALU – EndALU_CMP
		Mode X	Opn1 8	Opn2 9	
CALL DI	• Incrementa o registrador SP; • Salva o conteúdo do registrador PC na pilha (de acordo com SP); • Copia o conteúdo do registrador DI para PC; • Prepara um novo ciclo de busca.	Opcode 1D	Sfactor 000	ImmDis 00000000	CALL – R_PCL – IncSP – MBR←Opn2 – PushR – MAR←EA – WRITE – R_PCH – MBR←Opn2 – PushR – MAR←EA – WRITE – BR_R
		Mode Rr	Opn1 0	Opn2 E	
RET	• Copia o conteúdo da posição de memória referenciada pelo registrador SP para o registrador PC; • Decrementa SP; • Prepara um novo ciclo de busca.	Opcode 1E	Sfactor 000	ImmDis 00000000	RET – DecSP – MAR←EA – READ – EndOpnFetchToR – W_PCH – POP_PCL – MAR←EA – READ – EndOpnFetchToR – W_PCL
		Mode Rr	Opn1 0	Opn2 0	

O modelo RPSD desenvolvido para o processador proposto pode ser estendido de várias formas, como, por exemplo, através da inclusão de instruções para o tratamento de interrupções. Nossa intenção com o modelo exposto foi, contudo, demonstrar como a extensão RPSD pode ser utilizada de forma relativamente simples na modelagem de sistemas digitais.

CAPÍTULO 12: EQUIVALÊNCIA ENTRE REDE DE PETRI PARA SISTEMAS DIGITAIS E OUTRAS EXTENSÕES DE REDES DE PETRI

12.1. INTRODUÇÃO

Neste capítulo, são abordadas formas de **equivalência** entre a extensão desenvolvida neste trabalho de doutorado – **RPSD (Rede de Petri para Sistemas Digitais)** – e extensões de rede de Petri de baixo nível (tratadas na Parte II desta tese), principalmente as redes **Lugar/Transição**. Nos casos de transições **temporizadas**, são utilizados modelos em **redes de Petri Temporizadas de Merlin**, as quais nada mais são do que uma extensão das redes Lugar/Transição. Os modelos apresentados neste capítulo são validados, através de simulação, no apêndice II desta tese. Ao demonstrar a possibilidade de transcrição dos componentes estruturais de RPSD em modelos baseados em redes Lugar/Transição, prova-se que RPSD é uma rede de Petri. Assim, encontra-se uma via na qual resultados teóricos consolidados para extensões de redes de Petri possam ser estendidos para RPSD.

12.2. EQUIVALÊNCIA DE LUGARES

Para representar diferentes aspectos inerentes à modelagem de sistemas digitais embutidos, RPSD possui três tipos de lugares, quais sejam:

- a) **Tipo I (Estado Local)**: Utilizado para indicar que um ou mais **componentes do sistema de mesmo tipo** encontram-se num mesmo **estado local**;
- b) **Tipo II (Armazenamento)**: Utilizado para representar **componentes de armazenamento do sistema**, nos quais operações de *leitura/escrita* de dados podem ser feitas; e
- c) **Tipo III (Funcional)**: Utilizado para indicar que um ou mais *samples de dados* encontram-se em **processamento** por um ou mais **componentes funcionais de mesmo tipo**.

Nas seções seguintes (12.2.1 a 12.2.3), é visto como os diferentes tipos de lugares RPSD podem ser representados por **conjuntos de lugares de redes de Petri Lugar/Transição (L/T – capítulo 7)**. Nas figuras dessas seções, as **transições** das redes L/T foram **omitidas**: o objetivo é mostrar as **diferentes marcações** que tais conjuntos de lugares podem assumir, de acordo com **diferentes configurações** de lugares em RPSD. As transições responsáveis pela **interligação** dos lugares das redes L/T são apresentadas na seção 12.3. Os modelos em redes L/T, ilustrados ao longo deste capítulo, buscam comprovar que RPSD é uma extensão de redes de Petri válida, através da possibilidade de **conversão** de uma rede RPSD em uma rede L/T. A preocupação durante a concepção dos modelos L/T foi a de gerar modelos de equivalência válidos, não necessariamente otimizados.

12.2.1. TIPO I: ESTADO LOCAL

Lugares do tipo I são utilizados para indicar que um ou mais **componentes do sistema** (representados por *tokens*) de um **mesmo tipo** (por exemplo, processos, unidades funcionais, de armazenamento ou de comunicação), encontram-se num mesmo **estado local**.

Como a composição e os mecanismos de funcionamento de lugares do tipo I foram fornecidos na seção 11.2.1.1.1, passa-se para a representação destes lugares (e mecanismos correspondentes) em **redes de Petri Lugar/Transição (L/T)**.

As figuras 12.1, 12.2 e 12.3 mostram como um lugar do tipo I pode ser representado por um **conjunto de lugares** de uma rede L/T. Nestas figuras, a presença de um *token* ●, num lugar d_i , indica que o i -ésimo elemento do conjunto **domínio** de P ($D_S(P)$) **não encontra-se** no **repositório** de P ($R_S(P)$), isto é, indica que o *token* d_i não está no lugar P (está, sim, **disponível** para **inserção** em P). Por outro lado, a presença de um *token* ● num lugar $d_i[j]$ indica que o *token* d_i encontra-se na j -ésima posição do **repositório** de P (o **repositório** de um lugar do tipo I é um **conjunto ordenado**, representado por um **vetor**).

O lugar P da figura 12.1 tem **domínio** $\{d_1\}$. Isto significa que P admite apenas a inserção de um único *token*, qual seja, d_1 . Pelas convenções adotadas no parágrafo anterior, P é representado por um **conjunto de dois lugares** de uma rede L/T: d_1 e $d_1[1]$. **Transições** da rede L/T, conectadas a estes dois lugares, devem garantir que, em qualquer momento da execução da rede, apenas um destes lugares possua um *token* ● (o outro lugar deverá estar vazio). Na figura 12.1.a, a presença de um *token* ● no lugar d_1 indica que P pode receber um *token* d_1 ; em outras palavras, P está **vazio** (razão pela qual o lugar $d_1[1]$ não possui *token*). A figura 12.1.b mostra situação oposta, ou seja, a presença do *token* d_1 na única posição existente de P (posição 1), é representada pela presença de um *token* ● no lugar $d_1[1]$ e pela ausência de *token* no lugar d_1 .

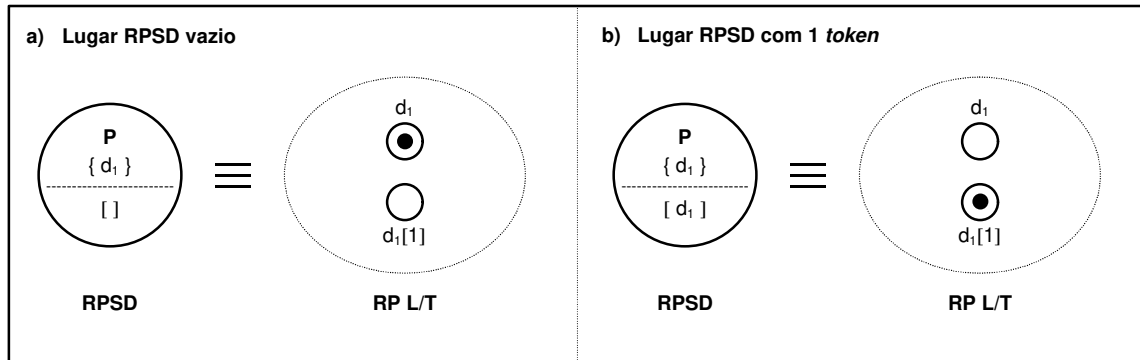


Figura 12.1. Correspondência entre lugar tipo I para 1 *token* e lugares de RP L/T.

Para um lugar P do tipo I com **domínio** de cardinalidade n ($n \geq 2$), seu **repositório** possui n posições **cronologicamente ordenadas**, representadas por um vetor $R_S(P)$, com índices variando de 1 a n . O *token* mais antigo no lugar deverá ocupar a posição $R_S(P)[1]$, o segundo mais antigo a posição $R_S(P)[2]$ e, assim, sucessivamente. *Tokens* podem ser **removidos** do lugar de forma **aleatória** ou de forma **cronológica**. Ao remover-se um *token* de uma posição $R_S(P)[i]$, $1 \leq i \leq n$, todos os demais *tokens* presentes em $R_S(P)$ são automaticamente deslocados uma posição para a esquerda, isto é, ocorre a seguinte seqüência de atribuições:

$$R_S(P)[i] := R_S(P)[i+1], R_S(P)[i+1] := R_S(P)[i+2], \dots, R_S(P)[j-1] := R_S(P)[j], i < j \leq n.$$

Desta forma, para o propósito de representação de lugares do tipo I com **domínio** de cardinalidade 2 (figura 12.2) ou superior (figura 12.3), através de redes L/T, adotou-se uma estrutura de **fila circular** para representação do **repositório** R_S . Uma fila circular tem tamanho fixo (dado por n , onde $n = \#D_S$) e possui dois ponteiros, **first** e **next**, que indicam,

respectivamente, a **posição do primeiro elemento** na estrutura (primeiro da fila) e a **próxima posição de inserção livre** (após o último da fila). A **remoção** de um elemento se dá através da retirada deste da célula da estrutura indicada por **first**. A **inserção** de um elemento ocorre na célula da estrutura indicada por **next**. Após uma **remoção/inserção**, o ponteiro correspondente (**first/next**) é atualizado. Assim, no caso de **remoção cronológica** de um ou mais *tokens* do lugar RPSD do tipo I, não é necessário, na rede L/T equivalente, fazer qualquer deslocamento de *tokens*, exceto os correspondentes aos ponteiros **first** e **next**. Entretanto, o mesmo não ocorre nos casos de **remoção aleatória** de *tokens*, conforme explicado adiante (seção 12.3).

O lugar **P** da figura 12.2 tem **domínio** $\{d_1, d_2\}$, de cardinalidade 2. Isto significa que **P** admite apenas a inserção dos *tokens* **d₁** e **d₂**. **P** é representado por um **conjunto de dez lugares** de uma rede L/T, assim distribuídos:

- lugares **d₁** e **d₂**, usados para indicar se o *token* correspondente está presente (lugar vazio) ou não (lugar com um *token* ●) no lugar **P**;
- lugares **d₁[1]** e **d₁[2]**, usados, respectivamente, para indicar se o *token* **d₁**, caso presente no lugar **P**, ocupa a posição de índice 1 ou 2 (nunca ambas) da estrutura **fila circular** representante de **R_S(P)**;
- lugares **d₂[1]** e **d₂[2]**, usados, respectivamente, para indicar se o *token* **d₂**, caso presente no lugar **P**, ocupa a posição de índice 1 ou 2 (nunca ambas) da estrutura **fila circular** representante de **R_S(P)**;
- lugares **first[1]** e **first[2]**, usados, respectivamente, para indicar se o *token* mais antigo em **P** ocupa a posição de índice 1 ou 2 (nunca ambas) da estrutura **fila circular** representante de **R_S(P)**;
- lugares **next[1]** e **next[2]**, usados, respectivamente, para indicar se o próximo *token* a ser inserido em **P** ocupará a posição de índice 1 ou 2 (nunca ambas) da estrutura **fila circular** representante de **R_S(P)**.

A figura 12.2 mostra as possíveis **configurações (marcações admissíveis)** que os lugares da rede L/T equivalente a **P** podem assumir. Na figura 12.2.a, **P** está vazio, o que é denotado pela presença do *token* ● nos lugares **d₁** e **d₂**. Além disso, os ponteiros **first** e **next** apontam para a mesma posição da estrutura **fila circular** representante de **R_S(P)**, qual seja, posição 1, na figura 12.2.a.1, e posição 2, na figura 12.2.a.2.

Tomando-se a configuração apresentada na figura 12.2.a.1 como ponto de partida, a inserção do *token* **d₁** em **P** fará com que a rede L/T assuma a configuração apresentada na figura 12.2.b.1. Observe que **R_S(P)[1] = d₁**, pois **first** aponta para a posição 1 da estrutura **fila circular** (*token* ● em **first[1]**), a qual contém o *token* **d₁** (lugar **d₁** vazio e um *token* ● em **d₁[1]**). A presença de um *token* ● em **next[2]** indica que a próxima inserção de *token* em **P** deverá ocorrer na posição 2 da estrutura **fila circular**. Neste ponto (figura 12.2.b.1), duas situações poderão ocorrer:

- a) o *token* **d₁** ser **removido** de **P**, o que deslocaria o ponteiro **first** uma fileira para cima (de **first[1]** para **first[2]**) e removeria o *token* do lugar **d₁[1]**, restituindo-o ao lugar **d₁**. A marcação resultante passaria para a descrita na figura 12.2.a.2; ou
- b) o *token* **d₂** ser **inserido** em **P**, o que deslocaria o ponteiro **next** uma fileira para cima (de **next[2]** para **next[1]**, devido à natureza circular da estrutura) e removeria o *token* do lugar **d₂**, inserindo-o no lugar **d₂[2]**. A marcação resultante passaria para a descrita na figura 12.2.d.1.

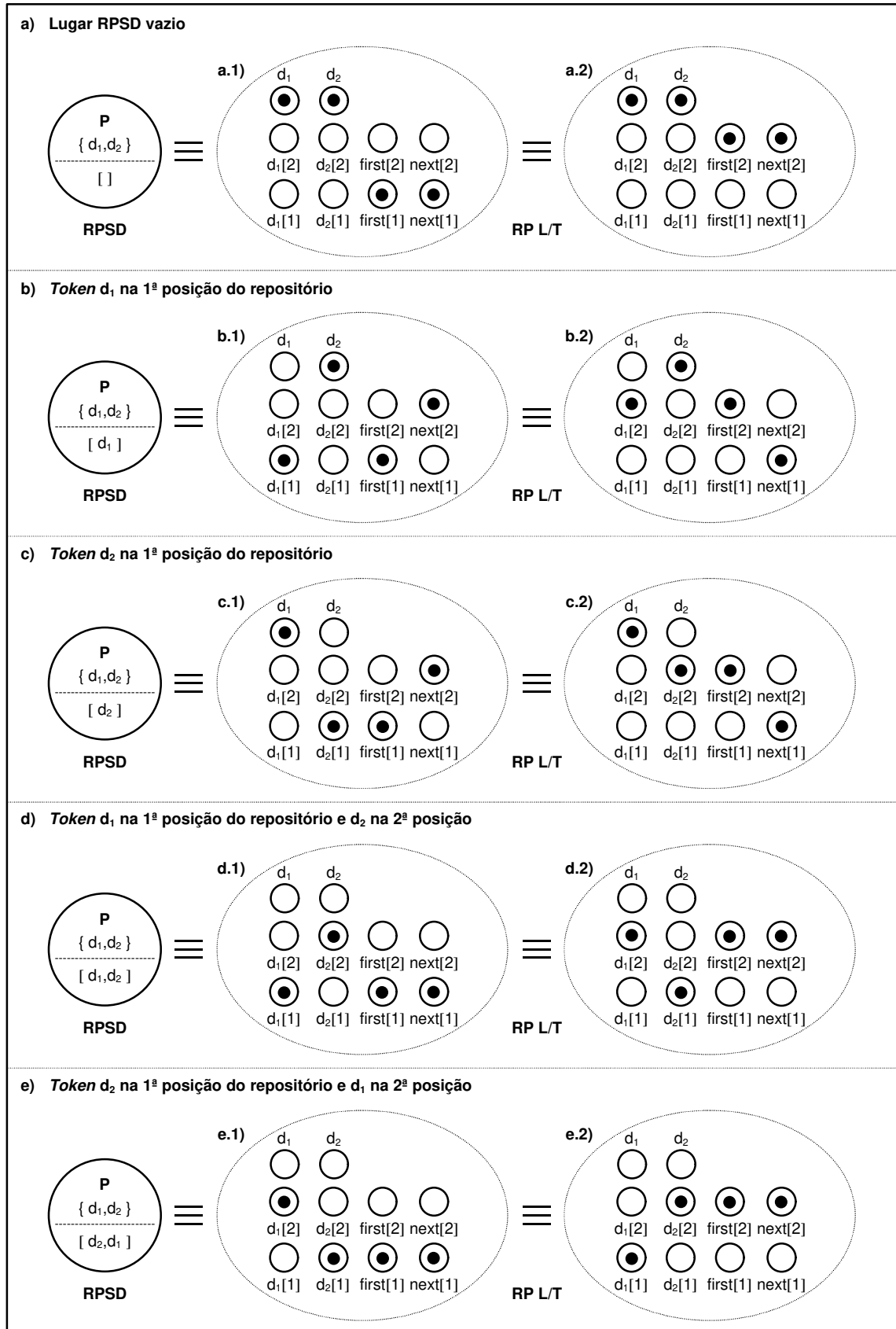


Figura 12.2. Correspondência entre lugar tipo I para 2 tokens e lugares de RP L/T.

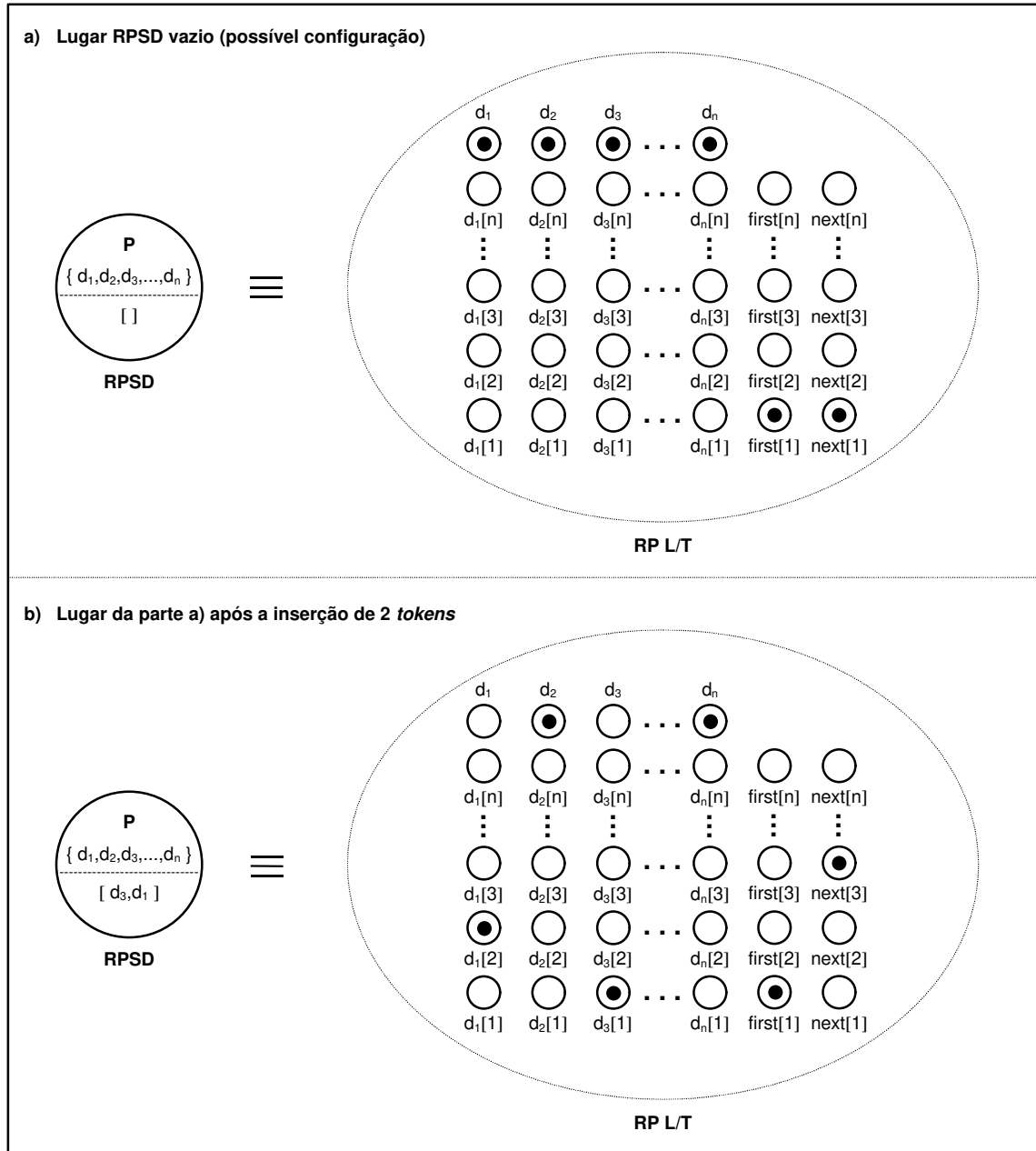


Figura 12.3. Correspondência entre lugar tipo I para n tokens e lugares de RP L/T.

Todas as marcações apresentadas na figura 12.2 podem ser alcançadas a partir de qualquer outra da figura.

As convenções adotadas para lugares de 2 tokens podem ser estendidas para lugares de n tokens (*domínio* de cardinalidade n). A rede L/T da figura 12.3 mostra um lugar P com *domínio* $\{d_1, d_2, \dots, d_n\}$, de cardinalidade n . Esta rede possui um **conjunto de lugares** assim distribuídos:

- lugares d_i , $1 \leq i \leq n$, usados para indicar se o *token* correspondente d_i está presente (lugar vazio) ou não (lugar com um *token* ●) no lugar P ;
- lugares $d_i[j]$, $1 \leq i \leq n$, $1 \leq j \leq n$, usados para indicar se o *token* d_i , caso presente no lugar P , ocupa a posição de índice j (e somente j) da estrutura **fila circular** representante de $R_S(P)$;

- lugares **first[k]**, $1 \leq k \leq n$, usados para indicar se o *token* mais antigo em **P** ocupa a posição de índice **k** (e somente **k**) da estrutura **fila circular** representante de $R_S(P)$;
- lugares **next[r]**, $1 \leq r \leq n$, usados para indicar se o próximo *token* a ser inserido em **P** ocupará a posição de índice **r** (e somente **r**) da **fila circular** representante de $R_S(P)$.

As transições conectadas à rede L/T da figura 12.3 deverão assegurar que os lugares assumam somente **marcações admissíveis**, definidas de acordo com as seguintes regras:

Regra I: Inserção de tokens. A inserção de um *token* d_i em **P** somente será possível se houver um *token* $\bullet$ no lugar d_i , $1 \leq i \leq n$. Neste caso, a **marcação corrente** da rede L/T será **modificada** realizando-se as seguintes **ações**:

- **remoção** do *token* $\bullet$ do lugar d_i ;
- **inserção** de um *token* $\bullet$ no lugar $d_i[j]$, onde **j** é o índice designado pelo ponteiro **next**, isto é, **next[j]** é o único dos lugares **next[r]**, $1 \leq r \leq n$, que possui um *token* $\bullet$;
- **atualização** do ponteiro **next**, através da **remoção** do *token* $\bullet$ do lugar **next[j]** e da **inserção** de um *token* $\bullet$ no lugar **next[j+1]**. Caso $(j+1) > n$, o *token* $\bullet$ será inserido no lugar **next[1]**. Isto decorre devido à **natureza circular** da **estrutura fila** representante de $R_S(P)$.

Regra II: Remoção de tokens. A remoção de um *token* d_i de **P** somente será possível se houver um *token* $\bullet$ em algum lugar $d_i[j]$, $1 \leq i \leq n$, $1 \leq j \leq n$. Há dois casos distintos, de acordo com a **expressão** do **ramo de entrada** associado à remoção (veja a descrição de ramos na seção 11.2.1.2):

Caso I: Remoção cronológica. Neste caso, a remoção do *token* d_i de **P** fará com que a **marcação corrente** da rede L/T seja **modificada** devido às seguintes **ações**:

- **remoção** do *token* $\bullet$ do lugar $d_i[j]$, onde **j** é o índice designado pelo ponteiro **first**, isto é, **first[j]** é o único dos lugares **first[k]**, $1 \leq k \leq n$, que possui um *token* $\bullet$;
- **inserção** de um *token* $\bullet$ no lugar d_i ;
- **atualização** do ponteiro **first**, através da **remoção** do *token* $\bullet$ do lugar **first[j]** e da **inserção** de um *token* $\bullet$ no lugar **first[j+1]**. Caso $(j+1) > n$, o *token* $\bullet$ será inserido no lugar **first[1]**. Isto decorre devido à estrutura de **fila circular** adotada para representar $R_S(P)$.

Caso II: Remoção aleatória. Neste caso, supondo-se que o *token* d_i ocupa a posição de índice **j** da estrutura **fila circular** (*token* $\bullet$ no lugar $d_i[j]$), a remoção deste *token* de **P** fará com que a **marcação corrente** da rede L/T seja **modificada** de acordo com uma das três situações seguintes:

Situação I: **first[j]** possui um *token* $\bullet$. Nesta situação, d_i ocupa a **primeira posição** do **repositório** $R_S(P)$. Assim, este caso de remoção aleatória equivale ao de remoção cronológica, seguindo-se, portanto, as etapas descritas em **Caso I**;

Situação II: **next[j]** possui um *token* $\bullet$ e **first[j]** está vazio. Nesta situação, d_i ocupa a **última posição** do **repositório** $R_S(P)$. Isto exige as seguintes etapas:

- **remoção** do *token* $\bullet$ do lugar $d_i[j]$;
- **inserção** de um *token* $\bullet$ no lugar d_i ;
- **atualização** do ponteiro **next**, através da **remoção** do *token* $\bullet$ do lugar **next[j]** e da **inserção** de um *token* $\bullet$ no lugar **next[j-1]**. Caso $(j-1) = 0$, o *token* $\bullet$ será inserido no lugar **next[n]**.

Situação III: $\text{first}[j]$ e $\text{next}[j]$ estão vazios. Nesta situação, d_i ocupa uma **posição intermediária** do **repositório** $R_S(P)$. Isto exige as seguintes etapas:

- **remoção** do *token* ● do lugar $d_i[j]$;
- **inserção** de um *token* ● no lugar d_i ;
- **deslocamento**, em uma posição para baixo, dos *tokens* presentes entre as posições $j+1$ e $r-1$, onde r indica o índice da próxima inserção na **fila circular**, isto é, existe um *token* ● no lugar $\text{next}[r]$, $1 \leq r \leq n$. Deve-se considerar a natureza circular da fila: se $(j+1) > n$, adota-se 1 como índice inferior, e se $(r-1) = 0$, adota-se n como índice superior;
- **atualização** do ponteiro **next**, através da **remoção** do *token* ● do lugar $\text{next}[r]$, $1 \leq r \leq n$, e da **inserção** de um *token* ● no lugar $\text{next}[r-1]$. Caso $(r-1) = 0$, o *token* ● será inserido no lugar $\text{next}[n]$.

A figura 12.3.b mostra um exemplo de **inserção** de 2 *tokens* (d_3 e d_1) a partir da marcação ilustrada na figura 12.3.a. Estes *tokens* podem ter sido inseridos um a um ou de uma só vez. No caso de inserção simultânea, a ordem de inserção no lugar corresponde à ordem em que os *tokens* são listados na **expressão** do ramo utilizado. Comparando-se a marcação resultante com a predecessora, observa-se que os lugares d_3 e d_1 ficaram vazios (denotando a inserção dos mesmos em P), ao passo que os lugares $d_3[1]$ e $d_1[2]$ ganharam um *token* cada (indicando que d_3 ocupa a primeira posição da estrutura **fila circular** e, d_1 , a segunda posição). Além disso, o lugar $\text{first}[1]$ permaneceu marcado (indicando que a primeira posição da **fila circular** é a primeira posição do **repositório**), e o *token* de $\text{next}[1]$ passou para $\text{next}[3]$ (denotando que a próxima inserção deverá ocupar a terceira posição da **fila circular**).

A **remoção** de *tokens* é ilustrada na figura 12.4, para um lugar P com **domínio** de cardinalidade 5. Nos exemplos desta figura, foram utilizados os procedimentos anteriormente descritos.

A presença de um *token* no lugar $\text{first}[3]$ da figura 12.4.a, denota que a fileira de lugares $d_i[3]$, $i = 1, \dots, 5$, possui o **primeiro** elemento do **repositório** de P (designado por $R_S(P)[1]$), qual seja, o associado ao lugar possuidor de um *token* (●). Neste caso, a presença de um *token* em $d_1[3]$ significa que $R_S(P)[1] = d_1$. De acordo com as convenções adotadas, para acessar índices crescentes no vetor $R_S(P)$, deve-se percorrer a estrutura de **fila circular** que representa tal vetor ($R_S(P)$) no sentido de baixo para cima. Desta forma, para $i = 1, \dots, 5$, a fileira de lugares $d_i[4]$ armazena $R_S(P)[2]$ e $d_i[5]$ armazena $R_S(P)[3]$. Devido à natureza circular da fila, a fileira seguinte à $d_i[5]$ é $d_i[1]$. Assim, a fileira de lugares $d_i[1]$ armazena $R_S(P)[4]$ e $d_i[2]$ armazena $R_S(P)[5]$. A partir destas conclusões e, da observação da marcação da rede na figura 12.4.a, tem-se que $R_S(P) = [d_1, d_2, d_3, d_4, d_5]$. Como P está cheio (com seus 5 *tokens*), o ponteiro **next** aponta para a mesma fileira de lugares que **first**, isto é, há um *token* em $\text{next}[3]$. Isto faz sentido, pois, para ocorrer uma inserção, será necessário primeiro acontecer uma remoção. Caso esta remoção seja cronológica, ela ocorrerá na fileira apontada por **first**, não implicando, portanto, na atualização do ponteiro **next** (pois este permanecerá na mesma fileira).

Tomando-se a marcação da rede L/T da figura 12.4.a como ponto de partida, a remoção do *token* presente na **primeira posição** do **repositório**, $R_S(P)[1]$, geraria a marcação presente na figura 12.4.b, de acordo com as ações descritas anteriormente, no **Caso I** e na **Situação I** do **Caso II**. Já a remoção do *token* presente na **última posição** do **repositório**, $R_S(P)[5]$, geraria a marcação da figura 12.4.c, de acordo com **Situação II** do **Caso II**. Por

último, a figura 12.4.d ilustra a marcação resultante da remoção de um *token* de uma **posição intermediária** do *repositório* (no caso $R_S(P)[2]$), aplicando-se as ações descritas na **Situação III** do **Caso II**, sobre a rede L/T da figura 12.4.a.

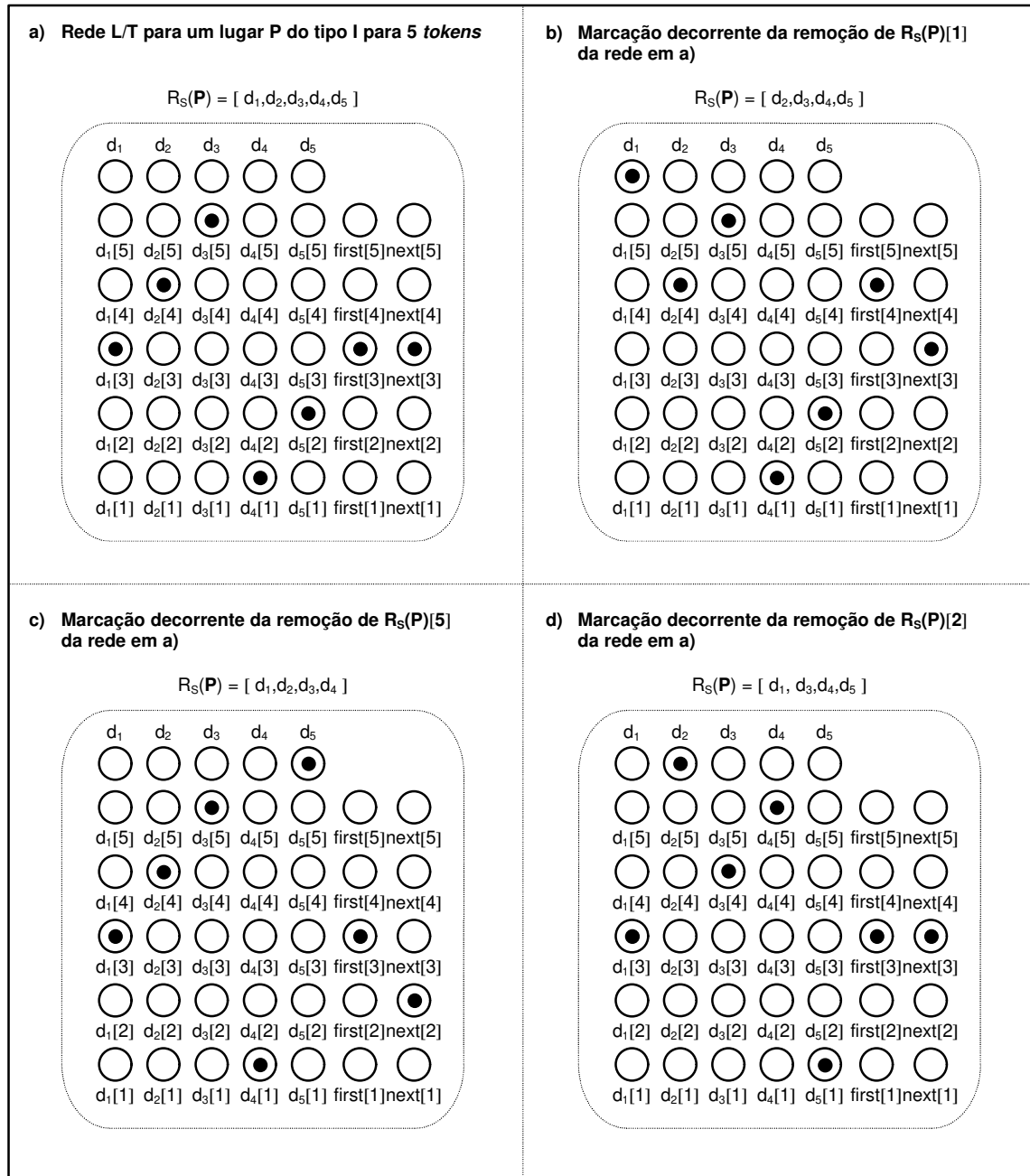


Figura 12.4. Efeito da remoção de *tokens* sobre a RP L/T associada a um lugar do tipo I.

Desta forma, conclui-se que lugares RPSD do tipo I podem ser representados por redes de Petri Lugar/Transição.

12.2.2. TIPO II: ARMAZENAMENTO

Lugares do tipo II são utilizados para representar **componentes de armazenamento** do sistema (tais como registradores e memórias), nos quais operações de *leitura/escrita* de dados (*tokens*) podem ser feitas.

A seção 11.2.1.1.2 mostrou a composição e os mecanismos de operação para lugares do tipo II. Desta forma, passa-se para sua representação em **redes de Petri Lugar/Transição (L/T)**, assim como foi feito para lugares do tipo I, na seção anterior.

As figuras 12.5 e 12.6 mostram como um lugar do tipo II pode ser representado por um **conjunto de lugares** de uma rede L/T. Ao contrário de lugares do tipo I, os de tipo II admitem **múltiplas cópias** de um mesmo *token*, desde que cada uma ocupe, naturalmente, uma **célula de armazenamento** distinta (entradas distintas no **vetor conteúdo**). Assim, há a necessidade somente de associar cada **célula de armazenamento** a um *token* do **domínio** do lugar. Na construção das redes L/T das figuras 12.5 e 12.6, adotou-se, por convenção, que a presença de um *token* $\bullet$ num lugar $d_i[j]$ indica que o *token* d_i encontra-se na j -ésima **célula de armazenamento** de **P** (as **células de armazenamento** de um lugar **P** do tipo II correspondem ao **vetor conteúdo** de **P**, representado por $C(P)$). Já a presença de um *token* $\bullet$, num lugar $dflt[j]$, indica que, a j -ésima **célula de armazenamento** de **P**, armazena o *token* correspondente ao valor *default* definido para o **domínio** D_M de **P**. Quando a classe **anotações** (A_n) da RPSD não explicita o valor *default* para um conjunto utilizado como **domínio**, assume-se como *default* o valor do primeiro elemento do conjunto, de acordo com a ordem estabelecida na enumeração do mesmo. Outra convenção adotada na construção das redes L/T das figuras 12.5 e 12.6, é a presença de um lugar **r/w**. Devido às **latências** de leitura e de escrita, em algumas ocasiões, o lugar de armazenamento estará **indisponível** para acesso, o que é representado pela **ausência** de um *token* em **r/w**. Por outro lado, quando **r/w** possuir um *token* $\bullet$, o lugar estará **disponível** para operações de leitura ou de escrita. O controle de lugares **r/w** é feito por **transições temporizadas** conectadas aos mesmos. Esta extensão, de redes de Petri L/T para redes de Petri **temporizadas** de **Merlin**, será tratada na seção 12.3, sem qualquer prejuízo à representação abordada nesta seção, uma vez que redes L/T constituem um subconjunto de redes temporizadas de **Merlin**, conforme descrito na seção 8.4.

Na figura 12.5, o lugar **P** possui diferentes configurações de **domínio**, para uma **única célula de armazenamento**. Na parte **a**) da figura, **P** tem **domínio** de cardinalidade 1, qual seja, $\{d_1\}$. A presença de um *token* no lugar $dflt[1]$ indica que a posição 1 do **vetor conteúdo**, referenciada pelo endereço a_0 do **espaço de endereçamento**, é ocupada pelo valor designado como *default* do **domínio**. Neste caso, como o **domínio** é unitário, o valor *default* é d_1 . Lugares $dflt[i]$ servem para os propósitos de **inicialização** e **reset** de **células de armazenamento**. Complementando o modelo da figura 12.5.a, a presença de um *token* em **r/w** indica que o lugar **P** está **disponível** para operações de **leitura** ou de **escrita**.

Comparando-se as partes **a**) e **b**) da figura 12.5, observa-se que o acréscimo de um elemento ao **domínio** do lugar **P** provoca o acréscimo de outro lugar ($d_2[1]$) à rede L/T. Na marcação da rede mostrada na parte **b**), a presença de um *token* em $d_2[1]$ indica que d_2 ocupa a primeira **célula de armazenamento** de **P** (**endereço** a_0), ao passo que, a presença de um *token* em **r/w**, indica a permissão de operações sobre **P**. A figura 12.5.c mostra a relação entre a cardinalidade do **domínio** $D_M(P)$ e os lugares da rede L/T utilizada para representar **P**: há um lugar $d_i[1]$, $i = 1, \dots, k$, para cada *token* d_i pertencente ao **domínio** $D_M(P)$.

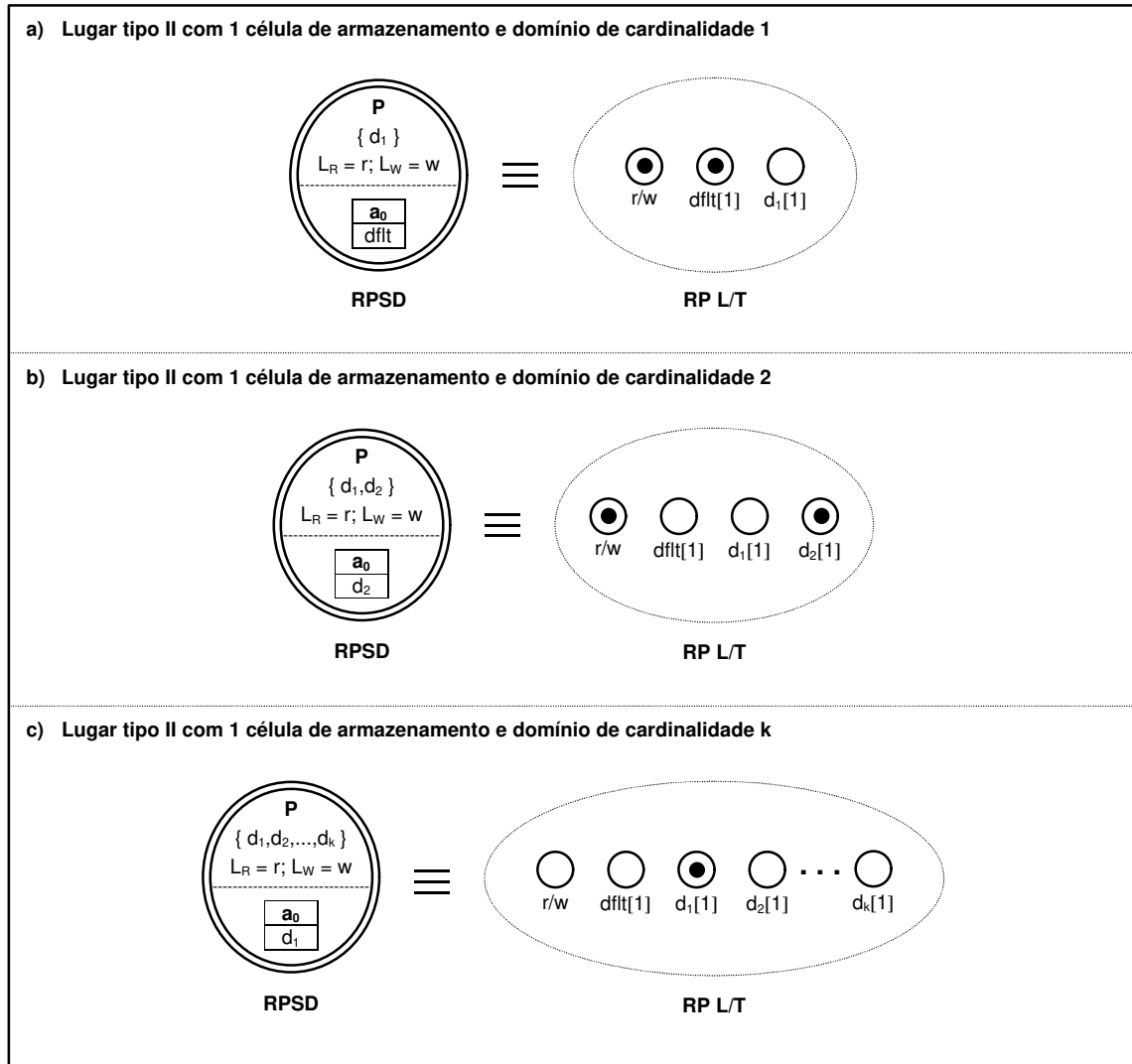
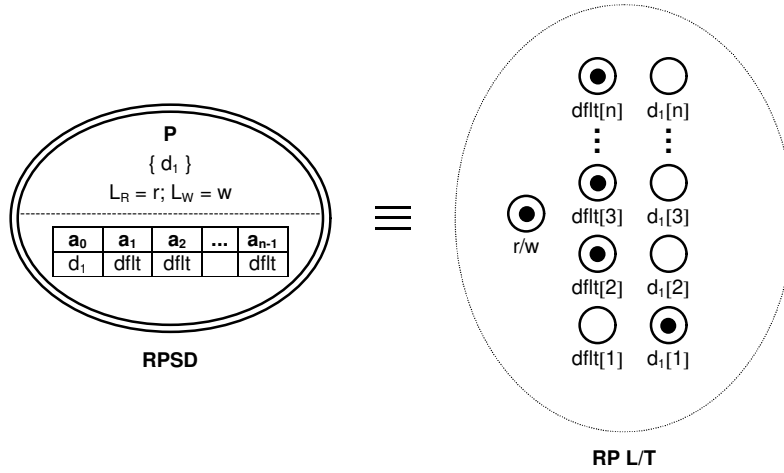


Figura 12.5. Correspondência entre lugares tipo II para 1 token e lugares de RP L/T.

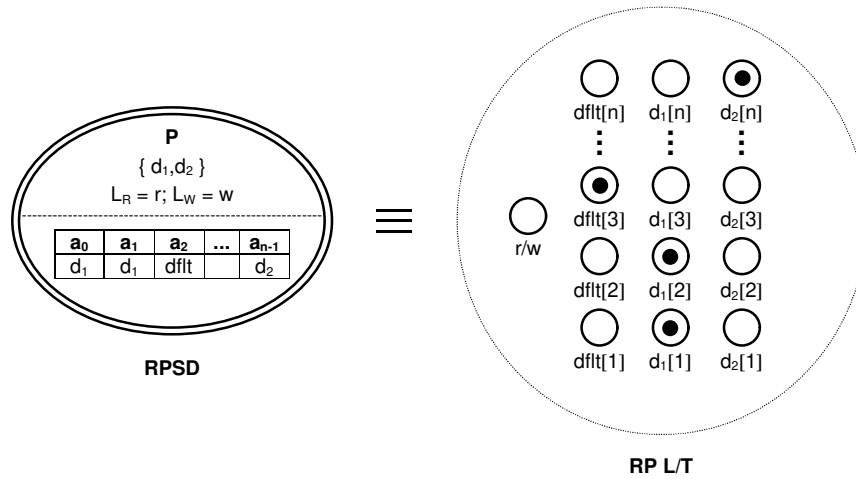
As redes L/T da figura 12.6 foram construídas a partir das redes da figura 12.5. Assim, as redes das partes a), b) e c) da figura 12.6 são, respectivamente, extensões das redes das partes a), b) e c) da figura 12.5. Observa-se que, o acréscimo de uma *célula de armazenamento* a_j , $j = 0, \dots, n-1$, ao *espaço de endereçamento* $A(P)$, implica no acréscimo de um conjunto de lugares $dflt[j]$ e $d_i[j]$, $i = 1, \dots, k$, à rede L/T.

Através das convenções adotadas, conclui-se que lugares RPSD do tipo II podem ser representados por redes de Petri Lugar/Transição.

a) Lugar tipo II com n células de armazenamento e domínio de cardinalidade 1



b) Lugar tipo II com n células de armazenamento e domínio de cardinalidade 2



c) Lugar tipo II com n células de armazenamento e domínio de cardinalidade k

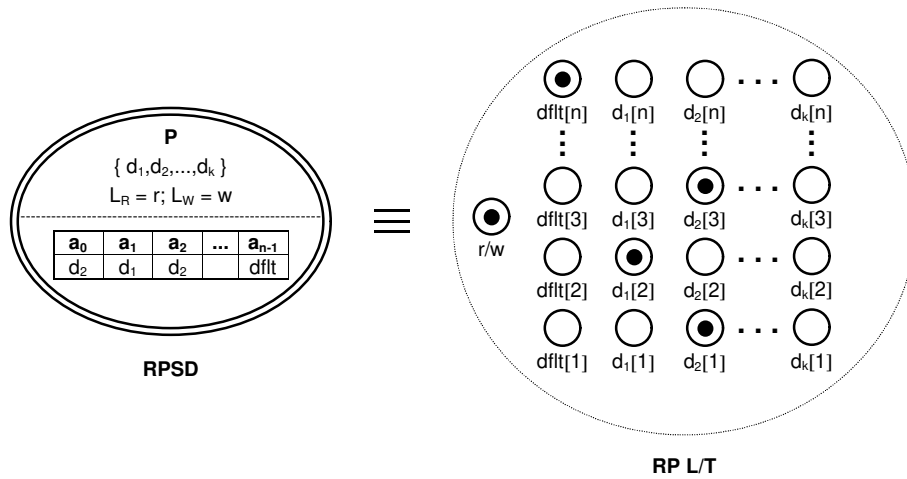


Figura 12.6. Correspondência entre lugares tipo II para n tokens e lugares de RP L/T.

12.2.3. TIPO III: FUNCIONAL

Um lugar funcional é utilizado para indicar que um ou mais *samples de dados* (representados por *listas ordenadas de tokens*) encontram-se em processamento por um ou mais **componentes funcionais de mesmo tipo** (cuja funcionalidade é descrita pelo parâmetro *comportamento* associado ao lugar).

A seção 11.2.1.1.3 mostrou a definição e os mecanismos de funcionamento embutidos em lugares funcionais. Deste modo, passa-se para a representação dos mesmos em **redes de Petri Lugar/Transição (L/T)**, assim como foi feito para lugares do tipo I e II, nas seções anteriores.

Na construção das redes L/T equivalentes a lugares funcionais, vamos, inicialmente, tratar da representação dos *repositórios de entrada e de saída*.

Os **parâmetros de entrada** de um lugar funcional são definidos através do seu *repositório de entrada R_I* . Independentemente do *comportamento* associado ao lugar ser descrito por uma *rotina* ou por uma *subrede* (RPSD-T), cada **parâmetro de entrada** pode ser representado por um lugar RPSD do tipo I, com *domínio D_S* igual ao *tipo* do **parâmetro**, e com *repositório R_S* fixado em *max_S* posições, onde *max_S* é o *número máximo de samples* que o lugar funcional admite concorrentemente. De forma análoga, os **parâmetros de saída** de um lugar funcional são definidos através do seu *repositório de saída R_O* . Cada **parâmetro de saída** pode ser representado por um lugar RPSD do tipo I, com *domínio D_S* igual ao *tipo* do **parâmetro**, e com *repositório R_S* fixado em *max_S* posições. Desta forma, a representação dos *repositórios* de um lugar funcional, através de redes L/T, é semelhante à representação dada para lugares do tipo I, na seção 12.2.1.

Começando com um exemplo simples, a figura 12.7 mostra os lugares de uma rede L/T, correspondentes aos *repositórios de entrada e de saída* de um lugar funcional **P**. Ambos *repositórios*, R_I e R_O , têm cardinalidade 1, isto é, possuem, cada qual, um único *tipo*: **X** para R_I , e **Z** para R_O . Isto quer dizer que, um **argumento** para o **parâmetro de entrada** de **P** deverá ser um *token* do *tipo X* (*tokens x_1, x_2 ou x_3*), ao passo que, o **resultado** gerado por uma das *max_S* *instâncias* do *comportamento B* de **P**, deverá ser um *token* do *tipo Z* (*z_1 ou z_2*). O valor *max_S(P)* varia nas partes **a)**, **b)** e **c)** da figura 12.7. Para cada **combinação** (*token de um tipo, posição numa célula de repositório*) define-se um **par de lugares** na rede L/T, **mutuamente exclusivos** (somente um dos lugares do par possuirá um *token* ● por vez). Este é o caso, por exemplo, do par de lugares $x_3[2,1]$ e $Nx_3[2,1]$ da rede L/T da figura 12.7.b, no qual, a presença de um *token* ● em $x_3[2,1]$, significa que o *token x_3 está armazenado* na posição [2,1] do *repositório* correspondente (no caso, R_I); por outro lado, a presença de um *token* ● em $Nx_3[2,1]$, significa que o *token x_3 não está armazenado* na posição [2,1] do repositório. Conforme será mostrado adiante, esta dualidade é importante para o mecanismo de deslocamento representante da função $L(B_i)$, $1 \leq i \leq max_S$ (veja a seção 11.2.1.1.3), ao manter **constante** o número total de *tokens* em cada uma das fileiras de lugares representantes, na rede L/T, das linhas dos repositórios. A **inserção** de um *token*, numa dada posição existente [i,j] de um *repositório de entrada*, só será possível se a posição estiver **vazia**, o que é indicado pela presença de um *token* ● no lugar **ins[i,j]** associado ao *repositório* (para cada lugar **ins[i,j]** existe o dual **Nins[i,j]**). Quando uma coluna **j** da matriz R_I está **totalmente** preenchida, **inserções** são temporariamente suspensas, o que é indicado pela presença de um *token* ● no lugar **full[j]** associado a R_I (obs.: os *repositórios* exemplificados na figura 12.7 apresentam uma única coluna, isto é, **j** é sempre 1); a cada lugar **full[j]** associa-se o dual **Nfull[j]**. Lugares

funcionais permitem a **remoção** somente de *samples completos* de *tokens*. Assim, é necessário que todas as colunas de uma determinada linha **i** da matriz R_O estejam preenchidas, para que os *tokens* da linha possam ser simultaneamente removidos, liberando a linha **i** do **repositório de entrada** para **novas inserções**. Quando duas ou mais linhas encontram-se preenchidas, os resultados presentes na linha de menor índice (os índices são crescentes de baixo para cima, na representação gráfica) terão prioridade de remoção. Neste contexto, foram definidos os lugares **rem** (e os duais **Nrem**): a presença de um *token* ● no lugar **rem[i]** indica que a linha **i** de R_O é a próxima a ter os *tokens* (resultados) removidos. Somente um dos lugares **rem** poderá possuir *token*, qual seja, o de menor índice. Desta forma, a transferência de *token* de um lugar **Nrem[i]** para **rem[i]** poderá acarretar, concomitantemente, na transferência de *token* de um lugar **rem[j]** para **Nrem[j]**, caso **rem[j]** possua *token* e **j** seja maior que **i**.

Na marcação da rede L/T da figura 12.7.a, é mostrado o instante de **inserção** do **argumento** (*token*) x_3 no **repositório de entrada** R_I de **P**. O *token* ● de $Nx_3[1]$ foi removido, e um *token* ● foi inserido em $x_3[1,1]$, denotando que x_3 ocupa a linha 1, coluna 1, da matriz representante de R_I , isto é, $R_I(P)[1,1] = x_3$. Observa-se que a presença de um *token* em **full[1]** (proveniente de **Nfull[1]**), indica que a coluna 1 de R_I , referente ao primeiro (e único) **parâmetro de entrada** de **P**, está cheia. Como **P** possui um único parâmetro de entrada, o *token* presente em $x_3[1,1]$ será imediatamente transferido para a instância B_1 (não ilustrada na figura) do **comportamento** $B(P)$ associado ao lugar, iniciando a execução da mesma; os *tokens* presentes nos demais lugares representantes da linha 1 de R_I serão removidos, deixando tal linha **indisponível** para inserções durante a execução da instância B_1 . Supondo-se que $B(P)$ mapeie os *tokens* x_1 e x_3 a z_1 , e o *token* x_2 a z_2 , a instância B_1 gerará o resultado z_1 , o qual será armazenado na posição [1,1] da matriz representante de R_O , o que será indicado, na parte da rede correspondente a R_O , pela presença de um *token* no lugar $z_1[1,1]$ (proveniente de $Nz_1[1,1]$). Ao término da execução da instância B_1 , o *token* presente em **Nrem[1]** será transferido para **rem[1]**, assinalando, para as transições que removam *tokens* de **P**, que a próxima remoção deverá ocorrer na linha 1 (a única, no caso). Quando tal remoção ocorrer, a linha 1 de R_I se tornará **disponível** novamente para inserções, através do restabelecimento da marcação *default* aos lugares representantes da linha (inserção de *token* ● nos lugares $Nx_1[1,1]$, $Nx_2[1,1]$ e $Nx_3[1,1]$; um *token* ● será inserido no lugar **ins[1,1]**, uma vez que R_I possui uma única linha – caso contrário, seria inserido em **Nins[1,1]**). Em relação ao controle de coluna, o *token* ● presente em **full[1]** será transferido para **Nfull[1]**.

O lugar funcional **P** da figura 12.7.b aceita dois *samples* de entrada, isto é, admite a **execução simultânea** de **duas instâncias** do **comportamento** $B(P)$, cada qual **ativada** pela presença de um **argumento** (*token*) do **tipo X** em uma das linhas do **repositório de entrada**. Na figura, a porção da rede L/T referente ao **repositório de entrada** $R_I(P)$ mostra que o mesmo encontra-se **cheio** (lugar **full[1]** com *token* e demais lugares vazios), isto é, suas células de armazenamento estão **indisponíveis**, o que é representado, em RPSD, pela presença de **u** no **repositório de entrada** do lugar. Em relação ao repositório de saída $R_O(P)$, a rede L/T mostra que a instância B_2 , associada à linha 2 ($L(B_2) = 2$), já gerou um resultado, o que pode ser constatado pela presença de um *token* em um dos lugares **positivos** (aqueles cujo rótulo não se inicia por N) associados à posição [2,1] de $R_O(P)$, mais especificamente, $z_1[2,1]$. A presença de um *token* no lugar **rem[2]** permite que o *token* z_1 , gerado pela instância B_2 , possa ser removido por alguma transição de saída conectada a **P**. A instância B_1 , associada à linha 1 ($L(B_1) = 1$), ainda está em execução (processamento), pois os lugares positivos da rede L/T designados para armazenar o resultado gerado por esta instância, quais sejam, lugares $z_1[1,1]$ e $z_2[1,1]$, estão **vazios**.

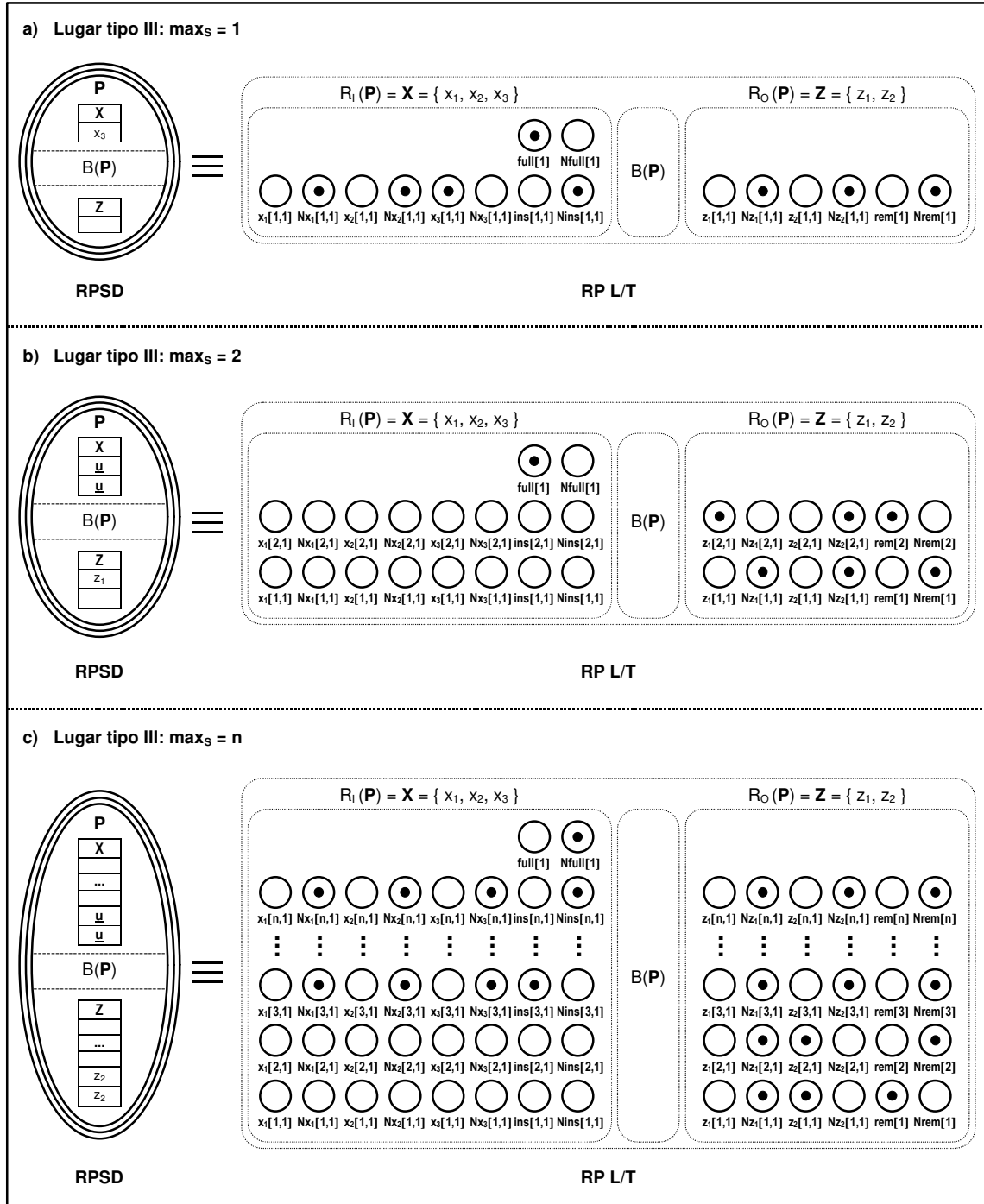


Figura 12.7. Representação dos repositórios de um lugar funcional através de RP L/T.

A figura 12.7.c apresenta um lugar funcional **P** para n *samples* de entrada, de tamanho 1. Assim, **P** admite a **execução simultânea** de até n instâncias de seu **comportamento**, cada uma delas ativada quando da presença de um **argumento** do **tipo X** em uma das linhas do **repositório de entrada**. Seguindo as convenções adotadas para as partes a) e b) da mesma figura, pode-se concluir que, pela marcação presente na figura 12.7.c, duas instâncias de **B(P)** foram **ativadas** e geraram **resultados** iguais (z_2), os quais foram **armazenados** nas duas primeiras linhas de **R_O(P)**. O resultado (*token*) presente na linha 1 de **R_O(P)**, tem **prioridade** de **remoção** sobre o resultado na linha 2, uma vez que o resultado da linha 1 foi gerado a

partir de um argumento mais **antigo** (*sample* mais antigo). Além disso, observa-se que o próximo argumento aceito por **P** deverá ser inserido na terceira linha de $R_I(P)$.

A figura 12.8 mostra um lugar funcional **P** para processamento simultâneo de até **5 samples**, com os *samples de entrada* compostos por **3 argumentos** e, os de *saída*, formados por **2 resultados**. Os *samples de entrada* são constituídos pela tripla ordenada de *tokens* (i_1, i_2, i_3) , onde i_1 e i_3 devem ser do **tipo X** e, i_2 , do **tipo Y**. Já os *samples de saída* são formados pelo par ordenado de *tokens* (o_1, o_2) , onde o_1 deve ser um *token* do **tipo Y** e, o_2 , do **tipo Z**. A classe **anotações** (A_n), da RPSD portadora de **P**, fornece a definição dos **tipos (conjuntos) X, Y e Z**. Portanto, ao contrário dos lugares funcionais da figura 12.7, o lugar **P** da figura 12.8 possui *repositórios de entrada e de saída* com mais de um **tipo**. A representação dos *repositórios* na figura 12.8 é uma extensão da representação adotada na figura 12.7.

Quando o *repositório de entrada* R_I apresenta dois ou mais componentes associados a um mesmo **tipo** (como é o caso da figura 12.8), a **inserção** de *tokens* em R_I deverá respeitar o mecanismo, segundo o qual, *tokens* de mesmo **tipo** devem ser "espalhados" por igual, nas colunas que os admitem, na matriz representante de R_I (veja a descrição da *operação de inserção de tokens*, na seção 11.2.1.1.3). Neste contexto, os lugares **ins** (na parte da rede L/T referente a R_I) são usados para indicar a **próxima posição de inserção**, para *tokens* de um mesmo **tipo**. Exemplificando, na marcação da rede L/T da figura 12.8, a presença de um *token* ● no lugar **ins**[3,3] indica que o próximo *token* de **tipo X** (**tipo** associado à coluna 3), a ser inserido em R_I , deverá ocupar a posição [3,3] deste *repositório*. Supondo que tal inserção ocorra, **ins**[3,3] ficará **vazio** e um *token* ● será inserido em **ins**[4,1], indicando que, a próxima inserção de um *token* de **tipo X** se dará na posição [4,1] de R_I . Observe que **não** há relação alguma, entre as indicações de posição de inserção, para *tokens* de **tipos diferentes**. Isto reflete o funcionamento definido para lugares do tipo III, nos quais argumentos para parâmetros de entrada não precisam ser inseridos no lugar de uma só vez. Na rede da figura 12.8, o argumento y_1 , de **tipo Y**, foi inserido em R_I antes que um argumento de **tipo X** fosse inserido na posição [3,3]. Os lugares duais **Nins** assumirão marcação contrária à dos lugares **ins**, excetuando-se, conforme explicado anteriormente, os casos nos quais a instância correspondente encontra-se em execução (todos os lugares da linha associada vazios).

Na figura 12.8, a marcação da rede L/T, referente ao *repositório de entrada* $R_I(P)$, mostra que as duas primeiras fileiras de lugares, de baixo para cima, não possuem *tokens*, denotando que instâncias de **B** (designadas pela função dinâmica **L**) estão processando, ou já processaram, *samples* correntemente associados a tais linhas. Ou seja, as duas primeiras linhas de R_I estão **indisponíveis** (o que é representado, no lugar RPSD, pelo símbolo u). A marcação mostra, também, que $R_I(P)[3,1] = x_1$ (*token* em $x_1[3,1]$), $R_I(P)[3,2] = y_3$ (*token* em $y_3[3,2]$) e $R_I(P)[4,2] = y_1$ (*token* em $y_1[4,2]$). Quanto às inserções, o próximo *token* de **tipo X** deverá ser inserido em $R_I(P)[3,3]$ (*token* em **ins**[3,3]), e o próximo de **tipo Y**, em $R_I(P)[5,2]$ (*token* em **ins**[5,2]). Em relação ao repositório de saída $R_O(P)$, a marcação da rede mostra que a instância B_i , $1 \leq i \leq 5$, associada à linha 1 pela função dinâmica **L** ($L(B_i) = 1$), gerou o resultado referente ao segundo **tipo** (coluna) de $R_O(P)$, o que é indicado pela presença de um *token* em $z_1[1,2]$. Entretanto, o *token* z_1 não poderá ser removido de R_O , até que o resultado referente ao primeiro **tipo** (coluna) de $R_O(P)$ seja gerado. Caso isto ocorra, o *token* presente em **Nrem**[1] será transferido para **rem**[1], liberando os resultados da linha 1 para remoção, de uma só vez, pelas transições de saída conectadas a **P**. Este é o caso da linha 2 de R_O , a qual armazena os *tokens* y_2 e z_2 , disponíveis para remoção simultânea (lugar **rem**[2] com *token*). A figura 12.9 mostra o efeito da remoção deste *sample de saída* sobre a marcação de **P**.

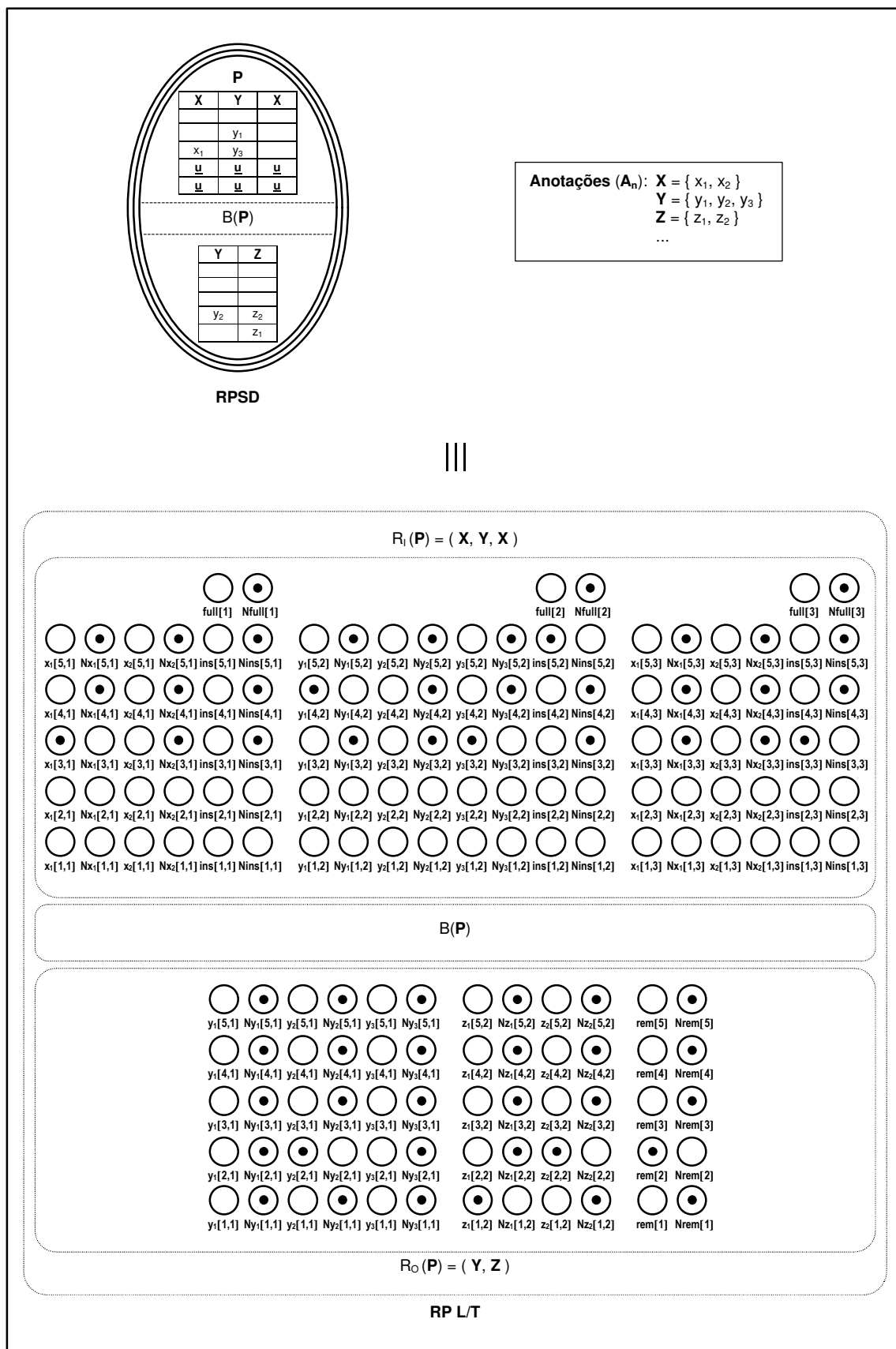


Figura 12.8. Lugar funcional com 3 parâmetros de entrada e 2 de saída.

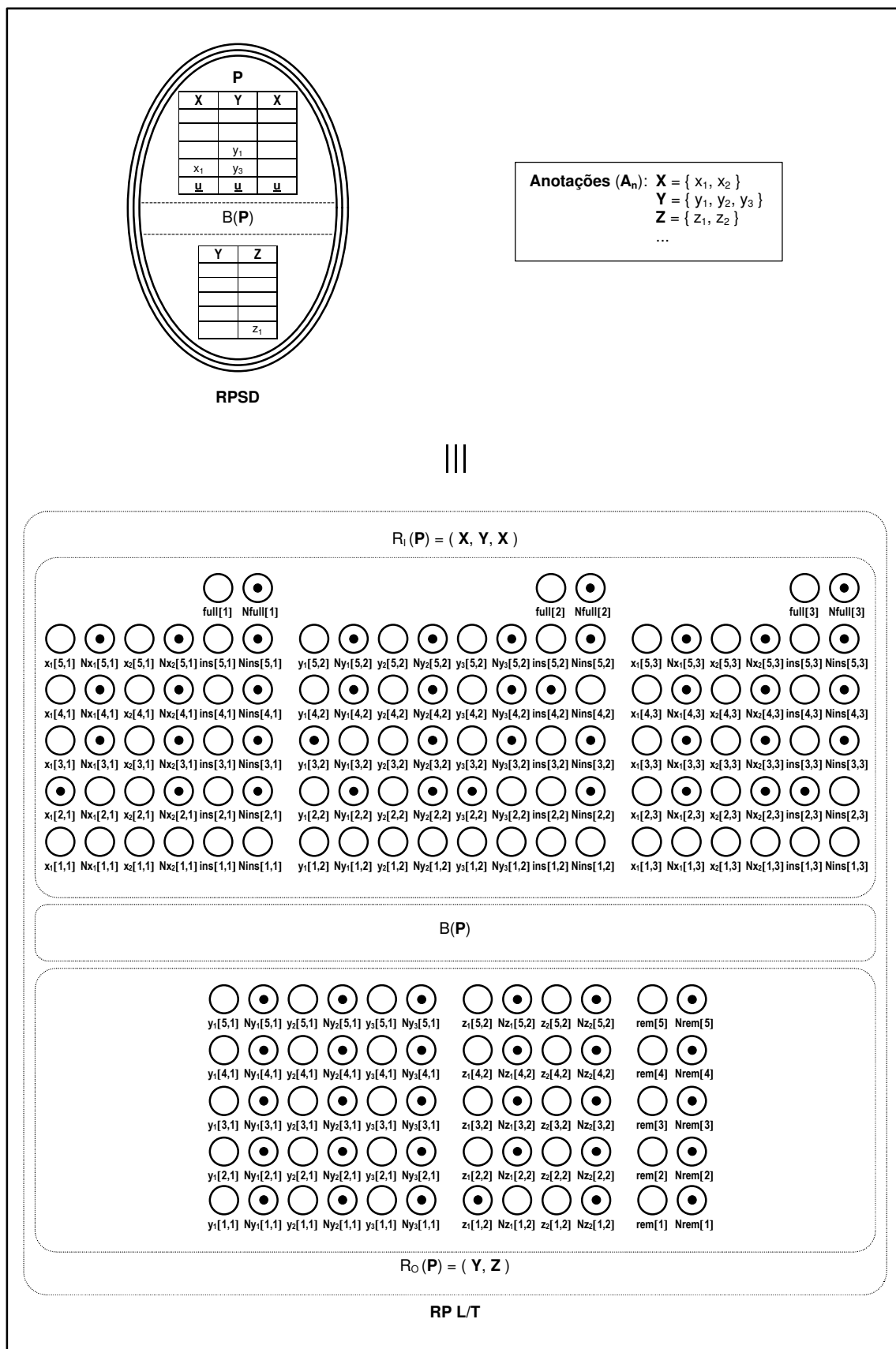


Figura 12.9. Lugar da figura 12.8 após a remoção do *sample* presente na linha 2 de R_o .

A remoção do *sample de tokens* (y_2, z_2) de $R_O(P)$, a partir da marcação ilustrada na figura 12.8, fez com que todos os *tokens* situados acima da linha 2 de $R_I(P)$ (tanto na representação em RPSD como na representação em rede L/T) fossem deslocados uma posição para baixo, resultando na marcação mostrada na figura 12.9. O mesmo deslocamento ocorreu em $R_O(P)$, com a linha 2 deste repositório recebendo os *tokens* presentes na linha 3, a linha 3 recebendo os da linha 4 e esta recebendo os *tokens* da linha 5, o que é visualizado apenas na rede L/T, uma vez que na RPSD as linhas situadas acima da segunda encontram-se vazias. Observe que, na rede L/T da figura 12.9, em ambos os repositórios, a última linha (a 5) assume a marcação *default*, a qual é caracterizada pela presença de um *token* ● em cada um dos lugares **negativos** (aqueles com rótulo iniciado por N). Na figura 12.8, se qualquer uma das colunas do repositório $R_I(P)$ estivesse cheia (*token* ● em $full[1]$, $full[2]$ ou $full[3]$), a mesma ficaria não cheia na marcação da figura 12.9 (transferência do *token* ● de $full[i]$ para $Nfull[i]$, $1 \leq i \leq 3$). Finalmente, caso existissem instâncias associadas a linhas de índice superior a 2, a função dinâmica L seria recalculada, para manter a correta correspondência entre os *samples* que entram e saem do lugar, de forma que, dados dois *samples* quaisquer, o mais antigo dos dois sempre estará associado a uma linha de índice menor. Este mecanismo interno de RPSD foi explicado na *operação de remoção de tokens*, na seção 11.2.1.1.3.

As convenções adotadas na representação do lugar funcional da figura 12.8, através de uma rede L/T, podem ser estendidas, sem modificação alguma, na representação de lugares funcionais com quaisquer configurações de *número máximo de samples*, *comportamentos* e *repositórios de entrada e de saída*.

Tratadas as representações dos *repositórios de entrada e de saída*, através de lugares de uma rede L/T, passa-se para a representação de *comportamentos*.

Até este ponto do texto, por uma questão de simplificação, os lugares RPSD foram representados através de conjuntos de lugares de redes de Petri Lugar/Transição. Contudo, como RPSD é um modelo temporizado, baseado nas *redes de Petri temporizadas de Merlin* [42,45] (RPTM – seção 8.4), e, como redes L/T são um **subconjunto** de RPTM (veja definição abaixo), **adotaremos o modelo RPTM em todas as representações de RPSD, dadas deste ponto em diante**, sem qualquer prejuízo para as representações já descritas, pois os lugares de uma rede L/T podem ser diretamente incorporados por uma RPTM. A representação de *comportamentos* em RPTM é necessária, pois, quando todos os **argumentos** de um lugar funcional encontram-se presentes numa linha l do *repositório de entrada*, a ativação da **instância** relacionada a l deve ser **imediata**, o que não seria possível de representar através de redes L/T. Por outro lado, RPTM possibilita tal representação, uma vez que se trata de uma extensão **fortemente temporizada** [43], ou seja, modelos RPTM podem **forçar o disparo** de transições.

Conforme visto na seção 8.4, RPTM é um par $(RP\ L/T, E)$, onde:

- $RP\ L/T$ é uma rede de Petri Lugar/Transição;
- E é uma função que associa um *intervalo fechado* de números racionais não-negativos $[a_t, b_t]$, $a_t \leq b_t$, para cada transição t da rede de Petri associada, chamado **intervalo de disparo estático** de t .

As **transições** de uma RPTM são **habilitadas** da mesma forma que nas redes L/T (definição 7.2 – capítulo 7). O disparo de uma transição t ocorre somente dentro do intervalo $[a_t, b_t]$, contado a partir do instante em que t torna-se habilitada. Do mesmo modo que nas

redes L/T, o **disparo** de uma transição é instantâneo e provoca o mesmo **efeito** (definição 7.3 – capítulo 7). Um caso especial é o intervalo de disparo $[0,0]$, o qual implica no **disparo imediato** da transição, tão logo ela torne-se habilitada.

Conforme exposto na seção 11.2.1.1.3, o **comportamento** B de um lugar funcional P , indicado por $B(P)$, pode ser expresso através de uma **rotina**, segundo a sintaxe da linguagem de programação adotada para o lugar, ou por uma **RPSD-Transformacional (RPSD-T)**, a qual constitui-se numa RPSD, com restrições em relação aos lugares associados aos parâmetros de entrada e de saída do **comportamento**.

A representação de uma **rotina** através de redes RPTM pode ser feita de diversas formas, em **diferentes níveis** de **detalhamento**. Contudo, para o propósito deste capítulo, qual seja, demonstrar a correspondência entre RPSD e outras extensões de redes de Petri, adotaremos uma representação de pouco detalhamento, sem qualquer prejuízo para o propósito estabelecido. Detalhamentos pormenorizados e otimizados podem ser adotados, fixando-se uma determinada linguagem de programação. Isto seria interessante na construção de ferramentas de síntese para RPSD, fugindo do escopo ora proposto.

Supondo-se que o **comportamento** $B(P)$ de um lugar funcional seja descrito por uma rotina que faça a soma de dois bits, gerando como resultados o valor desta soma e o vai-um, a figura 12.10 mostra como tal rotina poderia ser representada por uma RPTM.

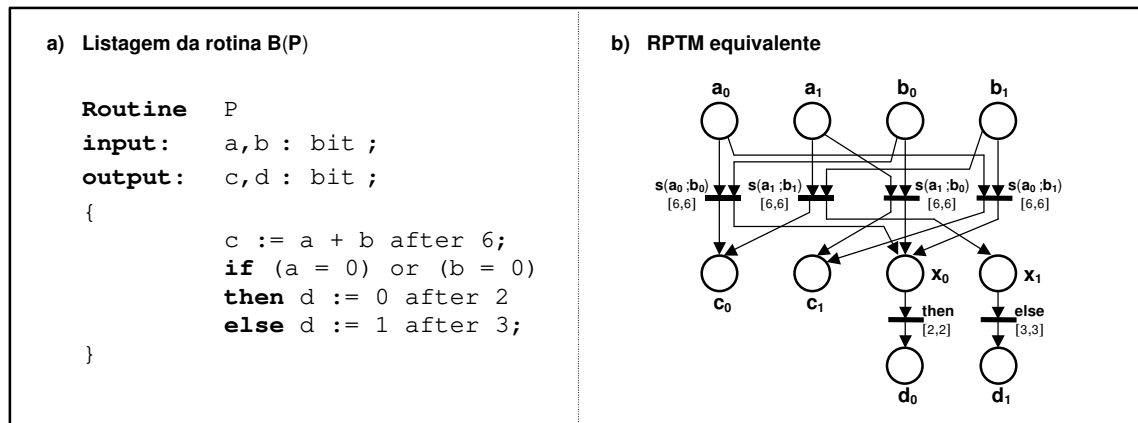


Figura 12.10. Representação de uma rotina através de uma RPTM.

Na rede da figura 12.10.b, a presença de um *token* ● em um lugar de índice 0 (zero), significa que a variável correspondente armazena o bit 0, enquanto a presença de um *token* ● em um lugar de índice 1, corresponde ao armazenamento do bit 1. A soma é efetivada através do disparo de uma, e somente uma, das transições com rótulo iniciado por s , após o transcorrimto de 6 unidades de tempo. Para representar a latência da geração do resultado correspondente ao vai-um (0 ou 1), os lugares x_0 e x_1 armazenam tal resultado, o qual somente é disponibilizado após o disparo da transição correspondente (**then**, consumidora de 2 unidades de tempo, ou **else**, consumidora de 3 unidades de tempo). Exemplificando, suponha que, ao executar a rotina, a variável a assumo o valor 1 e a variável b o valor 0. Desta forma, ocorrerá a inserção de um *token* ● nos lugares a_1 e b_0 , promovendo a habilitação da transição $s(a_1;b_0)$, a qual irá disparar após 6 unidades de tempo. O disparo de $s(a_1;b_0)$ promoverá a inserção de um *token* ● nos lugares c_1 (significando que o resultado da soma é o bit 1) e x_0 (significando que o vai-um é o bit 0), habilitando, assim, a transição **then**, a qual disparará, disponibilizando o resultado do vai-um (lugar d_0), após 2 unidades de tempo.

A figura 12.11 mostra como as representações do **comportamento** $B(P)$, da figura 12.10, são agregadas a um lugar funcional para **3 samples**.

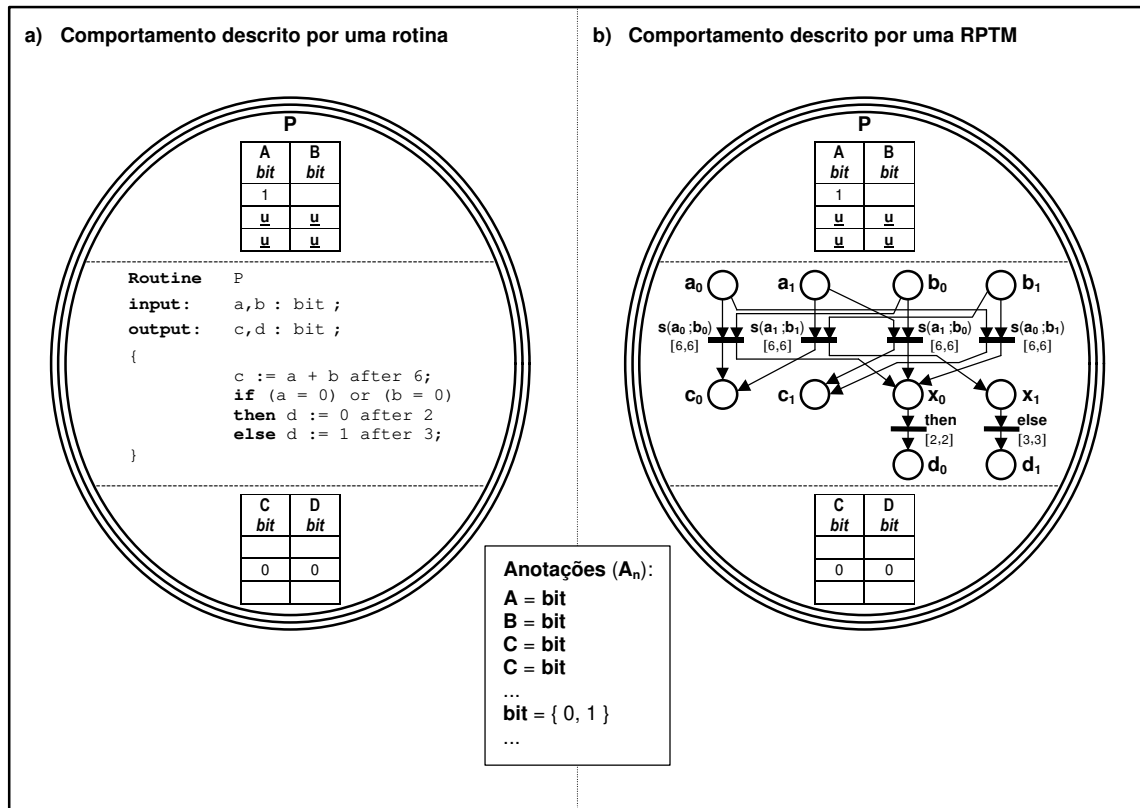
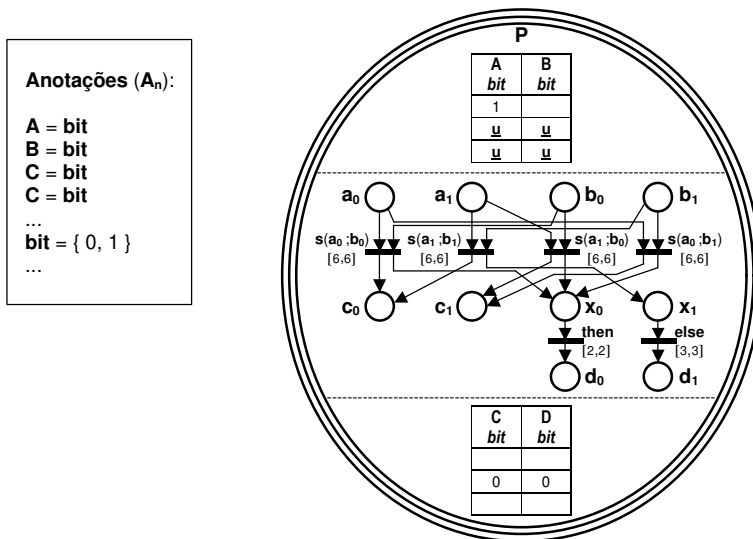


Figura 12.11. Representações equivalentes para um lugar funcional.

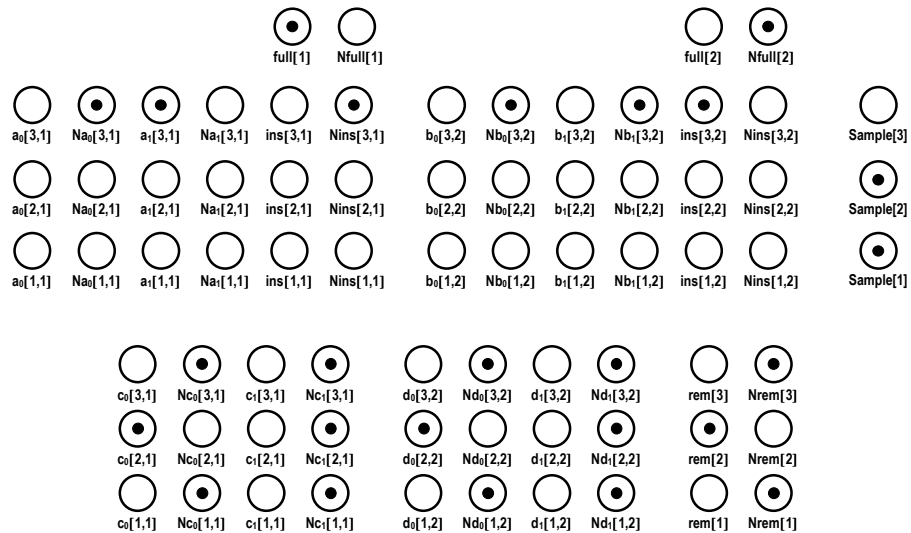
A figura 12.12 mostra a representação completa, em RPTM, do lugar funcional **P** da figura 12.11. Na figura 12.12.a, observa-se que os **repositórios de entrada** e **de saída** de **P** têm cardinalidade 2 (dois parâmetros de entrada e dois resultados de saída), seu comportamento é descrito pela RPTM representante da **rotina** de soma (descrita anteriormente), e seu **número máximo de samples** (max_S) é 3 (representando a possibilidade de até 3 somas simultâneas). De acordo com o **repositório de entrada**, das três **instâncias** do comportamento, apenas uma está **disponível** (a terceira linha do repositório, de baixo para cima, possui somente um **token**, associado ao parâmetro de entrada **A**); as outras duas instâncias estão **indisponíveis**: uma está em execução e a outra já disponibilizou seus resultados no **repositório de saída** (na segunda linha), os quais ainda não foram removidos de **P**.

Na figura 12.12.b, a marcação dos lugares RPTM reflete a configuração dos repositórios de **P**, de acordo com as convenções adotadas anteriormente. Observa-se, nesta figura, a presença de três novos lugares: **Sample[1]**, **Sample[2]** e **Sample[3]**. Estes lugares são utilizados por algumas transições da RPTM, para verificar se a linha correspondente do **repositório de entrada** está disponível ou não. Trata-se de um subterfúgio, uma vez que RPTM não fornece instrumentos para verificar se um lugar está vazio ou não. Desta forma, quando um **sample** (conjunto de **tokens** representando argumentos) é removido do conjunto de lugares representantes da **i**-ésima linha do **repositório de entrada**, o lugar **Sample[i]** receberá um **token** ●, denotando que tal linha encontra-se **indisponível**.

a) Lugar funcional – comportamento descrito por uma rotina representada por uma RPTM



b) RPTM equivalente: repositórios de entrada e de saída



c) RPTM equivalente: mecanismo de associação *sample*/instância

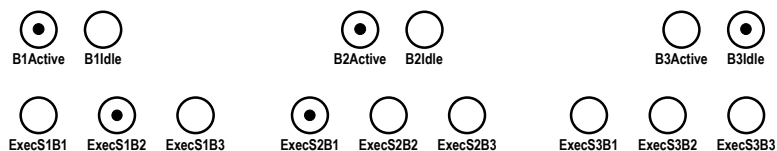
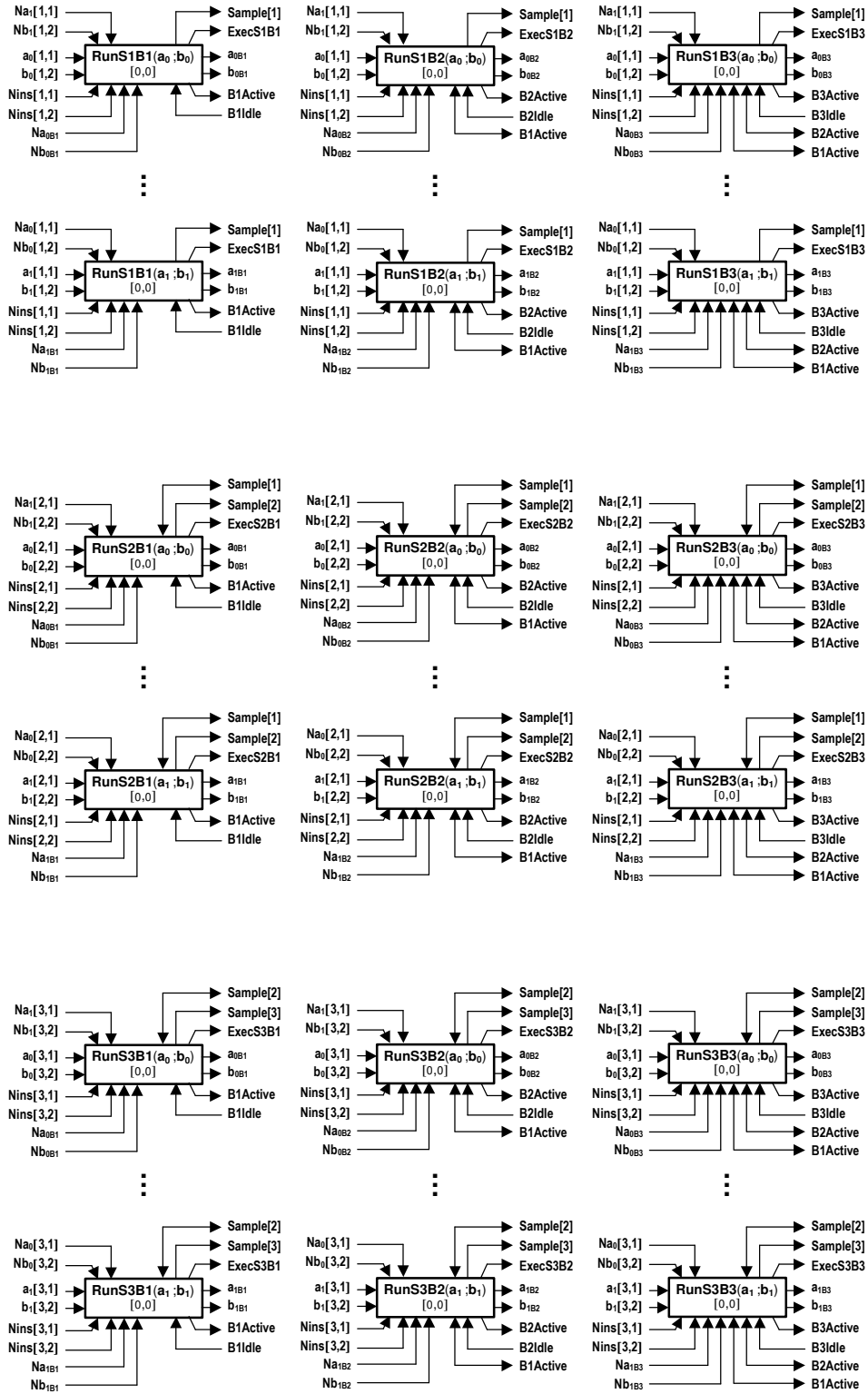


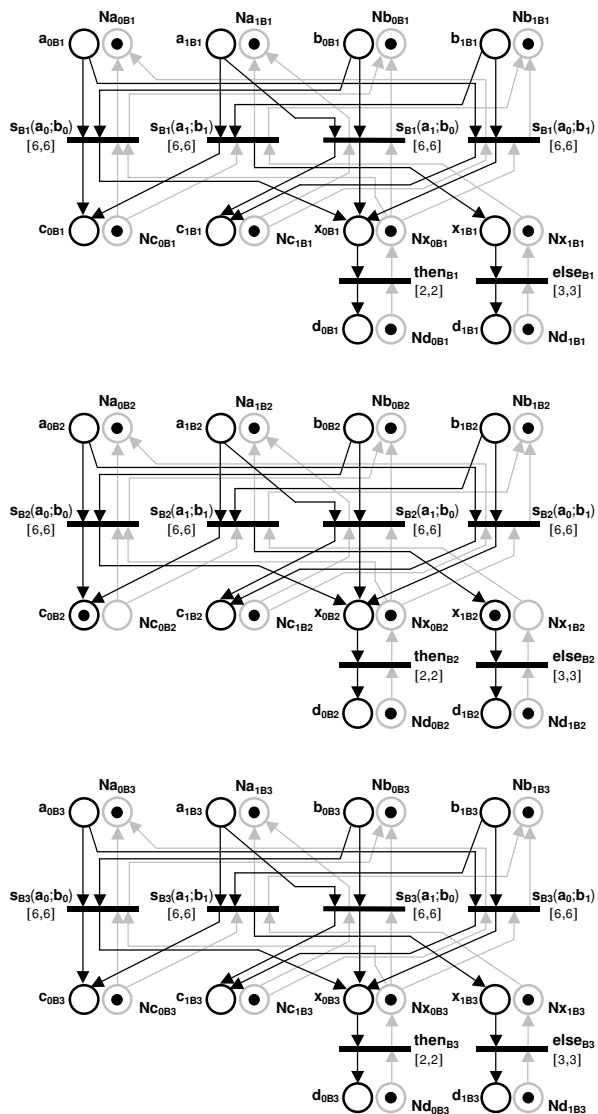
Figura 12.12. Representação completa de um lugar funcional.

c) cont.

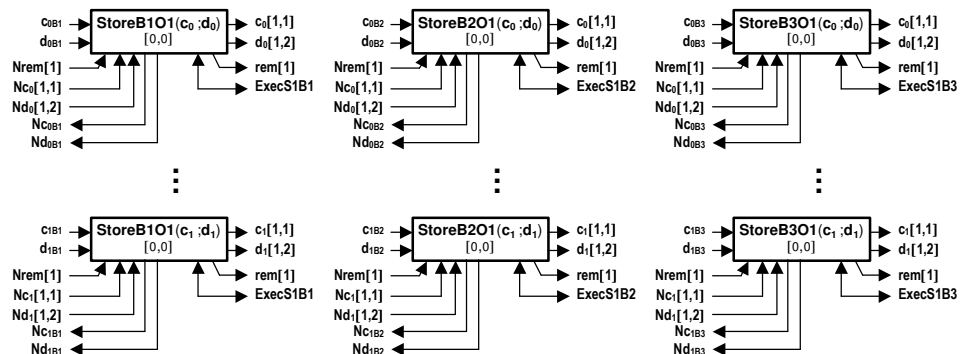


Continuação da figura 12.12.

d) RPTM equivalente: instâncias do comportamento

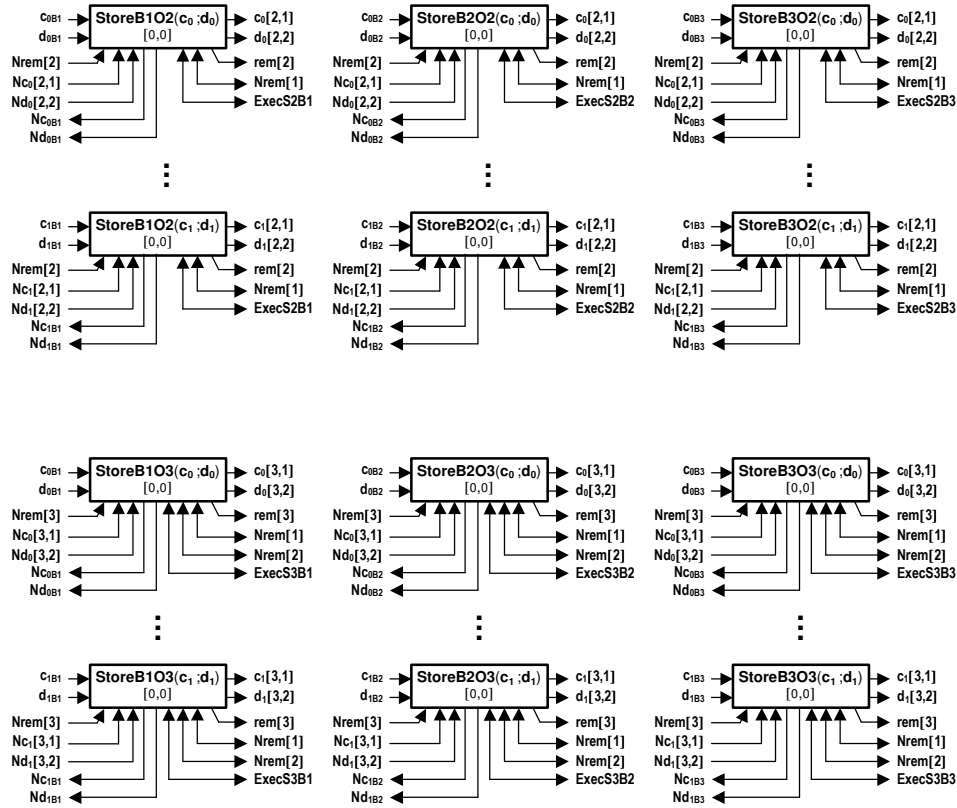


e) RPTM equivalente: mecanismo de transferência *sample* transformado/repositório de saída



Continuação da figura 12.12.

e) cont.



Continuação da figura 12.12.

As partes **c)** e **d)** da figura 12.12 estão diretamente relacionadas. A figura 12.12.c apresenta os lugares e transições componentes do mecanismo que associa, a cada *sample* removido do *repositório de entrada*, uma determinada *instância*, enquanto a figura 12.12.d mostra as três instâncias do comportamento, denominadas B_1 , B_2 e B_3 . Uma linha do *repositório de entrada* não fica permanentemente associada a uma mesma instância: quando um *sample* é removido do *repositório de entrada*, ele deverá ser associado à instância disponível de **menor índice**. Na figura 12.12.c, os lugares $B_iActive$ e B_iIdle ($i = 1, 2, 3$) são duais, isto é, quando $B_iActive$ estiver marcado, B_iIdle estará vazio e, vice-versa. Assim, observa-se que as instâncias B_1 e B_2 estão **ativas** (lugares $B_1Active$ e $B_2Active$ marcados), enquanto a instância B_3 está **inativa**. Os lugares $ExecS_jB_i$ ($i, j = 1, 2, 3$) indicam, para cada *sample* S_j , qual instância B_i o está executando. Isto se faz necessário, uma vez que a associação entre as linhas do *repositório de entrada* e as do *repositório de saída* não é direta. Por exemplo, suponha que os *samples* S_1 e S_2 estejam respectivamente associados às instâncias B_1 e B_2 (isto é, lugares $ExecS_1B_1$ e $ExecS_2B_2$ marcados). Quando a instância B_1 terminar sua execução, ela armazenará seus resultados na primeira linha do *repositório de saída*. Por sua vez, quando a instância B_2 terminar sua execução, poderá ocorrer o armazenamento de seus resultados na segunda ou na primeira linha do *repositório de saída*: o primeiro caso (segunda linha) ocorrerá se os dados produzidos por B_1 não tiverem sido removidos, ainda; o segundo caso (primeira linha) ocorrerá se os dados produzidos por B_1 já tiverem sido removidos, o que acarretará na mudança de denominação do *sample* S_2 para S_1 , transferindo o *token* $\bullet$ de $ExecS_2B_2$ para $ExecS_1B_2$. Este mecanismo, denominado *controle*

de deslocamento da função L , será descrito adiante, quando da representação RPTM de ramos coletores. Voltando à figura 12.12.c, o *sample* S_1 (o mais antigo) está associado à instância B_2 (lugar $ExecS_1B_2$ marcado), enquanto o *sample* S_2 (mais recente) está associado à instância B_1 (lugar $ExecS_2B_1$ marcado). As transições de associação entre *samples* e instâncias – $RunS_jB_i(a_x;b_y)$ ($i, j = 1, 2, 3$ e $x, y = 0, 1$) – têm as seguintes funções:

- Remoção dos *tokens* de um *sample* completo do **repositório de entrada**: Os índices x e y podem assumir os valores 0 ou 1. Denotando o complemento de x (y) por Nx (Ny), o disparo de $RunS_jB_i(a_x;b_y)$ removerá os *tokens* dos lugares $a_x[j,1]$, $b_y[j,2]$, $Na_{Nx}[j,1]$, $Nb_{Ny}[j,2]$, $Nins[j,1]$ e $Nins[j,2]$, deixando vazios os lugares correspondentes à j -ésima linha do **repositório de entrada**;
- Seleção da instância para recepção dos dados do *sample* removido: Tal instância corresponde a de menor índice, dentre as inativas. Desta forma, as transições dividem-se em três grupos, de acordo com a instância selecionada (B_1 , B_2 ou B_3). A habilitação de uma transição que seleciona a instância B_1 dependerá do lugar B_1Idle estar marcado. Seu disparo promoverá a remoção do *token* presente neste lugar e a inserção de um *token* em $B_1Active$. Já a habilitação de uma transição que seleciona a instância B_2 dependerá dos lugares B_2Idle e $B_1Active$ estarem marcados (observe a **seta dupla** para $B_1Active$, a qual denota um **ramo de realimentação**, isto é, um ramo que remove e insere o *token* no mesmo lugar, constituindo-se, desta forma, num ramo verificador da presença de *token* no lugar relacionado). Isto significa que a instância B_2 só poderá ser selecionada quando ela estiver inativa e a instância B_1 estiver ativa. Analogamente, a habilitação de uma transição que seleciona a instância B_3 dependerá da mesma estar inativa (*token* em B_3Idle) e das outras duas estarem ativas (*tokens* em $B_1Active$ e $B_2Active$). Além das condições de habilitação expostas, há uma outra condição, referente ao *sample* que será removido, uma vez que mais de um *sample* poderá estar completo, aguardando pela remoção. Neste caso, o *sample* que ocupa a linha de menor índice (os índices são crescentes de baixo para cima), no **repositório de entrada**, terá prioridade de remoção sobre os demais. Assim, pela figura 12.12.c, pode-se observar que uma transição para a remoção de um *sample* presente na terceira linha, somente poderá ficar habilitada caso o *sample* da linha 2 já estiver associado a alguma instância, o que é denotado pela presença de *token* no lugar **Sample[2]** (note a seta dupla relacionando tal lugar às transições $RunS_3B_i(a_x;b_y)$). Do mesmo modo, a habilitação de uma transição para a remoção de *sample* da segunda linha (pertencente ao grupo de transições $RunS_2B_i(a_x;b_y)$), dependerá do lugar **Sample[1]** estar marcado. Por último, a habilitação de uma transição do grupo $RunS_1B_i(a_x;b_y)$, não dependerá da marcação dos lugares **Sample[j]**, $j = 1, 2, 3$;
- Ajuste da marcação dos lugares **Sample[j]**, $B_iActive$, B_iIdle e $ExecS_jB_i$: O disparo de uma transição $RunS_jB_i(a_x;b_y)$ deverá inserir um *token* ● nos lugares **Sample[j]**, $B_iActive$ e $ExecS_jB_i$, além de remover o *token* do lugar B_iIdle , de forma a refletir a associação feita entre o *sample* S_j e a instância B_i ; e
- Inicialização da instância escolhida: A figura 12.12.d mostra que os lugares pertencentes às instâncias possuem, cada qual, um dual. Esta dualidade é necessária para a restauração do estado inicial destas redes, uma vez que o lugar funcional poderá estar conectado a uma transição através de um ramo preemptivo. Além disso, podem existir redes descritoras de comportamento que não esvaziem todos os seus lugares positivos, quando da passagem de resultados para o **repositório de saída** do lugar funcional, obrigando o restauro do estado inicial da instância (este restauro é mostrado em outros exemplos, mais adiante). Desta forma, a inicialização de uma instância por uma transição $RunS_jB_i(a_x;b_y)$ envolve, além da inserção de *token* nos lugares a_{xB_i} e b_{yB_i} , a remoção do *token* presente nos duais Na_{xB_i} e Nb_{yB_i} .

A figura 12.12.e mostra o mecanismo de transferência de resultados, produzidos pelas instâncias, para o **repositório de saída**. As transições de transferência de resultados – $\text{StoreB}_i\text{O}_j(\mathbf{c}_x;\mathbf{d}_y)$ ($i, j = 1, 2, 3$ e $x, y = 0, 1$) – têm as seguintes funções:

- Remoção de *tokens* da instância terminada: Denominando de **terminal** o lugar que não possui transições de saída (considerando-se somente as transições de uma instância), o término de uma instância é sinalizado quando, para cada **tipo** (conjunto de *tokens* que podem ser inseridos numa coluna) do **repositório de saída**, existir um lugar terminal marcado, representante de um elemento do tipo considerado. Nas instâncias da figura 12.12.d, tais lugares são $\mathbf{c}_{0B_i}$ e $\mathbf{c}_{1B_i}$, para o tipo **C**, e $\mathbf{d}_{0B_i}$ e $\mathbf{d}_{1B_i}$, para o tipo **D**, $i = 1, 2, 3$. Assim, o disparo da transição $\text{StoreB}_i\text{O}_j(\mathbf{c}_x;\mathbf{d}_y)$ promoverá a remoção dos *tokens* presentes nos lugares $\mathbf{c}_{xB_i}$ e $\mathbf{d}_{yB_i}$, além da inserção de um *token* em cada um de seus duais, quais sejam, $\mathbf{Nc}_{xB_i}$ e $\mathbf{Nd}_{yB_i}$; e
- Alteração na marcação dos lugares do **repositório de saída**, correspondentes à linha receptora da *sample de saída* (*sample* transformado, processado): O disparo da transição $\text{StoreB}_i\text{O}_j(\mathbf{c}_x;\mathbf{d}_y)$ promoverá a transferência do *token* ● dos lugares $\mathbf{Nc}_x[j,1]$ e $\mathbf{Nd}_y[j,2]$ para $\mathbf{c}_x[j,1]$ e $\mathbf{d}_y[j,2]$, refletindo o armazenamento dos resultados produzidos pela instância B_i , além da transferência do *token* ● do lugar $\mathbf{Nrem}[j]$ para $\mathbf{rem}[j]$, sinalizando, para as transições RPSD que tenham o lugar **P** como lugar de entrada, que a j -ésima linha do **repositório de saída** de **P** está **disponível** para remoção (obs.: as transições que removem *samples* transformados de **P** selecionarão a linha disponível de **menor índice**, conforme será visto adiante);

Outras condições para a habilitação de uma transição $\text{StoreB}_i\text{O}_j(\mathbf{c}_x;\mathbf{d}_y)$ são (observe as setas duplas):

- Lugar $\text{ExecS}_j\mathbf{B}_i$ marcado: Esta condição denota que o *sample* S_j , associado à j -ésima linha do **repositório de entrada**, executado pela instância B_i , deverá ser armazenado na j -ésima linha do **repositório de saída**;
- Lugar $\mathbf{Nrem}[1]$ marcado: Esta condição é utilizada somente se a transição pertencer ao grupo de transições $\text{StoreB}_i\text{O}_2(\mathbf{c}_x;\mathbf{d}_y)$. Isto significa que a transferência de resultados, para a segunda linha do **repositório de saída**, somente poderá ocorrer se a primeira linha não estiver disponível para remoção. O contrário é permitido, isto é, a transferência de resultados para a primeira linha do repositório, mesmo que a segunda linha esteja disponível para remoção; e
- Lugares $\mathbf{Nrem}[1]$ e $\mathbf{Nrem}[2]$ marcados: Esta condição é utilizada somente se a transição pertencer ao grupo de transições $\text{StoreB}_i\text{O}_3(\mathbf{c}_x;\mathbf{d}_y)$. Isto significa que a transferência de resultados, para a terceira linha do **repositório de saída**, somente poderá ocorrer se a primeira e a segunda linhas não estiverem disponíveis para remoção. O contrário é permitido, isto é, a transferência de resultados para a primeira ou para a segunda linha do repositório, mesmo que a terceira linha esteja disponível para remoção.

Resumidamente, a figura 12.12 mostra um lugar funcional **P** com **repositórios de entrada e de saída** de cardinalidade 2. O **número máximo de samples (max_s)** de um lugar funcional define a quantidade de **instâncias** (cópias) do **comportamento B** associado ao lugar. Assim, na representação do lugar funcional **P**, a RPTM representante da **rotina** associada a **P** é replicada 3 vezes (parte **d** da figura). Para que os *tokens* (**argumentos**) sejam admitidos por uma das **instâncias** da **rotina**, foram utilizadas transições $\text{RunS}_j\mathbf{B}_i(\mathbf{a}_x;\mathbf{b}_y)$, de disparo imediato, uma para cada combinação possível (e ordenada) envolvendo os *tokens* dos **tipos** presentes no **repositório de entrada R_I**. Tão logo os dois argumentos encontrem-se presentes numa linha de **R_I**, ambos são removidos, acionando a instância associada à linha portadora de tais *tokens*.

Em relação ao **repositório de saída** R_O , não há a necessidade de inserção simultânea de *tokens* (resultados) na estrutura representante deste **repositório**, tal como foi especificado para as transições $\text{Store}B_iO_j(c_x;d_y)$, as quais poderiam ter sido divididas em um conjunto maior de transições, obtendo-se, no entanto, o mesmo resultado. Por outro lado, transições RPSD que tenham um lugar funcional como entrada, só poderão remover ***samples completos de tokens***, isto é, **linhas inteiras** da matriz R_O . Para tal, existe um mecanismo de controle que libera os *tokens* de uma linha de R_O , tão logo a linha fique preenchida (lugares **rem[j]**, **j** = 1, 2, 3). O controle de prioridade de remoção é estabelecido pelas transições de saída do lugar funcional.

O funcionamento da RPTM representante de um lugar funcional ficará mais claro nos exemplos da próxima seção, envolvendo lugares deste tipo.

Todas as convenções adotadas na representação, em RPTM, de **comportamentos** descritos por **rotinas**, podem ser estendidas a **comportamentos** descritos por **RPSD-Transformacionais**, uma vez que estas são constituídas dos mesmos tipos de lugares de RPSD, bastando, portanto, substituir seus lugares RPSD pelas representações de lugares apresentadas neste capítulo (em rede L/T e RPTM) e, suas transições, pelas representações tratadas na sequência.

Destarte, pelas convenções de representação adotadas nesta seção, pode-se concluir que lugares funcionais de RPSD podem ser representados por extensões de redes de Petri de baixo nível (redes de Petri Lugar/Transição e redes de Petri Temporizadas de Merlin).

12.3. EQUIVALÊNCIA DE TRANSIÇÕES E RAMOS

Transições representam **eventos** que podem ocorrer no **sistema** modelado. Diz-se que uma transição está **habilitada** quando o **evento** que ela representa **pode ocorrer** no instante considerado, o que é constatado através de certas **regras** e **condições**. O **disparo** de uma transição habilitada corresponde à **ocorrência** do evento que ela representa. De acordo com os lugares envolvidos, o evento pode promover:

- A **inserção** ou a **remoção** de **tokens** representantes de componentes do sistema, em lugares do **tipo I**;
- A **leitura** ou a **atualização** (escrita) de **células de armazenamento**, em lugares do **tipo II**;
- A **inserção** de **argumentos** para os **parâmetros de entrada** necessários à ativação de **comportamentos** associados a lugares do **tipo III**, ou a **retirada** de **resultados** do processamento desses **comportamentos**.

Transições podem ser utilizadas na modelagem de **exceções**, nas quais os eventos podem ter como conseqüências:

- A **remoção** de **todos** os **tokens** representantes de componentes do sistema, em lugares do **tipo I**;
- A **limpeza** (**reset**) de **todas** as **células de armazenamento**, em lugares do **tipo II**;
- A **suspensão temporária** ou a **retomada** de **comportamentos** associados a lugares do **tipo III**;
- A **suspensão definitiva** (**preempção**) de **comportamentos** associados a lugares do **tipo III**, com a **remoção** de **todos** os **dados** (**tokens**) do lugar, inclusive das subredes existentes.

A seguir, na representação de transições RPSD através de redes de Petri Temporizadas de Merlin (RPTM), os *modelos de equivalência* são separados de acordo com o *tipo de lugar* e o *tipo de conexão* (determinado pelo ramo). O apêndice II apresenta a *simulação* destes modelos de equivalência. Ramos e transições foram tratados nas seções 11.2.1.2 e 11.2.1.3.

12.3.1. CONEXÕES COM LUGARES DO TIPO I

Antes de fazermos a representação de equivalência, em RPTM, de transições e ramos RPSD conectados a lugares do tipo I, vejamos as possíveis conexões envolvendo tais componentes.

Transições podem *remover tokens* de lugares do tipo I através de dois tipos de *ramos*, quais sejam:

- a) **Ramo de entrada comum**: Um ramo deste tipo permite que uma transição t remova um conjunto de dados (*tokens*) de um lugar P do tipo I. Este ramo é rotulado com uma tripla, denominada *expressão de remoção*, dada por $E(P,t) = \langle \min, \max, \text{ordem} \rangle$, onde *min* e *max* denotam, respectivamente, o número **mínimo** e **máximo** de *tokens* a retirar do lugar P , e $\text{ordem} \in \{R, C\}$ denota se os *tokens* serão retirados seguindo-se um critério **não-determinístico** (representado por R , de ordem randômica) ou **determinístico** (de acordo com o ordem de inserção no lugar – princípio FIFO, representado por C , de ordem cronológica). Caso o rótulo seja omitido, assume-se $E(P,t) = \langle 1, 1, R \rangle$. É representado por uma linha contínua;
- b) **Ramos de entrada conjugado**: Um ramo deste tipo permite que uma transição t remova um **mesmo conjunto de dados** (*tokens*) de um grupo de lugares P_i , $i = 1, 2, \dots, n$ ($n \geq 2$), do tipo I. Um ramo desse tipo retira, aleatoriamente, um mesmo conjunto de *tokens* de cada um dos lugares ligados às suas origens, razão pela qual é rotulado por uma *expressão de remoção* $E(P_i,t) = \langle \min, \max \rangle$, para algum i entre 1 e n , denotando, respectivamente, o número mínimo e máximo de *tokens* a retirar de cada um dos lugares relacionados. Caso o rótulo seja omitido, assume-se $E(P_i,t) = \langle 1, 1 \rangle$. É representado por uma linha contínua.

Em relação à *inserção* de *tokens* em lugares do tipo I, transições devem utilizar o chamado **ramo de saída comum**, o qual permite que uma transição t **adicione** um conjunto de dados (*tokens*) a um lugar P do tipo I. Um ramo desse tipo insere um conjunto de *tokens* no lugar quando do disparo da transição, de acordo com uma *função de inserção* $I(P,t)$, a qual é composta por um conjunto finito de *alternativas de pares*, cada qual formado por uma *condição booleana* e um *conjunto de tokens*, seguindo a formatação:

$$I(P,t) = \langle \text{boolean}_1, \text{set}_1 \rangle | \langle \text{boolean}_2, \text{set}_2 \rangle | \dots | \langle \text{boolean}_n, \text{set}_n \rangle, n \geq 0.$$

A *expressão booleana verdadeira* de **menor índice** – boolean_i – é responsável pela inserção do conjunto correspondente (set_i), desde que não existam *tokens* em comum entre o conjunto a ser inserido e o *repositório* (R_s) do lugar em foco. Caso ausente, a *função de inserção* corresponde ao conjunto de *tokens* retirados dos lugares ligados à transição considerada por *ramos de entrada comuns, conjugados, coletores ou de leitura*. Também neste caso, uma das condições de habilitação da transição relacionada é que não haja *tokens* em comum entre o conjunto a ser inserido e o *repositório* do lugar considerado. Ramos de saída comuns são representados por uma linha contínua.

RPSD tem 3 tipos de ramos relacionados ao **tratamento de exceções**: os *ramos de entrada preemptivos* e *suspensivos*, e os *ramos de saída restauradores*. Para lugares do tipo

I, somente ramos *preemptivos* podem ser utilizados. Um *ramo de entrada preemptivo* permite que uma transição **remova todos** os *tokens* do lugar, independentemente de quaisquer condições. Ramos de entrada preemptivos são representados por uma linha tracejada grossa.

A *regra de habilitação* referente a um *ramo de entrada comum*, rotulado com uma tripla $\langle \text{min}, \text{max}, \text{ordem} \rangle$, é que o lugar correspondente possua ao menos um subconjunto de *tokens* de cardinalidade entre *min* e *max*. Pode ocorrer de mais de um subconjunto satisfazer tal exigência, independentemente da ordem ser *aleatória* ou *cronológica*. Neste caso, somente um destes subconjuntos será removido do lugar quando do disparo da transição, o que dependerá de outras condições, relativas à guarda da transição e aos *tokens* presentes nos lugares de saída desta.

Um *ramo de entrada conjugado* estende a *regra de habilitação* de um ramo de entrada comum ao grupo de lugares ao qual está ligado.

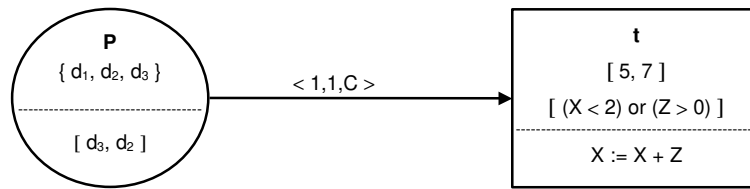
A *regra de habilitação* de um *ramo de saída comum* é que o lugar ligado a ele não possua *tokens* pertencentes ao conjunto resultante da *função de inserção de tokens*, além de estes *tokens* serem de tipos **compatíveis** com o *domínio do lugar* (D_S).

Para lugares do tipo I, a *regra de habilitação* referente a um *ramo preemptivo* é que o respectivo lugar possua ao menos um *token*. Caso a transição relacionada dispare, **todos** os *tokens* do lugar serão **removidos**. Isto significa que estados locais mantidos por diferentes componentes do sistema podem ser extintos de uma só vez.

Dadas as possíveis **conexões** envolvendo lugares do tipo I, passemos para a representação particularizada, em RPTM, de cada uma destas conexões. Assim, veremos como é feita a representação de uma transição RPSD, conectada a um lugar do tipo I, através de um *ramo de entrada comum* e de um *ramo de saída comum*. O *ramo de entrada conjugado* estende a forma de representação de um ramo de entrada comum. Já o *ramo de entrada preemptivo* será tratado somente para lugares do tipo III, nos quais sua atuação é mais abrangente. Este tratamento particularizado será adotado nas seções seguintes, as quais abordam **conexões** envolvendo lugares do tipo II e do tipo III. Caso uma mesma **transição** se **conecte** a um **conjunto de lugares de tipos iguais ou diferentes**, a representação em RPTM de tal transição corresponderá à união das representações RPTM relativas a cada conexão envolvendo a transição, tomando-se cuidado ao agregar as partes associadas às regras de habilitação referentes aos ramos. O objetivo proposto não é a obtenção de representações otimizadas e, sim, representações que atendam as **convencões adotadas na descrição de marcações admissíveis**, dadas para os diferentes tipos de lugares RPSD, na seção 12.2.

A figura 12.13 mostra a representação, em RPTM, de uma transição **t** conectada a um lugar **P**, do tipo I, por um *ramo de entrada comum* com *rótulo* $\langle 1, 1, C \rangle$ (remoção cronológica de um *token*). A *guarda* de **t**, dada por $G(t) = [(X < 2) \text{ or } (Z > 0)]$, indica uma das condições de *habilitação* da transição. A outra condição de habilitação é a presença de ao menos um *token* em **P**. O *intervalo de disparo* da transição é [5,7], significando que, após sua habilitação, **t** deverá disparar entre 5 e 7 unidades de tempo, a menos que seja desabilitada pelo disparo de outra transição, que possa afetar o **conteúdo** do lugar **P** ou tornar a *guarda* de **t** *falsa*, devido à realização de *operações* que possam alterar a **memória da rede** (rede cujos lugares são acessíveis por toda RPSD). Ao disparar, **t** removerá o *token* mais antigo presente em **P**, e realizará sua única *operação* ($To(t) = "X := X + Z"$), a qual atualizará a *representação* da variável **X** na **memória da rede**.

a) Conexão envolvendo ramo de entrada comum



b) RPTM equivalente

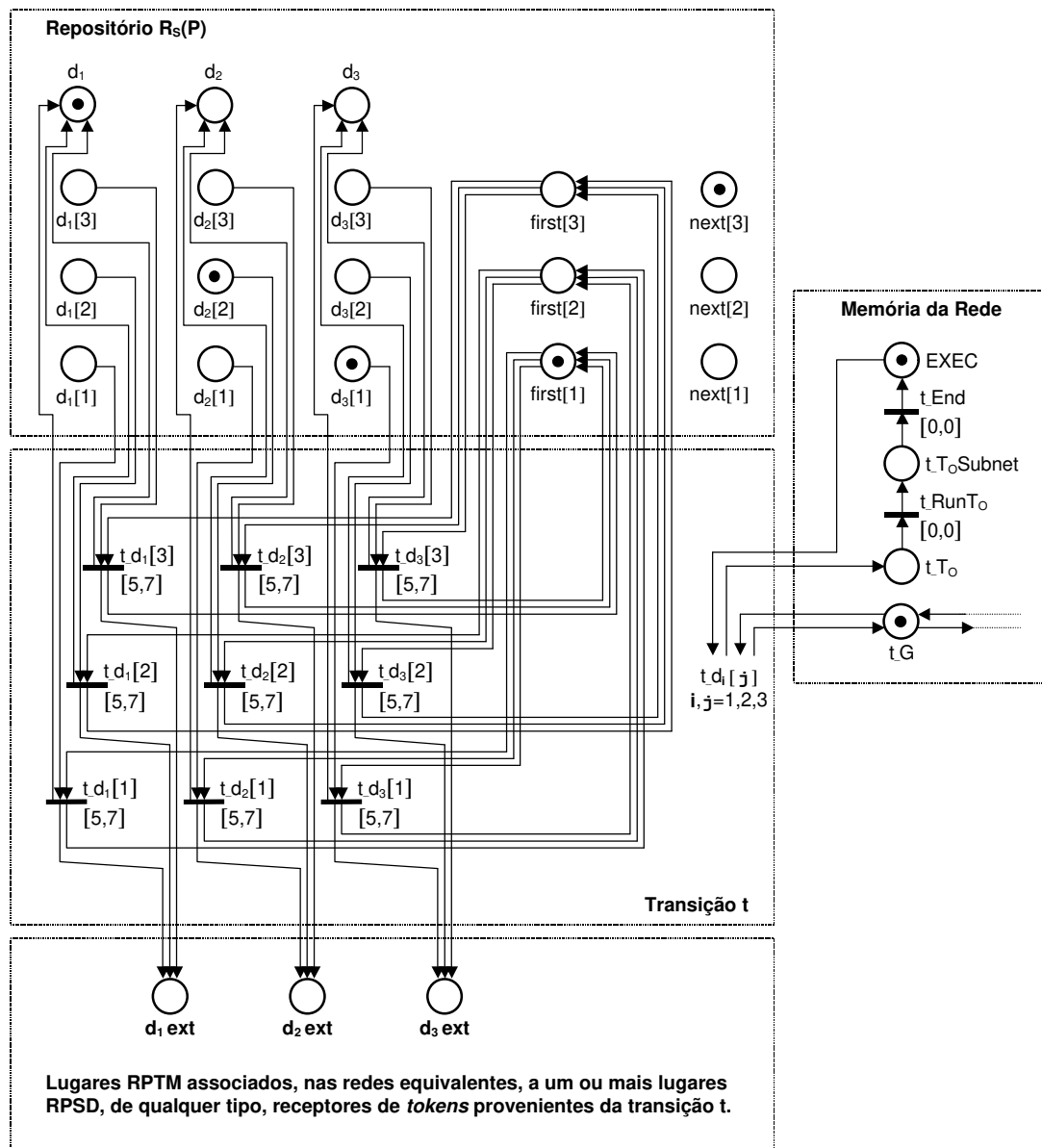


Figura 12.13. RPTM descrevendo uma conexão com ramo de entrada comum.

As transições da RPTM da figura 12.13 garantem a remoção de qualquer *token* do domínio $D_S(P)$, desde que tal *token* seja o **mais antigo** no **repositório** $R_S(P)$. Cada transição $t_{d_i[j]}$ é responsável pela **remoção** do *token* d_i , presente na j -ésima posição da **fila circular** representante de $R_S(P)$. Pelas **convenções adotadas**, a j -ésima posição da fila circular corresponderá à primeira posição do **repositório** $R_S(P)$, caso exista um *token* no lugar **first[j]**.

Somente uma transição $t_{d_i[j]}$ poderá ficar **habilitada** por vez, o que dependerá das seguintes condições:

- a) a existência de um *token* no lugar $d_i[j]$, indicando que o *token* d_i está presente no **repositório** de P ;
- b) a existência de um *token* no lugar **first[j]**, denotando que d_i é o *token* mais antigo no **repositório** de P ;
- c) a existência de um *token* no lugar t_G , indicando que a **guarda da transição** t é **verdadeira**; e
- d) a existência de um *token* no lugar **EXEC** associado a P (na **memória da rede**), denotando que transição alguma conectada ao lugar P está sendo executada.

Uma vez habilitada, $t_{d_i[j]}$ só poderá disparar após 5, e até 7, unidades de tempo, contadas após sua habilitação. O **disparo** de $t_{d_i[j]}$ promoverá:

- a) a inserção de um *token* no lugar d_i de $R_S(P)$, para indicar que o *token* d_i **não** está mais presente no **repositório** de P ;
- b) a atualização do ponteiro **first**, da fila circular representante do **repositório** de P , através da remoção do *token* de **first[j]** e da inserção de um *token* no lugar **first[j+1]**. Caso $j+1 = 4$, o *token* será inserido no lugar **first[1]**;
- c) a inserção de um *token* no lugar d_{iext} , associado ao **domínio** de outro lugar (**de qualquer tipo**), conectado a t por um **ramo de saída**. Caso t seja uma **transição somente consumidora de tokens**, isto é, uma transição sem lugares de saída, esta operação não é necessária;
- d) a remoção do *token* presente no lugar **EXEC** associado a P , denotando que t tem **acesso exclusivo** ao lugar P , isto é, nenhuma outra transição conectada a P poderá disparar enquanto o lugar **EXEC** (associado a P) estiver vazio; e
- e) a inserção de um *token* no lugar t_T_O , indicando para a **memória da rede** fazer as atualizações descritas pelas **operações da transição** (no caso, uma única operação). Estas operações são descritas na **subrede** representada pelo lugar $t\_T_OSubnet$.

Na marcação da figura 12.13, a transição $t_{d_3[1]}$ está habilitada, pois há um *token* em cada um de seus lugares de entrada ($d_3[1]$, **first[1]**, t_G e **EXEC**). Seu disparo promoverá a remoção dos *tokens* presentes nos lugares $d_3[1]$, **first[1]** e **EXEC**, e a inserção de um *token* nos lugares d_{3ext} , d_3 , **first[1]** e t_T_O . A marcação do lugar t_G não se alterará, a menos que a execução da atribuição sobre X (na subrede $t\_T_OSubnet$) torne a guarda da transição **falsa**.

A transição t da figura 12.13 é utilizada na figura 12.14, a qual descreve uma conexão entre t e um lugar Q , do tipo I, envolvendo um **ramo de saída comum**. Como a **função de inserção** deste ramo está ausente, ela corresponde ao conjunto de *tokens* retirados dos **lugares de entrada** da transição t . Na construção da RPTM equivalente, adotou-se que t possui um único lugar de entrada, qual seja, o lugar P da figura 12.13. Assim, a **função de inserção** corresponderá ao *token* retirado de P , pelo disparo de t . Caso a **função de inserção** seja modificada, a rede equivalente deverá ser alterada. **Ramos fornecedores**, utilizados para a

inserção de *tokens* em lugares do tipo III (seção 12.3.3), possuem a *função de inserção* definida da mesma forma que a utilizada para *ramos de saída comuns*. Mais adiante, na figura 12.18, será mostrado como uma *função de inserção* pode ser implementada, em RPTM, através de transições envolvendo diferentes combinações de lugares conectados.

As transições da RPTM da figura 12.14 garantem a inserção de qualquer *token* do domínio $D_S(Q)$ no *repositório* $R_S(Q)$, desde que tal *token* esteja **disponível** para inserção, isto é, não esteja presente no *repositório* $R_S(Q)$. Cada transição $t_{d_i[j]}$ é responsável pela **inserção** do *token* d_i , na j -ésima posição da **fila circular** representante de $R_S(Q)$. O ponteiro **next** indica a posição de inserção de um *token* nesta fila circular. Assim, em cada **coluna** de transições ($t_{d_1[j]}$, $t_{d_2[j]}$, $t_{d_3[j]}$) existirá, no máximo, uma transição habilitada por vez (aquela que possuir o lugar de entrada **next[j]** com um *token*). As convenções adotadas para a fila circular são as mesmas apresentadas na seção 12.2.1.

São condições para a **habilitação** de uma transição $t_{d_i[j]}$:

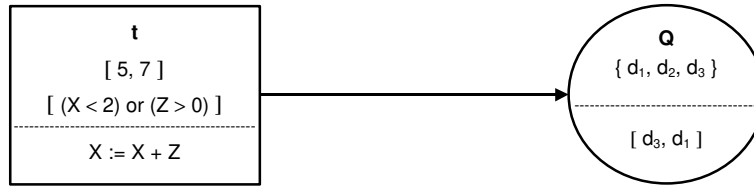
- a) a existência de um *token* no lugar **d_iext**, significando que alguma transição removera o *token* d_i de algum outro lugar;
- b) a existência de um *token* no lugar d_i , indicando que o *token* d_i não está presente no **repositório** de Q ;
- c) a existência de um *token* no lugar **next[j]**, indicando o final da fila circular representante do **repositório** de Q ;
- d) a existência de um *token* no lugar **t_G**, indicando que a *guarda da transição t* é **verdadeira**; e
- e) a existência de um *token* no lugar **EXEC** associado a Q (na **memória da rede**), denotando que transição alguma conectada ao lugar Q está sendo executada.

Caso não seja desabilitada pelo disparo de alguma outra transição que interfira nas suas condições de habilitação, $t_{d_i[j]}$ disparará entre 5 e 7 unidades de tempo, após sua habilitação. O **disparo** de $t_{d_i[j]}$ promoverá:

- a) a remoção de um *token* de ambos lugares **d_iext** e d_i , e a inserção de um *token* no lugar $d_i[j]$, para indicar que o *token* d_i passa a estar presente no **repositório** de Q , na j -ésima posição da fila circular representante deste **repositório**;
- b) a atualização do ponteiro **next**, da fila circular representante do **repositório** de Q , através da remoção do *token* de **next[j]** e da inserção de um *token* em **next[j-1]**. Caso $j-1 = 0$, o *token* será inserido em **next[3]**;
- c) a remoção do *token* presente no lugar **EXEC** associado a Q , denotando que t tem **acesso exclusivo** ao lugar Q , isto é, nenhuma outra transição conectada a Q poderá disparar enquanto **EXEC** (associado a Q) estiver vazio; e
- d) a inserção de um *token* no lugar **t_{To}**, indicando para a **memória da rede** fazer as atualizações descritas pelas **operações da transição** (no caso, uma única operação).

Na marcação da figura 12.14, a transição $t_{d_2[1]}$ está habilitada, pois há um *token* em cada um de seus lugares de entrada (**d₂ext**, d_2 , **next[1]**, **t_G** e **EXEC**). Seu disparo promoverá a remoção dos *tokens* presentes nos lugares **d₂ext**, d_2 , **next[1]** e **EXEC**, e a inserção de um *token* nos lugares $d_2[1]$, **next[3]** e **t_{To}**. Como nas demais representações deste capítulo, a marcação do lugar **t_G** não se alterará com o disparo de t , a menos que a execução das operações **t_{To}** (executadas na subrede representada pelo lugar **t_{To}Subnet**) afete a **guarda da transição**.

a) Conexão envolvendo ramo de saída comum



b) RPTM equivalente

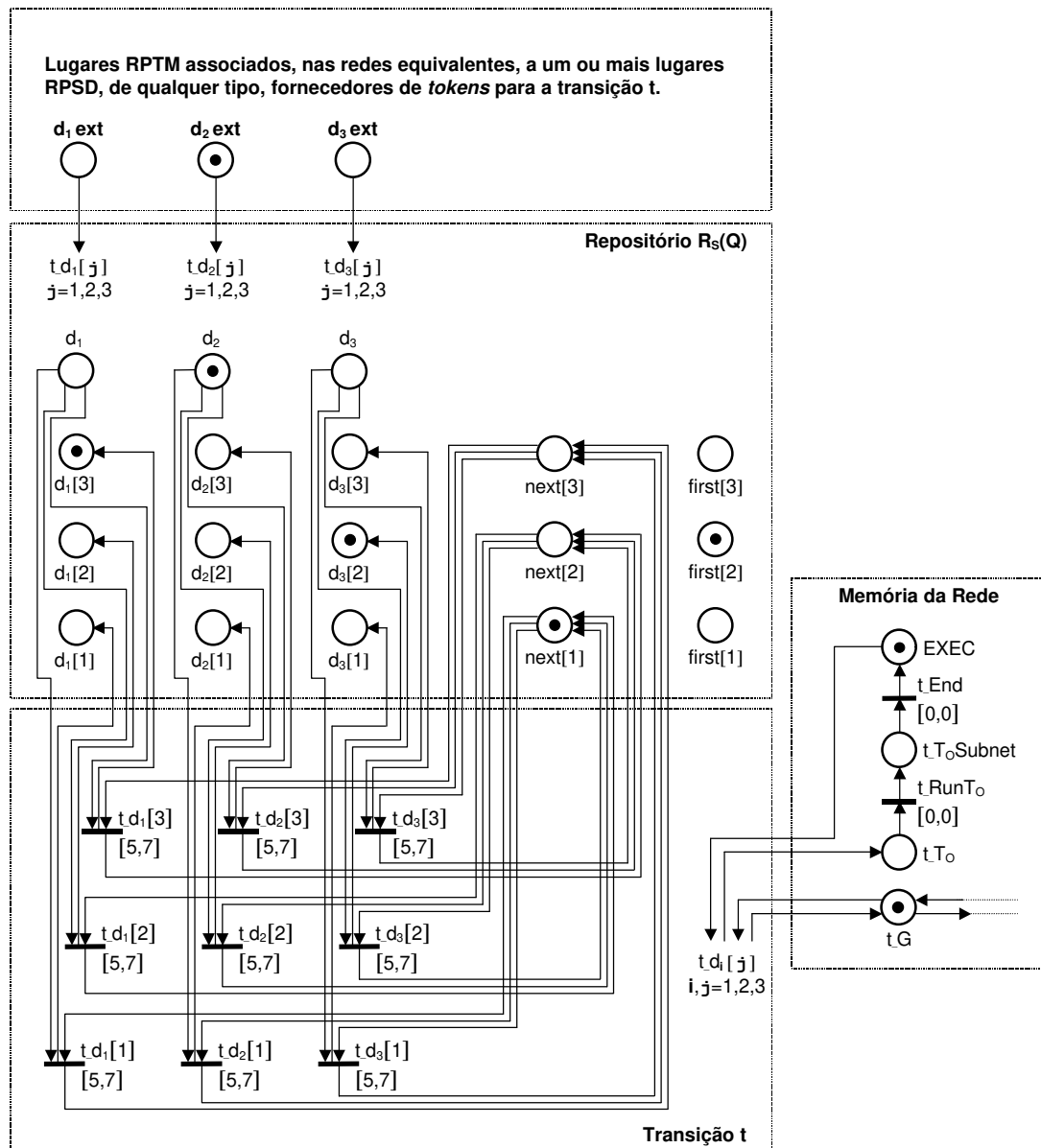


Figura 12.14. RPTM descrevendo uma conexão com ramo de saída comum.

As convenções utilizadas na representação dos dois exemplos dados nesta seção, podem ser estendidas e adaptadas para outras formas de **conexão**, envolvendo lugares do tipo I, transições e ramos correspondentes (ramos de entrada simples com rótulo descrevendo remoção aleatória, ramos de entrada conjugados e ramos de saída com funções de inserção com duas ou mais alternativas), desde que se atenda às restrições de **marcações admissíveis**, detalhadas na seção 12.2.1. No caso da utilização de ramos preemptivos com lugares do tipo I, ela é semelhante à empregada com lugares do tipo III, a qual encontra-se representada na figura 12.19. Até este ponto, consideramos nosso objetivo atendido, qual seja, o de demonstrar a viabilidade de representar, através de redes de Petri Temporizadas de Merlin, transições RPSD conectadas a lugares do tipo I. Na sequência, veremos a representação de transições e ramos relacionados a lugares do tipo II.

12.3.2. CONEXÕES COM LUGARES DO TIPO II

Assim como foi feito para lugares do tipo I, vejamos as possíveis conexões envolvendo transições, ramos e lugares do tipo II.

Transições podem fazer a *leitura* de *tokens* (sem remoção) armazenados em lugares do tipo II, através de **ramos de entrada de leitura**. Este tipo de ramo permite que uma transição **t** faça a leitura de um ou mais dados (*tokens*) presentes em um lugar de armazenamento **P**, de acordo com uma **expressão de leitura** **R(P,t)**, no seguinte formato:

$$R(P,t) = \langle t_1, a_1 \rangle \langle t_2, a_2 \rangle \dots \langle t_n, a_n \rangle$$

onde $t_1, t_2, \dots, t_n$ representam *identificadores* de *tokens* que armazenarão, cada qual, o conteúdo dos **endereços** $a_1, a_2, \dots, a_n$ do lugar relacionado. Caso os endereços sejam omitidos, $t_1, t_2, \dots, t_n$ assumirão o conteúdo das **n** primeiras posições de armazenamento do lugar. Ramos de leitura são representados por uma linha contínua.

Para fazer a **atualização** do conteúdo presente em células de armazenamento de um lugar do tipo II, uma transição deverá estar conectada a tal lugar por um **ramo de saída de escrita**, o qual permite a **escrita** de um ou mais dados (*tokens*) no lugar. Estes ramos são rotulados com uma **expressão de escrita** **W(P,t)**, no seguinte formato:

$$W(P,t) = \langle t_1, a_1 \rangle \langle t_2, a_2 \rangle \dots \langle t_n, a_n \rangle$$

onde $t_1, t_2, \dots, t_n$ representam *constantes* ou *identificadores* de *tokens*, compatíveis com o **domínio do lugar** (D_M), que deverão ser armazenados nos **endereços** $a_1, a_2, \dots, a_n$, **válidos** de acordo com a definição do **espaço de endereçamento** (**A**) do lugar relacionado. Caso os **endereços** sejam omitidos, $t_1, t_2, \dots, t_n$ serão escritos nas **n** primeiras posições de armazenamento do lugar. Ramos de escrita são representados por linha contínua.

O único tipo de ramo para o **tratamento de exceções** que pode ser utilizado com lugares do tipo II é o **ramo de entrada preemptivo**, o qual serve para **resetar** as células de armazenamento. O significado de **resetar** é o de assumir o valor *default* correspondente a cada **tipo** de *token* presente no **domínio lugar** (D_M), cuja definição encontra-se na classe **anotações** (A_n) da RPSD portadora do lugar tipo II. Conforme descrito anteriormente, ramos de entrada preemptivos são representados por uma linha tracejada grossa.

A **regra de habilitação** referente a um **ramo de leitura** possui duas condições. A primeira é que a **expressão de leitura** do ramo seja **compatível** com o lugar, de maneira que os **endereços**, caso fornecidos, sejam **válidos** (pertencentes ao **espaço de endereçamento A**).

A segunda condição refere-se às *latências* de *leitura* e *escrita*: para que uma operação de leitura possa ser feita no lugar relacionado **P**, é necessário que tenha transcorrido um intervalo de tempo mínimo **T**, desde a última operação **O** (de leitura ou de escrita) em **P**, tal que:

se **O** = operação de leitura $\Rightarrow T = L_R(\mathbf{P}) \times \mathbf{n}$, caso contrário

se **O** = operação de escrita $\Rightarrow T = L_W(\mathbf{P}) \times \mathbf{n}$,

onde **n** denota a quantidade de *tokens* envolvidos na operação **O**. Por exemplo, se a última operação envolvendo o lugar **P** foi a escrita de 5 *tokens*, então é necessário que tenha transcorrido um intervalo de $L_W \times 5$ unidades de tempo para que qualquer outra operação (escrita ou leitura) possa ser feita sobre o lugar.

Um *ramo de escrita* possui *regra de habilitação* semelhante à de um ramo de leitura. A primeira condição é que a *expressão de escrita* seja **compatível** com as definições do lugar, de forma que os *tipos* de seus *tokens* pertençam ao **domínio do lugar** (D_M) e os *endereços*, caso presentes, sejam **válidos** (pertencentes a **A**). A segunda condição componente da regra é que tenha transcorrido, desde a última operação envolvendo o lugar, um intervalo de tempo igual ou superior à *latência* correspondente (L_W para escrita e L_R para leitura) multiplicada pelo número de *tokens* envolvidos em tal operação.

Ramos preemptivos associados a lugares do tipo II não apresentam restrições quanto à habilitação da transição relacionada. Caso esta dispare, o conteúdo das células de armazenamento do lugar serão *resetadas*.

A figura 12.15 ilustra a representação, em RPTM, de uma transição **t** conectada a um lugar **P**, do tipo II, por um *ramo de entrada de leitura* com *expressão de leitura* $\langle t_1, \text{expr}_1 \rangle \langle t_2, \text{expr}_2 \rangle$, denotando a leitura dos *endereços* expr_1 e expr_2 , cujos *conteúdos* serão armazenados, respectivamente, nos *tokens* identificados pelos *nomes* t_1 e t_2 . A *guarda* de **t** é dada por $G(t) = [\text{count} < 10]$, onde **count** é uma variável presente na **memória da rede**. O *intervalo de disparo* de **t** é generalizado por $[x, y]$, significando que **t** deverá disparar entre **x** e **y** unidades de tempo após sua habilitação (a menos que seja desabilitada pelo disparo de outra transição). A única *operação* de **t** é o incremento da variável **count**.

O lugar **P** da figura 12.15 possui **n** *células de armazenamento*, identificadas pelos *endereços* $a_0, a_1, \dots, a_{n-1}$. Cada célula pode armazenar um único *valor*, representado pelos *tokens* $d_{flt}, d_1, d_2, \dots, d_k$, onde d_{flt} corresponde a um dos valores d_m , $1 \leq m \leq k$, designado como valor *default*. É permitido a duas ou mais células de **P** armazenarem o mesmo valor. As *latências de leitura* (L_R) e *de escrita* (L_W) de **P** são designadas por **r** e **w**, denotando os tempos para conclusão de operações de leitura e escrita para um único *token* (valor).

Apenas parte da representação de equivalência encontra-se ilustrada na figura 12.15, para não sobrecarregar o gráfico. Adotando-se as **convenções** dadas na seção 12.2.2, a presença de um *token* num lugar $d_{flt}[i]$ (ou $d_m[j]$) indica que o *token* d_{flt} (ou d_m) ocupa a *i*-ésima *célula de armazenamento* de **P**, isto é, $a_{i-1} = d_{flt}$ (ou $a_{i-1} = d_m$). O lugar **r/w** é utilizado para indicar se alguma operação pode ser feita envolvendo **P**, respeitando-se suas *latências de leitura e de escrita*. **P** estará liberado para operações quando da presença de um *token* em **r/w**. Os lugares com rótulo iniciado por t_1 referem-se à leitura do *token* t_1 , enquanto aqueles com rótulo iniciado por t_2 referem-se à leitura do *token* t_2 . Desta forma, a presença de um *token* no lugar t_1_read significa que a leitura do *token* t_1 pode ser feita; significado análogo apresenta o lugar t_2_read . A **memória da rede** possui, além dos lugares **t_G**, **t_To**, **t_ToSubnet** e **EXEC**, anteriormente explicados, um lugar para cada possibilidade

de associação entre os *endereços* $a_0, a_1, \dots, a_{n-1}$ e as variáveis $expr_1$ e $expr_2$ da *expressão de leitura* do ramo. A presença de um *token* nos lugares $expr_1=a_0$ e $expr_2=a_2$ denota que, caso t dispare, ocorrerá a leitura das células de armazenamento de endereços a_0 e a_2 . O controle dos lugares associados à *expressão de leitura* não está representado na figura 12.15. Entretanto, para garantir a atomicidade de t , enquanto o lugar **EXEC** (associado a **P**) estiver vazio, a marcação dos lugares $expr_1=a_i$ e $expr_2=a_j$, $i, j = 1, \dots, n-1$, somente poderá ser alterada por transições pertencentes à subrede representada por $t_{T_0Subnet}$. Os resultados da leitura são designados pela marcação dos lugares de terminação **ext**, na parte inferior direita da figura (há um grupo de lugares **ext** para representar o conteúdo do contêiner de *token* t_1 e um grupo para representar t_2). Tal como foi feito para conexões envolvendo lugares do tipo I, os lugares **ext** podem ser acessados externamente, pelas redes representantes de outros lugares RPSD.

As transições $t_{a_i a_j}$ ($i, j = 0, 1, \dots, n-1$) são responsáveis pelo início do processo de leitura, possuindo, desta forma, intervalo de disparo igual ao da transição t da RPSD, qual seja, $[x, y]$. Somente uma destas transições poderá ficar habilitada por vez. Uma transição $t_{a_i a_j}$ ficará habilitada se:

- os lugares $expr_1=a_i$ e $expr_2=a_j$ possuírem, cada qual, um *token*, indicando que as **expressões do rótulo** associado ao ramo de leitura considerado, na RPSD, são **verdadeiras**;
- existir um *token* no lugar **r/w**, denotando que **P** está **disponível** para uma operação de leitura (e, também, de escrita);
- existir um *token* no lugar **t_G**; e
- existir um *token* no lugar **EXEC** associado a **P** (na **memória da rede**), denotando que transição alguma conectada a **P** está sendo executada.

O disparo de uma transição habilitada $t_{a_i a_j}$ promoverá:

- a remoção do *token* presente em **r/w**, sinalizando que **P** está realizando uma operação (no caso, de leitura);
- a remoção do *token* presente no lugar **EXEC** associado a **P**, denotando que t tem **acesso exclusivo** a **P**; e
- a inserção de um *token* em **t₁_read**, habilitando a leitura cujo resultado será armazenado no *token* t_1 .

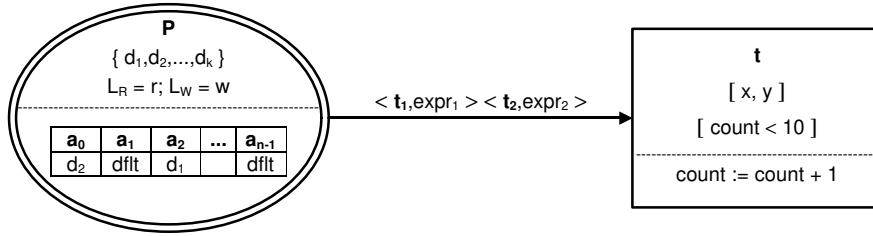
As transições iniciadas por t_1 são responsáveis pela leitura do valor a ser armazenado no *token* t_1 . Estas transições têm intervalo $[r, r]$, de forma a simular a **latência de leitura** de **P**. Somente uma destas transições poderá ficar habilitada por vez. A transição $t_{1\_d_m[i+1]}$, com d_m podendo ser **dflt**, ficará habilitada se:

- existir um *token* no lugar **t₁_read**;
- existir um *token* em $expr_1=a_i$, indicando a **leitura** da $(i+1)$ -ésima célula de armazenamento de **P**; e
- existir um *token* em $d_m[i+1]$, indicando que a $(i+1)$ -ésima célula de armazenamento de **P** está **ocupada** por d_m .

O disparo de uma transição habilitada $t_{1\_d_m[i+1]}$ promoverá:

- a inserção de um *token* no lugar **t₁_d_mext** (na porção da RPTM referente ao contêiner de *token* t_1);
- a remoção do *token* presente em **t₁_read**, sinalizando que t realizou a operação de leitura referente ao *token* t_1 ; e
- a inserção de um *token* em **t₂_read**, habilitando a leitura cujo resultado será armazenado no *token* t_2 .

- a) Conexão envolvendo ramo de leitura
Lugar tipo II com n células de armazenamento e domínio de cardinalidade k



- b) RPTM equivalente

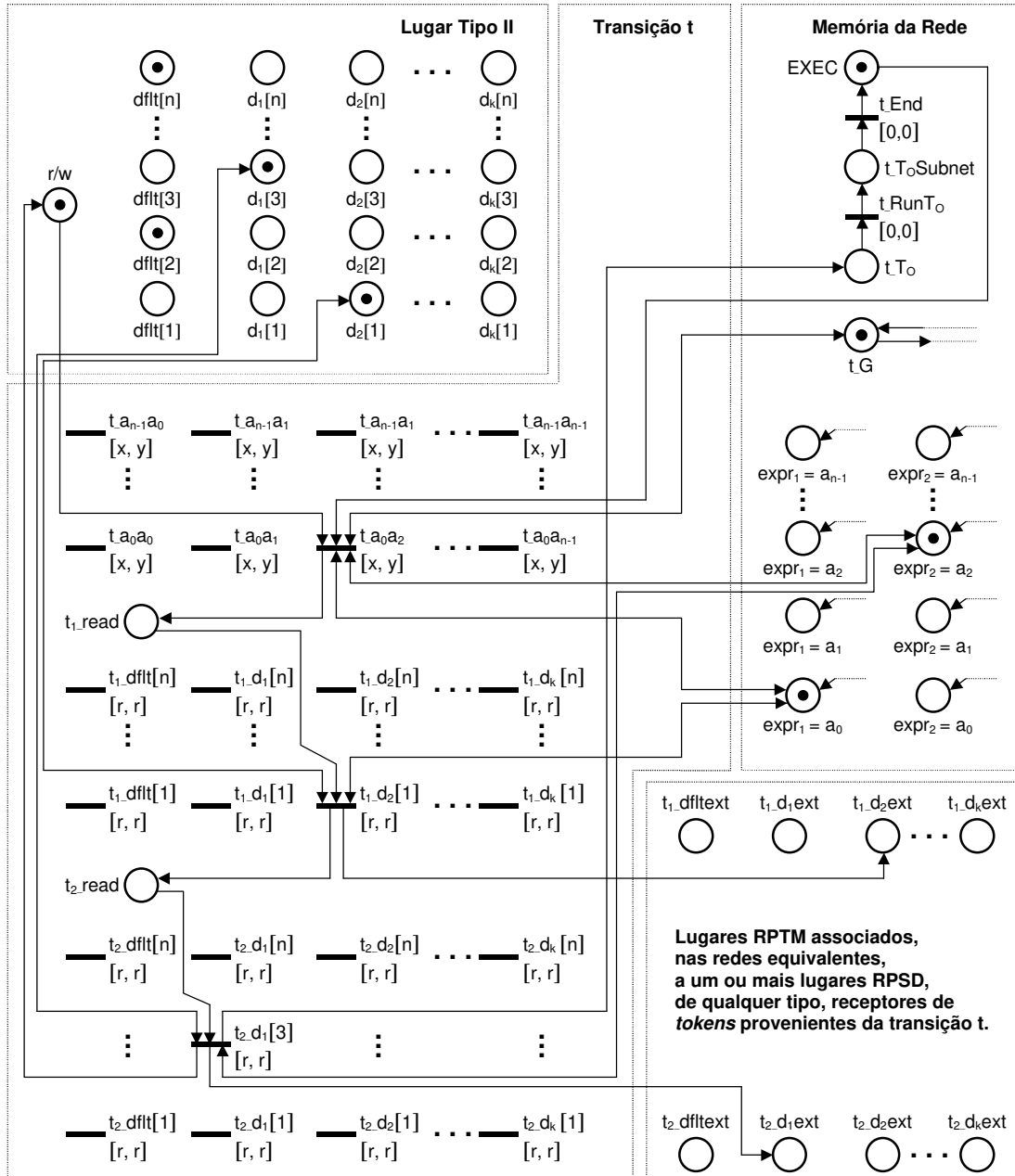


Figura 12.15. RPTM descrevendo uma conexão com ramo de leitura.

Por outro lado, as transições t_2 são responsáveis pela leitura a ser armazenada no *token* t_2 . Seu intervalo é dado por $[r, r]$, de forma a simular a **latência de leitura** de P . Analogamente a uma transição $t_1\_d_m[i+1]$, a transição $t_2\_d_m[j+1]$, com d_m podendo ser d_{flt} , ficará habilitada se:

- existir um *token* no lugar t_2_read ;
- existir um *token* em $expr_2=a_j$, indicando a **leitura** da $(j+1)$ -ésima célula de armazenamento de P ; e
- existir um *token* em $d_m[j+1]$, indicando que a $(j+1)$ -ésima célula de armazenamento de P está **ocupada** por d_m .

O disparo de uma transição habilitada $t_2\_d_m[j+1]$ promoverá:

- a inserção de um *token* no lugar $t_2\_d_mext$ (na porção da RPTM referente ao contêiner de *token* t_2);
- a remoção do *token* presente em t_2_read , sinalizando que t realizou a operação de leitura referente ao *token* t_2 ;
- a inserção de um *token* no lugar t_T_O , indicando para a **memória da rede** fazer as atualizações descritas pelas **operações da transição** (no caso, uma única operação); e
- a inserção de um *token* em r/w , disponibilizando P para uma nova operação de leitura/escrita de *tokens* (obs.: a habilitação de P para uma nova operação, tanto de escrita quanto de leitura, somente será possível quando for restituído o *token* ao lugar **EXEC** associado a P).

Na marcação mostrada na figura 12.15, ocorrerá a seqüência de disparo formada pelas transições $t\_a_0a_2$, $t_1\_d_2[1]$, $t_2\_d_1[3]$, t_RunT_O e t_End , após a qual os lugares $t_1\_d_2ext$ e $t_2\_d_1ext$ estarão marcados, sinalizando que o *token* t_1 receberá o conteúdo da célula a_0 (valor d_2) e que o *token* t_2 receberá o conteúdo da célula a_2 (d_1). Ambas as células preservarão seu conteúdo (observe a presença de ramos de realimentação conectados aos lugares $d_2[1]$ e $d_1[3]$).

A figura 12.16 mostra o mesmo lugar P e a mesma transição t da figura 12.15, conectados, no entanto, por um **ramo de saída de escrita**. Este ramo possui **expressão de escrita** $\langle t_1, expr_1 \rangle \langle t_2, expr_2 \rangle$, denotando a escrita do **conteúdo** dos *tokens* referenciados por t_1 e t_2 nos respectivos **endereços** $expr_1$ e $expr_2$, pertencentes ao **espaço de endereçamento** A do lugar.

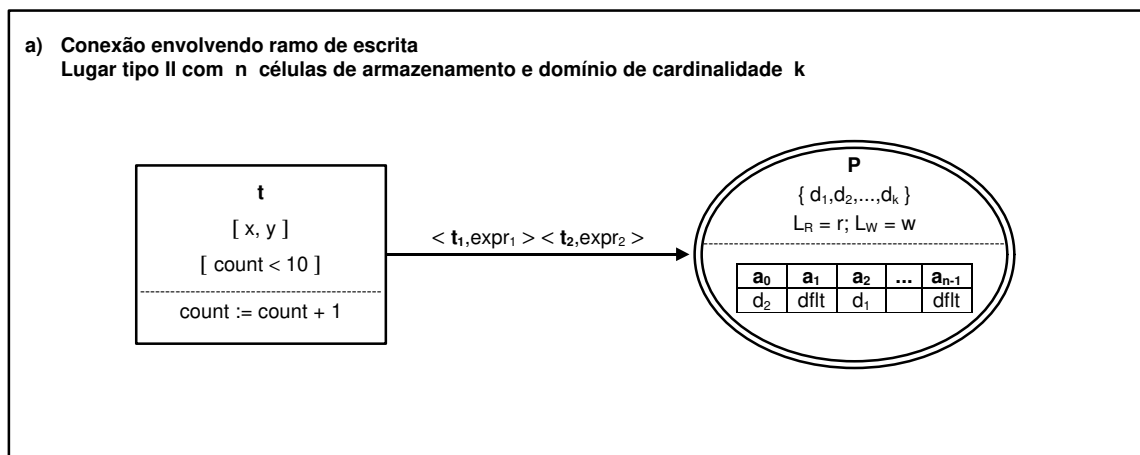
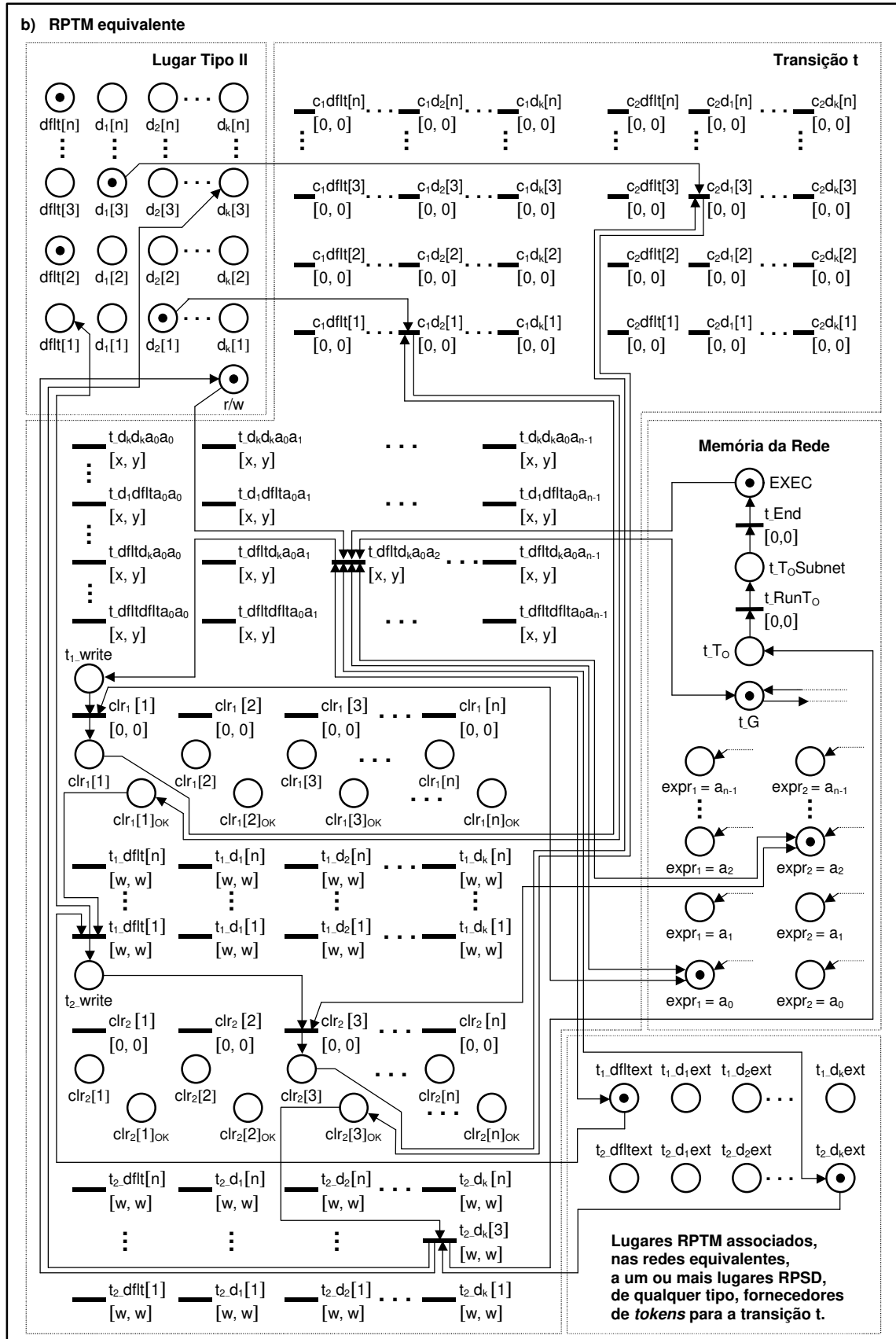


Figura 12.16. RPTM descrevendo uma conexão com ramo de escrita.



Continuação da figura 12.16.

Convenções semelhantes àsquelas adotadas na representação da conexão envolvendo o ramo de leitura, foram utilizadas na elaboração da RPTM da figura 12.16. Além dos lugares **t_G**, **t_To**, **t_ToSubnet** e **EXEC**, a **memória da rede** possui um lugar para cada possibilidade de associação entre os *endereços* **a₀**, **a₁**, ..., **a_{n-1}** e as variáveis **expr₁** e **expr₂** da *expressão de escrita* do ramo. A presença de um *token* nos lugares **expr₁=a_i** e **expr₂=a_j** denota que, caso **t** dispare, ocorrerá a escrita do conteúdo do *token* **t₁** na célula de armazenamento de endereço **a_i**, assim como a escrita de **t₂** na célula **a_j**. Os *conteúdos* de **t₁** e **t₂** são designados pela marcação dos lugares de terminação **ext**, na parte inferior direita da figura (há um grupo de lugares **ext** para representar o conteúdo do *token* **t₁** e um grupo para representar **t₂**). Se a transição **t** possuir lugares de entrada, a inserção de *tokens* nos lugares **ext** é realizada pela ação de *ramos de entrada* conectados a **t**. Caso contrário, isto é, se **t** for uma **transição somente geradora de tokens** (utilizada para o propósito de inicialização de lugares), a marcação escolhida para os lugares **ext** pode ser fixada, adotando-se, na RPTM da figura 12.16, a realimentação de tais lugares (fazendo com que as transições conectadas a eles tenham um ramo de saída para cada ramo de entrada).

Supondo-se que os lugares **t₁_d_gext**, **t₂_d_hext**, **expr₁=a_i**, **expr₂=a_j**, **r/w**, **t_G** e **EXEC** possuam, cada qual, um *token*, será feita a **atualização** do conteúdo das células de armazenamento de endereços **a_i** e **a_j**, respectivamente, para os valores dos *tokens* **t₁ = d_g** e **t₂ = d_h** (com **d_g** e **d_h** podendo ser o valor **dflt**). Este processo de atualização (escrita) é realizado em cinco etapas:

e.1) Ativação da operação de escrita.

As transições **t_d_gd_ha_ia_j** (**g, h = 1, 2, ..., k; i, j = 0, 1, ..., n-1**), com **d_g** e **d_h** podendo ser o valor **dflt**, são responsáveis pelo início do processo de escrita, possuindo, desta forma, intervalo de disparo igual ao da transição **t** da RPSD, qual seja, **[x,y]**. Somente uma destas transições poderá ficar habilitada por vez. Uma transição **t_d_gd_ha_ia_j** ficará habilitada se:

- os lugares **t₁_d_gext** e **t₂_d_hext** possuírem, cada qual, um *token*, indicando que os **contêineres de token** **t₁** e **t₂**, do **rótulo** associado ao ramo de escrita considerado, na RPSD, possuem valores válidos (**t₁ = d_g** e **t₂ = d_h**);
- os lugares **expr₁=a_i** e **expr₂=a_j** possuírem, cada qual, um *token*, indicando que as **expressões do rótulo** associado ao ramo de escrita considerado, na RPSD, são **verdadeiras**;
- existir um *token* no lugar **r/w**, denotando que **P** está disponível para uma operação de escrita (e, também, de leitura);
- existir um *token* no lugar **t_G**; e
- existir um *token* no lugar **EXEC** associado a **P** (na **memória da rede**), denotando que transição alguma conectada a **P** está sendo executada.

O disparo de uma transição habilitada **t_d_gd_ha_ia_j** promoverá:

- a remoção do *token* presente em **r/w**, sinalizando que **P** está realizando uma operação (no caso, de escrita);
- a remoção do *token* presente no lugar **EXEC** associado a **P**, denotando que **t** tem acesso exclusivo a **P**; e
- a inserção de um *token* em **t₁_write**, habilitando a **atualização** da célula **a_i**, a qual receberá o valor armazenado em **t₁** (no caso, **d_g**), após a realização das duas etapas a seguir.

e.2) Remoção do *token* presente na fileira de lugares correspondente à célula a_i .

Como sempre haverá um valor d_r ($r = 1, 2, \dots, k$) armazenado na célula a_i , o *token* presente no lugar $d_r[i+1]$ deverá ser removido, antes da realização da etapa **e.3**. Esta segunda etapa é realizada através do disparo seguido das transições $clr_1[i+1]$ e $c_1d_r[i+1]$.

A transição $clr_1[i+1]$ é responsável pela inserção de um *token* no lugar $clr_1[i+1]$, indicando a necessidade de limpar a $(i+1)$ -ésima fileira de lugares (formada por $dflt[i+1]$ e $d_c[i+1]$, $c = 1, 2, \dots, k$), correspondente à célula de armazenamento a_i . Além disso, esta transição remove o *token* presente em t_1_write , indicando que a escrita do *token* t_1 está sendo efetivada. Seu *intervalo de disparo* é imediato, ou seja, $[0,0]$.

Supondo-se que o valor armazenado em a_i é d_r , a inserção de um *token* em $clr_1[i+1]$ habilitará a transição $c_1d_r[i+1]$. O disparo desta é imediato (*intervalo de disparo* $[0,0]$), promovendo a remoção dos *tokens* presentes em $d_r[i+1]$ e $clr_1[i+1]$, assim como a inserção de um *token* em $clr_1[i+1]_{OK}$, possibilitando a execução da etapa seguinte.

e.3) Inserção de um *token* no lugar $d_g[i+1]$.

A inserção de um *token* em $clr_1[i+1]_{OK}$ indica que a célula de armazenamento a_i está limpa. Desta forma, a transição $t_1\_d_g[i+1]$ ficará habilitada e o seu disparo, após w unidades de tempo (w é a *latência de escrita* do lugar P) promoverá a inserção de um *token* no lugar $d_g[i+1]$. Pelas convenções adotadas, isto equivale, na RPSD, à inserção do *token* d_g na célula a_i do lugar P . Fora isso, o disparo desta transição promove a remoção dos *tokens* presentes nos lugares $clr_1[i+1]_{OK}$ e $t_1\_d_gext$, além da inserção de um *token* no lugar t_1_write , habilitando a **atualização** da célula a_j , a qual receberá o valor armazenado em t_2 (no caso, d_h), após a realização das duas etapas a seguir.

e.4) Remoção do *token* presente na fileira de lugares correspondente à célula a_j .

Esta etapa é análoga à etapa **e.2**. Supondo-se que o valor armazenado na célula de armazenamento a_j seja d_s ($s = 1, 2, \dots, k$), as transições $clr_2[j+1]$ e $c_2d_s[j+1]$ serão disparadas, nesta ordem, limpando (esvaziando) a $(j+1)$ -ésima fileira de lugares (formada por $dflt[j+1]$ e $d_c[j+1]$, $c = 1, 2, \dots, k$), correspondente à célula de armazenamento a_j .

e.5) Inserção de um *token* no lugar $d_h[j+1]$.

A inserção de um *token* em $clr_2[j+1]_{OK}$ (etapa anterior) habilitará a transição $t_2\_d_h[j+1]$ e esta disparará, após w unidades de tempo, promovendo a inserção de um *token* no lugar $d_h[j+1]$, atualizando, portanto, o conteúdo da célula de armazenamento de endereço a_j . O disparo desta transição promoverá, também, a remoção dos *tokens* presentes nos lugares $clr_2[j+1]_{OK}$ e $t_2\_d_hext$, além da inserção de um *token* nos lugares r/w e t_T_0 , finalizando a atualização sobre o lugar P da RPSD, o qual poderá ser acessado por uma transição após a realização das **operações** da transição t , isto é, após o lugar **EXEC** (associado a P) receber um *token*.

A marcação da RPTM da figura 12.16 mostra que $t_1 = dflt$ (*token* em $t_1\_dflttext$), $t_2 = d_k$ (*token* em $t_2\_d_kext$), $expr_1 = a_0$ (*token* em $expr_1=a_0$) e $expr_2 = a_2$ (*token* em $expr_2=a_2$). A presença de *tokens* em t_G , **EXEC** e r/w habilita a transição t , a qual é representada, na RPTM (como decorrência dos lugares marcados), pela transição $t\_dfltd_k a_0 a_2$. Esta transição disparará entre x e y unidades de tempo, removendo os *tokens* presentes nos lugares r/w e **EXEC**, e inserindo um *token* em t_1_write , o que habilitará a transição $clr_1[1]$. O disparo desta última será imediato, devido ao intervalo $[0,0]$, e promoverá a inserção de um *token* em $clr_1[1]$, indicando que a primeira fileira de lugares, na porção da RPTM denominada **Lugar Tipo II**, deverá ser limpa (todos os lugares vazios). Na sequência, a transição $c_1d_2[1]$ ficará

habilitada e disparará imediatamente, removendo o *token* de $\mathbf{d}_2[1]$, representando, na RPSD, a limpeza do conteúdo atual da célula $\mathbf{a}_0$; além disso, será removido o *token* de $\mathbf{clr}_1[1]$ e inserido um *token* em $\mathbf{clr}_1[1]_{OK}$. O disparo seguinte, de $\mathbf{t}_1\mathbf{dfflt}[1]$, gastará $\mathbf{w}$ unidades de tempo ($\mathbf{w}$ é a *latência de escrita* de $\mathbf{P}$), e promoverá a remoção dos *tokens* presentes em $\mathbf{clr}_1[1]_{OK}$ e $\mathbf{t}_1\mathbf{dfflt}\mathbf{text}$, além da inserção de um *token* nos lugares $\mathbf{dfflt}[1]$ e $\mathbf{t}_2\mathbf{write}$. Observe que, até este ponto (fim da etapa **e.3**), terá transcorrido entre $\mathbf{x}+\mathbf{w}$ e $\mathbf{y}+\mathbf{w}$ unidades de tempo (equivalente ao intervalo de disparo de $\mathbf{t}$, adicionado ao tempo gasto na atualização da célula $\mathbf{a}_0$). A atualização da célula $\mathbf{a}_2$ é análoga, decorrente da sequência de disparo das transições $\mathbf{clr}_2[3]$, $\mathbf{c}_2\mathbf{d}_1[1]$ e $\mathbf{t}_2\mathbf{d}_k[3]$. Ao término do processo de atualização das duas células, terá transcorrido entre $\mathbf{x}+2.\mathbf{w}$ e $\mathbf{y}+2.\mathbf{w}$ unidades de tempo.

A representação, em RPTM, de conexões envolvendo *ramos de entrada preemptivos* e lugares do tipo II, pode adotar o mecanismo descrito nas etapas **e.1**, **e.2** e **e.3**, estendendo as etapas **e.1** e **e.2** para todas as células de armazenamento $\mathbf{a}_i$, $i = 1, 2, \dots, \mathbf{k}$, e fixando a etapa **e.3** unicamente à inserção do valor *default dflt* em cada célula de armazenamento. O tempo gasto para a atualização (*reset* de todas as células de armazenamento) deverá ser equivalente à latência de escrita do lugar envolvido.

Dada a representação de diferentes formas de conexão envolvendo lugares do tipo II, passemos para a representação envolvendo lugares do tipo III.

12.3.3. CONEXÕES COM LUGARES DO TIPO III

Inicialmente, vejamos as possíveis conexões envolvendo transições, ramos e lugares do tipo III.

A *remoção* de *tokens* de um lugar funcional é feita através de um *ramo de entrada coletor*, o qual permite que uma transição $\mathbf{t}$ remova um *sample* de dados (*tokens*) completo do *repositório de saída* $\mathbf{R}_O$ de um lugar $\mathbf{P}$ do tipo III. Um ramo coletor deve ser rotulado com uma *expressão coletora*

$$\mathbf{C}(\mathbf{P}, \mathbf{t}) = \langle \mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_m \rangle,$$

onde $\mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_m$ são *tokens* com *tipos* implicitamente equivalentes àqueles definidos no *domínio de saída* do lugar, isto é, sendo o *domínio de saída* de $\mathbf{P}$ dado por

$$\mathbf{D}_O(\mathbf{P}) = (\mathbf{type}_{O1}, \mathbf{type}_{O2}, \dots, \mathbf{type}_{Om}), \mathbf{m} \geq 1,$$

então

$$\mathbf{t}_1 \in \mathbf{type}_{O1}, \mathbf{t}_2 \in \mathbf{type}_{O2}, \dots, \mathbf{t}_m \in \mathbf{type}_{Om}.$$

Somente *samples* completos podem ser retirados. Ramos coletores são representados por uma linha contínua.

A *adição* de *tokens* em um lugar do tipo III é feita através de um *ramo de saída fornecedor*. Um ramo fornecedor permite que uma transição $\mathbf{t}$ **adicione** um conjunto de **argumentos** (*tokens*) a um lugar funcional $\mathbf{P}$. Tal ramo insere um conjunto de *tokens* no *repositório de entrada* do lugar quando do disparo da transição, de acordo com uma *função de inserção* $\mathbf{I}(\mathbf{P}, \mathbf{t})$, a qual segue a mesma formatação dada para *ramos de saída comuns* (utilizados com lugares do tipo I), ou seja:

$$\mathbf{I}(\mathbf{P}, \mathbf{t}) = \langle \mathbf{boolean}_1, \mathbf{set}_1 \rangle | \langle \mathbf{boolean}_2, \mathbf{set}_2 \rangle | \dots | \langle \mathbf{boolean}_n, \mathbf{set}_n \rangle, \mathbf{n} \geq 0.$$

A expressão booleana **verdadeira** de **menor índice** – *boolean_i* – é responsável pela inserção do conjunto correspondente (*set_i*), desde que os *tokens* deste conjunto sejam **compatíveis** com os **tipos** dos parâmetros de entrada do **comportamento** (definidos no **domínio de entrada** do lugar) e a capacidade do lugar não seja estourada. Caso ausente, a **função de inserção** corresponde ao conjunto de *tokens* retirados dos lugares ligados à transição considerada por **ramos de entrada comuns, conjugados, coletores ou de leitura**. Ramos fornecedores são representados por linha contínua.

Os três tipos de ramos para o **tratamento de exceções** podem ser utilizados por transições conectadas a lugares do tipo III, da seguinte forma:

- a) **Ramo de entrada suspensivo**: Um ramo deste tipo permite que uma transição **suspenda temporariamente** as **instâncias ativas** do **comportamento** associado a um lugar do tipo III. Caso o comportamento do lugar seja descrito por uma **RPSD-T**, as **instâncias** em execução serão suspensas através da **impossibilidade de habilitação** de suas transições (há uma **variável de controle ENABLE** implicitamente associada a cada transição, cujo valor passa para **FALSE** quando da atuação de um ramo suspensivo). No caso do comportamento ser descrito por uma **rotina**, cada **instância** termina de executar o *statement* em processamento e fica **suspensa**. Ramos suspensivos são representados por uma linha tracejada fina;
- b) **Ramo de entrada preemptivo**: Quando relacionado a um lugar do tipo III, um ramo deste tipo permite que uma transição **remova**, independentemente de quaisquer condições, **todos** os *tokens* do lugar, incluindo aqueles presentes nos **repositórios de entrada e de saída**, além dos *tokens* presentes nas **instâncias de comportamento**, caso este seja descrito por uma **RPSD-T**. Este tipo de ramo é representado por uma linha tracejada grossa;
- c) **Ramo de saída restaurador**: Um ramo deste tipo permite que uma transição **retome** um **comportamento** descrito por um lugar do tipo III, que tenha sido **suspenso** por outra transição ligada ao lugar por um ramo *suspensivo*. Caso o comportamento do lugar seja descrito por uma **RPSD-T**, as **instâncias suspensas voltam a executar**, através da **não interferência** nas regras de habilitação de suas transições (valor da **variável de controle ENABLE** passa para **TRUE**). Caso o comportamento seja descrito por uma **rotina**, cada **instância** retoma sua execução no *statement seguinte* àquele em que o processamento havia sido suspenso. É representado por uma linha tracejada fina.

Um **ramo fornecedor** tem por **regra de habilitação** que a **cardinalidade** do conjunto de *tokens* resultante da **função de inserção de tokens** seja **menor ou igual** à quantidade de **células livres (disponíveis)** no **repositório de entrada (R_I)** do lugar associado, além dos tipos destes *tokens* serem compatíveis com o **domínio de entrada (D_I)** do lugar.

A **regra de habilitação** de um **ramo coletor** possui duas condições: primeira, deve existir pelo menos um *sample* de *tokens* no **repositório de saída (R_O)** do lugar relacionado; segunda, a quantidade de *tokens* da **expressão de remoção** deve ser igual à cardinalidade do **domínio de saída (D_O)** do lugar.

Para lugares do tipo III, a **regra de habilitação** referente a um **ramo preemptivo** é que o respectivo lugar possua ao menos um *token*. Caso a transição relacionada dispare, **todos** os *tokens* do lugar serão **removidos**. Isto significa que **instâncias** de computação associadas a um lugar do tipo III poderão ser **abortadas** a qualquer momento, mesmo antes de sua execução, quando argumentos ainda estão sendo recebidos.

A **regra de habilitação** de um **ramo suspensivo** é que o lugar funcional possua ao menos uma **instância** em execução.

Complementando o ramo suspensivo, tem-se o **ramo restaurador**. Este **não interfere** na **habilitação** da transição à qual está ligado na RPSD, servindo apenas para **retomar** a execução de **instâncias** do **comportamento** de um lugar funcional, que possam **eventualmente** ter sido suspensas pelo disparo de uma transição ligada ao lugar por um ramo suspensivo.

A figura 12.17 mostra o lugar funcional **P** da figura 12.12, desta vez conectado a uma transição **t** por um **ramo de entrada coletor**, o qual é rotulado com a **expressão coletora** $\langle t_1, t_2 \rangle$, denotando que os **resultados** produzidos por **P** (**sample de saída**) serão **associados** aos **tokens** identificados por t_1 e t_2 , quando do **disparo** de **t**. A **guarda** de **t** é $[w = m]$, e seu **intervalo de disparo** é generalizado por $[r, s]$. A única **operação** de **t** é o incremento da variável **sample** em uma unidade.

As partes **b)** a **g)** da figura 12.17 mostram parte da representação de equivalência, em RPTM, para a conexão da parte **a)**, adotando-se, para a representação dos **repositórios de entrada** (R_I) e **de saída** (R_O) de **P**, as **convenções** dadas na seção 12.2.3. A figura 12.17 deve ser complementada com os grupos de transições $RunS_j B_i(a_x; b_y)$ e $StoreB_i O_j(c_x; d_y)$, presentes nas partes **c)** e **e)** da figura 12.12. Com esta complementação (omitida para não sobrecarregar a figura 12.17), observe que a figura 12.12 está contida na figura 12.17. Desta forma, a descrição a seguir restringe-se às etapas envolvidas no disparo da transição **t**, considerando-se que os mecanismos de **associação sample/instância** (figura 12.12.c) e de **transferência de sample transformado/repositório de saída** (figura 12.12.e), já estejam consolidados para o leitor.

A transição **t** somente poderá disparar para remover **samples de saída completos** de **P**. Isto significa que os dois resultados, a serem armazenados nos **contêineres de token** t_1 e t_2 , deverão estar **disponíveis** para remoção. O **token** t_1 é indicado pelo conjunto de lugares c_0ext e c_1ext (o lugar com **token** indica o valor de t_1), enquanto t_2 é indicado pelo conjunto de lugares d_0ext e d_1ext . Os **tokens** t_1 e t_2 poderão ser removidos da linha **j** da matriz de lugares representante de $R_O(P)$ se, e somente se, existir um **token** no lugar $rem[j]$ e não existir um **token** num lugar $rem[k]$, tal que $k < j$. Por sua vez, $rem[j]$ possuirá um **token** somente após o disparo de uma das transições $StoreB_i O_j(c_x; d_y)$, conforme mostrado na descrição da figura 12.12. De acordo com as convenções dadas na seção 12.2.3, quando a instância do comportamento $B(P)$, associada à linha **i**, produzir um **token** do **tipo C**, será inserido um **token** no lugar RPTM correspondente ao valor produzido ($c_0[i,1]$ ou $c_1[i,1]$); procedimento análogo é adotado quando da produção de um **token** do **tipo D**. Os lugares $rem[j]$ e $Nrem[j]$ ($j = 1, 2, 3$) são utilizados para controlar a **prioridade** de remoção do **repositório de saída** $R_O(P)$: **tokens** presentes na linha 1 têm prioridade de remoção sobre **tokens** presentes nas linhas 2 e 3, ao passo que, **tokens** presentes na linha 2 têm prioridade de remoção sobre os da linha 3. A marcação dos lugares $rem[j]$, $Nrem[j]$, $ExecS_j B_i$, $B_iActive$, $Sample[j]$, $c_x[j,1]$ e $d_y[j,2]$ determinará qual transição $t_{B_i O_j}(c_x; d_y)$ ($i, j = 1, 2, 3$ e $x, y = 0, 1$) irá disparar, sinalizando o início do processo que resultará nas seguintes ações: remoção de **tokens** de uma linha **m** de $R_O(P)$ ($m = 1, 2, 3$), atribuindo valores aos **contêineres** t_1 e t_2 ; deslocamento dos valores contidos nas linhas de índice superior a **m** (se houver), uma posição para baixo, em ambos repositórios $R_I(P)$ e $R_O(P)$, representando o recálculo da função dinâmica **L** (descrita na seção 11.2.1.1.3); restauro do estado inicial da instância B_i ; e liberação da instância B_i (B_i **inativa**). O detalhamento deste processo é dado na sequência.

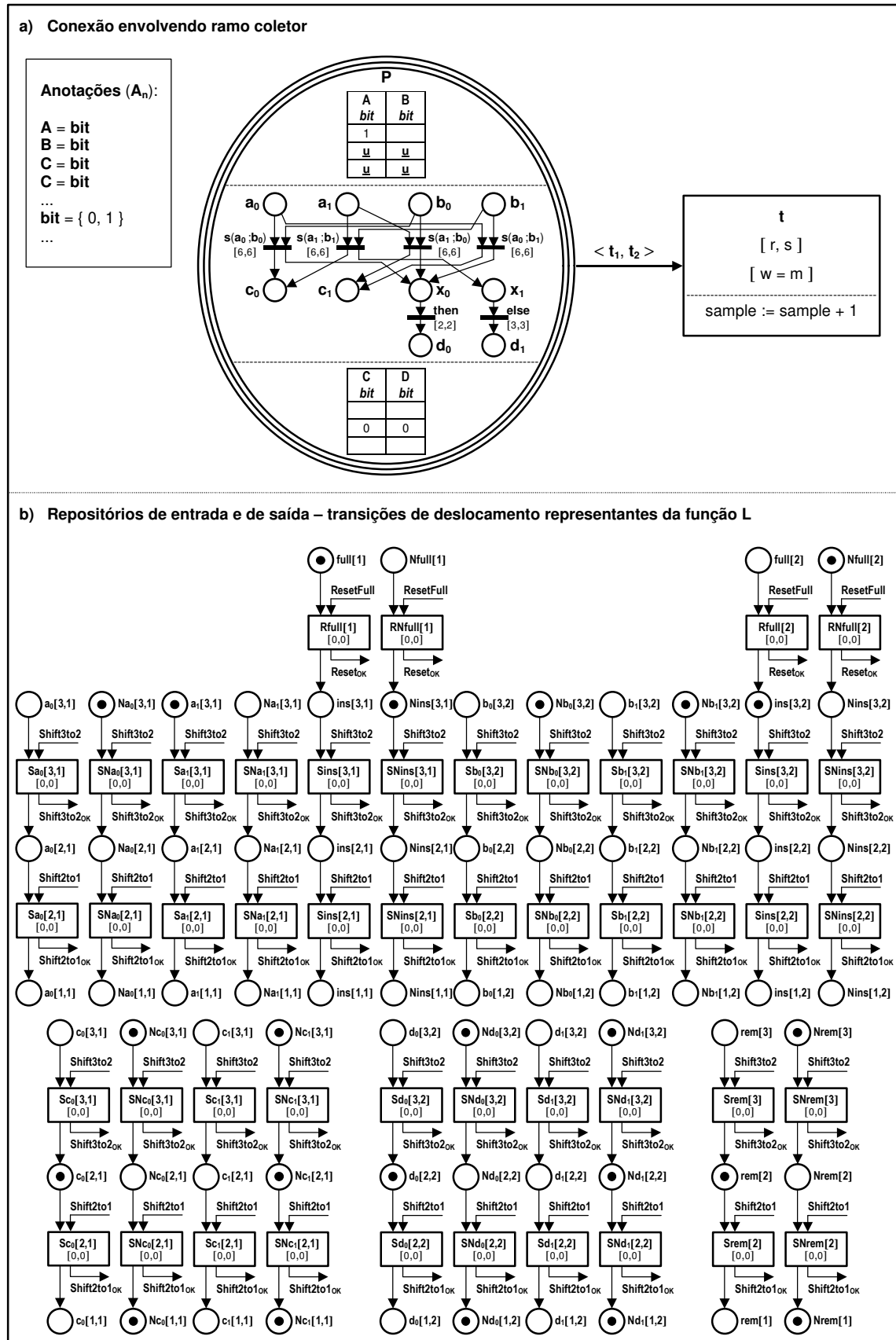
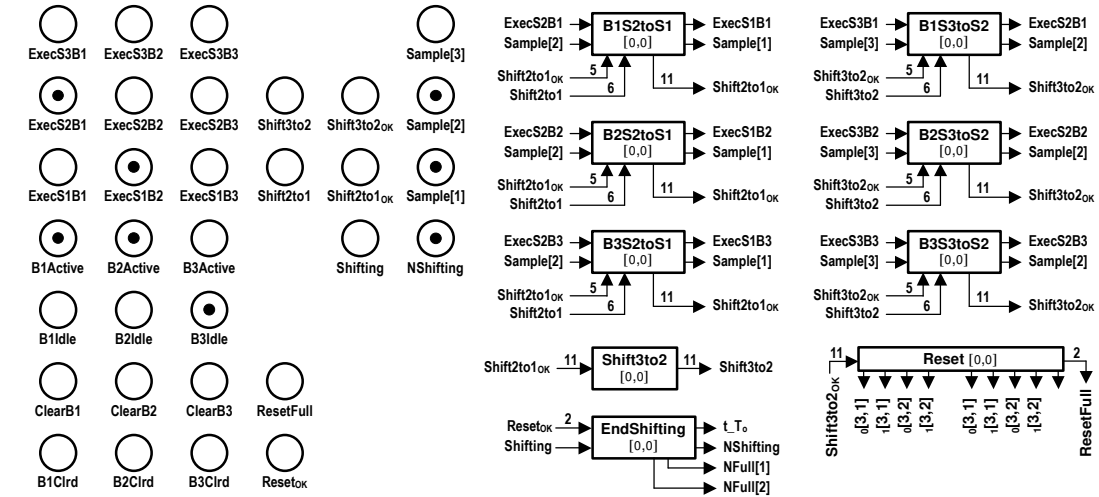
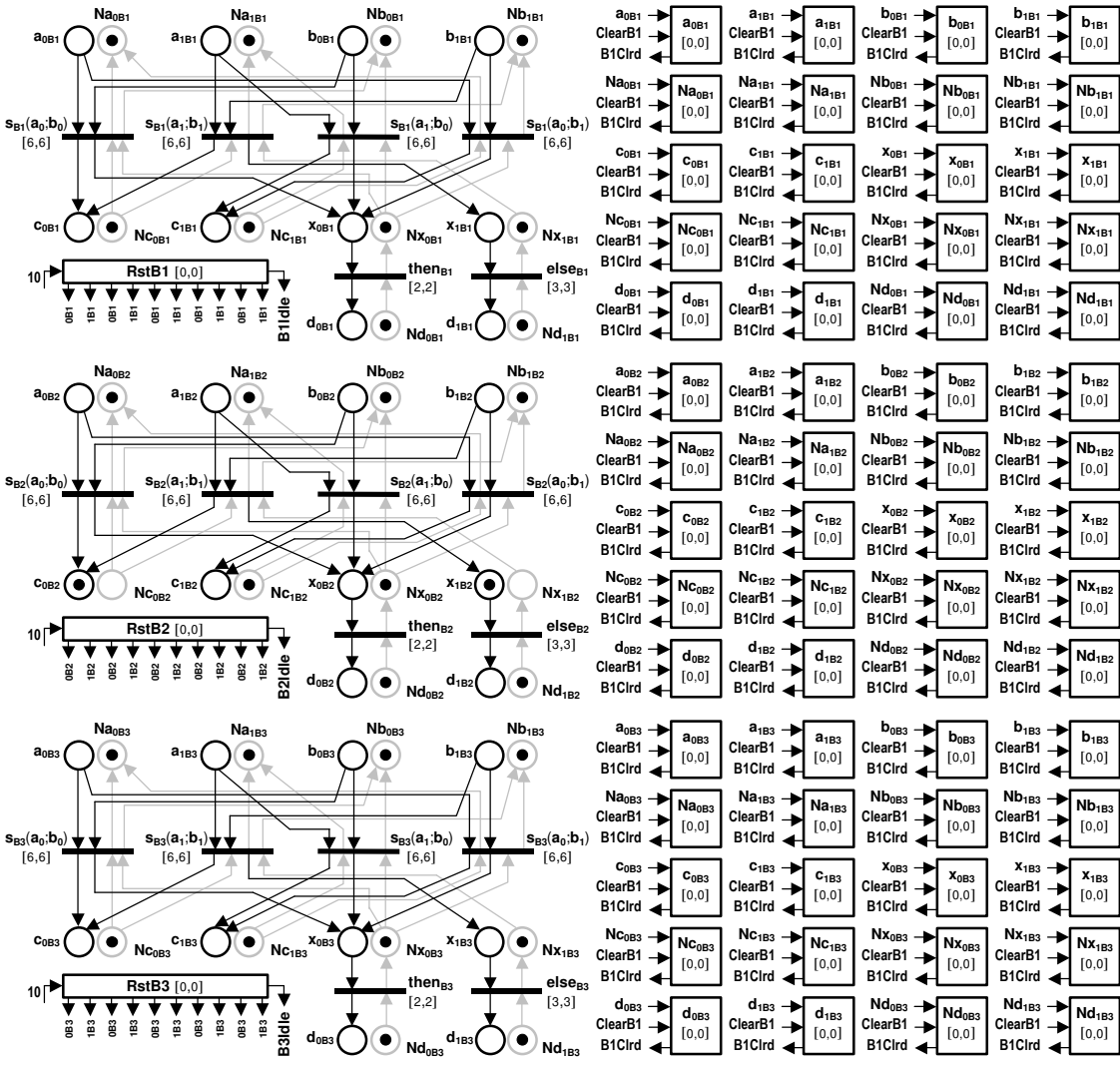


Figura 12.17. RPTM descrevendo uma conexão com ramo coletor.

c) Mecanismo de associação *sample*/instância – mecanismo de deslocamento da função L

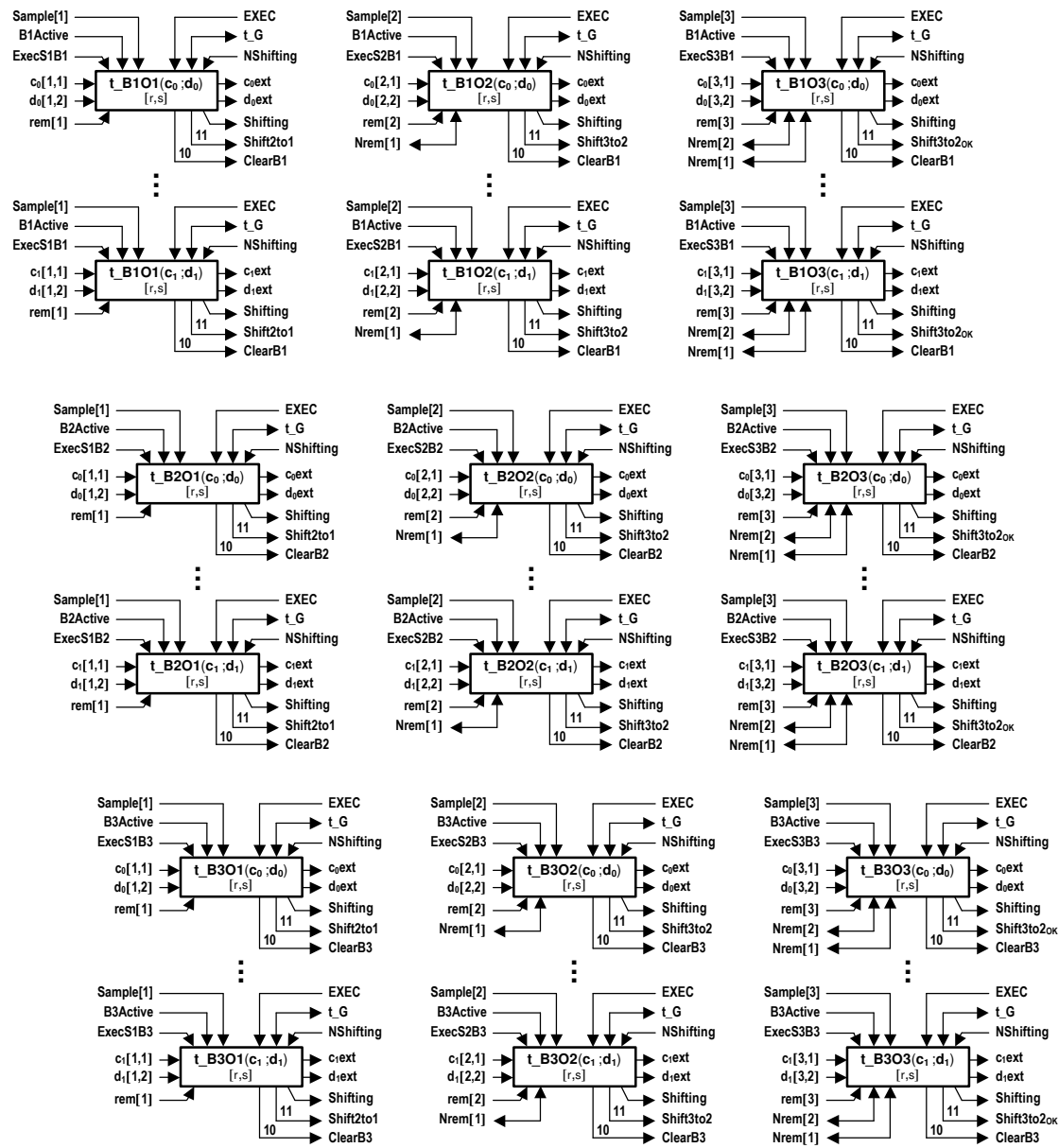


d) Instâncias do comportamento – transições para o restauro do estado original

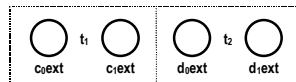


Continuação da figura 12.17.

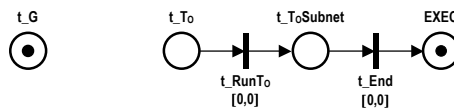
e) Transição t



f) Lugares RPTM associados, nas redes equivalentes, a um ou mais lugares RPSD, de qualquer tipo, receptores de *tokens* provenientes da transição t



g) Memória da rede



Continuação da figura 12.17.

A transição t da RPSD da figura 12.17.a é representada, na RPTM, pelo conjunto de transições $t\_B_iO_j(c_x;d_y)$ ($i, j = 1, 2, 3$ e $x, y = 0, 1$), presente na figura 12.17.e. O disparo de uma transição $t\_B_iO_j(c_x;d_y)$ corresponde à remoção do *sample* presente na j -ésima linha do *repositório de saída*, *sample* este que foi transformado (processado) pela *instância* B_i , e é composto pelos resultados c_x e d_y . Considerando-se as condições de habilitação, o conjunto de transições $t\_B_iO_j(c_x;d_y)$ pode ser dividido em três grupos, conforme a linha ocupada pelo *sample* a ser removido (1, 2 ou 3). As condições de habilitação, válidas para os três grupos que compõem o conjunto de transições $t\_B_iO_j(c_x;d_y)$, são:

- lugares t_G e **EXEC** (associado a P) marcados (localizados na figura 12.17.g), denotando que a *guarda da transição* definida para t é **verdadeira** e que nenhuma outra transição conectada a P , se existir, está em execução;
- lugar **NShifting** (figura 12.17.c) marcado, denotando que a **operação de deslocamento de tokens**, nos *repositórios de entrada e de saída*, **não** está sendo realizada (esta operação encontra-se descrita adiante);
- lugares **Sample[j]**, **B_iActive** e **ExecS_jB_i** (figura 12.17.c) marcados, indicando, conforme explanação da figura 12.12, que a *instância* B_i está amarrada ao *sample* S_j (*sample* relacionado às linhas de índice j em ambos os *repositórios*); e
- lugares $c_x[j,1]$, $d_y[j,2]$ (figura 12.17.d) e **rem[j]** (figura 12.17.b) marcados, indicando que o *sample* S_j encontra-se totalmente processado (S_j é formado pelos valores c_x e d_y), podendo, portanto, ser removido.

Para as transições associadas ao *sample* S_1 , quais sejam, as do grupo $t\_B_iO_1(c_x;d_y)$, as condições acima são suficientes. Para as transições $t\_B_iO_2(c_x;d_y)$, relativas ao *sample* S_2 , deve-se adicionar, à lista acima, a presença de um *token* no lugar **Nrem[1]**, denotando que a linha 1 do *repositório de saída* não está disponível para remoção (*sample* S_1 não processado totalmente). De forma análoga, as transições $t\_B_iO_3(c_x;d_y)$, associadas ao *sample* S_3 , necessitam que os lugares **Nrem[1]** e **Nrem[2]** estejam marcados.

O disparo de uma transição habilitada $t\_B_iO_j(c_x;d_y)$, dentro do intervalo $[r,s]$ (igual ao de t), tem como efeito:

- remoção do *token* presente no lugar **EXEC** associado a P , denotando que t passa a ter acesso exclusivo a P ;
- remoção do *token* presente em **NShifting** e inserção de um *token* em **Shifting**, viabilizando o início da operação de deslocamento de *tokens* nos repositórios (veja adiante);
- inserção de 11 *tokens* em **Shift2to1** (caso $j = 1$), **Shift3to2** (caso $j = 2$) ou **Shift3to2OK** (caso $j = 3$), determinando a partir de qual linha deverá ser realizada a operação de deslocamento de *tokens* (por construção, de acordo com as convenções adotadas, 11 lugares por linha deverão sofrer deslocamento de *tokens*);
- remoção dos *tokens* presentes em **Sample[j]**, **B_iActive** e **ExecS_jB_i**, para que a função dinâmica L possa ser recalculada (veja adiante);
- remoção dos *tokens* presentes em $c_x[j,1]$, $Nc_{Nx}[j,1]$, $d_y[j,2]$, $Nd_{Ny}[j,2]$ e **rem[j]** – onde Nx e Ny denotam o complemento de x e de y (os índices x e y podem assumir os valores 0 ou 1), **esvaziando** a j -ésima linha do *repositório de saída* (obs.: a j -ésima linha do *repositório de entrada* já se encontra **vazia**);
- inserção de um *token* em $c_{x,ext}$ e em $d_{y,ext}$, lugares RPTM referentes, respectivamente, aos **contêineres** t_1 e t_2 da RPSD (representados na figura 12.17.f); e
- inserção de 10 *tokens* no lugar **ClearB_i**, para possibilitar o restauro da marcação *default* da rede representante da *instância* B_i , definida pela presença de um *token* em cada um dos lugares **negativos** (10 lugares no total, para cada *instância*).

Ao alterar a marcação de determinados lugares da RPTM, o disparo de uma transição $t_{B_iO_j}(c_x;d_y)$ desencadeia um processo de transformações sobre a rede (em tempo zero), o qual pode ser descrito pelas etapas a seguir, as quais podem ser executadas em qualquer ordem, inclusive com intercalações, sem qualquer prejuízo para o resultado final:

e.1) Operação de deslocamento de *tokens* / recálculo da função dinâmica L .

As convenções adotadas na seção 12.2.3 estabelecem que, após a remoção de um *sample de tokens* da j -ésima linha do *repositório de saída*, todas as linhas de índice superior, nos *repositório de entrada* e de *saída*, terão seus *tokens* (representando, respectivamente, argumentos e resultados) **deslocados** uma posição para baixo (índice decrescente de linha).

Neste contexto, o disparo de $t_{B_iO_j}(c_x;d_y)$ promoverá a inserção de 11 *tokens* em **Shift2to1** (se $j = 1$), **Shift3to2** (se $j = 2$) ou **Shift3to2OK** (se $j = 3$), determinando a partir de qual linha deverá ser realizada a **operação de deslocamento de tokens**. Os lugares mencionados, juntamente com os lugares **Shift2to1OK**, **ResetFull** e **ResetFullOK**, são denominados *lugares de controle de deslocamento*. Pela construção da RPTM, observa-se que, independentemente do estado da rede, cada linha do *repositório de saída* terá sempre 5 lugares marcados. Por outro lado, cada linha do *repositório de entrada* terá ou 6 (se **disponível**) ou 0 (se **indisponível**) lugares marcados (linhas **indisponíveis** têm **todos** os seus lugares **vazios**). Considerando-se os dois *repositórios*, como a soma de lugares marcados em uma linha será 5 ou 11, o total de *tokens* inseridos no lugar de controle de deslocamento utilizado, quando do disparo de $t_{B_iO_j}(c_x;d_y)$, será 11 (o número máximo de lugares marcados), independentemente da $(j+1)$ -ésima linha do *repositório de entrada* (se $j < max_S$) estar **disponível** ou **não** (neste caso, conforme será visto adiante, há transições de ajuste cujo disparo promove a remoção dos 6 *tokens* excedentes). Conforme já mencionado, após o disparo de $t_{B_iO_j}(c_x;d_y)$, a j -ésima linha, em ambos os repositórios, encontrar-se-á **vazia** (viabilizando tal linha receber a configuração da $(j+1)$ -ésima linha, através de um simples deslocamento vertical de *tokens* entre os lugares correspondentes).

Para tratarmos das operações de deslocamento, considere as seguintes ações:

- a.1)** deslocar verticalmente os *tokens* presentes nos lugares representantes da linha 2, em ambos os *repositórios*, para os lugares representantes da linha 1;
- a.2)** deslocar verticalmente os *tokens* presentes nos lugares representantes da linha 3, em ambos os *repositórios*, para os lugares representantes da linha 2; e
- a.3)** restaurar, à sua marcação *default*, os lugares correspondentes à linha 3, assim como os lugares para controle de coluna cheia (**full**[1], **Nfull**[1], **full**[2], **Nfull**[2]), de acordo com as convenções adotadas na seção 12.2.3 (lugares **positivos** vazios e lugares **negativos** marcados);

Considerando-se o valor de j , de acordo com a transição representante de t disparada, há três possibilidades de operações de deslocamento:

Caso I ($j = 1$). Neste caso, deverão ser executadas as ações **a.1**, **a.2** e **a.3**, nesta ordem;

Caso II ($j = 2$). Neste caso, deverão ser executadas as ações **a.2** e **a.3**, nesta ordem; e

Caso III ($j = 3$). Neste caso, por não haver linhas situadas acima da terceira, deverá ser executada somente a ação **a.3**.

As considerações a seguir servem para redes cujos repositórios possuem mais de três linhas. De forma geral, dado um lugar RPSD P , com $max_S(P)$ linhas em seus repositórios, o valor de j estará entre 1 e $max_S(P)$. Disso resulta que a **operação de deslocamento de tokens** será composta por $max_S(P)$ ações, onde as primeiras $(max_S(P)-1)$ ações serão análogas à ação **a.1**, de deslocamento, e a última ação será análoga à ação **a.3**, de restauro da última linha e do controle de coluna cheia.

Restringindo-se à RPTM da figura 12.17, qualquer **operação de deslocamento de tokens** pode ser definida em função das ações **a.1**, **a.2** e **a.3**. Desta forma, vejamos como cada uma destas é executada.

A ação **a.1** inicia-se com o disparo das transições **Srótulo_lugar₂** habilitadas, onde **rótulo_lugar₂** corresponde a um lugar representante da linha 2 de um dos repositórios. Na figura 12.17.b, são exemplos de **rótulo_lugar₂**, os lugares **a₀[2,1]**, **Nins[2,2]**, **c₁[2,1]** e **Nd₀[2,2]**; observe-se, na mesma figura, que a cada lugar **rótulo_lugar₂**, encontra-se conectada uma **transição de deslocamento Srótulo_lugar₂**, como, por exemplo, as transições **Sa₀[2,1]**, **SNins[2,2]**, **Sc₁[2,1]** e **SNd₀[2,2]**. O disparo de uma transição **Srótulo_lugar₂** promoverá a remoção do **token** presente em **rótulo_lugar₂**, a inserção de um **token** no lugar **rótulo_lugar₁** (situado, na figura 12.17.b, imediatamente abaixo de **rótulo_lugar₂**), a remoção de um **token** de **Shift2to1** e a inserção de um **token** em **Shift2to1_{OK}**. Se a linha 2 do **repositório de entrada** estiver **indisponível**, o número total de transições **Srótulo_lugar₂** disparadas será 5; caso contrário (linha **disponível**), será 11. Disso resulta que, após o disparo de todas as transições **Srótulo_lugar₂** habilitadas, o número de **tokens** em **Shift2to1_{OK}** será 5 ou 11, assim como o número de **tokens** em **Shift2to1** será 6 ou 0. Desta forma, considera-se duas possíveis ações subsequentes:

- a.1.1)** caso os lugares **Shift2to1_{OK}** e **Shift2to1** possuam, respectivamente, 5 e 6 **tokens**, deverá ocorrer a remoção dos 6 **tokens** excedentes de **Shift2to1**, repassando-os para **Shift2to1_{OK}** (para encerrar o processo de deslocamento de **tokens** nos repositórios). Além disso, deverá ocorrer o recálculo da função **L**, refletindo o deslocamento da linha 2 para a linha 1 (como, logo após o disparo de **t_{B_iO_j}(c_x;d_y)**, a linha 2 estava **indisponível**, há uma instância associada a ela que deverá ficar associada à linha 1). Por último, deverá ocorrer a habilitação da ação **a.2**; ou
- a.1.2)** caso os lugares **Shift2to1_{OK}** e **Shift2to1** possuam, respectivamente, 11 e 0 **tokens**, deverá ocorrer somente a habilitação da ação **a.2**.

A ação **a.1.1** será executada através do disparo das transições **B_iS2toS1** e **Shift3to2** (figura 12.17.c), onde **i** é o mesmo índice da transição representante de **t** disparada, isto é, **t_{B_iO_j}(c_x;d_y)**. Já a ação **a.1.2** será executada apenas com o disparo de **Shift3to2**. A transição **B_iS2toS1** é responsável pela transferência dos **tokens** presentes em **ExecS₂B_i** e **Sample[2]** para **ExecS₁B_i** e **Sample[1]** (significando que a instância **B_i**, antes associada à linha 2, passa a associar-se à linha 1), pela remoção dos 6 **tokens** excedentes em **Shift2to1** e, também, pela atualização do lugar **Shift2to1_{OK}**, o qual passa de 5 para 11 **tokens**. A presença de 11 **tokens** em **Shift2to1_{OK}** habilitará a transição **Shift3to2**, a qual disparará imediatamente, removendo os 11 **tokens** deste lugar, repassando-os para o lugar **Shift3to2**, iniciando, desta forma, a ação **a.2**, através da habilitação das transições **Srótulo_lugar₃**, onde **rótulo_lugar₃** corresponde a um lugar representante da linha 3 de um dos repositórios (figura 12.17.b).

Tal como a ação **a.1**, a ação **a.2** inicia-se com o disparo das transições **Srótulo_lugar₃** que estiverem habilitadas. Segue-se comportamento análogo ao descrito para a ação **a.1**, substituindo-se as referências à linha 2 por 3 e as referências à linha 1 por 2. Desta forma, são equivalentes os seguintes pares de lugares:

- a) Shift2to1** e **Shift3to2**;
- b) Shift2to1_{OK}** e **Shift3to2_{OK}**;
- c) ExecS₂B_i** e **ExecS₃B_i**;
- d) ExecS₁B_i** e **ExecS₂B_i**;
- e) Sample[2]** e **Sample[3]**; e
- f) Sample[1]** e **Sample[2]**.

Da mesma forma, a descrição dada para a ação **a.1** é análoga à descrição da ação **a.2**, considerando-se equivalentes os seguintes pares de transições:

- a) **Srótulo_lugar₂** e **Srótulo_lugar₃**; e
- b) **B_iS2toS1** e **B_iS3toS2**.

Caso existisse mais linhas nos repositórios da figura 12.17.b, a transição **Shift3to2** teria como equivalente uma transição **Shift4to3**. Contudo, como a linha 3 é a última, não existe tal transição. De qualquer forma, o disparo de **Shift3to2** inserirá 11 *tokens* em **Shift3to2OK**, os quais necessitarão ser removidos para o encerramento da ação **a.2** e habilitação da ação **a.3**.

Após o deslocamento da última linha, a transição **Reset** (figura 12.17.c) ficará habilitada e disparará imediatamente, removendo todos os *tokens* do **lugar de controle de deslocamento** mais recentemente utilizado (**Shift3to2OK**, no caso), de forma a encerrar a ação corrente (**a.2**). Ao mesmo tempo, o disparo de **Reset** inicia a **última ação** referente à **operação de deslocamento de tokens** (**a.3**), através da inserção de *tokens* nos lugares **negativos** da última linha (em ambos os repositórios), representando o restauro destas linhas à sua marcação *default* (exceto os lugares **Nins[3,1]** e **Nins[3,2]**), assim como da inserção de 2 *tokens* no lugar **ResetFull**, necessários à correta configuração dos lugares utilizados para controle de coluna cheia. A inserção de 2 *tokens* justifica-se pela existência de duas colunas (representando os parâmetros **A** e **B**), considerando-se que, em cada uma delas, somente um dos lugares **rótulo_lugar** estará marcado. Conseqüentemente, somente duas das quatro transições **Rrótulo_lugar** ficarão habilitadas. A presença de um *token* em **full[1]** (**full[2]**) habilitará a transição **Rfull[1]** (**Rfull[2]**). Observe que o disparo de **Rfull[1]** (**Rfull[2]**) promoverá o deslocamento do *token* presente em **full[1]** (**full[2]**) para **ins[3,1]** (**ins[3,2]**), significando que a **coluna** correspondente, outrora cheia, passa a admitir a inserção de um novo **argumento** em sua terceira linha. Por outro lado, a presença de um *token* em **Nfull[1]** (**Nfull[2]**) habilitará a transição **RNfull[1]** (**RNfull[2]**), cujo disparo promoverá o deslocamento do *token* presente em **Nfull[1]** (**Nfull[2]**) para **Nins[3,1]** (**Nins[3,2]**), apenas para o propósito de completar o restauro da terceira linha do **repositório de entrada** à sua marcação *default* (restauro este iniciado com o disparo da transição **Reset**). Independentemente da transição disparada (**Rfull[1]**, **RNfull[1]**, **Rfull[2]** ou **RNfull[2]**), haverá a necessidade inserir um *token* nos lugares **Nfull[1]** e **Nfull[2]**, uma vez que as colunas não encontrar-se-ão mais cheias. Isto será feito através do disparo da transição **EndShifting** (figura 12.17.c), a qual ficará habilitada quando os 2 *tokens* do lugar **ResetFull** passarem para o lugar **ResetOK** (em decorrência do disparo das duas transições **Rrótulo_lugar** habilitadas). O disparo de **EndShifting** promoverá, também, a transferência do *token* presente em **Shifting** para **NShifting** – indicando o término da **operação de deslocamento de tokens**, assim como a inserção de um *token* em **t_TO** – sinalizando, para a **memória da rede** (figura 12.17.g), realizar as operações relacionadas à transição **t** da RPSD.

e.2) Restauro do estado original e liberação da instância **B_i**.

O disparo da transição **t_BiOj(c_x;d_y)** promoverá a inserção de 10 *tokens* em **ClearB_i**, determinando que a RPTM representante da instância **B_i** deverá ser restaurada à sua marcação *default*, a qual é definida pela presença de um *token* em cada um de seus lugares **negativos**. No caso do **comportamento B(P)** adotado na RPSD da figura 12.17, após uma instância **B_i** repassar os resultados produzidos para o **repositório de saída**, todos os lugares **positivos** da RPTM correspondente a **B_i** ficarão vazios (e os duais correspondentes, marcados). Como esta característica está intrinsecamente relacionada à rede RPTM utilizada para representar o **comportamento B(P)**, a descrição a seguir trata

de casos gerais, nos quais quaisquer lugares (**positivos** ou **negativos**), da RPTM representante de B_i , poderão permanecer marcados, mesmo após a transferência de resultados para o *repositório de saída* (respeitando-se, evidentemente, as convenções de dualidade adotadas, as quais impedem a marcação simultânea de um lugar e de seu dual).

Neste contexto, visando o restauro da marcação *default*, como há 10 pares de duais na rede representante de uma instância B_i , faz-se necessária a remoção dos 10 *tokens* presentes na rede, independentemente de estarem presentes em lugares **positivos** ou **negativos**. Para controlar esta remoção, para cada *token* removido da rede, há a transferência de um *token* do *lugar de controle de restauro* $ClearB_i$ para B_iClrd . Ao término do processo de remoção, sinalizado pela presença de 10 *tokens* em B_iClrd , será feita a inserção de um *token* em cada um dos lugares **negativos** da rede associada a B_i .

Na figura 12.17.d, para cada lugar *rótulo_lugar* B_i , pertencente à rede representante da instância B_i , há uma transição de mesmo nome (*rótulo_lugar* B_i), a qual ficará habilitada quando da presença de um *token* no lugar *rótulo_lugar* B_i e da presença de, pelo menos, um *token* em $ClearB_i$. O disparo de uma transição *rótulo_lugar* B_i será responsável pela remoção do *token* presente no lugar *rótulo_lugar* B_i , assim como pela transferência de um *token* de $ClearB_i$ para B_iClrd . Após o disparo das transições habilitadas *rótulo_lugar* B_i , o lugar B_iClrd conterá 10 *tokens*, habilitando a transição $RstB_i$. De disparo imediato, $RstB_i$ promoverá a remoção dos 10 *tokens* presentes em B_iClrd , o restauro, à marcação *default*, da rede representante de B_i (todos os seus lugares **negativos** marcados) e, também, a **desativação** da instância B_i , através da inserção de um *token* no lugar B_iIdle , liberando a instância para a recepção e o processamento de um novo *sample*.

Seja $t_{B_iO_j(c_x;d_y)_1}$ a transição representante de t cujo disparo habilitou a execução das etapas **e.1** e **e.2**. Para que uma nova transição $t_{B_iO_j(c_x;d_y)_2}$ fique habilitada, será necessário, dentre as condições já analisadas, que a etapa **e.1**, desencadeada por $t_{B_iO_j(c_x;d_y)_1}$, já esteja encerrada (*token* em $NShifting$), assim como a transição $t_{B_iO_j(c_x;d_y)_1}$ não tenha mais **acesso exclusivo** a P (lugar **EXEC**, associado a P na **memória da rede**, deverá estar marcado, sinalizando que as **operações** da transição t já foram encerradas).

Na marcação mostrada na figura 12.17 (considere todas as partes da figura), a transição t está habilitada, pois a linha 2 do *repositório de saída* está completa (*token* em $rem[2]$), a guarda de t é *verdadeira* (*token* em t_G) e o lugar P não está sendo processado, nem por t (*token* em $NShifting$) e nem por uma outra transição (*token* em **EXEC**). Seu disparo ocorrerá entre r e s unidades de tempo, após sua habilitação. O disparo de t corresponderá, na RPTM, ao disparo de $t_{B_iO_2(c_0;d_0)}$, o qual promoverá:

- remoção do *token* presente em **EXEC** associado a P (t passa a ter acesso exclusivo a P);
- remoção do *token* presente em $NShifting$ e inserção de um *token* em **Shifting**, viabilizando o início da operação de deslocamento de *tokens*;
- inserção de 11 *tokens* em **Shift3to2**, indicando que a operação de deslocamento de *tokens* será iniciada pela linha 2;
- remoção dos *tokens* presentes em **Sample[2]**, $B_1Active$ e $ExecS_2B_1$, para que a função dinâmica L possa ser recalculada;
- remoção dos *tokens* presentes em $c_0[2,1]$, $Nc_1[2,1]$, $d_0[2,2]$, $Nd_1[2,2]$ e $rem[2]$, esvaziando a linha 2 do *repositório de saída*;
- inserção de um *token* em c_0ext e em d_0ext , determinando, respectivamente, os valores dos **contêineres de token** t_1 e t_2 da RPSD; e
- inserção de 10 *tokens* no lugar $ClearB_1$, para possibilitar o restauro da marcação *default* da rede representante da instância B_1 .

O disparo de $t\_B_1O_2(c_0;d_0)$ desencadeará a execução das etapas **e.1** e **e.2**, em qualquer ordem, consumindo, para tal, zero unidades de tempo.

A etapa **e.1** consistirá das ações **a.2** e **a.3**, uma vez que $j = 2$. A ação **a.2** inicia-se com o disparo das transições $Sc_0[3,1]$, $Snc_1[3,1]$, $Sd_0[3,2]$, $Snd_1[3,2]$ e $Srem[3]$, deslocando, respectivamente, os *tokens* presentes em $c_0[3,1]$, $nc_1[3,1]$, $d_0[3,2]$, $nd_1[3,2]$ e $rem[3]$ para $c_0[2,1]$, $nc_1[2,1]$, $d_0[2,2]$, $nd_1[2,2]$ e $rem[2]$ (observe que nenhuma transição *Srótulo_lugar₃*, associada à terceira linha do *repositório de entrada*, ficou habilitada, devido à tal linha estar **indisponível**). Após esta sequência de disparo de transições, os lugares **Shift3to2OK** e **Shift3to2** possuirão, respectivamente, 5 e 6 *tokens*, habilitando, desta forma, a ação **a.2.1**. A ação **a.2.1** inicia-se com o disparo da transição **B₁S3toS2**, transferindo os *tokens* presentes em **ExecS₃B₁** e **Sample[3]** para **ExecS₂B₁** e **Sample[2]**. Isto significa que, a instância **B₁**, antes associada à linha 3, passa a associar-se à linha 2. O disparo da transição **B₁S3toS2** será responsável, também, pela remoção dos 6 *tokens* excedentes em **Shift3to2** e pela atualização do lugar **Shift3to2OK**, o qual passará de 5 para 11 *tokens*. A presença de 11 *tokens* em **Shift3to2OK** habilitará a transição **Reset**, a qual disparará imediatamente, removendo os 11 *tokens* deste lugar, além de inserir *tokens* nos lugares **negativos** da terceira linha, em ambos os repositórios, iniciando, assim, a ação **a.3**. Os lugares negativos que receberão *tokens* são **Na₀[3,1]**, **Na₁[3,1]**, **Nb₀[3,2]**, **Nb₁[3,2]**, **Nc₀[3,1]**, **Nc₁[3,1]**, **Nd₀[3,2]**, **Nd₁[3,2]** e **Nrem[3]**. O disparo de **Reset** inserirá, também, 2 *tokens* no lugar **ResetFull**, habilitando, assim, as transições **Rfull[1]** e **RNfull[2]**. O disparo de **Rfull[1]** promoverá o deslocamento do *token* presente em **full[1]** para **ins[3,1]**, assim como a transferência de um *token* de **ResetFull** para **ResetOK**. Por outro lado, o disparo de **RNfull[2]** promoverá o deslocamento do *token* presente em **Nfull[2]** para **Nins[3,2]**, assim como a transferência de um *token* de **ResetFull** para **ResetOK**. Após o disparo de **Rfull[1]** e **RNfull[2]**, o lugar **ResetOK** possuirá 2 *tokens*, habilitando, assim, a transição **EndShifting**. O disparo de **EndShifting** promoverá a inserção de um *token* nos lugares **Nfull[1]** e **Nfull[2]**, a transferência do *token* presente em **Shifting** para **NShifting** e, também, a inserção de um *token* em **t_{TO}**, encerrando, desta forma, a etapa **e.1**.

A etapa **e.2** será iniciada através do disparo, em qualquer ordem, das transições **Na₀B₁**, **Na₁B₁**, **Nb₀B₁**, **Nb₁B₁**, **Nc₀B₁**, **Nc₁B₁**, **Nx₀B₁**, **Nx₁B₁**, **Nd₀B₁** e **Nd₁B₁**. Após esta sequência de disparo de transições, todos os lugares da RPTM representante da instância **B₁** estarão vazios, assim como o lugar **B₁Clrd** passará a contar com 10 *tokens*. Esta quantidade de *tokens* em **B₁Clrd** habilitará a transição **RstB₁**, a qual disparará, restaurando a rede representante de **B₁** à sua marcação *default* (através da inserção de um *token* em cada um dos lugares **Na₀B₁**, **Na₁B₁**, **Nb₀B₁**, **Nb₁B₁**, **Nc₀B₁**, **Nc₁B₁**, **Nx₀B₁**, **Nx₁B₁**, **Nd₀B₁** e **Nd₁B₁**) e liberando a instância **B₁** para a recepção e o processamento de um novo *sample* (através da inserção de um *token* no lugar **B₁Idle**).

Em relação à **memória da rede**, o *token* inserido em **t_{TO}** (através do disparo de $t\_B_1O_2(c_0;d_0)$) habilitará a transição **t_{RunTO}**, cujo disparo promoverá a remoção do *token* de **t_{TO}**, assim como a inserção de um *token* em **t_{TO}Subnet**, modelando a execução das **operações** associadas à transição **t** (o lugar **t_{TO}Subnet** deverá ser substituído por um modelo RPTM, representante das operações de **t**). A presença de um *token* em **t_{TO}Subnet** habilitará a transição **t_{End}**, a qual disparará, removendo o *token* de **t_{TO}Subnet** e inserindo um *token* em **EXEC**.

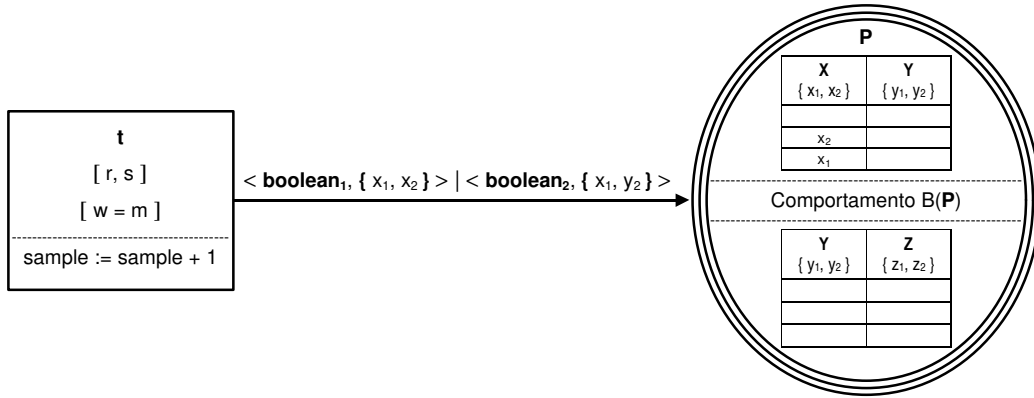
A conexão entre um lugar funcional e uma transição, através de um *ramo de saída fornecedor*, é ilustrada na figura 12.18, envolvendo um lugar **P** e uma transição **t**. Em relação ao lugar **P**, este possui 2 **parâmetros de entrada** e 2 de **saída** (resultados). Argumentos de entrada deverão ser dos **tipos X e Y**, enquanto os resultados gerados serão dos **tipos Y e Z**. A definição dos **tipos** encontra-se distribuída nos cabeçalhos dos **repositórios**, entre chaves (mesmo conteúdo da classe **anotações** da RPSD portadora de **P**). No exemplo adotado, **X, Y e Z** possuem cardinalidade 2. O **número máximo de samples** $max_s(P)$ é 3. O **comportamento** $B(P)$ não se encontra ilustrado na figura. A **guarda** de **t** é $[w = m]$, e seu **intervalo de disparo** é generalizado por $[r, s]$. A única **operação** de **t** é o incremento da variável **sample** em uma unidade. A **função de inserção** $\langle boolean_1, \{x_1, x_2\} \rangle | \langle boolean_2, \{y_1, y_2\} \rangle$, associada ao ramo fornecedor, estabelece que, caso a expressão booleana **boolean₁** seja **verdadeira**, e **t** encontra-se habilitada, o disparo de **t** inserirá os **tokens** **x₁** e **x₂** em **P** (no seu **repositório de entrada**). No caso de **boolean₂** ser **verdadeira**, e **boolean₁** ser **falsa**, o disparo de **t** inserirá os **tokens** **x₁** e **y₂** em **P**.

Na representação de conexão mostrada na figura 12.18, quando a expressão **boolean₁** for **verdadeira**, o lugar **boolean₁** possuirá um **token** e o lugar **Nboolean₁** estará vazio. Convenção equivalente é adotada para a expressão **boolean₂**, com os lugares **boolean₂** e **Nboolean₂**. A RPTM apresenta uma transição para cada **possibilidade de inserção** e de **marcação corrente** do lugar **P**. Assim, considerando-se que a **guarda** de **t** seja **verdadeira** e que nenhuma outra transição conectada a **P**, se existir, esteja em execução (condições expressas pela presença de **token** em **t_G** e em **EXEC**), **t** poderá ficar **habilitada** em dois casos:

Caso I: Expressão **boolean₁** **verdadeira** (presença de **token** em **boolean₁**). Neste caso, somente uma das transições que tenham **boolean₁** como lugar de entrada poderá ficar habilitada. Estas transições são **t_{b1}(1,2)** e **t_{b1}(2,3)**. A primeira serve para inserir **x₁** e **x₂**, respectivamente, nas posições [1,1] e [2,1] da matriz representante do **repositório de entrada** $R_I(P)$, enquanto a segunda é responsável pela inserção de **x₁** e **x₂**, respectivamente, nas posições [2,1] e [3,1] da matriz. Se a coluna **X** de $R_I(P)$ estiver vazia (presença de um **token** no lugar **ins[1,1]**), a transição **t_{b1}(1,2)** ficará habilitada. Caso a coluna **X** de $R_I(P)$ possua um único **token** (presença de um **token** em **ins[2,1]**), a transição **t_{b1}(2,3)** ficará habilitada. Nos demais casos (coluna **X** com dois ou três **tokens**), nenhuma transição da RPTM estará habilitada, isto é, **t** estará **desabilitada**. O **intervalo de disparo** de **t_{b1}(1,2)** e **t_{b1}(2,3)** é o mesmo de **t**. O disparo de **t_{b1}(1,2)** ou **t_{b1}(2,3)** promoverá alterações na marcação corrente da RPTM, de acordo com as convenções descritas na seção 12.2.3;

Caso II: Expressão **boolean₁** **falsa** e expressão **boolean₂** **verdadeira** (presença de **token** em **Nboolean₁** e **boolean₂**). Neste caso, somente uma das transições que tenham **boolean₂** como lugar de entrada poderá ficar habilitada. Estas transições são **t_{b2}(i,j)**, **i, j = 1, 2, 3**. A transição **t_{b2}(i,j)** serve para inserir **x₁** e **y₂**, respectivamente, nas posições [i,1] e [j,2] da matriz representante do **repositório de entrada** $R_I(P)$, isto é, **x₁** será inserido na **i**-ésima linha da coluna **X** de $R_I(P)$, enquanto **y₂** será inserido na **j**-ésima linha da coluna **Y** de $R_I(P)$. A transição **t_{b2}(i,j)** ficará habilitada se existir um **token** em ambos os lugares **ins[i,1]** e **ins[j,2]**. Nos demais casos (coluna **X** ou **Y** cheia), nenhuma transição da RPTM estará habilitada, isto é, **t** estará **desabilitada**. O **intervalo de disparo** de **t_{b2}(i,j)** é o mesmo de **t**, e seu disparo promoverá alterações na marcação corrente da RPTM, compatíveis com as convenções descritas na seção 12.2.3.

a) Conexão envolvendo ramo fornecedor



b) RPTM equivalente (somente repositório de entrada)

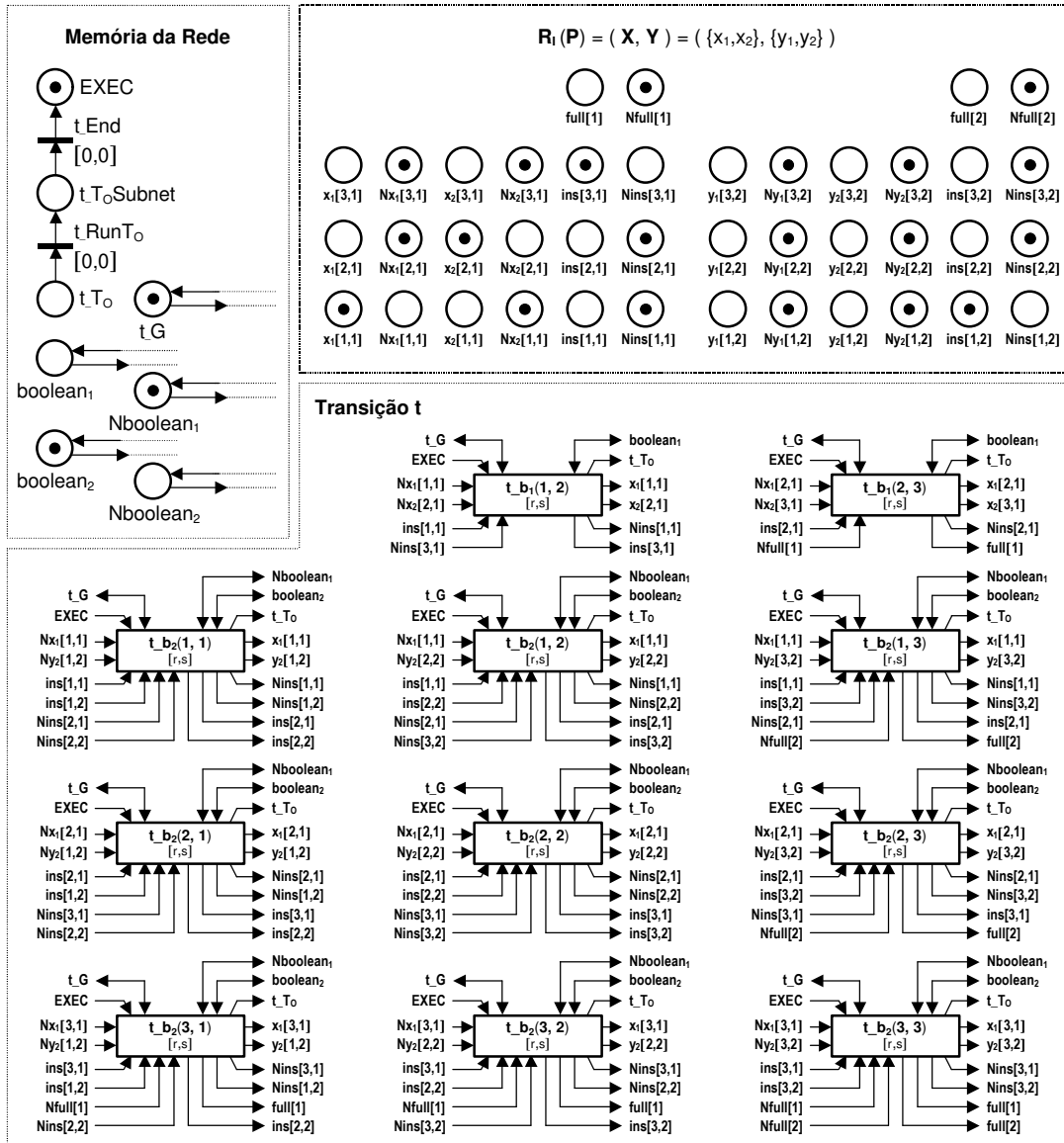


Figura 12.18. RPTM descrevendo uma conexão com ramo fornecedor.

Na marcação mostrada na figura 12.18, a única transição habilitada é $t\_b_2(3,1)$, cujo disparo promoverá, na RPSD, a inserção do *token* x_1 na posição [3,1] da matriz $R_I(P)$, a inserção do *token* y_2 na posição [1,2] da matriz $R_I(P)$ e, também, a ativação das operações associadas à transição t . Considerando-se a RPTM, o disparo de $t\_b_2(3,1)$ promoverá:

- a) a transferência do *token* presente em $Nx_1[3,1]$ para o seu dual $x_1[3,1]$, representando a inserção do *token* x_1 na posição [3,1] da matriz $R_I(P)$;
- b) a transferência do *token* presente em $Ny_2[1,2]$ para o seu dual $y_2[1,2]$, representando a inserção do *token* y_2 na posição [1,2] da matriz $R_I(P)$;
- c) a transferência dos *tokens* presentes em $ins[3,1]$ e $ins[1,2]$ para os seus duais $Nins[3,1]$ e $Nins[1,2]$, representando que as posições [3,1] e [1,2], da matriz $R_I(P)$, não estão mais livres;
- d) a transferência dos *tokens* presentes em $Nfull[1]$ e $Nins[2,2]$ para os seus duais $full[1]$ e $ins[2,2]$, representando que a primeira coluna da matriz $R_I(P)$ está cheia, ao passo que a linha 2 da segunda coluna está livre;
- e) a remoção do *token* presente em **EXEC** (associado a P), indicando que t tem acesso exclusivo a P ;
- f) a inserção de um *token* em t_to , possibilitando a execução das operações associadas a t .

Continuando a descrição de conexões envolvendo lugares do tipo III, a atuação de *ramos de entrada suspensivos* e de *ramos de saída restauradores* consiste, respectivamente, da atribuição de **FALSE** ou **TRUE** para a **variável de controle** **ENABLE** associada ao lugar funcional. Isto é implementado, na RPTM, pela adição de um lugar denominado **ENABLE**, associado ao lugar funcional, o qual refletirá o valor da variável de controle **ENABLE**. Este lugar deverá ser conectado, através de ramos de entrada, às transições pertencentes às **instâncias** do **comportamento B** do lugar funcional. Assim, a presença de um *token* no lugar **ENABLE** constituir-se-á em uma das condições de habilitação destas transições.

A figura 12.19 estende a figura 12.17, ao incluir uma nova transição – **p** – conectada a P através de um *ramo de entrada preemptivo*. A transição **p** ficará habilitada quando a variável **Error**, pertencente à **memória da rede**, for igual a **5**. Caso **p** dispare, **todos** os *tokens* presentes no lugar P serão **removidos** e o valor da variável **Error** passará para **0**. Os *tokens* de P encontram-se distribuídos nos *repositórios de entrada* e *de saída*, e nas três **instâncias** do **comportamento B**(P). Em relação à RPTM equivalente à RPSD, o disparo de **p** fará com que as subredes representantes dos *repositórios* e das **instâncias** sejam **restauradas** à sua **marcação default**, conforme as convenções adotadas (seção 12.2.3). As extensões feitas sobre a rede da figura 12.17, na composição da rede da figura 12.19, serão explanadas no decorrer da descrição a seguir.

A **memória da rede** (figura 12.19.i) apresenta novos lugares e transições, de forma a refletir a modelagem da execução das operações $To(p)$ e, também, agregar a guarda dessa nova transição da rede RPSD. Observa-se, porém, que o lugar **EXEC** não foi duplicado, uma vez que ele é associado ao lugar P , e não às transições. Desta forma, por vez, somente uma das transições conectadas ao lugar P poderá ter acesso ao mesmo.

O disparo da transição **p** é representado, na RPTM, por uma seqüência de disparo de transições, a qual é iniciada pelo disparo da transição **p_Empty** (figura 12.19.g) e finalizada pelo disparo da transição **p_End** (figura 12.19.i).

A habilitação da transição **p_Empty** depende da marcação dos lugares **p_G**, **EXEC** e **ENABLE**. A presença de um *token* em **p_G** significa que a guarda de **p** é *verdadeira* (**TRUE**). Já a presença de um *token* em **EXEC** denota que o lugar RPSD associado (no caso, **P**) não está sendo acessado por transição alguma. Finalmente, conforme visto na página anterior (na descrição sobre conexões envolvendo *ramos de entrada suspensivos* e *ramos de saída restauradores*), a presença de um *token* no lugar **ENABLE** significa que a *variável de controle* **ENABLE**, associada ao lugar funcional (**P**, no caso), é *verdadeira* (**TRUE**). A utilização do lugar **ENABLE**, nas condições de habilitação de **p_Empty**, se deve à necessidade, durante o processo de remoção de *tokens* desencadeado pelo disparo de **p**, deste lugar (**ENABLE**) não possuir *token*, para que as transições das três instâncias fiquem **desabilitadas** (evitando-se, desta forma, possíveis anomalias). Uma vez habilitada, a transição **p_Empty** disparará entre **x** e **y** unidades de tempo (intervalo de disparo igual ao de **p**, na RPSD), promovendo a remoção dos *tokens* presentes em **EXEC** e **ENABLE**, assim como a inserção de 20 *tokens* em **ClearI_R** e de 15 *tokens* em **ClearO_R**. O disparo de **p_Empty** desencadeará o restauro, ao estado original (restituição à marcação *default*), das redes representantes do *repositório de entrada* (a partir da inserção de 20 *tokens* em **ClearI_R**), do *repositório de saída* (a partir da inserção de 15 *tokens* em **ClearO_R**) e das instâncias (nas quais o restauro do estado original se dará após a remoção de todos os *tokens* do *repositório de entrada*, baseando-se nas linhas **indisponíveis** do mesmo). O processo de restauro envolve, também, os diversos lugares de controle da figura 12.19.d.

O restauro à marcação *default* do *repositório de entrada* inicia-se com o disparo das transições **rótulo_lugar_{IR}** (figura 12.17.b), onde **rótulo_lugar_{IR}** corresponde a qualquer um dos lugares da rede representante do *repositório de entrada*. Uma transição **rótulo_lugar_{IR}** tem como lugares de entrada **rótulo_lugar_{IR}** e **ClearI_R**, e, como lugar de saída, **I_Rcleared**. Desta forma, tal transição ficará habilitada quando existir um *token* no lugar **rótulo_lugar_{IR}** e existir, pelo menos, um *token* em **ClearI_R**. O disparo de **rótulo_lugar_{IR}** promoverá a remoção do *token* presente no lugar de mesmo nome, assim como a transferência de um *token* de **ClearI_R** para **I_Rcleared**. Como o disparo de **p_Empty** insere 20 *tokens* em **ClearI_R**, ocorrerá, no máximo, 20 disparos de transições **rótulo_lugar_{IR}**. Pelas convenções adotadas na representação RPTM do *repositório de entrada*, o número total de *tokens* em uma de suas fileiras de lugares (representante de uma das linhas do repositório) será, logo após o disparo de **p_Empty**, 6 ou 0, dependendo, respectivamente, da linha estar **disponível** ou **não**. Em relação aos lugares de controle de coluna cheia, apenas um deles estará marcado, por coluna. Somando-se o número máximo de *tokens* nas três fileiras de lugares ($3 \times 6 = 18$) com o número de *tokens* nos lugares de controle de coluna cheia ($2 \times 1 = 2$), obtém-se o valor 20. Após o disparo de todas as transições **rótulo_lugar_{IR}** habilitadas, haverá uma sobra de 6 *tokens* em **I_Rcleared**, para cada linha **indisponível** do *repositório de entrada*. Para finalizar a operação de remoção de *tokens* do *repositório de entrada*, os *tokens* excedentes em **ClearI_R** deverão ser removidos, o que é feito com o auxílio das transições **S_jB_i** (figura 12.19.d), cada qual associada ao grupo de lugares **Sample[j]**, **B_iActive** e **ExecS_jB_i**. Conforme visto na descrição da figura 12.17, estes lugares estarão marcados caso a instância **B_i** esteja amarrada ao *sample* **S_j** (**j**-ésima linha **indisponível**). O disparo de uma transição **S_jB_i** promoverá o restauro dos lugares **Sample[j]**, **B_iActive** e **ExecS_jB_i** (deixando-os vazios), a transferência de 6 *tokens* excedentes em **ClearI_R** para **I_Rcleared** e, também, a inserção de 10 *tokens* em **ClearB_i** (necessários ao restauro da rede representante da instância **B_i**). No exemplo adotado, até 3 transições **S_jB_i** poderão ficar habilitadas e disparar (uma vez que o número máximo de linhas **indisponíveis** é 3). O restauro à marcação *default* do *repositório de entrada* encerra-se com o disparo da transição **p_EmptyOK** (figura 12.17.g), conforme será descrito adiante.

O disparo de uma transição S_jB_i , se houver, promoverá a inserção de 10 *tokens* no lugar $ClearB_i$, possibilitando, desta forma, o restauro da rede representante da instância B_i (através da remoção de todos os *tokens* presentes na rede B_i , seguida pela inserção de um *token* em cada um de seus lugares **negativos**), assim como a liberação da instância B_i (denotada pela inserção de um *token* B_iIdle). Esta operação de restauro e liberação de instância é idêntica à etapa **e.2**, descrita na explanação da figura 12.17.

A remoção de todos os *tokens* presentes no **repositório de saída** ocorre de forma semelhante a do **repositório de entrada**, substituindo-se $ClearI_R$ por $ClearO_R$ e $I_Rcleared$ por $O_Rcleared$. Entretanto, não há transições semelhantes às transições S_jB_i , uma vez que inexistente sobra de *tokens* em $ClearO_R$ (logo após o disparo de p_Empty , cada linha do **repositório de saída** terá exatamente 5 *tokens*, motivo pelo qual tal transição insere 15 *tokens* em $ClearO_R$).

Após o disparo de todas as transições referentes à remoção de *tokens* dos **repositórios de entrada e de saída**, os lugares $I_Rcleared$ e $O_Rcleared$ terão, respectivamente, 20 e 15 *tokens*. O restauro de uma instância **ativa** B_i conduz à inserção de um *token* em B_iIdle ; caso a instância B_i achava-se **inativa** (quando do disparo de p_Empty), o lugar B_iIdle já estava marcado. Desta forma, a presença de *tokens* em B_1Idle , B_2Idle , B_3Idle , $I_Rcleared$ (20) e $O_Rcleared$ (15), constitui-se na condição de habilitação da transição $p\_EmptyOK$, cujo disparo finaliza as operações de restauro. O disparo de $p\_EmptyOK$ promoverá a remoção dos *tokens* presentes em $I_Rcleared$ e $O_Rcleared$, a inserção de um *token* em p_To (para que as operações referentes à transição p possam ser executadas), a inserção de um *token* em **ENABLE** (marcação *default* deste lugar, utilizada para não interferir na habilitação das transições das redes representantes das instâncias) e, também, a inserção de um *token* em cada um dos lugares **negativos**, tanto na rede representante do **repositório de entrada** (figura 12.19.b) quanto na representante do **repositório de saída** (figura 12.19.c) – caracterizando a marcação *default* destas redes.

A execução da RPTM representante do disparo da transição p encerra-se com o disparo de p_End (figura 12.19.i), o qual restabelece um *token* ao lugar **EXEC** associado a P .

Na marcação presente na RPTM da figura 12.19, a transição p_Empty está habilitada e o seu disparo promoverá a sequência de disparo acima descrita, após a qual a marcação da rede apresentará a seguinte configuração (partes da figura 12.19):

- i) partes **b**) e **c**): somente lugares **negativos** (aqueles cujo rótulo inicia-se por **N**) marcados;
- ii) parte **d**): somente lugares B_1Idle , B_2Idle e B_3Idle marcados;
- iii) parte **e**): somente lugares **negativos** marcados;
- iv) parte **h**): marcação inalterada (trata-se de lugares externos à RPSD da figura 12.19.a); e
- v) parte **i**): mesma marcação (considerando-se que as transições p_RunTo e p_End já tenham disparado).

Destarte, as **convenções** adotadas, ao longo deste capítulo, permitem mostrar que **Rede de Petri para Sistemas Digitais** é uma extensão de rede de Petri **válida**, uma vez que a representação de seus **componentes estruturais** através de redes de Petri de baixo nível é possível. Os exemplos deste capítulo empregaram modelos descritos em **redes de Petri Lugar/Transição** e **redes de Petri Temporizadas de Merlin**, os quais são validados, via simulação, no apêndice II desta tese.

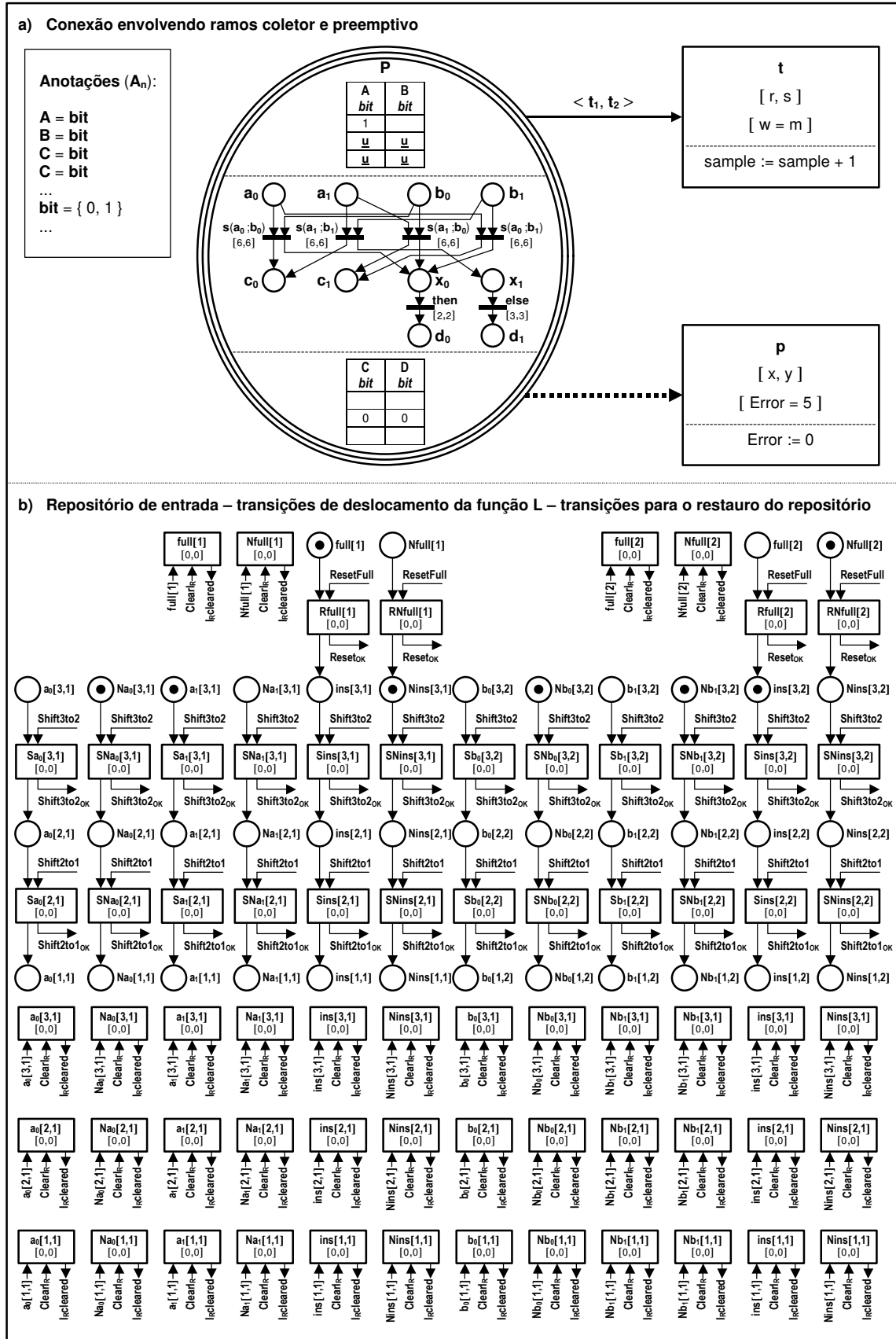
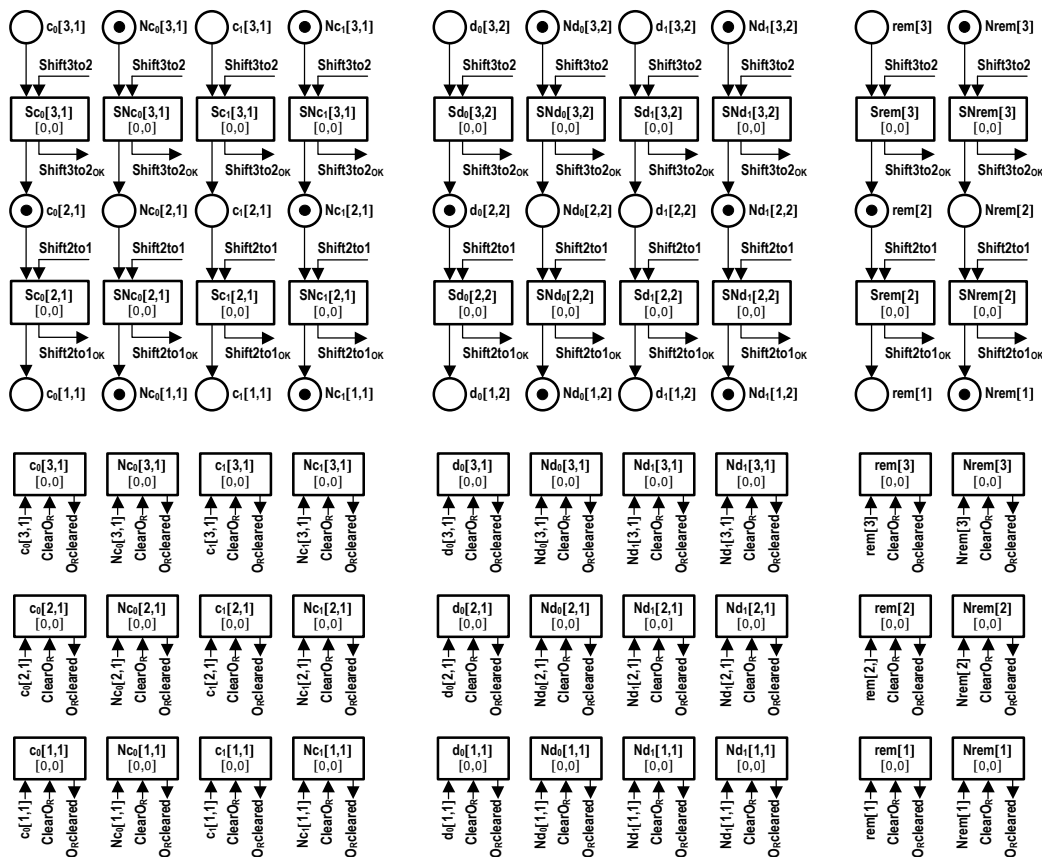
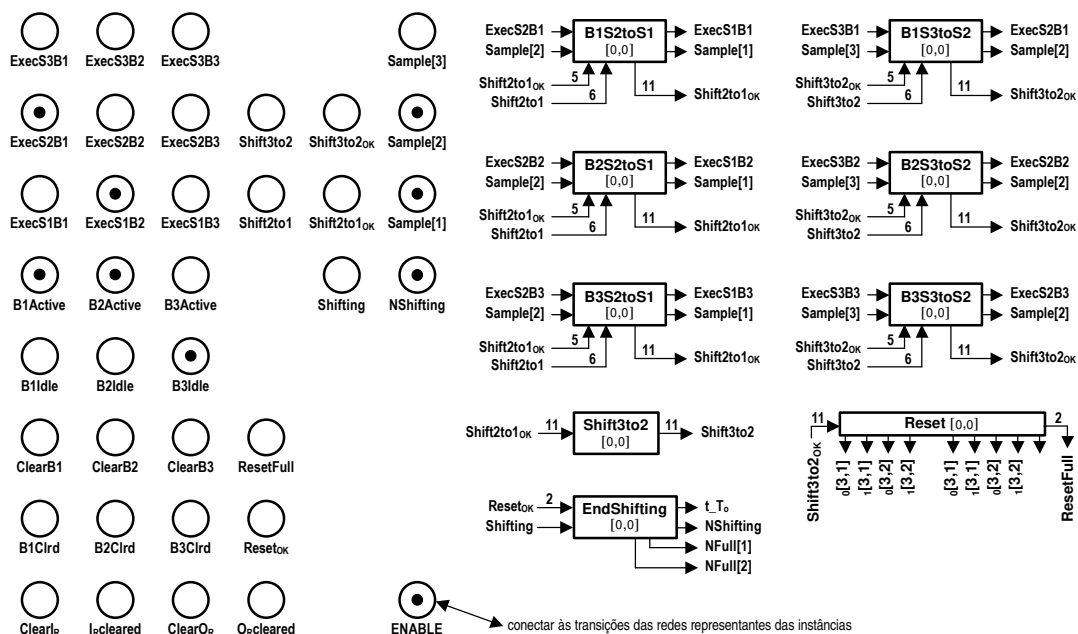


Figura 12.19. RPTM descrevendo uma conexão com ramo preemptivo.

c) Repositório de saída – transições de deslocamento da função L – transições para o restauro do repositório

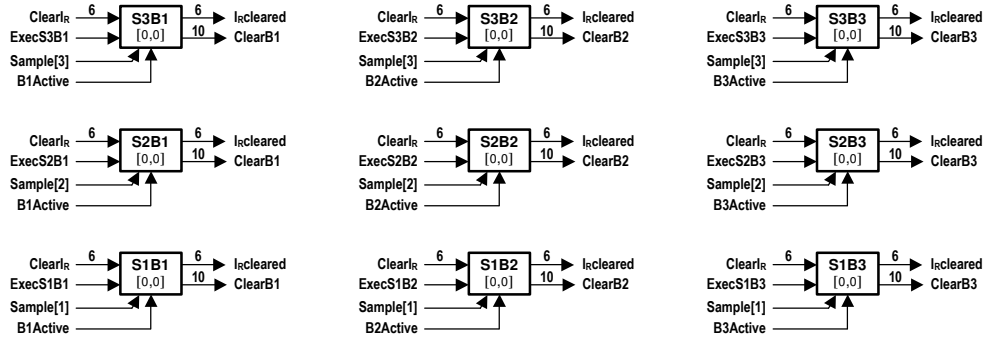


d) Mecanismo de associação *sample*/instância – mecanismo de deslocamento da função L – controle de restauro

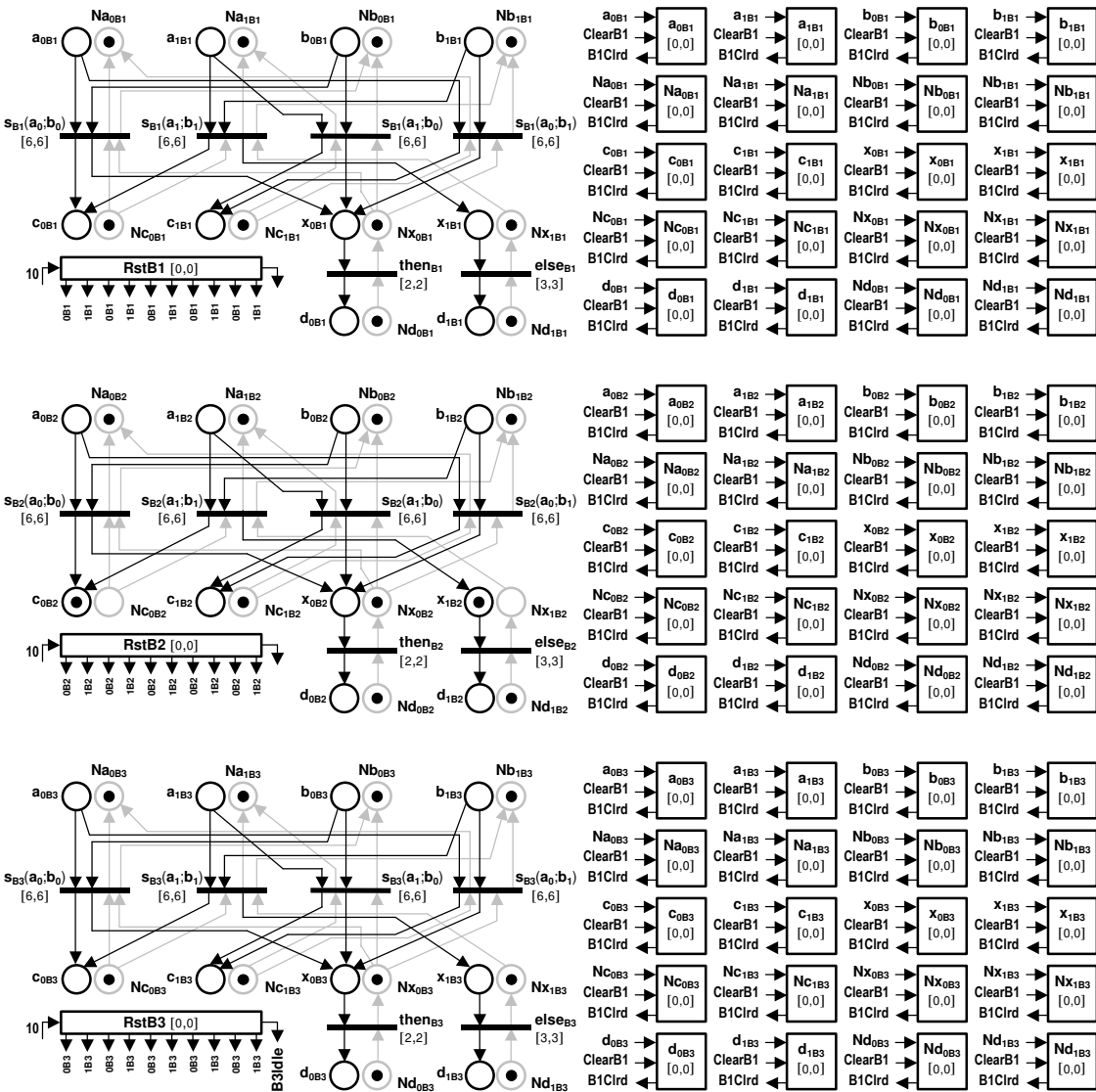


Continuação da figura 12.19.

d) cont.

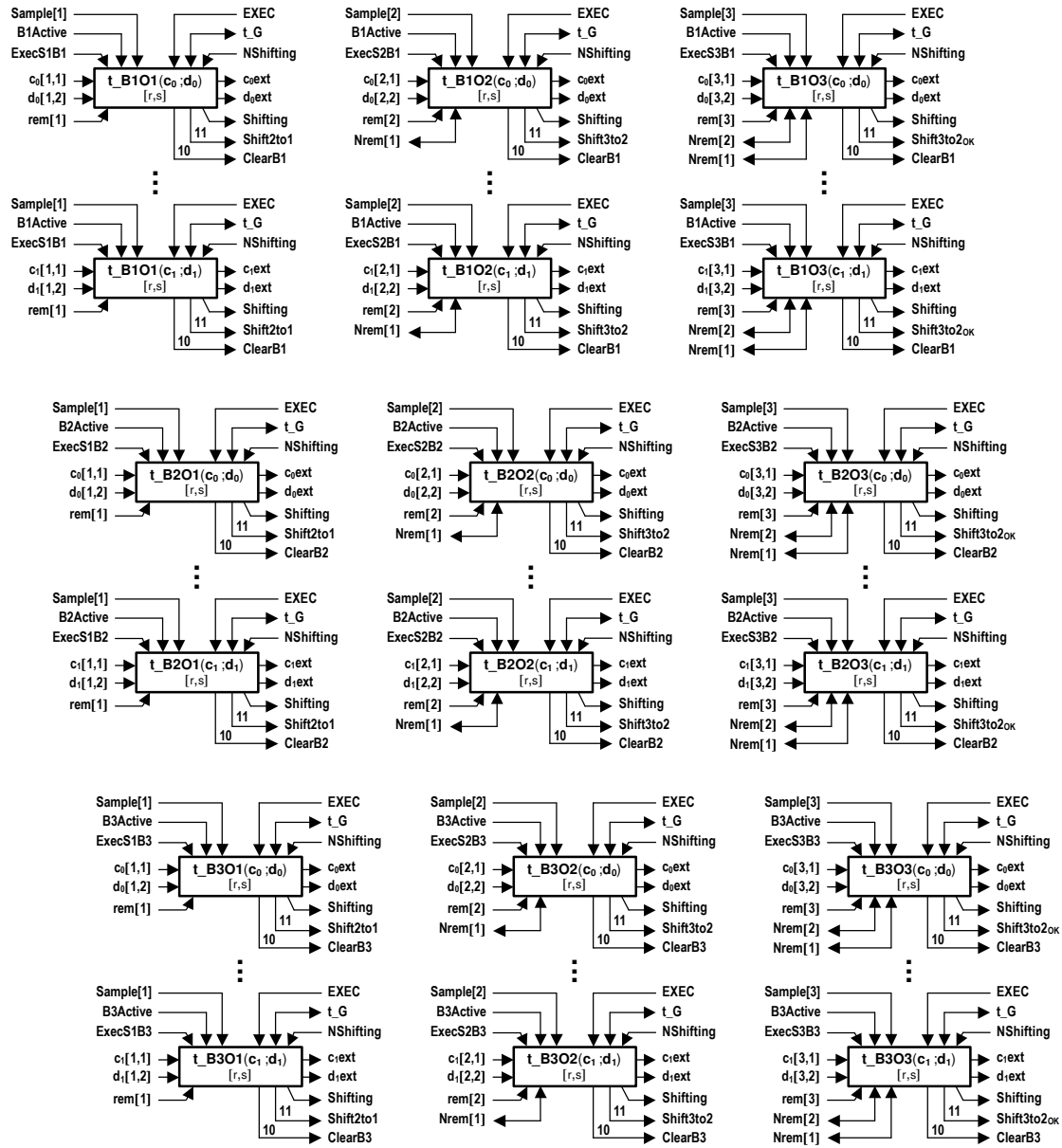


e) Instâncias do comportamento – transições para o restauro do estado original

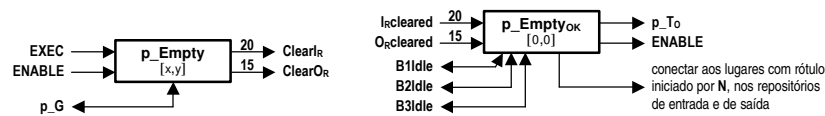


Continuação da figura 12.19.

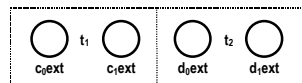
f) Transição t



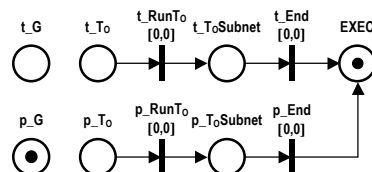
g) Transição p



h) Lugares RPTM associados, nas redes equivalentes, a um ou mais lugares RPSD, de qualquer tipo, receptores de tokens provenientes da transição t



i) Memória da rede



Continuação da figura 12.19.

CAPÍTULO 13: MÉTODOS DE ANÁLISE BASEADOS EM SIMULAÇÃO

13.1. INTRODUÇÃO

Rede de Petri e suas extensões permitem a modelagem de diversos aspectos em diferentes tipos de sistemas computacionais. Um dos propósitos de um modelo de sistema é permitir que o mesmo seja verificado quanto ao atendimento, ou não, das características **funcionais e temporais** que o sistema real deva atender, de acordo com as restrições de seu projeto. **Propriedades** do sistema real podem ser inferidas a partir da **validação, ou não**, de propriedades do modelo de tal sistema. O conceito de **análise** de um modelo em redes de Petri está relacionado com a **verificação** de propriedades no modelo. Uma das formas de análise de redes de Petri é a **simulação**, na qual diferentes **seqüências de disparo** de **transições** são realizadas, a partir de diferentes **marcações iniciais** (disposição dos *tokens* nos lugares quando do início da simulação).

Em RPSD, o **estado** do sistema modelado é denominado **classe** (veja seção 11.2.2). Assim, a simulação de um modelo em RPSD consiste na geração do **grafo de classes** associado a tal modelo, a partir de uma **classe** considerada **inicial**. Na seção 13.2 é apresentado um **algoritmo de simulação** (e **estruturas de dados** associadas), capaz de gerar o grafo de classes de uma RPSD. Nas seções que complementam o capítulo, é apresentado um conjunto de **propriedades** (e suas **técnicas de análise**) que podem ser verificadas a partir da simulação de modelos RPSD.

13.2. ALGORITMO DE SIMULAÇÃO PARA REDE DE PETRI PARA SISTEMAS DIGITAIS

Para melhor compreensão do **algoritmo de simulação** de modelos RPSD, exploremos, inicialmente, suas **estruturas de dados**. Objetivando implementação futura, foram elaboradas estruturas **dinâmicas**, isto é, estruturas que deverão ser implementadas usando-se **ponteiros**. Desta forma, o tamanho máximo de tais estruturas ficaria limitado à capacidade de gerenciamento de memória do compilador empregado em sua implementação.

São quatro as **principais** estruturas de dados (há estruturas complementares, de importância secundária, cuja ausência não interfere no foco deste capítulo):

- a) **INFO_LUGARES**: Estrutura que armazena as informações pertinentes à definição de cada um dos **lugares** da rede;
- b) **INFO_TRANSICOES**: Estrutura que armazena as informações referentes à definição de cada uma das **transições** da rede e seus respectivos **ramos**;
- c) **GRAFO_CLASSES**: Estrutura que armazena as informações que definem cada uma das diferentes classes que são progressivamente geradas pelo algoritmo durante o processo de simulação;
- d) **CLASSES_PROVISORIAS**: Estrutura auxiliar que armazena provisoriamente as definições das classes que estão sendo geradas pelo algoritmo, as quais serão confrontadas com as classes já armazenadas em **GRAFO_CLASSES**.

As figuras de 13.1 a 13.4 são **representações gráficas** das **estruturas** mencionadas. Em cada uma dessas figuras, o nome escrito dentro de cada retângulo representa um nome de **campo de dados**. Os campos de dados nada mais são do que variáveis, as quais pertencem a **diferentes tipos**, podendo ser números inteiros, números reais, valores booleanos, cadeias de caracteres ou ponteiros. Cada estrutura é composta de subestruturas. Os **ponteiros** são representados por **setas inteiras** ou **pontilhadas**. A origem da seta é o campo que armazena o valor de um **endereço de memória**, o qual designa a posição inicial de uma subestrutura (ponta da seta) agregada à estrutura pai (estrutura que contém o campo do qual parte a seta). As **setas inteiras** representam **ponteiros para estruturas diferentes** da estrutura que contém o campo do qual parte o ponteiro, enquanto **setas pontilhadas** representam **ponteiros para estruturas iguais** à estrutura que contém o campo de origem do ponteiro. Dependendo da rede simulada, pode acontecer de **nem todos os campos ponteiros** serem utilizados. Neste caso, tais campos permanecem com o valor *default* de quando são criados, qual seja, o valor **NIL** (ponteiro para nenhuma estrutura).

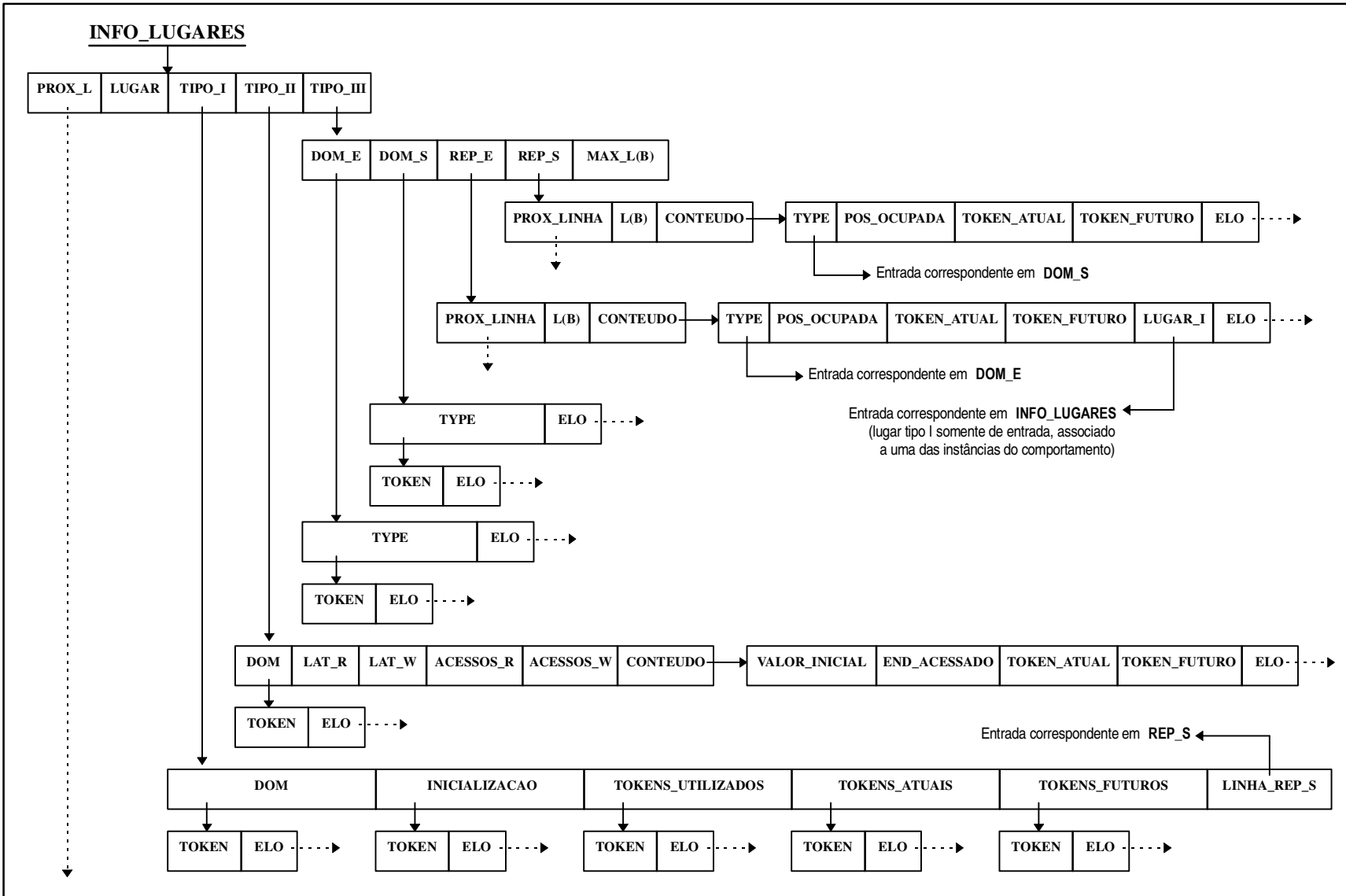
Os principais campos que compõem cada uma das estruturas de dados são descritos a seguir. Nestas descrições, **lugares (transições)** são referidos por **números naturais seqüenciais**. Isto é feito para facilitar o trato computacional dos mesmos, bastando-se associar a cada lugar (transição) um identificador numérico único, quando da sua definição.

Os campos **PROX_L**, **PROX_LINHA** (figura 13.1), **PROX_T** (figura 13.2), **PROX_C** (figura 13.3), **SUCESSORAS** (figura 13.4) e **ELO** (figuras 13.1 a 13.4) armazenam, cada qual, um ponteiro que **aponta** (este é o termo usado para designar o valor de um ponteiro) para a **parte subsequente da estrutura** a que pertencem. Por exemplo, o valor de **PROX_L** da parte da estrutura referente ao lugar **n** apontará para a parte da estrutura referente ao lugar **n+1**, caso **n** não seja o último lugar definido para a rede. Sendo **n** o número do último lugar, **PROX_L** armazenará **NIL**.

Os campos da estrutura **INFO_LUGARES** (figura 13.1) são:

- a) **LUGAR**: Armazena o identificador numérico associado ao lugar;
- b) **TIPO_I**: Armazena **NIL** para lugares do tipo II e III. Para lugares do tipo I, armazena um ponteiro para uma estrutura com os seguintes campos:
 - b.1) **DOM**: Armazena um ponteiro para a lista de *tokens* que podem ocupar o lugar, isto é, corresponde ao **domínio D_S** ;
 - b.2) **INICIALIZACAO**: Armazena um ponteiro para a lista de *tokens* presentes no lugar quando da inicialização da rede. Esta lista é utilizada, também, para o *reset* do lugar, quando da atuação de uma transição conectada ao mesmo por um ramo preemptivo;
 - b.3) **TOKENS_UTILIZADOS**: Armazena um ponteiro para a lista de *tokens* que ocupam o lugar, ao menos uma vez, durante o processo de simulação. Esta lista serve para verificar quais *tokens* não foram utilizados pelo lugar, quando confrontada com o domínio do lugar (lista **DOM**);
 - b.4) **TOKENS_ATUAIS**: Lista usada durante o processo de simulação. Armazena um ponteiro para a lista de *tokens* que ocupam o lugar, na classe que está sendo simulada;
 - b.5) **TOKENS_FUTUROS**: Lista usada durante o processo de simulação. Armazena um ponteiro para a lista de *tokens* que ocupam o lugar, em uma classe que está sendo gerada a partir da classe que está sendo simulada;

Figura 13.1. Estrutura de dados INFO_LUGARES.



b.6) **LINHA_REP_S**: Campo usado quando o lugar representa um dado **parâmetro de entrada** do **comportamento** associado a um lugar do tipo III. Armazena um ponteiro para uma das linhas do **repositório de entrada** do referido lugar, pertencente à lista **REP_S** do lugar (veja descrição de **REP_S** adiante).

Todas as listas de *tokens* anteriores seguem a mesma formatação, dada pela estrutura:

- **TOKEN**: Campo que armazena o identificador do *token*;
- **ELO**: Conforme explicado anteriormente, campo que aponta para a célula subsequente, qual seja, a estrutura representativa do próximo *token* da lista, se houver;

c) **TIPO_II**: Armazena **NIL** para lugares do tipo I e III. Para lugares do tipo II, armazena um ponteiro para uma estrutura com os seguintes campos:

c.1) **DOM**: Armazena um ponteiro para a lista de *tokens* que podem ocupar o lugar, isto é, corresponde ao **domínio** D_M ;

c.2) **LAT_R, LAT_W**: Armazenam, respectivamente, as **latências** de **leitura** e de **escrita** do lugar;

c.3) **ACESSOS_R, ACESSOS_W**: Armazenam, respectivamente, a quantidade de vezes que o lugar foi acessado para operações de leitura e de escrita, durante o processo de simulação;

c.4) **CONTEUDO**: Armazena um ponteiro para uma lista de informações utilizadas na manipulação do **vetor conteúdo** associado ao lugar. Cada entrada da lista corresponde a um **endereço (célula de armazenamento)** do lugar, sendo composta por cinco campos:

c.4.1) **VALOR_INICIAL**: Armazena o *token* presente no endereço quando da inicialização da rede. Este valor é utilizado, também, para o *reset* do endereço, quando da atuação de uma transição conectada ao lugar por um ramo preemptivo;

c.4.2) **END_ACESSADO**: Armazena o valor booleano **TRUE** caso o endereço seja acessado pelo menos uma vez durante a simulação, caso contrário, armazena o valor booleano **FALSE**. O valor deste campo é utilizado para identificar endereços não acessados;

c.4.3) **TOKEN_ATUAL**: Campo usado durante o processo de simulação. Armazena o *token* que ocupa o lugar, na classe que está sendo simulada;

c.4.4) **TOKEN_FUTURO**: Campo usado durante o processo de simulação. Armazena o *token* que ocupa o lugar, em uma classe que está sendo gerada a partir da classe que está sendo simulada;

c.4.5) **ELO**: Aponta para o endereço subsequente, se houver.

d) **TIPO_III**: Armazena **NIL** para lugares do tipo I e II. Para lugares do tipo III, armazena um ponteiro para uma estrutura com os seguintes campos:

d.1) **DOM_E**: Armazena um ponteiro para uma lista ordenada de **tipos** de *tokens*, correspondente ao **domínio de entrada** D_I . Cada um dos **tipos** da lista é armazenado em uma estrutura com os seguintes campos:

d.1.1) **TYPE**: Armazena um ponteiro para a lista de *tokens* que definem o tipo. Tal lista é formada por células com dois campos:

d.1.1.1) **TOKEN**: Armazena o identificador do *token*;

d.1.1.2) **ELO**: Aponta para o *token* subsequente, se houver;

d.1.2) **ELO**: Aponta para o tipo subsequente, se houver;

- d.2) **DOM_S**: Armazena um ponteiro para uma lista ordenada de **tipos** de *tokens*, correspondente ao **domínio de saída** D_O . Estrutura análoga a **DOM_E**;
- d.3) **REP_E**: Armazena um ponteiro para um conjunto de listas correspondente à matriz **repositório de entrada** R_I do lugar. Cada entrada, associada a uma linha de R_I , possui três campos:
- d.3.1) **PROX_LINHA**: Armazena um ponteiro para a próxima linha da matriz R_I , se houver;
 - d.3.2) **L(B)**: Armazena o valor da função de associação dinâmica $L(B)$ (veja seção 12.2.3). Esta função relaciona as instâncias do **comportamento** B do lugar com as linhas dos **repositórios de entrada** R_I e de **saída** R_O ;
 - d.3.3) **CONTEUDO**: Armazena um ponteiro para uma lista de informações utilizadas na manipulação dos **argumentos** para os **parâmetros de entrada** do lugar. Cada entrada da lista corresponde a uma **célula** da matriz R_I (**argumento** associado a uma das colunas da matriz R_I), sendo composta por seis campos:
 - d.3.3.1) **TYPE**: Armazena um ponteiro para a entrada de **DOM_E** correspondente ao tipo do parâmetro;
 - d.3.3.2) **POS_OCUPADA**: Campo usado durante o processo de simulação. Armazena o valor booleano **TRUE** caso a **célula** já possua um argumento, caso contrário, armazena o valor booleano **FALSE**;
 - d.3.3.3) **TOKEN_ATUAL**: Campo usado durante o processo de simulação. Armazena o *token* que ocupa a **célula**, na classe que está sendo simulada;
 - d.3.3.4) **TOKEN_FUTURO**: Campo usado durante o processo de simulação. Armazena o *token* que ocupa a **célula**, em uma classe que está sendo gerada a partir da classe que está sendo simulada;
 - d.3.3.5) **LUGAR_I**: Armazena um ponteiro para a entrada de **INFO_LUGARES** correspondente a um lugar do tipo I, usado para armazenar o valor de **TOKEN_ATUAL**, quando da **ativação** da **instância** do **comportamento** associada à linha da qual este campo faz parte;
 - d.3.3.6) **ELO**: Aponta para o endereço subsequente, se houver;
- d.4) **REP_S**: Armazena um ponteiro para um conjunto de listas correspondente à matriz **repositório de saída** R_O do lugar. Estrutura análoga a **REP_E**, exceto pela ausência do campo **LUGAR_I**;
- d.5) **MAX_L(B)**: Armazena o maior valor associado ao conjunto de campos **L(B)**, durante o processo de simulação. Este valor determina o número máximo de instâncias do comportamento executadas simultaneamente.

Por construção, cada linha das estruturas **REP_E** e **REP_S** está estaticamente associada a uma instância (cópia) do comportamento B . O valor do campo **L(B)** serve para estabelecer a **correspondência** entre as linhas de **REP_E** e **REP_S**, além de indicar a **prioridade de remoção** (a linha de **REP_S** com o menor valor no campo **L(B)** terá prioridade de remoção). Os valores dos campos **L(B)** de ambas estruturas (**REP_E** e **REP_S**) são recalculados após cada operação de inserção/remoção.

Os campos da estrutura de dados **INFO_TRANSICOES** (figura 13.2) são:

- a) **TRANSICAO**: Armazena o identificador numérico associado à transição;
- b) **POTENCIALMENTE_DISPARAVEL**: Armazena o valor booleano **TRUE** caso a transição seja habilitada pelo menos uma vez durante a simulação, caso contrário, armazena o valor booleano **FALSE**. O valor deste campo é utilizado pelo algoritmo de verificação da propriedade de **vivacidade** (seção 13.3.3), e para identificar as transições que nunca disparam (seção 13.3.4);
- c) **ACESSOS_NEGADOS**: Armazena a **quantidade de vezes** que a transição **não disparou** devido, somente, à ação das **latências de leitura/escrita** de algum lugar do tipo II, conectado à transição;
- d) **INSERCOES_NEGADAS**: Armazena a **quantidade de vezes** que a transição **não disparou** devido, somente, à **falta** de alguma instância de comportamento **livre**, associada a algum lugar do tipo III, conectado à transição;
- e) **L_ENTRADA**: Armazena um ponteiro para uma lista dos **lugares de entrada** da transição. Tal lista é formada por células de três campos, quais sejam:
 - e.1) **LUGAR**: Armazena um ponteiro para a parte da estrutura **INFO_LUGARES** relativa a um dos lugares de entrada da transição;
 - e.2) **INFO**: Armazena uma *STRING*, denotando informações referentes ao **tipo de ramo** empregado na conexão com o lugar; e
 - e.3) **ELO**: Armazena um ponteiro para célula referente a outro dos lugares de entrada da transição;
- f) **L_SAIDA**: Armazena um ponteiro para uma lista dos **lugares de saída** da transição. Tal lista é formada por células de três campos, quais sejam:
 - f.1) **LUGAR**: Armazena um ponteiro para a parte da estrutura **INFO_LUGARES** relativa a um dos lugares de saída da transição;
 - f.2) **INFO**: Armazena uma *STRING*, denotando informações referentes ao **tipo de ramo** empregado na conexão com o lugar; e
 - f.3) **ELO**: Armazena um ponteiro para célula referente a outro dos lugares de saída da transição;
- g) **GUARDA**: Armazena uma *STRING*, representante da guarda da transição;
- h) **LDI_E, LDS_E**: Armazenam, respectivamente, os limites de disparo inferior e superior **estáticos** da transição;
- i) **LDI_D, LDS_D**: Armazenam, respectivamente, os limites de disparo inferior e superior **dinâmicos** da transição, durante o processo de simulação;
- j) **LDI_L, LDS_L**: Armazenam, respectivamente, os limites de disparo inferior e superior **dinâmicos** da transição, durante o processo de simulação, sob influência das **latências de leitura/escrita**. Campos utilizados somente pelas transições conectadas a lugares de armazenamento (tipo II);
- k) **OPERACOES**: Armazena um ponteiro para a estrutura representativa do conjunto de operações a serem realizadas quando do disparo da transição.

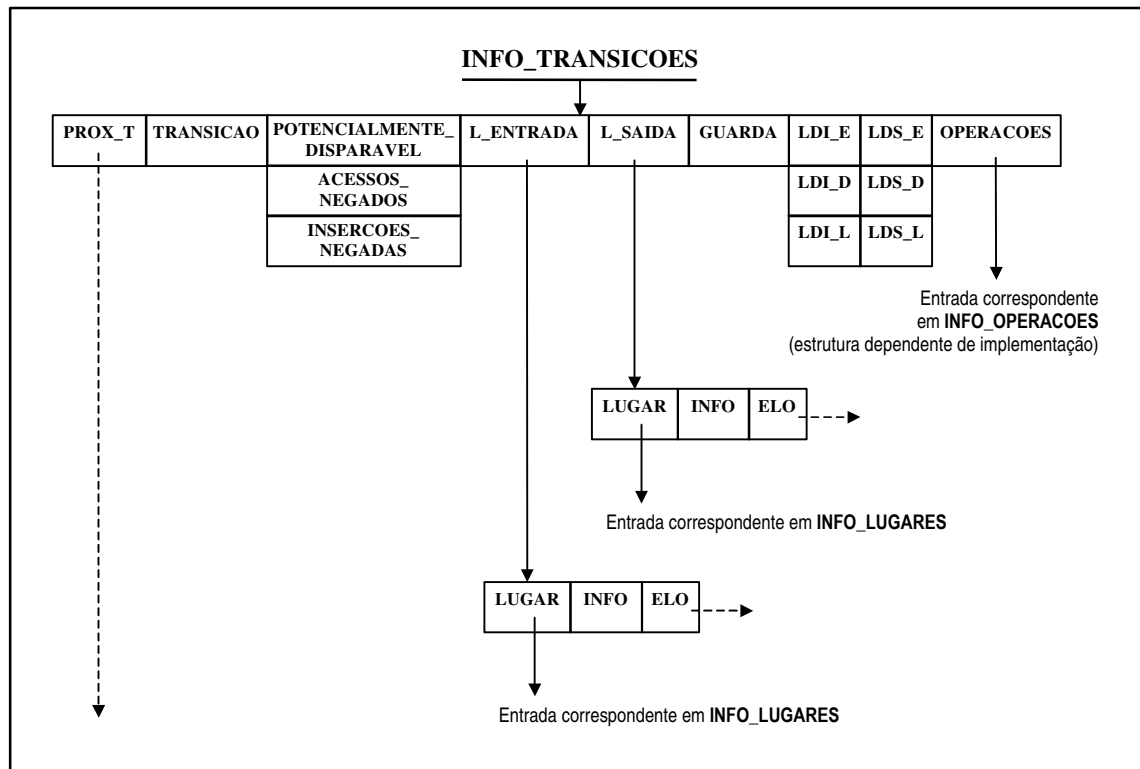


Figura 13.2. Estrutura de dados INFO_TRANSICOES.

Os campos da estrutura **GRAFO_CLASSES** (figura 13.3) são:

- a) **CLASSE**: Armazena o identificador numérico associado à classe;
- b) **MARCACAO**: Armazena uma *STRING*, representante da marcação da classe;
- c) **DOM_TEMPO**: Armazena uma *STRING*, composta pela seqüência $t[LDI, LDS]$, onde t é uma transição habilitada na classe, **LDI** o seu limite de disparo inferior e **LDS** o seu limite de disparo superior;
- d) **VARIAVEIS**: Armazena uma *STRING*, representante da valoração das variáveis na classe;
- e) **SUC_IMEDIATAS**: Armazena um ponteiro para uma lista indicadora das classes imediatamente sucessoras da classe. Tal lista é composta por células de quatro campos, quais sejam:
 - e.1) **TRANSICAO**: Armazena o número associado ao identificador da transição cujo disparo levou a rede da classe indicada no campo **a)** para a classe sucessora imediata, indicada no item **e.3)**;
 - e.2) **AMARRACAO**: Armazena uma *STRING*, composta pela seqüência

$$L(l) \leftarrow \{tokens \text{ obtidos do lugar } l\},$$
 indicadora dos conjuntos de *tokens* obtidos, por remoção ou leitura, de cada um dos lugares de entrada da transição disparada;
 - e.3) **CLASSE**: Armazena o número da classe sucessora imediata;
 - e.4) **ELO**: Armazena um ponteiro para a célula referente a outra das classes sucessoras imediatas da classe indicada no item **a)**;

- f) **CFC**: Armazena o número do **componente fortemente conexo** (veja apêndice I) possuidor da classe. Este campo é utilizado durante a verificação da propriedade de **vivacidade** (seção 13.3.3);
- g) **NÃO_REINICIAVEL**: Armazena inicialmente o valor booleano **FALSE**. Quando da verificação da propriedade de **reiniciabilidade** (seção 13.3.2), este campo passa a armazenar o valor booleano **TRUE**, caso a classe tenha como sucessora imediata a classe inicial C_0 (indicada pelo número 0) ou uma classe que leve à classe C_0 .

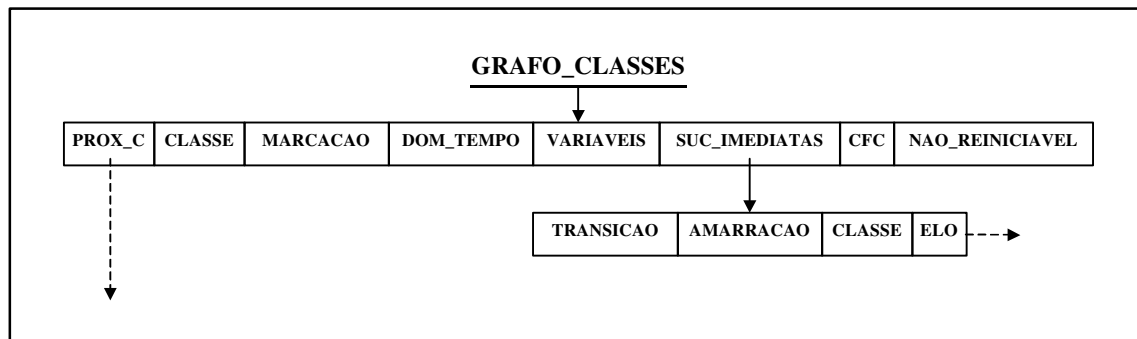


Figura 13.3. Estrutura de dados GRAFO_CLASSES.

A estrutura **CLASSES_PROVISORIAS** (figura 13.4) é utilizada para guardar provisoriamente as informações referentes às classes sucessoras imediatas da classe que está sendo simulada. Pode-se observar na figura 13.4 a existência de **três níveis** do mesmo tipo de subestrutura. O **primeiro nível** refere-se a uma **fila de classes cujas sucessoras imediatas devem ser calculadas**. Tais sucessoras são armazenadas no **segundo nível** da estrutura. O **terceiro nível** representa a **segunda geração de sucessoras do primeiro nível**. Os campos encontram-se repetidos nestes três níveis com o intuito de uma maior rapidez no processo de simulação. Após a simulação da primeira classe da fila de classes do primeiro nível, ocorre:

- 1) Sendo a primeira célula do primeiro nível uma classe não pertencente à estrutura **GRAFO_CLASSES**, o conteúdo do seu campo **SUCESSORAS** é transferido para o campo **ELO** da última célula do primeiro nível e as informações da nova classe são inseridas em **GRAFO_CLASSES**;
- 2) A primeira célula do primeiro nível é removida da fila; e
- 3) Passa-se à simulação da nova primeira célula do primeiro nível.

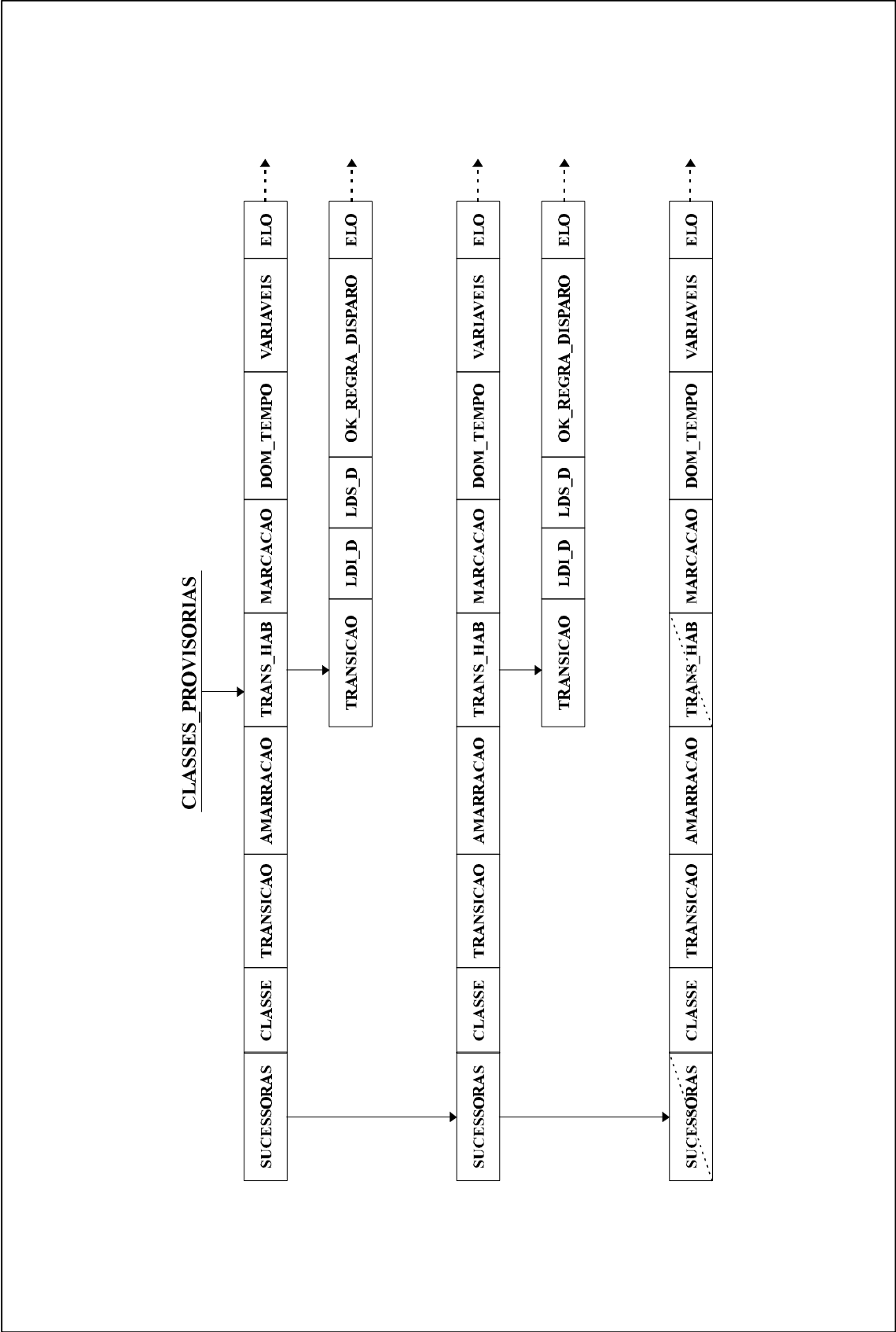


Figura 13.4. Estrutura de dados CLASSES_PROVISORIAS.

Os principais campos da estrutura **CLASSES_PROVISORIAS** são:

- a) **SUCCESSORAS**: Armazena um ponteiro que aponta para a fila de classes sucessoras imediatas da classe do nível a que pertence o campo. No último nível, assume o valor **NIL**, uma vez que as classes sucessoras do último nível só poderão ser calculadas quando as mesmas encontrarem-se no segundo nível;
- b) **CLASSE**: Armazena o identificador numérico associado à classe;
- c) **TRANSICAO**: Armazena o identificador numérico da transição cujo disparo levou a rede da classe antecessora para a classe sucessora. Útil somente nos níveis 2 e 3 das estruturas;
- d) **AMARRACAO**: Armazena uma *STRING*, composta pela seqüência

$$L(I) \leftarrow \{tokens \text{ obtidos do lugar } I\},$$
indicadora dos conjuntos de *tokens* obtidos, por remoção ou leitura, de cada um dos lugares de entrada da transição disparada. Útil somente nos níveis 2 e 3 da estrutura;
- e) **TRANS_HAB**: Armazena um ponteiro para uma lista das transições habilitadas na classe;
- f) **MARCACAO**: Armazena uma *STRING*, representante da marcação da classe;
- g) **DOM_TEMPO**: Armazena uma *STRING*, composta pela seqüência **t[LDI,LDS]**, onde **t** é uma transição habilitada na classe, **LDI** o seu limite de disparo inferior e **LDS** o seu limite de disparo superior. Este campo só é definido nos níveis 1 e 2 da estrutura;
- h) **VARIAVEIS**: Armazena uma *STRING*, representante da valoração das variáveis na classe;

Na seqüência, veremos como as estruturas de dados apresentadas são utilizadas no algoritmo de simulação para RPSD. Levando-se em conta as regras de funcionamento de RPSD dadas no capítulo 11, o algoritmo de simulação apresentado na figura 13.5 pode ser simplificado pelos seguintes passos:

- 1) Fazer inicializações;
- 2) Calcular, usando-se a lista **INICIALIZACAO** (para lugares do tipo I) e os campos **VALOR_INICIAL** (para lugares do tipo II) de **INFO_LUGARES**, a marcação **M₀** da classe **C₀** e adicioná-la à definição da classe **C₀**;
- 3) Inicializar **CLASSES_PROVISORIAS**;
- 4) Para cada transição de **INFO_TRANSICOES**, verificar as condições de habilitação em **C₀**. Caso a transição for habilitada para alguma amarração, armazená-la na lista de transições habilitadas (**TRANS_HAB**) da primeira entrada do nível 1 da estrutura de **CLASSES_PROVISORIAS**;
- 5) Definir o domínio de tempo **D₀** da classe inicial **C₀** e inseri-lo na definição da classe. Inserir **C₀** na estrutura **GRAFO_CLASSES**. Caso **D₀** seja vazio, encerrar o algoritmo. Caso contrário, prosseguir;
- 6) Fazer **C_{corrente}** igual a **C₀**;
- 7) Tomando a lista de transições habilitadas (**TRANS_HAB**) em **C_{corrente}**, verificar as que estão de acordo com a regra de disparo;
- 8) Para cada transição **t** que estiver de acordo com a regra de disparo, calcular todas as amarrações possíveis e, para cada amarração α que torne habilitada a transição, efetuar as

operações da transição e inserir a marcação $M_{t\alpha}$ da classe decorrente de $C_{corrente}$ pelo disparo de t , mediante a amarração α , na lista **SUCCESSORAS** da primeira entrada do nível 1 de **CLASSES_PROVISORIAS**. Neste instante, as sucessoras imediatas C_k da classe $C_{corrente}$ ainda não estão completamente definidas, pois falta a definição dos domínios de tempo das mesmas. Para definir esses domínios de tempo, deve-se continuar o algoritmo;

- 9) Para cada classe C_k pertencente à lista **SUCCESSORAS** da primeira entrada do nível 1 da estrutura **CLASSES_PROVISORIAS**, fazer:
 - 9.1) Para cada transição de **INFO_TRANSICOES**, verificar as condições de habilitação na classe C_k . Caso a transição seja habilitada para alguma amarração, armazená-la na lista de transições habilitadas da classe C_k ;
 - 9.2) Tomando a lista de transições habilitadas em C_k , verificar as que estão de acordo com a regra de disparo;
 - 9.3) Para cada transição s que estiver de acordo com a regra de disparo, calcular todas as amarrações possíveis e, para cada amarração α que torne habilitada a transição, efetuar as operações da transição e inserir a marcação $M_{s\alpha}$ da classe decorrente de C_k pelo disparo de s mediante a amarração α , na lista **SUCCESSORAS** da entrada do nível 2 de **CLASSES_PROVISORIAS**, correspondente a C_k . Neste instante, os domínios de tempo das sucessoras imediatas C_k da classe $C_{corrente}$ estão definidos e, conseqüentemente, as classes C_k estão completamente definidas;
- 10) Para cada classe C_k (agora totalmente definida) pertencente à lista **SUCCESSORAS** da primeira entrada do nível 1 da estrutura **CLASSES_PROVISORIAS**, verificar se a mesma é uma classe de **GRAFO_CLASSES**. São duas as possibilidades:
 - a) Classe C_k pertence a **GRAFO_CLASSES**: Neste caso, faz-se a atualização do campo **SUC_IMEDIATAS** da classe correntemente simulada ($C_{corrente}$) e desaloca-se a classe C_k da lista **SUCCESSORAS** da primeira entrada do nível 1 da estrutura **CLASSES_PROVISORIAS**;
 - b) Classe C_k não pertence a **GRAFO_CLASSES**: Neste caso, insere-se as informações de C_k ao final de **GRAFO_CLASSES**, faz-se a atualização do campo **SUC_IMEDIATAS** da classe correntemente simulada ($C_{corrente}$) e desloca-se o ponteiro que aponta para a classe C_k da lista **SUCCESSORAS** da primeira entrada do nível 1 da estrutura **CLASSES_PROVISORIAS** para o último campo **ELO** do nível 1 de **CLASSES_PROVISORIAS**;
- 11) Desalocar a primeira entrada do nível 1 da estrutura **CLASSES_PROVISORIAS**. Caso **CLASSES_PROVISORIAS** fique vazia, encerrar a simulação. Caso contrário, fazer $C_{corrente}$ igual à classe correspondente à primeira entrada do nível 1 da estrutura **CLASSES_PROVISORIAS** e repetir os passos do algoritmo a partir do passo 7.

Ao final da simulação, a estrutura **GRAFO_CLASSES** conterá as informações necessárias à construção do **grafo de classes** da rede. Nas seções seguintes, será apresentado um conjunto de propriedades que podem ser verificadas a partir da simulação de modelos RPSD. A análise destas propriedades baseia-se nas alterações feitas nas estruturas **INFO_LUGARES** e **INFO_TRANSICOES** e no algoritmo de simulação de RPSD.

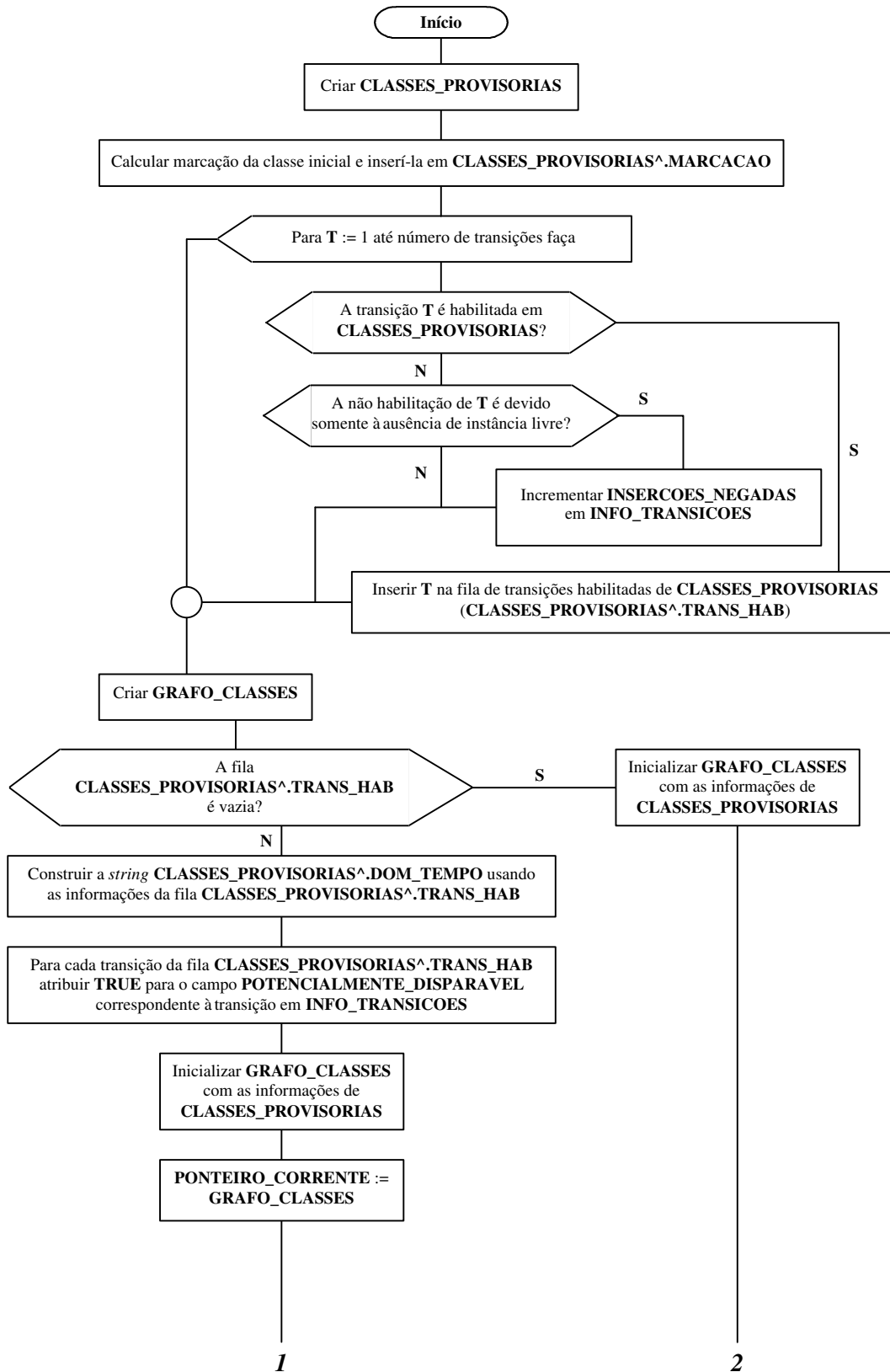
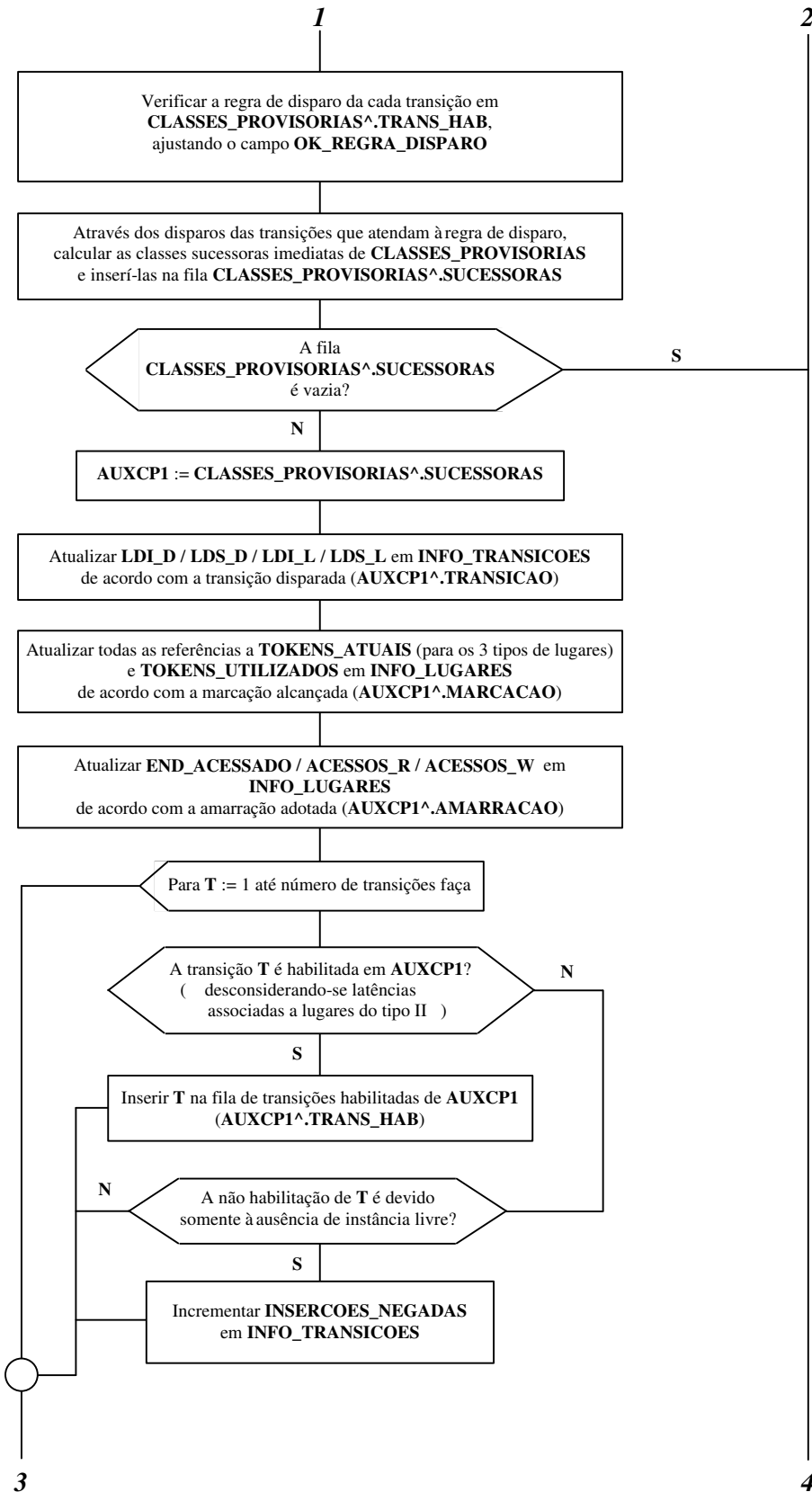
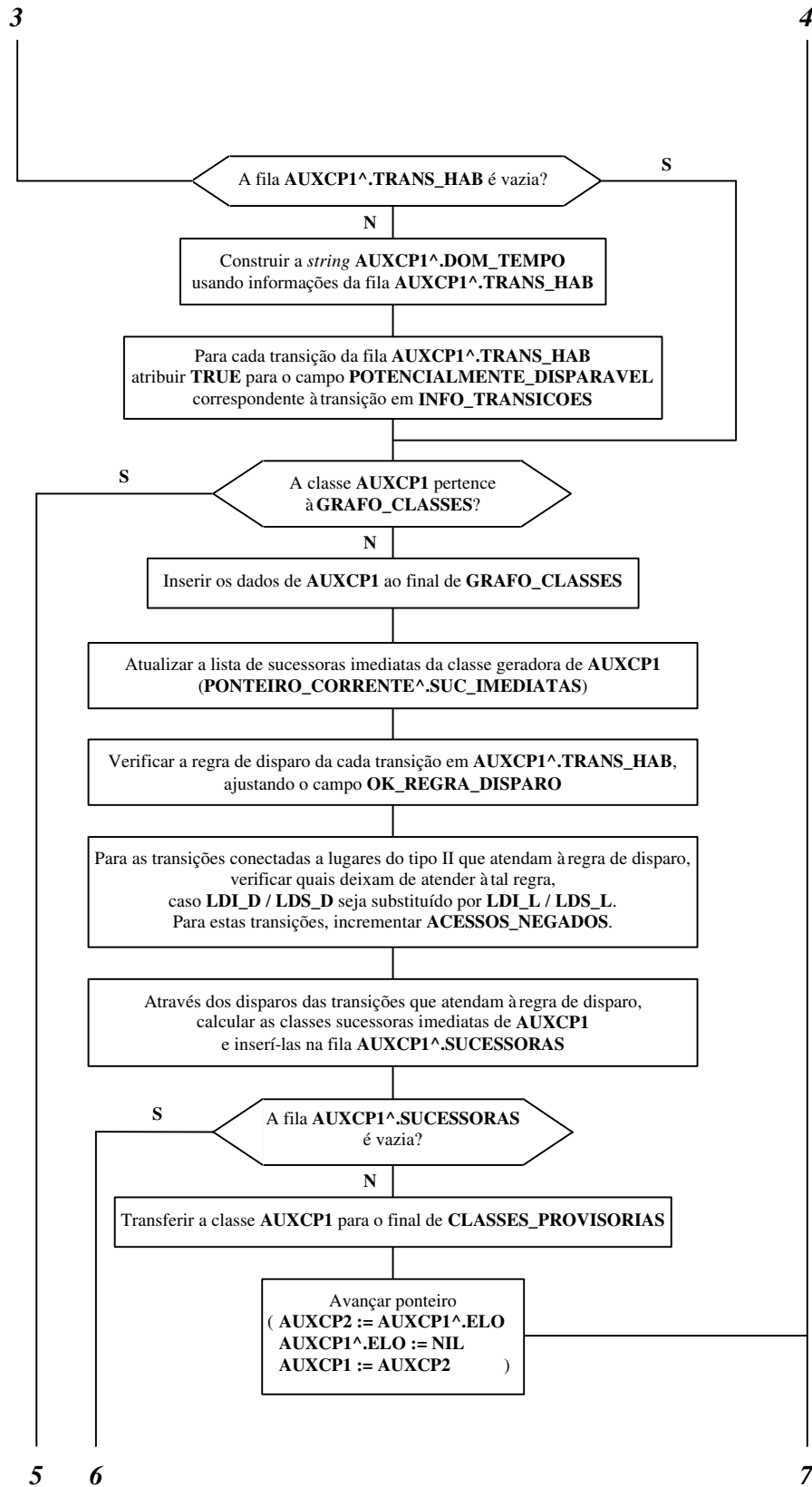


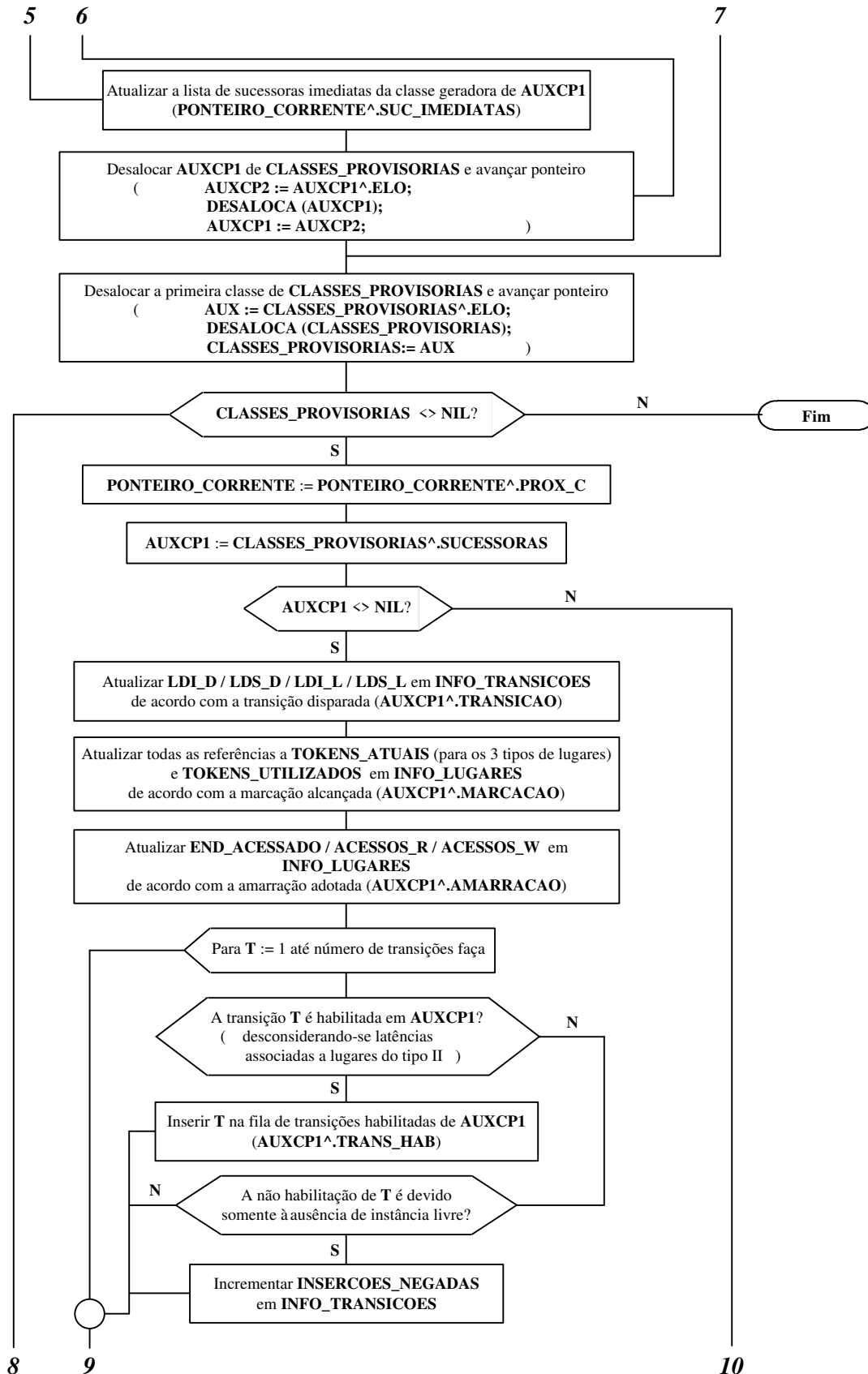
Figura 13.5. Fluxograma do algoritmo de simulação para RPSD.



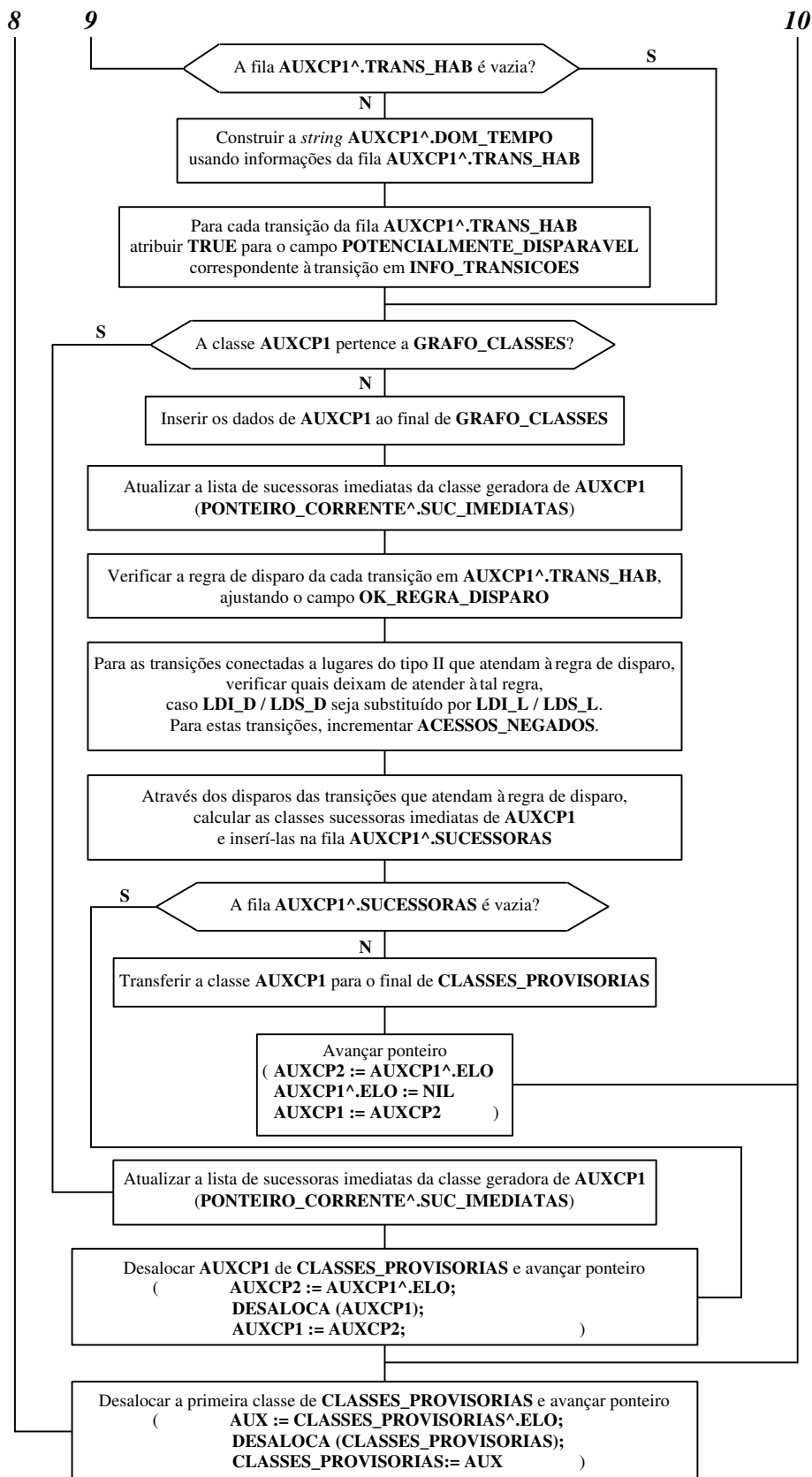
Continuação da figura 13.5.



Continuação da figura 13.5.



Continuação da figura 13.5.



Continuação da figura 13.5.

13.3. ANÁLISE DE PROPRIEDADES

Na teoria de redes de Petri, as **propriedades** de um modelo são responsáveis pela **validação ou não** de propriedades do sistema modelado. As principais propriedades estudadas em modelos baseados em redes de Petri são conhecidas como **limitabilidade**, **reiniciabilidade** e **vivacidade**. Outras propriedades são estudadas, de acordo com a extensão adotada. O conceito de **análise** de um modelo em redes de Petri está relacionado com a **verificação** de propriedades no modelo.

As diferentes extensões de rede de Petri apresentam diferentes **técnicas de análise de propriedades**, elaboradas a partir das características particulares de cada extensão. Basicamente, as **técnicas de análise** enquadram-se em dois grupos: técnicas de análise **dinâmicas** e técnicas de análise **estruturais**. As técnicas dinâmicas têm seu ponto de partida na **árvore de alcançabilidade** (máquina de senhas, grafo de estados, grafo de classes) da rede, obtida a partir da **simulação** do modelo. Estas técnicas dependem, portanto, da **marcação inicial** da rede (disposição dos *tokens* no início da simulação). Por outro lado, as técnicas estruturais baseiam-se na **interconexão dos componentes** que formam a rede (ou parte dela): **lugares** e **transições**. Estas técnicas **não dependem** da **marcação inicial** da rede.

Técnicas de análise dinâmicas foram desenvolvidas para **RPSD**, encontrando-se demonstradas nesta seção, a qual descreve como a **simulação** de modelos RPSD permite a **verificação** de um conjunto ilustrativo de **propriedades**, no sistema modelado. Após a geração do **grafo de classes** de uma RPSD, estas propriedades podem ser inferidas a partir de informações coletadas junto às **estruturas de dados** manipuladas pelo **algoritmo de simulação** da seção 13.2 (ilustrado no fluxograma da figura 13.5).

Técnicas de análise estruturais para RPSD são tratadas no próximo capítulo.

Para melhor compreensão dos termos inerentes à **Teoria dos Grafos**, sugere-se a leitura do **apêndice I** desta tese.

13.3.1. LIMITABILIDADE

Nas redes de Petri de baixo nível, como as **redes de Petri Lugar/Transição** e as **redes de Petri Temporizadas de Merlin**, um modelo é definido como **limitado** a um valor inteiro **n** se, para qualquer marcação **M** decorrente da marcação inicial **M₀**, o número de *tokens* em cada lugar da rede for menor ou igual a **n**. Numa **rede limitada**, o **número de marcações** é **finito**. Por outro lado, numa **rede não limitada**, há **infinitas marcações**, significando a **impossibilidade** de **implementação física** do sistema modelado.

Conforme visto na seção 11.2.2, RPSD estende o conceito de **marcação** para o de **classe**. Uma **classe** representa um **estado** do sistema modelado. A classe é descrita por uma tripla $C_i = (M_i, D_i, V_i)$, $i \in \mathbb{IN}$, onde **M_i** é a **marcação** da rede, **D_i** o seu **domínio de tempo** e **V_i** a **lista de variáveis** da rede e seus respectivos valores. Assim, traçando-se um paralelo com a definição de limitabilidade para redes de baixo nível, um modelo RPSD é **limitado** se o número de **classes decorrentes** de uma dada classe inicial **C₀** for finito. Isto ocorre quando é finito o número de **combinações** envolvendo **marcações**, **domínios de tempo** e **valorações**

de variáveis. Conseqüentemente, a prova de limitabilidade é feita verificando-se a finitude de cada um destes três conjuntos, para uma dada classe inicial. É o que faremos na seqüência.

A finitude de **marcações** é comprovada para qualquer modelo RPSD, pois, por definição, seus lugares têm quantidade de *tokens* limitada, da seguinte forma:

- a) **Lugares do tipo I:** A quantidade de *tokens* de um lugar **P** do tipo I é limitada pela cardinalidade do *domínio* $D_S(\mathbf{P})$, uma vez que $D_S(\mathbf{P})$ é um conjunto finito e o *repositório* $R_S(\mathbf{P})$ não admite cópias múltiplas de um mesmo *token* $\mathbf{d}_i$ ($\mathbf{d}_i \in D_S(\mathbf{P})$);
- b) **Lugares do tipo II:** A quantidade de *tokens* de um lugar **P** do tipo II é limitada pela cardinalidade do *espaço de endereçamento* $A(\mathbf{P})$, o qual define a identificação das *células de armazenamento* do lugar (*vetor conteúdo* $C(\mathbf{P})$);
- c) **Lugares do tipo III:** A quantidade de *tokens* de um lugar **P** do tipo III é limitada, uma vez que as diferentes partes de **P** têm número de *tokens* finito:

- c.1) **Repositório de entrada** $R_I(\mathbf{P})$: Sua quantidade de *tokens* é limitada em

$$\#D_I(\mathbf{P}) \times \max_S(\mathbf{P}),$$

onde $\#D_I(\mathbf{P})$ é a cardinalidade do *domínio de entrada* $D_I(\mathbf{P})$ e $\max_S(\mathbf{P})$ é o *número máximo de samples* suportado por **P**;

- c.2) **Comportamento** $B(\mathbf{P})$: Sua quantidade de *tokens* é limitada, pois existirão, no máximo, $\max_S(\mathbf{P})$ *instâncias* do mesmo, onde cada instância é uma **RPSD-T** ou uma **rotina**. Como uma rotina pode ser descrita por uma RPSD-T e, como RPSD-T é formada por lugares dos tipos I, II e III, conclui-se que tal tipo de rede tem quantidade limitada de *tokens*. Logo, $B(\mathbf{P})$ apresenta um número finito de *tokens*;

- c.3) **Repositório de saída** $R_O(\mathbf{P})$: Sua quantidade de *tokens* é limitada em

$$\#D_O(\mathbf{P}) \times \max_S(\mathbf{P}),$$

onde $\#D_O(\mathbf{P})$ é a cardinalidade do *domínio de saída* $D_O(\mathbf{P})$ e $\max_S(\mathbf{P})$ é o *número máximo de samples* suportado por **P**.

A quantidade de diferentes **domínios de tempo** é limitada para qualquer modelo em RPSD, devido ao **mecanismo de atualização do tempo da rede**, utilizado quando do disparo de uma transição. Este mecanismo, descrito na seção 11.2.2, baseia-se na subtração de determinados números inteiros, os quais são utilizados como **limites de disparo inferior (LDI)** e **superior (LDS)** dos **intervalos de disparo** associados às transições da rede. Como haverá um valor máximo para **LDS** e, como **LDI** não pode ser inferior a 0 (zero), haverá uma faixa finita de números inteiros empregados como limites **LDI** e **LDS dinâmicos**. Assim, haverá um número finito de possíveis intervalos de disparo e, conseqüentemente, um número finito de possíveis domínios de tempo.

As variáveis utilizadas nos rótulos do ramos e nas guardas e operações das transições, compõem a **memória da rede** (veja definição na seção 11.2.2). **Valoração** é o nome dado ao conjunto de valores associados à memória da rede, numa dada classe. O número de valorações será **finito** se a **memória da rede** empregar somente **tipos finitos**. Caso contrário, o número de valorações **poderá** ser infinito. Os tipos das variáveis da rede são definidos no conjunto **Mem** da RPSD.

Desta forma, pelo exposto nesta seção, uma RPSD é **limitada** para uma dada classe inicial $\mathbf{C}_0$ se o seu grafo de classes for **finito**, o que ocorre somente se todas as **variáveis**

usadas na rede assumirem um conjunto **finito** de valores, o que dependerá da definição do conjunto **Mem (memória da rede)** associado ao modelo RPSD.

13.3.2. REINICIABILIDADE

Um modelo em rede de Petri de baixo nível é definido como **reiniciável** para uma dada marcação inicial M_0 se, para qualquer marcação decorrente de M_0 , existir uma **seqüência de disparo de transições** que faça a rede voltar à marcação inicial M_0 . O reflexo desta propriedade no **sistema real** se dá através da **possibilidade** deste **retornar** ao seu **estado inicial**, após a execução de uma ou mais tarefas.

Em relação à RPSD, uma rede **limitada** é **reiniciável** para uma dada classe inicial C_0 se, para qualquer classe C_k decorrente da mesma, existir um caminho no **grafo de classes** com início em C_k e término em C_0 . Uma RPSD não limitada não pode ser reiniciável.

Para a **análise** desta propriedade em modelos RPSD, desenvolvemos um **algoritmo** capaz de determinar se uma RPSD é reiniciável ou não. Este algoritmo tem como entrada o **grafo de classes** gerado pela simulação da rede, o qual é armazenado na estrutura de dados **GRAFO_CLASSES**, descrita na seção 13.2 (figura 13.3).

A figura 13.6 mostra o fluxograma do **algoritmo** de verificação da propriedade de **reiniciabilidade** (adaptado de [40]). Este algoritmo é bastante simples. Inicialmente, para cada classe de **GRAFO_CLASSES**, atribui-se para o campo **NAO_REINICIAVEL** o valor booleano **TRUE**. Em seguida, monta-se uma pilha com os números das classes que tenham C_0 como sucessora imediata. Na seqüência, sendo C_r uma classe dessa pilha, retira-se C_r da pilha, atribui-se **FALSE** ao seu campo **NAO_REINICIAVEL** e adiciona-se à pilha as classes que tenham C_r como sucessora imediata, e que tenham o seu campo **NAO_REINICIAVEL** igual a **TRUE**, se existirem tais classes. O algoritmo prossegue até que a pilha fique vazia. No final, basta verificar se existe alguma classe que tenha o seu campo **NAO_REINICIAVEL** em **TRUE**. Caso exista, a rede será **não reiniciável**; caso contrário, será **reiniciável**.

13.3.3. VIVACIDADE

Nas extensões de rede de Petri, uma rede é definida como **viva** perante uma dada marcação inicial M_0 se, para qualquer marcação decorrente de M_0 , e para cada transição t da rede, existir uma **seqüência de disparo de transições** que faça a rede atingir uma marcação na qual t seja **habilitada**. Caso isto ocorra, o sistema físico correspondente estará livre de situações de **deadlock** e **livelock**, isto é, o sistema real não possuirá situações de bloqueio.

Em relação à RPSD, uma rede **limitada** é **viva** para uma dada classe inicial C_0 se, para qualquer classe decorrente C_k e para uma transição t considerada, existir uma seqüência σ de disparo de transições, de forma que, ao final dessa seqüência, a transição t se encontre habilitada.

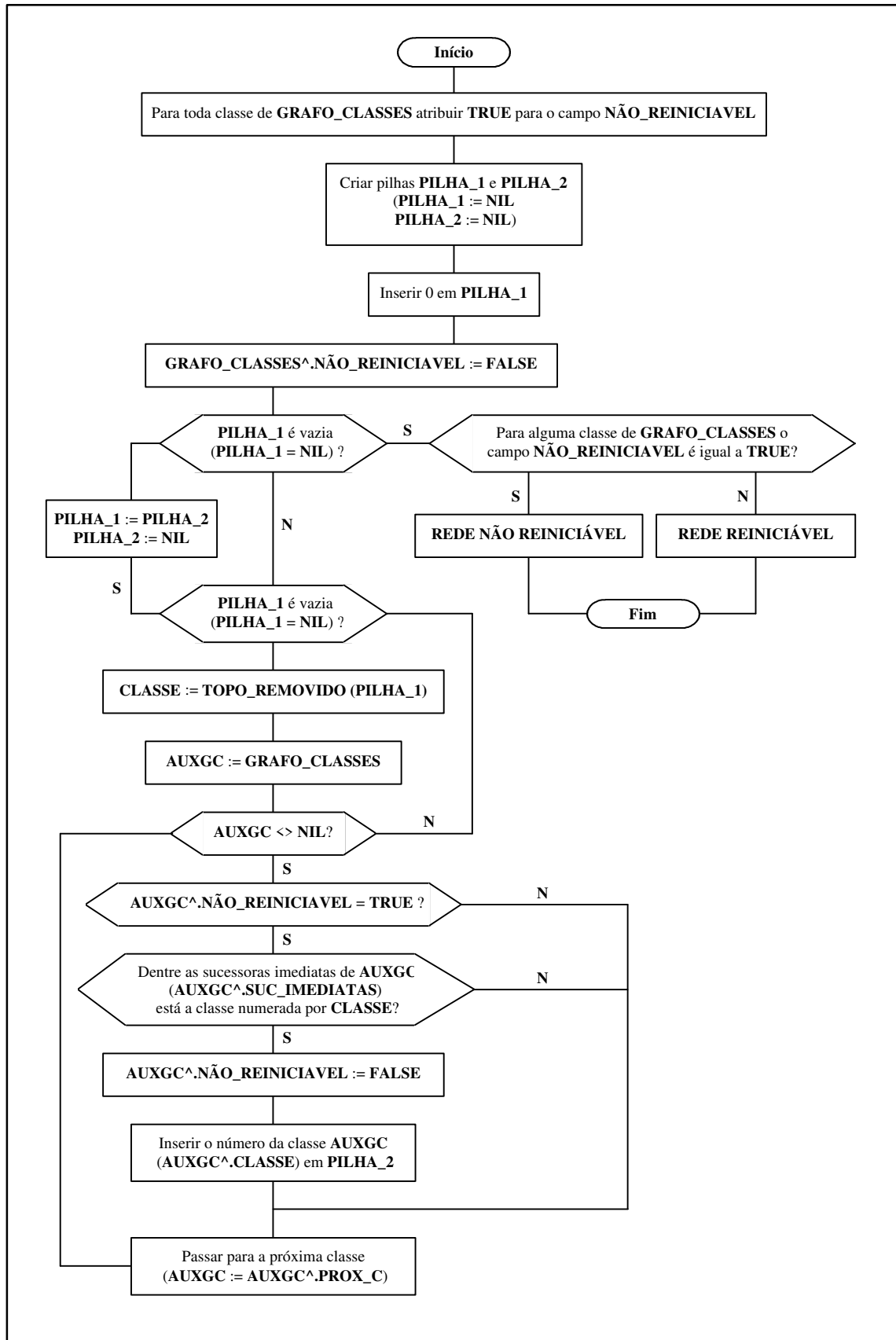


Figura 13.6. Fluxograma do algoritmo de verificação de reiniciabilidade.

A estrutura de dados **GRAFO_CLASSES**, definida na seção 13.2, também é utilizada pelo **algoritmo** de verificação da propriedade de **vivacidade** (fluxograma na figura 13.7). Este algoritmo baseia-se nos dois casos seguintes:

Caso 1: RPSD reiniciável.

Quando a rede for **reiniciável** e todas as **transições** forem **potencialmente disparáveis**, a rede é **viva**. Uma transição é **potencialmente disparável** quando, pelo menos uma vez, fica **habilitada** em alguma classe C_i decorrente de C_0 . Uma vez que, a partir de qualquer classe pode-se retornar à classe inicial e a partir dessa chega-se a qualquer classe, basta verificar se todas as transições são potencialmente disparáveis. Assim, o algoritmo de vivacidade limita-se a pesquisar se existe alguma transição de **INFO_TRANSICOES** com o valor booleano **FALSE** no campo **POTENCIALMENTE_DISPARAVEL**. Caso exista alguma transição nesta situação, a rede é **não viva**; caso contrário, é **viva**.

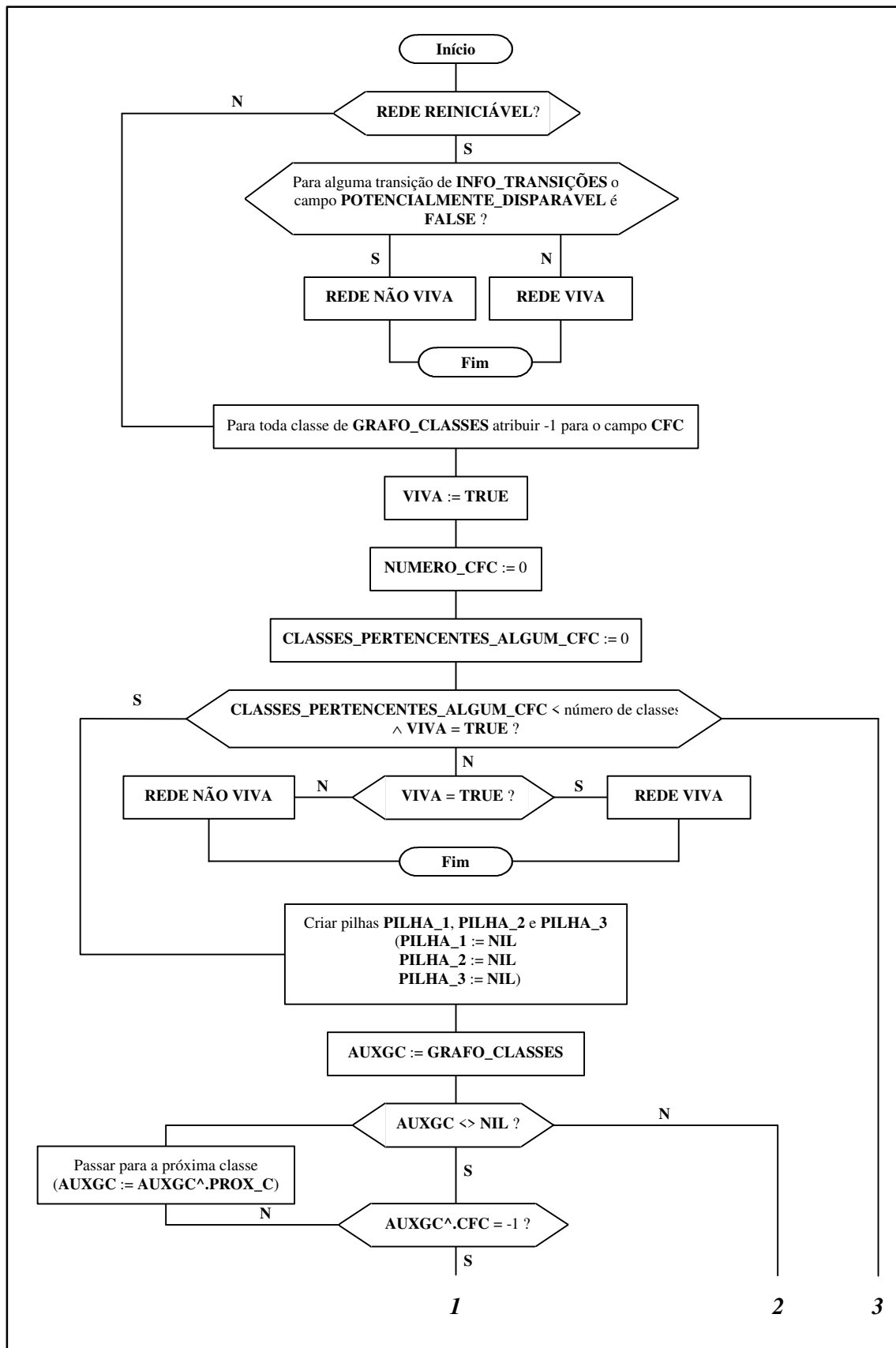
Caso 2: RPSD não reiniciável.

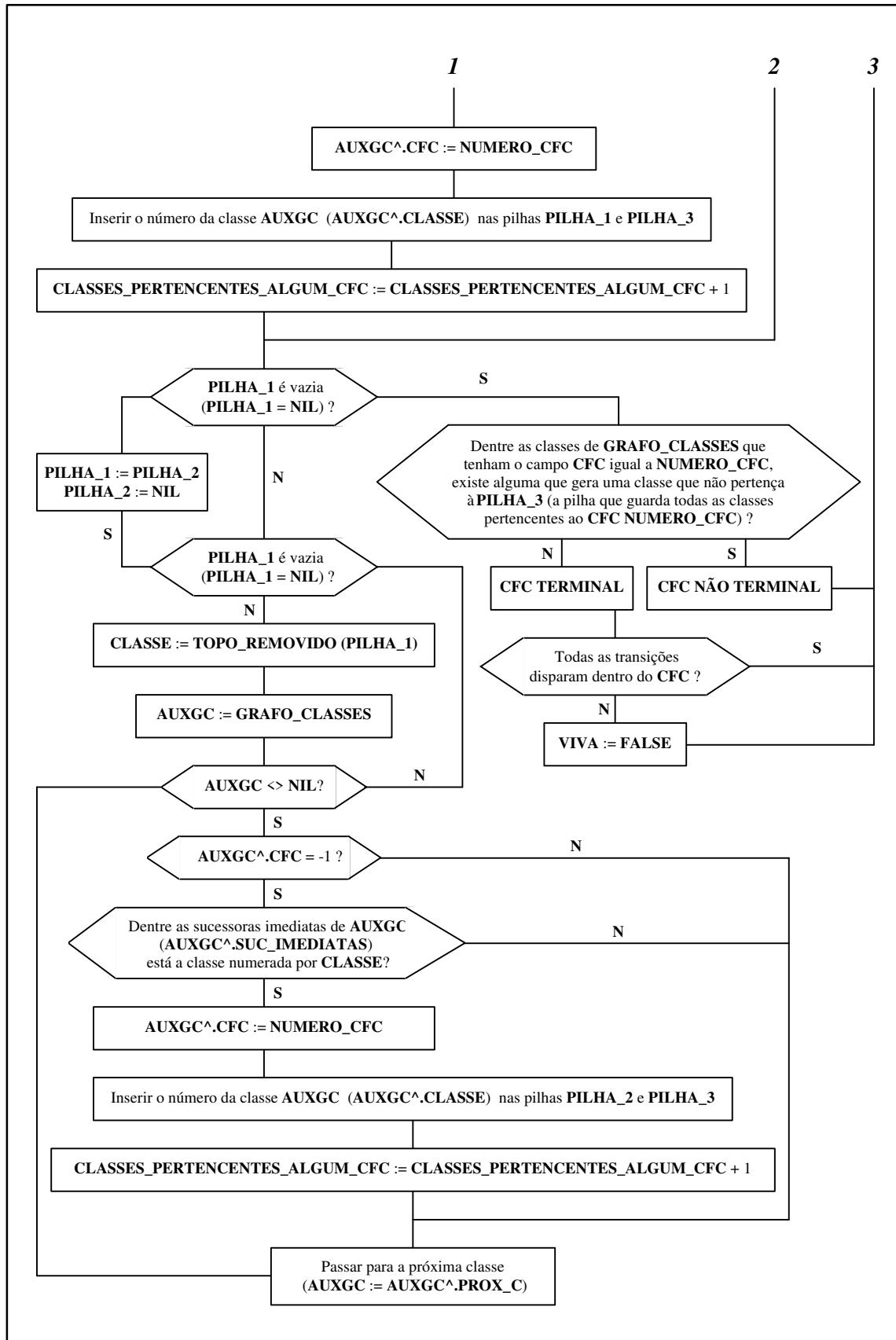
De acordo com o exposto no apêndice I desta tese, um grafo de classes é **fortemente conexo** se existir, para duas classes quaisquer C_i e C_j consideradas nessa ordem, um caminho com início em C_i e término em C_j .

Um **grafo de classes** qualquer G de n classes pode ser **dividido** em vários **subgrafos de classes** fortemente conexos $S_1, S_2, \dots, S_q$, com $n_1, n_2, \dots, n_q$ classes cada um. Assim, tais subgrafos são chamados **componentes fortemente conexos (CFC's)**.

Utilizando-se dos resultados da referência [40] para redes Lugar/Transição e, considerando-se o **rótulo** de um **arco** do **grafo de classes** como sendo o número associado ao identificador da transição cujo disparo (mediante uma dada **amarração**) conduz a rede de uma **classe origem** (a origem do arco) a uma **classe destino** (a ponta do arco), tem-se que RPSD é **viva** se, e somente se, **cada transição** da rede for **rótulo** de **pelo menos um** dos **arcos do grafo de classes** e existirem arcos que se dirigem dos componentes fortemente conexos, que não contém todas as transições como rótulos de seus arcos, aos componentes que as contém.

Cada classe (nó) do grafo pertence a um único **CFC**. Chamando de **terminais** os **CFC's** que não têm saída (ramo) para qualquer outro **CFC**, o **algoritmo** de validação da propriedade de vivacidade verifica se os **CFC's** terminais têm seus arcos rotulados com **todas** as transições. Em caso negativo, a rede é **não viva**, uma vez que o **CFC** não tem saída e existe alguma transição que não vem a ser habilitada em classe alguma do mesmo. Em caso positivo, prossegue-se o algoritmo, até que todas as classes pertençam a algum dos **CFC's**. A rede só é viva se todas as transições forem rótulos dos arcos de todos os **CFC's** terminais, tomados um a um.





Continuação da figura 13.7.

13.3.4. TRANSIÇÕES NÃO DISPARADAS

Transições representam **eventos** do sistema modelado, os quais podem ocorrer ou não. O disparo de uma transição representa a ocorrência do evento relacionado. Este disparo dependerá das **condições de habilitação** da transição (relativas aos lugares e ramos conectados à transição, além de variáveis utilizadas no controle de fluxo da rede) e do atendimento de **certas restrições temporais**.

Neste contexto, faz-se importante verificar quais **transições não disparam**, ao longo da **simulação da rede**, para corrigir eventuais **erros de especificação** do sistema, uma vez que, em última análise, tais transições representarão um custo desnecessário na implementação do sistema correspondente. Além disso, faz-se necessário identificar o **motivo** pelo qual uma transição não disparou durante a simulação, dentre dois possíveis:

- i) A transição não disparou devido à sua **não habilitação**: neste caso, as **condições** envolvendo lugares, ramos e variáveis, relacionados à transição, não foram satisfeitas em classe alguma da rede (a classe representa o estado global do sistema); e
- ii) A transição chegou a ficar **habilitada** em alguma classe, mas **não disparou**: neste caso, as **restrições temporais** não foram atendidas, seja pela especificação do **intervalo de disparo** da transição, ou seja pela especificação das **latências de leitura/escrita**, envolvendo lugares de entrada/saída do tipo II (armazenamento), conectados à transição.

Para cada uma das transições da rede modelada, a estrutura **INFO_TRANSICOES** (figura 13.2) fornece o campo booleano **POTENCIALMENTE_DISPARAVEL**, o qual expressa se a transição foi habilitada ao menos uma vez, durante a simulação. No início da simulação, este campo recebe o valor **FALSE**. Durante a simulação, as transições que atendem as suas condições de habilitação, para alguma amarração, são inseridas na lista **TRANS_HAB** de transições habilitadas da classe correntemente simulada, pertencente à estrutura **CLASSES_PROVISORIAS** (figura 13.4). Para cada uma das transições de **TRANS_HAB**, o campo **POTENCIALMENTE_DISPARAVEL** recebe o valor **TRUE**. Após a montagem da lista **TRANS_HAB**, verifica-se quais das transições habilitadas atendem as regras de disparo, conforme especificado na seção 11.2.1.3. Para cada transição habilitada de acordo com as regras de disparo, o algoritmo de simulação gerará classes decorrentes da classe correntemente simulada, adotando todas as amarrações possíveis. As classes geradas são armazenadas na estrutura **GRAFO_CLASSES** (figura 13.3), na qual cada entrada da lista **SUC_IMEDIATAS**, formada de células de três campos (**TRANSICAO**, **AMARRACAO**, **CLASSE**), denota a **classe alcançada** através do disparo de uma **transição**, mediante uma **amarração**, a partir da classe correntemente simulada.

Desta forma, ao final da simulação, é possível obter informações que distingam as transições não disparadas, da seguinte forma:

i) **Transições que não ficaram habilitadas em classe alguma.**

Percorre-se a estrutura **INFO_TRANSICOES**, listando as transições que apresentam o valor **FALSE** no campo **POTENCIALMENTE_DISPARAVEL**;

ii) **Transições que ficaram habilitadas mas não dispararam devido às regras de disparo.**

Percorre-se a estrutura **INFO_TRANSICOES**, listando as transições com as seguintes características:

- valor **TRUE** no campo **POTENCIALMENTE_DISPARAVEL**;

- identificador ausente no campo **TRANSICAO** das listas **SUC_IMEDIATAS** de todas as classes da estrutura **GRAFO_CLASSES**.

Assim, foi demonstrado como a verificação de transições não disparadas pode ser feita em RPSD, através da simulação da rede.

13.3.5. TOKENS NÃO UTILIZADOS EM LUGARES DO TIPO I

Lugares do tipo I são utilizados para indicar que um ou mais **componentes análogos do sistema** (como, por exemplo, processos, unidades funcionais, unidades de armazenamento ou unidades de comunicação) encontram-se num **mesmo estado local** do sistema modelado. Um componente **C** do sistema encontra-se num dado estado local **S** quando o *token* representante de **C** (simplicadamente, *token C*) está presente no lugar representante de **S** (lugar **S**). O *token C* deve pertencer ao **domínio** do lugar, o qual descreve o conjunto de componentes que podem assumir o estado local correspondente.

Assim, considerando-se a simulação de um sistema modelado em RPSD, a **não inserção** de um determinado *token C* em um lugar **S** do tipo I, que o aceitaria, indica que o **componente C não assume**, em instante algum, o **estado local S**. Isto pode ser decorrente de um erro na **especificação** do estado local **S** (não fazer sentido **C** apresentar o estado local **S**), ou na especificação dos **eventos** que levariam **C** a assumir o estado local **S** (erro na especificação das transições relacionadas ao lugar **S**).

A estrutura **INFO_LUGARES** (figura 13.1) apresenta, para cada lugar do tipo I, o campo **TOKENS_UTILIZADOS**, o qual armazena um ponteiro para a lista de *tokens* que ocupam o lugar, ao menos uma vez, durante a simulação da rede. O fluxograma da figura 13.5 mostra os pontos de atualização da lista **TOKENS_UTILIZADOS**.

Para identificar quais *tokens não são utilizados* por um lugar do tipo I, basta verificar quais *tokens pertencem* à lista descritora de seu **domínio** (apontada pelo campo **DOM**) e **não pertencem** à lista **TOKENS_UTILIZADOS**.

13.3.6. ENDEREÇOS NÃO ACESSADOS EM LUGARES DO TIPO II

Lugares do tipo II são utilizados para representar **componentes de armazenamento do sistema** modelado (tais como registradores, *buffers* e memórias), nos quais operações de **leitura/escrita** de dados (representados por *tokens*) podem ser feitas. Um lugar deste tipo descreve um conjunto unidimensional de **células de armazenamento**, cada qual acessada por um **endereço**. Para representar conjuntos bidimensionais, pode-se utilizar dois ou mais lugares. **Operações de leitura/escrita** são acionadas por **eventos** (representados por transições) e envolvem **endereços variáveis** (descritos por **expressões** anexadas aos ramos relacionados). Tais operações respeitam **restrições temporais**, dadas pelas **latências** de **leitura** e de **escrita** associadas ao componente do sistema real.

Após a simulação de um modelo RPSD, pode ocorrer de determinados **endereços não terem sido acessados**. Isto configura, no sistema real, um **desperdício de recursos**, o qual

pode ser evitado (ou atenuado) com a escolha de outros componentes de armazenamento ou, ainda, com modificações no esquema de endereçamento. Assim, faz-se importante a **identificação** de tais endereços.

O conjunto de células de armazenamento de um lugar do tipo II é denominado **vetor conteúdo**. Na estrutura **INFO_LUGARES** (figura 13.1), o campo **TIPO_II.CONTEUDO** acessa uma lista de informações utilizadas na manipulação do vetor conteúdo associado a um lugar do tipo II. Cada entrada da lista **TIPO_II.CONTEUDO** relaciona-se a um **endereço** e possui, dentre outros atributos, a indicação de **acesso** (ou não) do endereço, através do campo **END_ACESSADO**. No início da simulação, este campo recebe o valor **FALSE**. Durante a simulação, toda vez que uma transição conectada ao lugar **dispara**, as entradas de **TIPO_II.CONTEUDO**, correspondentes aos endereços envolvidos, têm seu campo **END_ACESSADO** atualizado para **TRUE**. Tal atualização encontrada-se relacionada no fluxograma da figura 13.5.

Desta forma, para identificar quais **endereços não são acessados** pelos lugares do tipo II, deve-se percorrer a lista **TIPO_II.CONTEUDO** de cada lugar, verificando quais entradas têm o campo **END_ACESSADO** igual a **FALSE**.

13.3.7. QUANTIDADE DE ACESSOS EM LUGARES DO TIPO II

Outra importante propriedade relacionada a componentes de armazenamento, que pode ser calculada a partir dos resultados de simulação de modelos RPSD, é a **quantificação** dos **acessos** a tais componentes. Para melhor precisão, esta quantificação pode ser dividida quanto ao **tipo** de acesso: **leitura** ou **escrita**. Esta separação é importante, uma vez que pode ajudar na alocação dos componentes de armazenamento do sistema.

Após a simulação do sistema modelado, a verificação da **quantidade de acessos de leitura e de escrita**, associada a um lugar do tipo II, pode ajudar na determinação do componente mais adequado, conforme ilustrado nos casos seguintes:

i) **Lugar apresenta somente acessos de leitura.**

Desconsiderando-se erros na especificação do lugar, o componente correspondente poderá ser substituído por uma memória apenas de leitura ou por um circuito *hardwired*, o que dependerá de outros fatores como, por exemplo, seu espaço de endereçamento e a arquitetura adotada para o sistema;

ii) **Lugar apresenta somente acessos de escrita.**

Salvo erros de especificação e casos nos quais o sistema modelado faça parte de um sistema maior, o componente correspondente deverá ser extinto, uma vez que não faz sentido escrever dados que não serão lidos;

iii) **Lugar apresenta grande diferença entre acessos de leitura e de escrita.**

Neste caso, pode-se decompor o componente em outros, levando a uma análise comparativa das alternativas possíveis, uma vez que componentes diferentes podem implicar em latências diferentes. Além disso, deve-se considerar que a decomposição de um componente pode alterar significativamente a arquitetura adotada.

Na estrutura **INFO_LUGARES** (figura 13.1), os campos **ACESSOS_R** e **ACESSOS_W** armazenam, respectivamente, a **quantidade de vezes** que o lugar foi **acessado** para **operações** de **leitura** e de **escrita**, durante o processo de simulação. No início da simulação, estes campos recebem o valor 0 (zero). Durante a simulação, toda vez que uma transição conectada ao lugar **dispara**, um destes campos é **incrementado**, de acordo com o **ramo** correspondente ser de **leitura** ou de **escrita** (a informação do **tipo de ramo** encontra-se no campo **INFO** de **INFO_TRANSICOES** – figura 13.2). Este incremento é igual ao **número de endereços acessados** na operação. Por exemplo, se a transição disparada estiver relacionada com o lugar através de um ramo de escrita que faz a atualização de 8 endereços, o campo **ACESSOS_W** do lugar será incrementado em 8. A atualização dos campos **ACESSOS_R** e **ACESSOS_W** encontrada-se ilustrada no fluxograma da figura 13.5.

Concluída a simulação da RPSD, a **quantidade de acessos** de **leitura** e de **escrita**, relacionadas a um lugar do tipo II, são obtidas, respectivamente, dos campos **ACESSOS_R** e **ACESSOS_W**, da entrada correspondente ao lugar, em **INFO_LUGARES**.

13.3.8. VERIFICAÇÃO DA ADEQUAÇÃO DAS LATÊNCIAS EM LUGARES DO TIPO II

Conforme relatado na subseção anterior (13.3.7), lugares do tipo II são importantes na escolha dos componentes de armazenamento do sistema modelado. Neste contexto, elaboramos a conceitualização e a forma de verificação de uma outra propriedade, qual seja, a **adequação das latências** de **leitura** e de **escrita**, a partir da **quantificação** de **acessos negados** a um lugar deste tipo, mediante o **não disparo** das transições relacionadas, devido, **somente**, às **restrições de latência**. Ou seja, após a simulação da rede, quantas vezes uma **transição não disparou**, porque não havia transcorrido **tempo** suficiente desde a **última operação** envolvendo o lugar, tempo este determinado pela multiplicação

$$(\text{número de endereços acessados}) \times (\text{latência de leitura ou de escrita}). \quad (I)$$

Em relação ao sistema modelado, esta **quantificação** de **acessos negados** pode auxiliar na **escolha** de componentes mais apropriados, de acordo com as **restrições de projeto** (componentes mais rápidos *versus* componentes mais baratos). Componentes mais lentos (e baratos) tendem a elevar a quantidade de acessos negados, o que pode deteriorar o desempenho, ao passo que, componentes mais rápidos (e caros) tendem a diminuir tal quantidade, o que pode resultar numa melhora de desempenho.

O campo **ACESSOS_NEGADOS** da estrutura **INFO_TRANSICOES** (figura 13.2) armazena, durante a simulação, a **quantidade de vezes** que a transição correspondente **não disparou**, devido, **somente**, à ação das **latências** de **leitura/escrita** de algum lugar do tipo II, conectado à transição. No início da simulação, este campo recebe o valor 0 (zero). Durante a simulação, as transições que atendem as suas condições de habilitação, para alguma amarração, são inseridas na lista **TRANS_HAB** de transições habilitadas da classe correntemente simulada, pertencente à estrutura **CLASSES_PROVISORIAS** (figura 13.4). Após a montagem de **TRANS_HAB**, verifica-se quais das suas transições atendem as **regras de disparo** (veja seção 11.2.1.3), e atribui-se **TRUE** ao campo **OK_REGRA_DISPARO** correspondente. Para as **transições conectadas a lugares do tipo II** que **atendem as regras de disparo**, verifica-se quais **deixam de atender tais regras**, caso seus **limites de disparo** inferior e superior **dinâmicos** (**LDI_D** e **LDS_D**) sejam substituídos por limites de disparo que levem as **latências** em consideração (**LDI_L** e **LDS_L**): para estas transições,

incrementar o campo **ACESSOS_NEGADOS**. Ao longo do processo de simulação, os limites **LDI_L** e **LDS_L** de uma transição t_i , conectada a um lugar **P** do tipo II, são atualizados sempre que ocorre o disparo de alguma transição t_j conectada a **P**. Esta atualização consiste na soma do resultado da multiplicação (*I*), considerando-se a transição disparada t_j , com os limites **LDI_D** e **LDS_D** da transição t_i .

Após a simulação da rede, a **quantidade de vezes** que uma **transição conectada a um lugar do tipo II não disparou**, devido, **somente**, às **restrições temporais** impostas pelas **latências de leitura/escrita** do lugar, é obtida consultando-se o valor armazenado no campo **ACESSOS_NEGADOS**, da entrada correspondente à transição, em **INFO_TRANSICOES**.

13.3.9. VERIFICAÇÃO DE SUBDIMENSIONAMENTO EM LUGARES DO TIPO III

Lugares do tipo III, ou funcionais, servem para representar, cada qual, um **conjunto de funções idênticas**, que podem ser executadas em **paralelo**, e que possuem **parâmetros** de **entrada** e de **saída** bem definidos. A **função** associada a um lugar funcional é denominada **comportamento**, o qual pode ser representado por uma listagem de comandos ou por outra rede, chamada RPSD-Transformacional, permitindo hierarquização. Denomina-se **instâncias** as cópias do **comportamento**, as quais são limitadas ao valor **número máximo de samples** (*max_s*). Cada **sample** corresponde a um conjunto de parâmetros para o **comportamento** e deverá ser processado por uma **instância livre**. Quando todas as **instâncias** estiverem **ocupadas** (em **execução** ou na **espera de remoção** de seu **sample** já **processado**), nenhum **sample** poderá ser inserido no lugar.

Assim, pelo exposto, é importante verificar se o valor **número máximo de samples** de um lugar funcional está **corretamente especificado** pois, se ele estiver **subdimensionado** (abaixo do necessário), poderá provocar **gargalos de processamento** no sistema físico correspondente e, se ele estiver **superdimensionado** (acima do necessário), poderá acarretar **desperdício de recursos** computacionais no sistema real.

A verificação do **sub/superdimensionamento** de *max_s* pode ser feita a partir de dados gerados pelo algoritmo de simulação apresentado. Nesta subseção, é visto como detectar o **subdimensionamento**, enquanto na subseção seguinte será abordada a detecção do **superdimensionamento**.

O campo **INSERCOES_NEGADAS** da estrutura **INFO_TRANSICOES** (figura 13.2) armazena, durante a simulação, a **quantidade de vezes** que a transição correspondente **não disparou**, devido, **somente**, à **falta** de alguma **instância** do **comportamento livre**, associada a um lugar do tipo III conectado à transição. No início da simulação, este campo recebe o valor 0 (zero). No decorrer da simulação, as transições que atendem as suas condições de habilitação, para alguma amarração, são inseridas na lista **TRANS_HAB** da estrutura **CLASSES_PROVISORIAS** (figura 13.4), conforme já explicado anteriormente. Durante a construção de **TRANS_HAB**, incrementa-se o campo **INSERCOES_NEGADAS** de cada transição conectada a um lugar funcional, a qual atenda todas as condições de habilitação (dadas na seção 11.2.1.3), exceto a condição referente à possibilidade de inserção de *tokens* no lugar, devido ao **repositório de entrada** do mesmo não possuir células em quantidade suficiente para admitir, pelo menos, um conjunto de *tokens* fornecido pela

transição. O fluxograma da figura 13.5 mostra os pontos de incremento do campo **INSERCOES_NEGADAS**.

Concluída a simulação da rede, a **quantidade de vezes** que uma **transição conectada a um lugar funcional não ficou habilitada**, devido, **somente**, a um **subdimensionamento** do número de *samples* que podem ser processados pelo lugar, é obtida consultando-se o valor armazenado no campo **INSERCOES_NEGADAS**, da entrada correspondente à transição, em **INFO_TRANSICOES**.

13.3.10. VERIFICAÇÃO DE SUPERDIMENSIONAMENTO EM LUGARES DO TIPO III

Para detectar o **superdimensionamento** de max_S em um lugar funcional, basta verificar se o maior número de instâncias executadas em paralelo no lugar é inferior a max_S . Caso isto aconteça, deve-se adotar tal número como novo valor para max_S .

Os campos **L(B)** da estrutura **INFO_LUGARES** (figura 13.1) armazenam o valor da função de associação dinâmica **L(B)** (veja seção 12.2.3), a qual relaciona as **instâncias** do **comportamento** com as linhas do **repositório de entrada** e de **saída** do lugar. O valor de **L(B)** varia entre 0 (**instância livre**) e max_S . Por construção, cada linha das estruturas **REP_E** e **REP_S** está estaticamente associada a uma **instância** do **comportamento**. O valor do campo **L(B)** serve para estabelecer a **correspondência** entre as linhas de **REP_E** e **REP_S**, além de indicar a **prioridade de remoção** (a linha de **REP_S** com o menor valor no campo **L(B)** terá prioridade de remoção). Os valores dos campos **L(B)** de ambas estruturas (**REP_E** e **REP_S**) são recalculados após cada operação de inserção/remoção. Para ilustrar a atuação da função **L(B)**, a figura 13.8 apresenta um trecho da estrutura **INFO_LUGARES**, no qual o valor max_S é igual a 4 e $L(B_1) = 3$, $L(B_2) = 0$, $L(B_3) = 2$ e $L(B_4) = 0$. Isto significa que as **instâncias** B_2 e B_4 estão **livres**, B_3 está associada com a segunda linha de ambos os **repositórios**, enquanto B_1 está associada com a terceira linha.

O campo **MAX_L(B)** da estrutura **INFO_LUGARES** serve para armazenar o **maior valor** associado ao conjunto de campos **L(B)**, durante o processo de simulação. Este valor determina o número **máximo** de **instâncias** do **comportamento** executadas simultaneamente. Na figura 13.8, o valor de **MAX_L(B)** é 3. Isto significa que, se até o final da simulação, este valor não for alterado, o lugar correspondente está **superdimensionado**, uma vez que uma de suas **instâncias** estará sempre **livre**.

Desta forma, a **verificação e quantificação** do **superdimensionamento** de um lugar do tipo III, quanto ao número máximo de **instâncias** do comportamento associado, consiste no resultado da subtração

$$max_S - MAX\_L(B).$$

Caso a subtração resulte em 0 (zero), o lugar não estará superdimensionado; caso contrário, o resultado da subtração indicará a quantidade de instâncias excedentes.

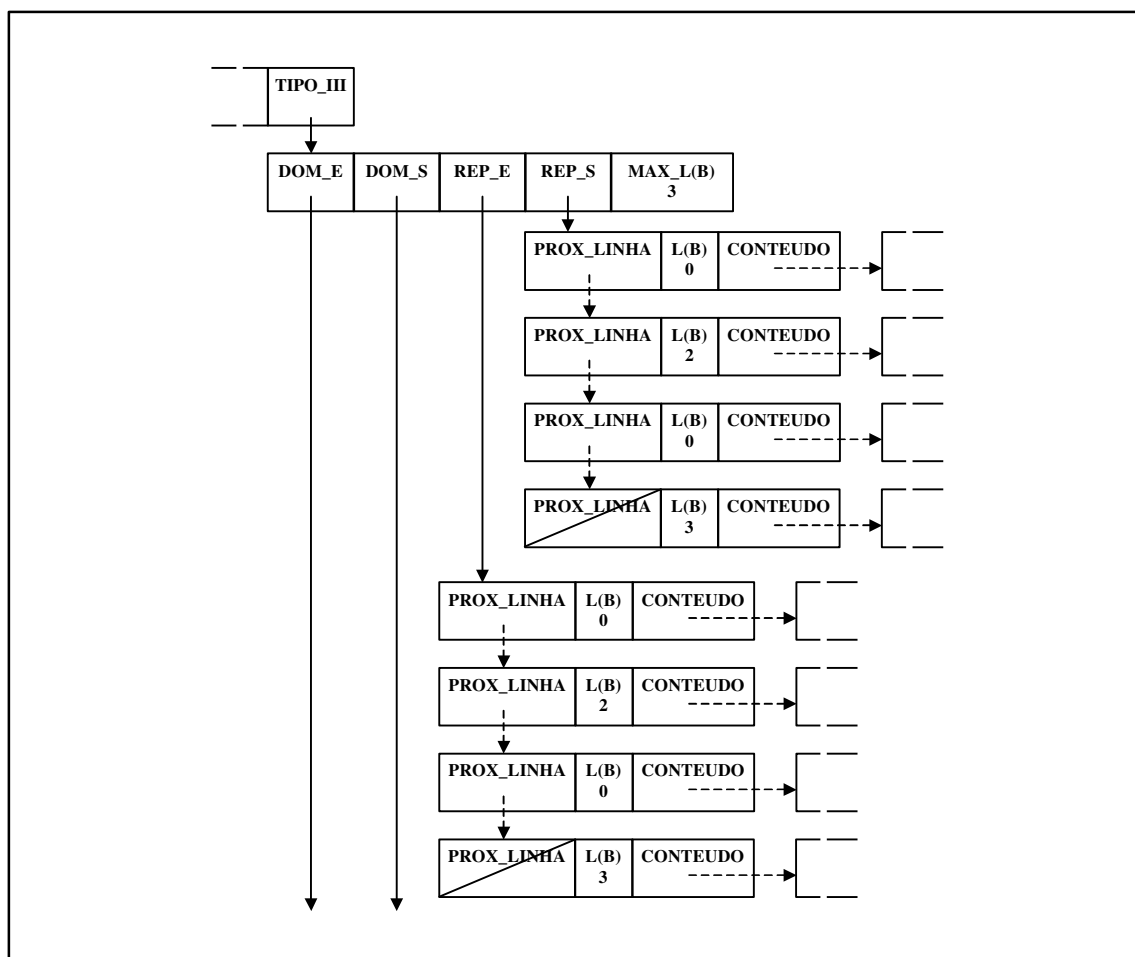


Figura 13.8. Trecho da estrutura de dados INFO_LUGARES.

CAPÍTULO 14: MÉTODOS DE ANÁLISE BASEADOS EM INVARIANTES

14.1. INTRODUÇÃO

Conforme visto na seção 13.3, as **técnicas de análise** de propriedades em redes de Petri dividem-se em dois grupos: técnicas de análise **dinâmicas** e técnicas de análise **estruturais**. As técnicas dinâmicas têm seu ponto de partida na **árvore de alcançabilidade** da rede, obtida a partir da **simulação** do modelo, dependendo, portanto, da **marcação inicial** da rede. Por outro lado, as técnicas estruturais baseiam-se na **interconexão dos componentes** que formam a rede (ou parte dela): **lugares** e **transições**. Estas técnicas **não dependem** da **marcação inicial** da rede.

Técnicas de análise dinâmicas foram desenvolvidas para modelos RPSD, encontrando-se demonstradas no capítulo 13. Técnicas de análise estruturais são tratadas no presente capítulo.

A seção 14.2 mostra a aplicação de técnicas de análise estruturais em redes de Petri, enquanto a seção 14.3 apresenta um estudo da aplicabilidade e restrições semânticas de tais técnicas na extensão RPSD.

14.2. ANÁLISE DE INVARIANTES EM REDES DE PETRI

A **verificação formal** de determinadas propriedades em redes de Petri pode ser feita com base em duas técnicas de análise estruturais, quais sejam, a análise de **invariantes de lugar** e a análise de **invariantes de transição** [86].

Para redes de Petri **Lugar/Transição (RP L/T – capítulo 7)**, um **invariante de lugar** corresponde a um **conjunto de lugares** cuja **soma de tokens é constante**, em **qualquer** marcação da rede. Por outro lado, um **invariante de transição** corresponde a um **conjunto de transições** que, disparadas em **qualquer ordem**, numa **quantidade fixa de vezes para cada uma**, **não alteram**, ao final da seqüência de disparo, a **marcação** da rede.

Uma rede L/T é **limitada** se **cada um** de seus **lugares pertence** a pelo menos um **invariante de lugar**. Além disso, **em toda rede L/T limitada e viva**, suas **transições pertencem** a pelo menos um **invariante de transição**. O **inverso** desta afirmação **nem sempre é válido**, isto é, uma rede L/T cujas transições pertençam a pelo menos um invariante de transição não é, **necessariamente**, limitada e viva.

Para ilustrar os conceitos envolvidos, considere a rede L/T da figura 14.1, a qual modela o **problema dos processos de leitura/escrita**. O modelo representa um sistema no qual vários processos têm acesso de **leitura** ou de **escrita** a um **arquivo compartilhado**. Vários processos de leitura podem acessar o arquivo simultaneamente, entretanto, quando um processo de escrita deseja acessá-lo, tal acesso será **exclusivo** (os demais processos ficam impedidos de acessar o arquivo). O lugar p_0 indica o **número de processos de leitura inativos**, enquanto p_1 indica o **número de processos de leitura ativos**. As transições t_0 e t_1

movem os processos de leitura entre estes dois estados. Similarmente, o lugar p_3 representa o **número de processos de escrita inativos**, p_4 o **número de tais processos ativos**, ao passo que, as transições t_2 e t_3 alteram processos de escrita entre estes dois estados. O lugar p_2 representa um **semáforo** que **controla o acesso** ao arquivo compartilhado.

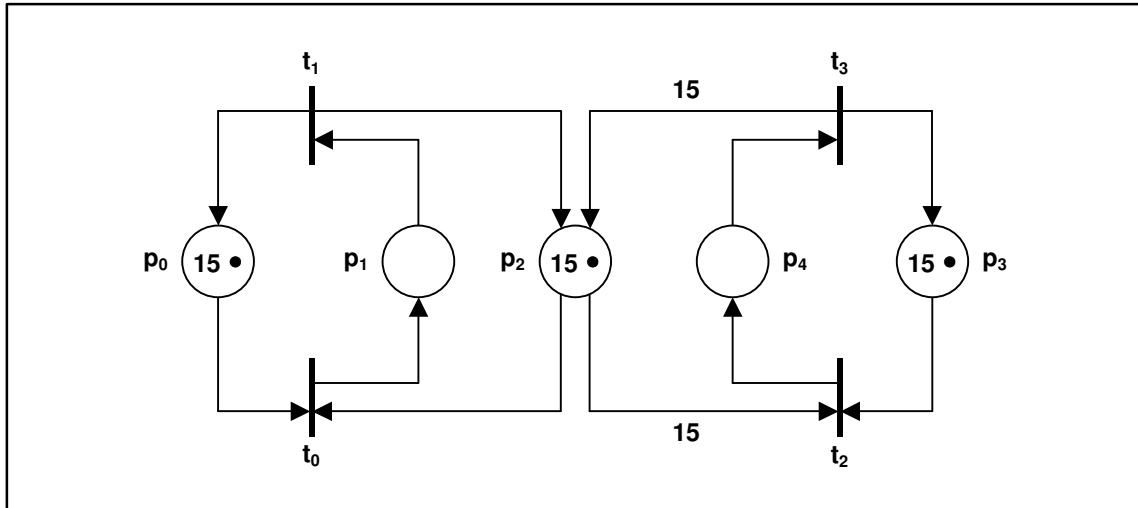


Figura 14.1. Modelo em RP L/T do problema dos processos de leitura/escrita.

A rede da figura 14.1 possui três invariantes de lugar e dois invariantes de transição. Os **invariantes de lugar** são:

- i) $\{p_0, p_1\}$, pois $p_0 + p_1 = 15$: a soma do número de *tokens* nos lugares p_0 e p_1 é igual a 15, em todas as marcações da rede, isto é, o **número total de processos de leitura é constante**;
- ii) $\{p_1, p_2, p_4\}$, pois $p_1 + p_2 + 15p_4 = 15$: a soma dos *tokens* em p_1 , p_2 e 15 vezes p_4 é sempre igual a 15;
- iii) $\{p_3, p_4\}$, pois $p_3 + p_4 = 15$: a soma dos *tokens* em p_3 e p_4 é sempre igual a 15, isto é, o **número total de processos de escrita é constante**.

Os **invariantes de transição** são:

- i) $\{t_0, t_1\}$: a seqüência de disparo $\{t_0, t_1\}$ sempre retorna o sistema à marcação na qual ele estava, antes da execução da seqüência. Neste caso, a passagem de um **processo de leitura** pelos estados **inativo-ativo-inativo** ou pelos estados **ativo-inativo-ativo**, não altera a marcação inicial;
- ii) $\{t_2, t_3\}$: as mesmas considerações acima, para **processos de escrita**.

A rede da figura 14.1 é **limitada**, uma vez que **todos** os seus **lugares** pertencem a pelo menos um **invariante de lugar**. **Todas** as transições pertencem a um **invariante de transição**, entretanto, a partir desta informação, **não é possível inferir** que a rede seja **viva**. Caso alguma transição da rede **não pertencesse** a um **invariante de transição**, **poder-se-ia concluir** que a rede é **não limitada** e **não viva**.

Para realizar a **análise de invariantes**, as **funções de incidência** da rede são **convertidas** para **matrizes de incidência** (veja informações de **representação matemática** de redes L/T no capítulo 7). A **função de incidência direta** é **convertida** na **matriz de incidência direta** A ; a de **incidência reversa**, na **matriz de incidência reversa** B . A **matriz de incidência** C , válida para redes sem lugares realimentados, é o resultado da subtração da matriz B pela A , isto é, $C = B - A$.

Na seqüência, considere uma rede L/T com **m** lugares e **n** transições. Desta forma, sua matriz de incidência **C** conterá as seguintes informações:

- a) se $C_{ij} = 0$, o lugar I_i não é nem lugar de entrada e nem lugar de saída da transição t_j ;
- b) se $C_{ij} = x$, $x > 0$, o lugar I_i é um **lugar de entrada** da transição t_j , com **peso de entrada** igual a x ;
- c) se $C_{ij} = y$, $y < 0$, o lugar I_i é um **lugar de saída** da transição t_j , com **peso de saída** igual a y .

O disparo de uma transição pode ser expresso usando a matriz de incidência. O mesmo é válido para uma **seqüência de disparo**. Representando por σ uma seqüência de disparo aplicada a uma rede com marcação **M**, a marcação resultante **M'** pode ser descrita por

$$M' = M + Cf, \quad (1)$$

onde **f** é **vetor de inteiros**, de tamanho igual ao **número de transições da rede** ($f \in \mathbb{Z}^m$), no qual cada **posição** (entrada) armazena a **quantidade de vezes** que a **transição** correspondente aparece em σ .

Em (1), se $M' = M$ (isto é, a seqüência retorna a rede à mesma marcação), então

$$Cf = 0 \text{ (matriz nula)}. \quad (2)$$

Se a **equação (2)** for **verdadeira**, o vetor **f** (representante de σ) é um **invariante de transição**, pois deixa a marcação **M** **inalterada**. Desta forma, **invariantes de transição** podem ser calculados resolvendo-se o **sistema linear homogêneo (2)**.

Na seqüência, vejamos como podem ser calculados invariantes de lugar.

Se a equação (1) for multiplicada por um **vetor** $v \in \mathbb{Z}^n$, onde **n** é o **número de lugares da rede**, obtém-se a equação

$$v^T M' = v^T M + v^T Cf. \quad (3)$$

Representando por $\{M_0\}$ o **conjunto de marcações decorrentes** da **marcação inicial** M_0 , se $v^T C = 0$ na equação (3), então

$$v^T M' = v^T M, \quad \forall M \in \{M_0\}. \quad (4)$$

Se a **equação (4)** for **verdadeira**, o vetor **v** é um **invariante de lugar**. Assim, **invariantes de lugar** podem ser **obtidos** resolvendo-se o **sistema linear homogêneo**

$$v^T C = 0. \quad (5)$$

Portanto, para redes L/T, o cálculo de invariantes de lugar e de transição baseia-se na resolução dos **sistemas lineares homogêneos (2) e (5)**.

Para **redes de alto nível**, os conceitos envolvidos na análise de invariantes são os mesmos. Nestas redes, o tamanho da **matriz de incidência** é muito **grande**, uma vez que se trabalha, geralmente, com **representações equivalentes** em redes de **baixo nível**, como as redes L/T. Desta forma, é **alto** o **custo computacional** na resolução dos sistemas lineares destacados.

Para **redes de Petri Coloridas**, a referência [53] introduz um conjunto de **regras de transformação**, sobre a matriz de incidência, que permitem a **redução gradual** desta em matrizes mais simples, sem alterar o conjunto de invariantes de lugar. Baseadas no **método de eliminação de Gauss**, tais regras são **aplicáveis** nos casos onde as submatrizes geradas possuem **inversa**. Nos demais casos, faz-se necessário **desdobrar** a rede colorida numa **rede L/T equivalente**. Quando isto ocorre, os invariantes de lugar da rede equivalente podem ser calculados aplicando-se o método de eliminação de Gauss, gerando, normalmente, um **grande número de invariantes de lugar**, os quais deverão ser **agrupados**, representando os **invariantes da rede Colorida**. Entretanto, **não há um método genérico** para fazer tal agrupamento. Informações adicionais sobre a análise de invariantes em redes Coloridas podem ser encontradas em [58].

Para aqueles que desejarem se aprofundar na análise de invariantes, recomenda-se a leitura das referências [58,87].

Na seção seguinte, abordaremos questões envolvendo conceitos de análise de invariantes e **RPSD**.

14.3. APLICABILIDADE DA ANÁLISE DE INVARIANTES EM RPSD

Ao estudar a aplicação de técnicas de análise de invariantes em RPSD, constatamos alguns fatores que, **se não inviabilizam, limitam** consideravelmente os **resultados** que podem ser obtidos a partir desse tipo de análise. Tais fatores encontram-se descritos na seqüência.

1) Propriedades a serem analisadas.

Limitabilidade e **vivacidade** são as duas principais **propriedades** que podem ser analisadas utilizando-se de **invariantes**, conforme exposto na seção anterior. A princípio, tal análise basear-se-ia na **representação em baixo nível** de modelos RPSD. Esta representação é possível, conforme demonstrado no capítulo 12. Entretanto, cair-se-ia no mesmo problema destacado para redes Coloridas, qual seja, o da **necessidade** de criação de um **mecanismo** genérico de **correspondência**, entre **agrupamentos** de invariantes de lugar, da **rede de baixo nível**, e **invariantes de lugar**, da **rede de alto nível**.

Mesmo que tal problema fosse resolvido, a **simulação** continuaria sendo **necessária**, pois, **outras propriedades**, como as destacadas no capítulo 13, exigiriam a concepção de **novas abordagens** para a realização da análise estrutural, o que demandaria **tempo de desenvolvimento considerável**.

Neste sentido, uma vez que a **simulação** faz-se **obrigatória**, estima-se que o **tempo** gasto na **computação** das propriedades de **limitabilidade** e **vivacidade** (**descontando-se**, portanto, o tempo de **simulação**) seja **menor** que o **tempo** gasto na **resolução** dos **sistemas lineares** envolvendo lugares da rede de baixo nível.

2) Características da rede L/T equivalente.

Supondo-se que, desconsiderando as observações dadas no fator 1), desejássemos trabalhar na **representação em baixo nível** de modelos **RPSD**, encontraríamos alguns problemas, relativos à representação de equivalência de lugares RPSD.

Na representação adotada no capítulo 12, as **redes L/T descritoras de lugares** do tipo I, II e III são **conservativas**, isto é, sua quantidade de *tokens* é constante. Desta forma, não faria sentido fazer a análise de invariantes de lugar sobre tais redes.

Assim, caberia **identificar** quais **subredes** (não necessariamente conservativas) deveriam ser **analisadas**.

3) Concepção semântica de RPSD.

De forma simplificada, invariantes de lugar relacionam-se à conservação de uma quantidade fixa de *tokens* em um conjunto de lugares, enquanto invariantes de transição relacionam-se à conservação de qualquer estado da rede.

Trabalhando somente no **nível de abstração de RPSD** (não considerando sua representação de baixo nível), os **conceitos de invariantes**, apresentados na forma acima, **fazem sentido?**

Aparentemente, invariantes de transição manteriam seu sentido original.

Quanto a invariantes de lugar, alguns problemas foram detectados. O principal deles refere-se às **diferenças semânticas** dos lugares RPSD, os quais representam **estados locais, componentes de armazenamento ou componentes funcionais**. Faria sentido um **invariante de lugar** englobar lugares de **tipos distintos**? Outro problema refere-se à **contagem de tokens**, uma vez que tais elementos apresentam **semântica diferenciada**, de acordo com o lugar onde estão inseridos. Além disso, *tokens* podem ter seu valor agregado **alterado** dentro de um mesmo lugar (funcional, no caso).

A análise de invariantes, em alto nível de abstração, exige a criação de novas abordagens de tratamento, que trabalhem com os **elementos estruturais** de tais níveis.

4) Tempo de desenvolvimento.

Devido ao interesse despertado e ferramentas disponíveis, redes de Petri Coloridas constitui-se numa das extensões melhor desenvolvidas ao longo das duas últimas décadas. Entretanto, suas técnicas de análise de invariantes apresentam **limitações computacionais** e de **aplicabilidade** [88], constituindo-se num conjunto de problemas/questões **em aberto**. Igual situação apresenta outras extensões de alto nível.

Considerando-se a experiência de outros pesquisadores, e a relação dos problemas destacados nos fatores acima, pode-se concluir que o **desenvolvimento** de técnicas de análise baseadas em invariantes demandaria **tempo considerável**, não compatível com os prazos de desenvolvimento deste trabalho de doutorado.

No **contexto** exposto nos **fatores** anteriores, preferimos dedicar esforço maior no desenvolvimento da **análise dinâmica de propriedades** (capítulo 13), uma vez que tal análise responde a um **número** muito **maior de questões** inerentes a um modelo de sistema digital em RPSD.

CAPÍTULO 15: METODOLOGIA DE COPROJETO *HARDWARE/SOFTWARE* BASEADA EM RPSD

15.1. INTRODUÇÃO

Este capítulo mostra as linhas gerais de uma metodologia de coprojeto *hardware/software* baseada em **RPSD**, proposta para desenvolvimento futuro, sob supervisão do *Grupo de Micro-Eletrônica* do *Departamento de Ciências da Computação e Estatística* do **IBILCE/UNESP (GME/DCCE/IBILCE/UNESP)**. Trata-se de uma primeira versão de metodologia, uma vez que suas etapas podem sofrer alterações, decorrentes de seu próprio desenvolvimento. A intenção deste capítulo, no entanto, é a de apresentar os conceitos básicos que norteiam a metodologia proposta.

15.2. FLUXO DE PROJETO

A metodologia proposta visa a **representação, simulação, análise, validação e síntese** em alto nível de sistemas digitais. O fluxo de projeto da metodologia é mostrado na figura 15.1.

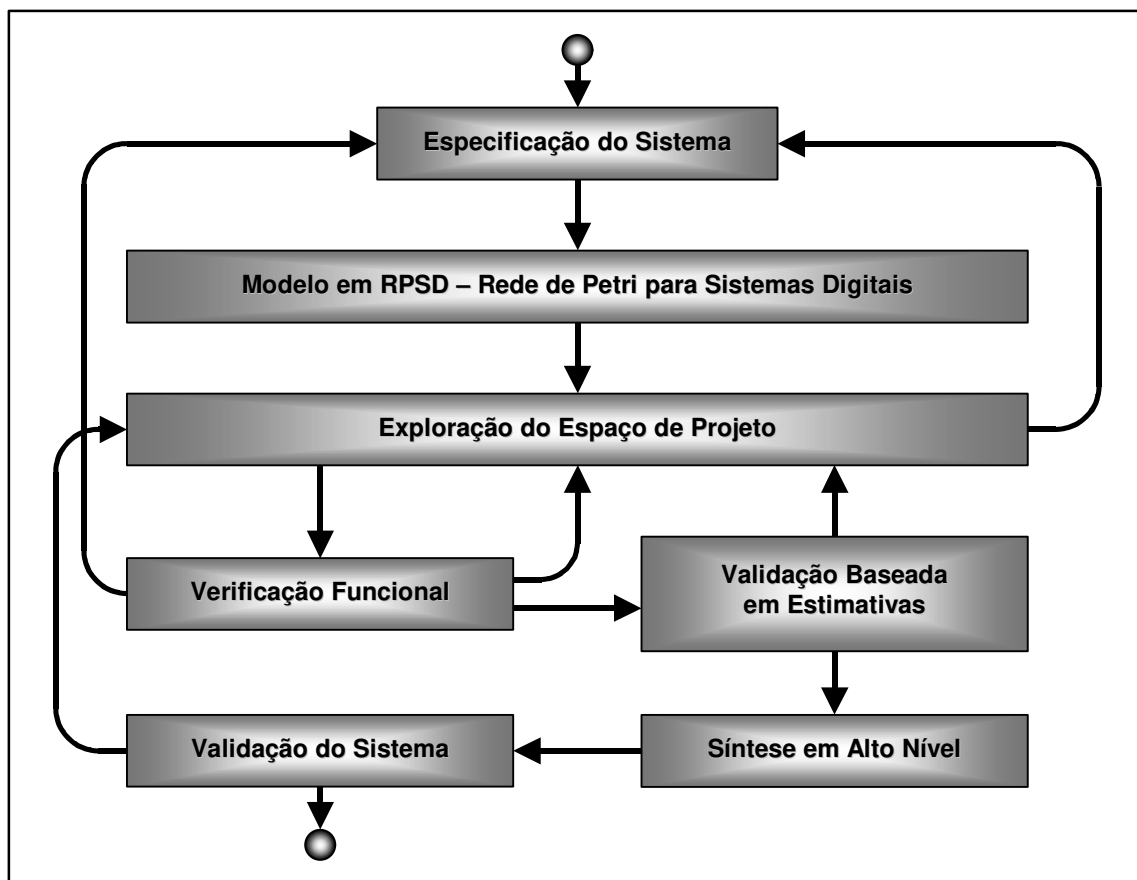


Figura 15.1. Fluxo de projeto da metodologia proposta.

Na sequência, uma descrição das etapas da metodologia é fornecida.

1) Especificação.

Nesta etapa será feita a **especificação do sistema digital**, através de uma **interface gráfica amigável**, que deverá operar em alto nível de abstração. A interface deverá fornecer recursos que permitam tanto a especificação da **funcionalidade do sistema** quanto das **restrições de projeto** (desempenho, área e consumo, dentre outras).

2) Modelo em RPSD.

Nesta etapa, a **especificação do sistema** deverá ser **convertida** em uma rede **RPSD**, para tratamento nas etapas subseqüentes. Alguns exemplos de possíveis associações, entre componentes da especificação e componentes RPSD, são:

- a) **item de dado**: mapeado em um ou mais *tokens*;
- b) **conjunto seqüencial de funções/tarefas**: mapeado em um lugar do tipo III (com comportamento descrito por uma rotina);
- c) **conjunto paralelo de funções/tarefas**: mapeado em um lugar do tipo III (com comportamento descrito por uma RPSD-T), ou em um conjunto de lugares do tipo III (com comportamentos descritos por rotinas – um lugar para cada função/tarefa);
- d) **elemento estruturado (complexo) de armazenamento**: mapeado em um conjunto de lugares do tipo II;
- e) **elemento simples de armazenamento**: mapeado em um lugar do tipo I;
- f) **evento inerente ao fluxo normal de execução do sistema**: mapeado em uma transição;
- g) **evento de exceção ao fluxo normal de execução do sistema / evento interativo do ambiente, percebido pelo sistema**: mapeado em uma transição conectada com o ramo (ou ramos) apropriado – suspensivo/restaurador/preemptivo;
- h) **evento escalonado do sistema**: mapeado em uma transição temporizada;
- i) **evento aleatório do sistema**: mapeado em uma transição não temporizada (sem intervalo de disparo);
- j) **canal de comunicação**: mapeado em um conjunto de ramos (denotando relações de interdependência entre as partes comunicantes) e lugares do tipo I (denotando sinais em barramentos);
- k) **protocolo de comunicação**: mapeado em um trecho de rede (subrede) pré-definido, obtido de uma biblioteca de modelos.

Associações mais complexas deverão ser concebidas, o que dependerá da definição dos elementos utilizados na interface gráfica de especificação.

3) Exploração do Espaço de Projeto.

Com base no modelo RPSD gerado na etapa 2, deve ser feita a exploração do espaço de projeto. Esta etapa busca responder as seguintes questões:

- qual a melhor **arquitetura** para implementar o sistema?

- quais **funções** do sistema devem ser **implementadas** em componentes de *hardware* e quais em componentes de *software*?
- qual deve ser o **escalonamento** (tempo para início de execução) das funções, de forma a atender as restrições de **desempenho**?

Para responder a tais questões, esta etapa foi dividida nas três sub-etapas seguintes.

3.1) Seleção da arquitetura: Dados um conjunto de modelos (*templates*) de **arquitetura** e um conjunto de modelos de **componentes** usados na implementação das mesmas, deve-se escolher uma **configuração arquitetura/alocação de componentes**, sobre a qual possam ser **mapeados** os **componentes estruturais** do **modelo RPSD** (lugares, transições, ramos), de forma que as **restrições de projeto** (tais como custo, desempenho e consumo) possam ser **atendidas**. Para tal, devem ser desenvolvidas **técnicas de análise** e **algoritmos** que **avaliem configurações** distintas, com base nas restrições estabelecidas. Assim, a configuração **arquitetura/alocação** melhor avaliada deve ser submetida às etapas seguintes e, caso seja rejeitada, outra configuração deve ser adotada. Caso contrário, o sistema pode ser implementado.

Limitando-se as restrições de projeto somente a custo e desempenho, uma possível abordagem consiste na seleção, dentre um conjunto pré-definido, de uma configuração **arquitetura/alocação** que atenda as restrições de custo. Desta forma, as restrições de desempenho passam a ser tratadas somente nas etapas de validação e de particionamento temporal. Apesar de facilitar a seleção da arquitetura, tal abordagem pode aumentar o tempo do processo de projeto, caso as arquiteturas inicialmente selecionadas não atendam as restrições de desempenho.

3.2) Particionamento espacial: Dadas uma arquitetura e uma alocação de componentes (configuração **arquitetura/alocação**), deve-se escolher quais **funções/tarefas** do sistema serão associadas a **componentes de hardware** e quais serão associadas a **componentes de software**, de forma que as **restrições de projeto** possam ser atendidas. Geralmente, funções/tarefas críticas em relação a tempo são mapeadas para *hardware*, enquanto as menos críticas (ou que exijam flexibilidade) são mapeadas para *software*. Considerando-se o modelo RPSD do sistema, o particionamento espacial significa determinar o **mapeamento** (*hardware* ou *software*) dos **lugares funcionais** (tipo III). Há vários **algoritmos** de particionamento espacial (veja o capítulo 5), os quais estão sendo analisados quanto à sua adaptação para RPSD.

3.3) Particionamento temporal: Dados uma **arquitetura**, uma **alocação** de componentes e o **particionamento espacial** das **funções/tarefas** do sistema, deve-se procurar explorar o paralelismo existente entre **funções/tarefas**, de forma que **restrições críticas** de **desempenho** possam ser **atendidas**. A seção 5.4 apresentou um algoritmo de particionamento temporal, proposto por Bakshi e Gajski [33], cuja aplicabilidade a modelos RPSD está sendo estudada.

A exploração do espaço de projeto trata-se de uma etapa **recorrente** no fluxo de projeto, uma vez que seus resultados podem não passar pelo crivo das etapas **4**, **5** e **7**. Quando isto ocorre, nem todas as suas sub-etapas são executadas novamente no retorno, pois há várias combinações envolvendo arquiteturas, componentes alocados e escolhas de particionamento (tanto espacial quanto temporal).

4) Verificação Funcional.

Esta etapa consiste na **simulação e análise** do modelo obtido na etapa 2: os resultados da simulação devem ser analisados não só quanto à verificação da **funcionalidade** do sistema modelado mas, também, quanto à **validação** (ou não) de **propriedades** do sistema. Conforme mostrado no capítulo 13, a análise de propriedades é importante, uma vez que ela pode detectar **erros** tanto na **especificação** do sistema quanto na **alocação** de componentes. No primeiro caso, o fluxo de projeto retorna à etapa 1, ao passo que, no segundo caso, retorna à etapa 3.

Para executar esta etapa, deverão ser implementados os algoritmos de simulação e análise de propriedades apresentados no capítulo 13, os quais deverão ser estendidos.

5) Validação Baseada em Estimativas.

Uma vez certificada a correta funcionalidade do sistema, o mesmo deverá ser **validado**, isto é, deverá atender as **restrições de projeto** fornecidas na etapa de **especificação**. Estas restrições referem-se, principalmente, a **características físicas e temporais** do sistema como, por exemplo, **área** ocupada pelos componentes funcionais associados à implementação em *hardware* (ou em *software* – memória), **desempenho** (*throughput* – taxa de processamento), e **consumo**. Como estas **restrições** são **altamente dependentes** da **implementação** física do sistema e, como a implementação envolve **custos**, esta etapa busca validar o sistema sem ter que implementá-lo, com base no cálculo de **estimativas**. Neste contexto, deverão ser desenvolvidos **modelos matemáticos** que permitam estimar valores para tais atributos do sistema, com base em informações sobre a **arquitetura** selecionada, os **componentes** alocados e a **partição** (espacial/temporal) adotada.

6) Síntese em Alto Nível.

Após a validação do sistema modelado, o mesmo poderá ser sintetizado. A síntese em alto nível consiste na síntese dos **componentes RPSD** associados a **hardware dedicado**, **segmentos de software** e **interfaces**.

6.1) Síntese de hardware: Nesta etapa, deve-se gerar uma **representação estrutural**, no **nível de abstração RTL** (*Register Transfer Level* – Nível de Transferência entre Registradores), dos **componentes RPSD** associados a **hardware dedicado**, visando implementação através de **ferramentas de síntese RTL**.

A **geração** da representação RTL de um componente, ou conjunto de componentes, do modelo RPSD, poderá ser feita com base na **descrição de baixo nível** do(s) componente(s), utilizando-se das **regras de equivalência** mostradas no capítulo 12. Tais regras permitem a representação de componentes estruturais de RPSD através de conjuntos de lugares e transições de uma rede de Petri de baixo nível. Estes componentes, por sua vez, deverão ser mapeados para uma descrição RTL.

6.2) Síntese de *software*: Nesta etapa, deve-se gerar os **segmentos de código (rotinas, programas)** correspondentes aos **componentes** do modelo **RPDS** associados à implementação em *software* (na etapa **3.2**). Tais segmentos serão executados nos processadores alocados (na etapa **3.1**), de acordo com o escalonamento fornecido pelo particionamento temporal (na etapa **3.3**), excetuando-se os casos nos quais o sistema modelado apresente escalonamento dinâmico. Lugares funcionais com comportamento descrito por rotina têm síntese direta. Os demais casos ainda não foram estudados.

6.3) Síntese de interfaces: Nesta etapa, deve-se verificar a necessidade de geração de componentes adicionais de *hardware* (*latches*, *buffers*) e de *software* (rotinas), relativos à comunicação entre os demais componentes do sistema.

7) Validação do Sistema.

Feita a síntese em alto nível do sistema, o mesmo deverá ser **validado** novamente, uma vez que a validação baseada em estimativas pode apresentar algumas distorções, comprometendo o atendimento às **restrições de projeto**. Esta validação poderá ser feita utilizando-se de simuladores RTL (para validar os componentes dedicados de *hardware*), programas de *benchmark* (para validar os segmentos de *software*) e medições no sistema implementado.

Mudanças poderão ser adotadas nas etapas da metodologia descrita, uma vez que a mesma se encontra em fase inicial de desenvolvimento.

CONCLUSÕES

A partir da constatação de que as extensões de rede de Petri não são completas para a modelagem de sistemas digitais, nos empreendemos na elaboração de uma extensão possuidora de temporização, estruturas de programação, mecanismos para o tratamento de exceções, hierarquia comportamental e que fosse de alto nível, facilitando, dessa forma, a criação de modelos de sistemas digitais (mais especificamente, embutidos), para suas posteriores simulação, análise e síntese. Os resultados deste trabalho estão expostos nesta tese, através da formalização da extensão denominada **Rede de Petri para Sistemas Digitais (RPSD)**.

RPSD foi desenvolvida a partir de um estudo abrangente das áreas de sistemas digitais (envolvendo modelos conceituais, linguagens de especificação, arquiteturas e metodologias de projeto) e redes de Petri (extensões, representações e técnicas de análise). Ao longo de seu desenvolvimento, foi agregado um conjunto de características à extensão RPSD, possibilitando a definição de um modelo conceitual de grande capacidade descritiva. Tal capacidade foi demonstrada, nesta tese, através da modelagem de um processador hipotético.

Como se pretende, no futuro próximo, implementar uma plataforma em *software* capaz de capturar descrições em RPSD de sistemas digitais embutidos, foi realizado um estudo comparativo entre RPSD e linguagens de especificação usadas no projeto de tais sistemas. Este estudo demonstrou que RPSD atende integralmente as características elencadas por Daniel Gajski, para a representação adequada do comportamento desses sistemas, superando, desta forma, as linguagens analisadas.

Em relação às redes de Petri, foi demonstrada a equivalência entre RPSD e extensões de rede de Petri de baixo nível, o que permitiu concluir que a extensão proposta é uma extensão de rede Petri válida. Assim, abriu-se a possibilidade de estender, para RPSD, métodos de verificação formal, simulação e síntese de baixo nível, desenvolvidos para outras extensões de rede de Petri. Neste contexto, faz-se necessárias algumas considerações:

- Simulação: A tese apresentou técnicas de simulação e de análise dinâmica de propriedades de modelos RPSD. Estas técnicas buscam responder questões inerentes a requisitos de projeto de sistemas digitais embutidos. O conjunto proposto pode ser estendido, ampliando a capacidade de análise de RPSD, não necessitando, no entanto, alterar a formalização da extensão. Por exemplo, pode-se desenvolver um algoritmo para a verificação da propriedade de limitabilidade nas redes que usam variáveis pertencentes a tipos infinitos. Este algoritmo identificaria classes de um modelo RPSD que difiram somente na valoração das variáveis da rede, buscando componentes fortemente conexos, no grafo de classes da rede analisada, que apresentem diferenças somente quanto ao conteúdo de suas variáveis. A identificação de componentes fortemente conexos com tal característica seria um indício de rede ilimitada;
- Verificação formal: Ao estudar a aplicabilidade de técnicas de análise de invariantes em RPSD, surgiram algumas questões que, se não inviabilizam, limitam consideravelmente os resultados que podem ser obtidos a partir desse tipo de análise. Contudo, é um campo em aberto, como nas demais extensões de rede de Petri de alto nível, demandando novas abordagens, além de considerável tempo;

- Síntese de baixo nível: A representação de um modelo RPSD em uma rede de baixo nível demonstrou ser bastante extensa, dificultando a síntese a partir de tal representação, para sistemas complexos. Nesta questão, pode-se desenvolver novas formas de equivalência, o que pode reduzir um pouco o número de lugares da rede de baixo nível. Outra opção consiste na agregação de lugares de baixo nível, formando conjuntos de lugares, cada qual mapeado para um componente estrutural do sistema.

A tese apresentou, também, a primeira versão de uma metodologia de coprojeto *hardware/software* baseada em RPSD. Esta metodologia visa a descrição, simulação, análise, validação e síntese em alto nível de sistemas digitais embutidos. A extensão RPSD entra em todas as etapas da metodologia, possibilitando:

- Fluxo de projeto homogêneo: A adoção de um modelo unificado, para representar todo o sistema, permite maior grau de automatização das etapas de projeto, uma vez que menos conversões são necessárias. Isto facilita o uso de etapas recorrentes no fluxo de projeto;
- Representação independente de implementação: As técnicas de análise de propriedades auxiliam na escolha da implementação dos componentes do sistema modelado (para *hardware* ou para *software*). Esta característica facilita a aplicação de algoritmos de particionamento;
- Alocação e configuração de componentes: As técnicas de análise de propriedades auxiliam, também, na alocação e configuração de componentes do sistema. Por exemplo, é possível:
 - verificar se lugares funcionais estão superdimensionados (*número máximo de samples* acima do necessário, o que acarretaria desperdício de recursos computacionais no sistema real) ou subdimensionados (*número máximo de samples* abaixo do necessário, o que provocaria gargalos de processamento no sistema físico correspondente);
 - verificar a taxa média de uso (acesso) das células de armazenamento de lugares do tipo II, o que auxilia na definição dos subsistemas de armazenamento do sistema modelado;
 - verificar se as latências de leitura e de escrita dos lugares do tipo II estão adequadas, o que auxilia na escolha de componentes mais apropriados de acordo com as restrições de projeto (componentes mais rápidos *versus* componentes mais baratos);
 - verificar quais *tokens* não são utilizados em lugares do tipo I, permitindo uma melhor configuração do sistema;
 - verificar quais transições não disparam, permitindo ajustar determinadas condições de habilitação das mesmas.

Mudanças poderão ser adotadas nas etapas da metodologia proposta, uma vez que a mesma encontra-se em fase inicial de desenvolvimento. Atualmente, estamos analisando alguns algoritmos de particionamento espacial e temporal, quanto à sua adaptabilidade para RPSD. Os passos seguintes, no desenvolvimento da metodologia proposta, incluem a implementação dos algoritmos de simulação e de análise de propriedades, o desenvolvimento de técnicas de síntese em alto nível, o desenvolvimento de modelos matemáticos para o cálculo de estimativas para restrições de projeto e o desenvolvimento de uma interface gráfica de especificação de sistemas digitais embutidos.

Finalmente, esperamos, com esta extensão, contribuir na interação entre a comunidade de redes de Petri e a de coprojeto *hardware/software*.

REFERÊNCIAS BIBLIOGRÁFICAS / INTERNET

- [1] R. Ernst, "Codesign of Embedded Systems: Status and Trends", IEEE Design & Test, pp. 45-54, April-June 1998.
- [2] M.F. Jacome, G. de Veciana, "Design Challenges for New Application-Specific Processors", IEEE Design & Test of Computers, pp. 40-50, April-June 2000.
- [3] D.D. Gajski, F. Vahid, S. Narayan, J. Gong, "Specification and Design of Embedded Systems", Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1994.
- [4] VSIA, "System-Level Design Model Taxonomy Document", version 2.1 (SLD 2 2.1), July, 2001.
(<http://www.vsi.org/resources/specs/sld220.pdf> – data de acesso: 01/03/2004)
- [5] RASSP Taxonomy Working Group (RTWG), "VHDL Modeling Terminology and Taxonomy", Revision 2.4, February 23, 1998.
(<http://www.eda.org/rassp/> – data de acesso: 01/03/2004)
- [6] D.D. Gajski, N.D. Dutt, C.H. Wu, Y.L. Lin, "High-Level Synthesis: Introduction to Chip and System Design", Kluwer Academic Publishers, Boston, Massachusetts, 1991.
- [7] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, vol 8, pp. 231-274, 1987.
- [8] P.S. Chen, "The Entity-Relationship Approach to Logical Data Base Design", Q.E.D. Information Sciences, Wellesley, Massachusetts, 1977.
- [9] A. Sutcliffe, "Jackson System Development", Prentice-Hall, New York, 1988.
- [10] A. Orailloglu, D.D. Gajski, "Flow Graph Representation", In Proceedings of the Design Automation Conference, 1986.
- [11] P.T. Ward, S.J. Mellor, "Structured Development for Real-Time Systems", Yourdon Press, New York, 1985.
- [12] D.D. Gajski, F. Vahid, S. Narayan, "A System-Design Methodology: Executable-Specification Refinement", In Proceedings of the European Conference on Design Automation, 1994.
- [13] V. Madiseti, "System-Level Synthesis and Simulation VHDL: A Taxonomy and Proposal Towards Standardization", VIUF Spring, 1995 Proceedings.
- [14] Yi Gao, Shilang Tang, Zhongli Ding, "Comparison between CISC and RISC", Department of Electrical and Computer Engineering, University of Massachusetts Dartmouth, 2003.
(http://www.cc.gatech.edu/grads/z/Howard.Zhou/micellaneous/gre_cs_sub/risc_cisc.pdf – data de acesso: 01/03/2004 – responsáveis: { [ygao1](mailto:ygaol@cs.umbc.edu), [stang2](mailto:stang2@cs.umbc.edu), [zding1](mailto:zding1@cs.umbc.edu) }@cs.umbc.edu)
- [15] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, "Treadmarks: Shared Memory Computing on Networks of Workstations", IEEE Computer, Vol. 29, No. 2, pp. 18-28, February 1996.
- [16] M.J. Flynn, "Some computer organizations and their effectiveness", IEEE Transactions on Computing, C-21, (1972) 948-960.
- [17] Alexander Wolfe, "VLIW Architecture Emerges as Embedded Alternative", 2001.
(<http://www.embedded.com/story/OEG20010222S0039> – data de acesso: 01/03/2004)
- [18] Jack Dongarra, Aad van der Steen, "Distributed-memory SIMD machines", 2003.
(<http://www.euroben.nl/reports/web03/dm-simd.html> – data de acesso: 01/03/2004)
- [19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, "MPI: The Complete Reference", The MIT Press, 1996.
(<http://www.netlib.org/utk/papers/mpi-book/mpi-book.html> – data de acesso: 01/03/2004)

- [20] H. Ishihata, T. Horie, T. Shimizu, "Architecture for the AP1000 highly parallel computer", Fujitsu Sci. Tech. J. 29, 6-14, 1993.
- [21] IEEE Inc., N.Y., "IEEE Standard VHDL Language Reference Manual", 1988.
- [22] Verilog Homepage.
(<http://www.verilog.com/> – data de acesso: 01/03/2004)
- [23] C.A.R. Hoare, "Communicating sequential processes", Communications of the ACM, 21(8): 666-677, 1978 – Reprinted in J.P. Bowen and M.G. Hinchey, "High-Integrity System Specification and Design", Springer-Verlag, FACIT series, pages 303-329, 1999.
- [24] Daniel C. Hyde, "Introduction to the Programming Language Occam", Department of Computer Science, Bucknell University, 1995.
(<http://www.eg.bucknell.edu/~cs366/occam.pdf> – data de acesso: 01/03/2004)
- [25] International Engineering Consortium, "Specification and Description Language (SDL)", 2003.
(<http://www.iec.org/online/tutorials/sdl/index.html> – data de acesso: 01/03/2004)
- [26] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, H. De Man, "DSP Specification using the Silage Language", Proceedings IEEE ICASSP Conference, Albuquerque, pp. 1057-1060, April 1990.
- [27] A. Kalavade, E.A. Lee, "A Hardware-Software Codesign Methodology for DSP Applications", IEEE Des. & Test, vol. 10, no. 3, pp. 16-28, Sept. 1993.
- [28] S.L. Coumeri, D.E. Thomas, "A simulation environment for hardware-software codesign", In Proc. 32nd Design Automation Conference, pp. 58-63, 1995.
- [29] J. Smith, G. De Micheli, "A Methodology for Synthesis with Reusable Components from an Arithmetic Specification", In Proc. 14th European Conference on Circuit Theory and Design (ECCTD'99), Stresa, Italy, 29 August – 2 September 1999.
(<http://akebono.stanford.edu/users/nanni/research/sys/ecctd.pdf> – data de acesso: 01/03/2004)
- [30] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vicentelli, "Formal verification of embedded systems based on CFMS networks", in Proc. 33rd Design Automation Conference, 1996.
- [31] N.S. Woo, A.E. Dunlop, W. Wolf, "Codesign from cospecification", IEEE Computer, vol. 27, no. 1, pp. 42-47, Jan. 1994.
- [32] D.D. Gajski, J. Zhu, R. Dömer, "Essential Issues in Codesign", Chapter 1 in "Hardware/Software Co-Design: Principles and Practice" (ed. J. Staunstrup, W. Wolf), Kluwer Academic Publishers – ISBN 0-7923-8013-4, October 1997.
- [33] S. Bakshi, D.D. Gajski, "A scheduling and pipelining algorithm for hardware/software systems", Proceedings of the 10th International Symposium on System Synthesis, pp. 113-118, 1997.
- [34] A. Kalavade, E.A. Lee, "A Global Criticality/Local Phase Driven Algorithm for Constrained Hardware/Software Partitioning Problem", In Proc. Codes/CASHE'94, Third Intl. Workshop on Hardware/Software Codesign, pp. 42-48, Grenoble, France, Sept, 22-24, 1994.
(<http://ptolemy.eecs.berkeley.edu/publications/papers/94/codesign/codes94.pdf> – tese correlata, de Asawaree Kalavade, disponível eletronicamente em <http://ptolemy.eecs.berkeley.edu/publications/papers/95/kalavadeThesis/> – datas de acesso: 01/03/2004)
- [35] R.K. Gupta, C.N. Coelho Jr., G. De Micheli, "Program implementation schemes for hardware/software systems", IEEE Computer, vol. 27, no. 1, pp. 48-55, Jan. 1994.

- [36] R. Bawa, E. Encrenaz, "A Platform for the Formal Verification of VHDL Programs", Proceedings of the 4th International Workshop on Symbolic Methods and Applications in Circuit Design, Louvain, Belgique, 1996.
- [37] C. Heuser, "Modelagem Conceitual de Sistemas: Redes de Petri", V EBAI, 1990.
- [38] W. Reisig, "Petri Nets, An Introduction", EATCS, Monographs on Theoretical Computer Science, W. Brauer, G. Rozenberg, A. Salomaa (Eds.), Springer Verlag, Berlin, 1985.
- [39] J.L. Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall, N.J., 1981, ISBN 0-13-661983-5.
- [40] R.T. Tamura, S. Motoyama, W.C. Borelli, "Rede de Petri: Análise e Simulação", Relatório Técnico 029/85, FEE/UNICAMP, 1985.
- [41] C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets", Ph.D. Thesis, MIT, Project MAC, MAC-TR 120, MIT 1974.
- [42] P.M. Merlin, "A study of recoverability of communication protocols", Ph.D. Thesis, Computer Science Department, University of California, 1974.
- [43] A. Cerone, A. Maggiolo-Schettini, "Time-based expressivity of time Petri nets for system specification", Theoretical Computer Science, vol. 216, pp. 1-53, 1999.
- [44] J.E. Coholahan, N. Roussopoulos, "Timed requirements for timed-driven systems using augmented Petri nets", IEEE Trans. Software Eng., vol. 9, pp. 603-616, 1983.
- [45] P.M. Merlin, D.J. Farber, "Recoverability of Communication Protocols: Implications of a Theoretical Study", IEEE Trans. Commun., vol. 24, pp. 1036-1043, 1976.
- [46] B. Walter, "Timed Petri-Nets for Modelling and Analyzing Protocols with Real-Time Characteristics", in: H. Rudin, C.H. West (Eds.), Proc. 3rd IFIP Workshop on Protocol Specification, Testing, and Verification, North-Holland, Amsterdam, pp. 149-159, 1983.
- [47] C. Ghezzi, D. Mandrioli, S. Morasca, M. Pezzè, "A unified high-level Petri net formalism for time-critical systems", IEEE Trans. Software Eng., vol. 17, pp. 160-172, 1991.
- [48] M. Marton, W.C. Borelli, S. Motoyama, "Rede de Petri Temporizada: Modelo e Exemplo", Relatório Técnico 077/86, FEE/UNICAMP, 1986.
- [49] J. Sifakis, "Use of Petri nets for Performance Evaluation", 3rd Intl. Symposium on Modeling and Evaluation, IFIP, North Holland, pp. 75-93, 1977.
- [50] B. Berthomieu, M. Diaz, "Modeling and Verifications of Time Dependent Systems Using Time Petri Nets", IEEE Transactions on Software Engineering, vol. 17, no. 3, pp. 259-273, March 1991.
- [51] B. Berthomieu, F. Vernadat, "State class constructions for branching analysis of Time Petri nets", 9th International Conference TACAS 2003, Warsaw, Poland, Springer-Verlag LNCS 2619, pp. 442-457, February 2003.
(<http://www.laas.fr/~bernard/tina/papers/ascg2.ps.gz> – data de acesso: 01/03/2004)
- [52] H.J. Genrich, K. Lautenbach, "System Modelling with High-Level Petri Nets", Theoretical Computer Science, North-Holland, vol. 13, pp. 109-136, 1981.
- [53] K. Jensen, "Coloured Petri Nets and the Invariant Method", Theoretical Computer Science, North-Holland, vol. 14, pp. 317-336, 1981.
- [54] K. Jensen, "High-level Petri Nets", In: A. Pagnoni, G. Rozenberg (eds.): Applications and Theory of Petri Nets, Informatik-Fachberichte, Springer-Verlag, vol. 66, pp. 166-180, 1983.
- [55] K. Jensen, "An Introduction to the Theoretical Aspects of Coloured Petri Nets", In: J. W. de Bakker, W. P. de Roever, G. Rozenberg (eds.): "A Decade of Concurrency", Lecture Notes in Computer Science, Springer-Verlag, vol. 803, pp. 230-272, 1994.

- [56] Petri Nets World Homepage.
(<http://www.daimi.au.dk/PetriNets/> – data de acesso: 01/03/2004)
- [57] K. Jensen, "Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use". Volume 1, Basic Concepts – Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997, ISBN: 3-540-60943-1.
- [58] K. Jensen, "Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use". Volume 2, Analysis Methods – Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997, ISBN: 3-540-58276-2.
- [59] K. Jensen, "Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use", Volume 3: *Practical Use*, In: EATCS Monographs on Theoretical Computer Science. Berlin: Springer-Verlag, 1997. ISBN 3-540-62867-3.
- [60] L. Petrucci, "Combining Finkel's and Jensen's Reduction Techniques to Build Covering Trees for Coloured Nets", Petri Nets Newsletter, Special Interest Group on Petri Nets and Related System Models, Gesellschaft für Informatik (GI), Germany, vol. 36, pp. 32-36, August 1990.
- [61] R.J. Machado, J.M. Fernandes, H.D. Santos, "A Methodology for Complex Embedded Systems Design – Petri Nets within a UML Approach", 2nd IFIP International Workshop on Distributed and Parallel Embedded Systems – DIPES'00, Session of Methodologies, Paderborn, Germany, 2000.
(<http://get.dsi.uminho.pt/bib/addes.pdf> – data de acesso: 01/03/2004)
- [62] W.L.A. Oliveira, "Proposta de uma Rede de Petri de Alto-Nível Incluindo Temporização", IBILCE/UNESP, Dissertação de Mestrado, 131 pp., 1997.
- [63] N. Marranghello, J. Mirkowski and K. Bilinski, "Synthesis of Synchronous Digital Systems Specified by Petri Nets", In: "Hardware Design and Petri Nets", Chapter 7, pp.129-150, A. Yakovlev, L. Gomes, and L. Lavagno (eds.), Kluwer Academic Publishers Inc., 2000, ISBN: 0-7923-7791-5.
- [64] Z. Peng, K. Kuchcinski, "Automated Transformation of Algorithms into Register-transfer Level Implementation", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 13, no. 2, pp. 150-166, 1994.
- [65] Z. Peng, "A Formal Methodology for Automated Synthesis of VLSI Systems", Ph.D. Thesis, nr. 170, Department of Computer and Information Sciences, Linköping University, Sweden, 1987.
- [66] R. Bawa, E. Encrenaz, "A Tool for Translation of VHDL Descriptions into a Formal Model and its Application to Formal Verification and Synthesis", Lecture Notes in Computer Science, Springer-Verlag, vol. 1135, pp. 471-474, 1996.
- [67] E. Encrenaz, "A Symbolic Relation for a Subset of VHDL'87 Descriptions and its Application to Symbolic Model Checking", Proceedings of the IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science, Springer-Verlag, vol. 987, pp. 328-342, 1995.
- [68] R. Bawa, L. Jacomme, "Synthèse de Descriptions Comportementales Séquentielles en Conformité Avec la Sémantique VHDL", Actes de Colloque CAO de Circuits Intégrés et Systèmes, pp. 303-306, 1997.
- [69] L. Gomes, "Fuzzy modeling for reactive real-time systems control using Reactive Petri Nets; In "Fuzziness in Petri Nets"; J. Cardoso, H. Scarpelli (Eds.), Physica-Verlag (Springer), 1999, ISBN 3-7908-1158-0.
- [70] P. Chevaillier, J. Tisseau, F. Harrouet, R. Querrec, "Prototyping Manufacturing Systems. Contribution of Virtual Reality, Agent Systems and Synchronous Interpreted Petri Nets",

- In Preprints of the 9th symposium on Information Control in Manufacturing (INCOM'98), volume 3, pp. 249-254, Nancy-Metz (France), June 1998.
- [71] R.J. Machado, J.M. Fernandes, A.J. Esteves, H.D. Santos, "An Evolutionary Approach to the Use of Petri Net based Models: From Parallel Controllers to HW/SW Co-Design", In: "Hardware Design and Petri Nets", Chapter 11, pp. 205-222, A. Yakovlev, L. Gomes and L. Lavagno (eds.), Kluwer Academic Publishers Inc., 2000, ISBN: 0-7923-7791-5.
- [72] R.J. Machado, J.M. Fernandes, A.J. Proença, "SOFHIA: A CAD Environment to Design Digital Control Systems", In C. Delgado Kloos and E. Cerny (Eds.) Proceedings of the XIII IFIP Conference on Computer Hardware Description Languages and their Applications, pp. 8688, Chapman & Hall, 1997.
- [73] R. Esser, "An Object Oriented Petri Net Language for Embedded System Design", Proc. STEP'97 – 8th International Workshop on Software Technologies and Engineering Practice, London, UK, July 1997.
(<ftp://ftp.tik.ee.ethz.ch/pub/people/esser/paper/Step97.ps.gz> – data de acesso: 01/03/2004)
- [74] R. Esser, J.W. Janneck, M. Naedele, "Applying an Object-Oriented Petri Net Language to Heterogeneous System Design", Proc. of the Workshop Petri Nets in System Engineering PNSE'97, Hamburg, Germany, 25-26 September 1997.
(http://ptolemy.eecs.berkeley.edu/~janneck/Publications/1997_PNSE.pdf – data de acesso: 01/03/2004)
- [75] K. Bilinski, E. Dagless, "High-Level Synthesis of Synchronous Parallel Controllers", Proceedings of the 17th International Conference on Application and Theory of Petri Nets, J. Billington and W. Reisig (Eds.), Lecture Notes in Computer Science, vol. 1091, pp. 93-112, 1996.
- [76] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers", IEICE Transactions on Information and Systems, Vol. E80-D, No. 3, pp. 315-325, 1997.
(<http://www.lsi.upc.es/~jordic/petrify/refs/ieice97.ps.gz> – data de acesso: 01/03/2004)
- [77] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Synthesis of control circuits from STG specifications – Practical Exercise Manual", Tutorial in ASYNC'2000, Eilat, Israel, 2000.
(<http://www.staff.ncl.ac.uk/alex.yakovlev/home.formal/talks/async2k/ASYNC00handson-frame.html> – data de acesso: 01/03/2004)
- [78] R.H. Klenke, M. Meyassed, J.H. Aylor, B.W. Johnson, R. Rao, A. Ghosh, "An Integrated Design Environment for Performance and Dependability Analysis", In Proc. DAC'97, Anaheim, California, June 9-13 1997.
(http://www.sigda.org/Archives/ProceedingArchives/Dac/Dac97/papers/1997/dac97/pdffiles/10_2.pdf – versão longa em <http://www.people.vcu.edu/~rhklenke/docs/970528.pdf> – datas de acesso: 01/03/2004)
- [79] P. Maciel, "Register estimation for digital system design: an approach based on timed Petri nets", Proceedings of the second IEEE International Conference on Systems, Man and Cybernetics (SMC'02), October 6-9, 2002, Hammamet, Tunisia, Volume 4. IEEE Computer Society Press, October 2002.
- [80] P. Maciel, T. Barros, "A Petri Net Based Approach for Resource Estimation in Manufacturing Control Systems", In: Invited Session on Modelling and Analysis based on Petri nets – International Conference on Information Systems, Analysis, and Synthesis. Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics (SCI'2001), Computer Science and Engineering: Part II, Volume 14, 2001.

- [81] F.C. Filho, P. Maciel, E. Barros, "*Using Petri nets for data dependency analysis*", Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'2000), 8-11 October 2000, Nashville, TN, Vol. 4, pp. 2998-3003, 2000.
- [82] P. Maciel, F. C. Filho, E. Barros, W. Rosenstiel, "*Methods based on Petri net for resource sharing estimation*", Proc. 13th Symp. on Integrated Circuits and Systems Design, 18-24 September 2000, Manaus, Brazil, pages 75-80. 2000.
- [83] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, A. Yakovlev, "*Coupling Asynchrony and Interrupts: Place Chart Nets*", Proceedings of 18th International Conference on Application and Theory of Petri Nets, Lecture Notes in Computer Science, ed. P. Azéma, G. Balbo, Vol. 1248, pp. 328-347, Toulouse, France, 1997.
- [84] J. Mirkowski, A. Yakovlev, "*A Petri Net Model for Embedded Systems*", Proceedings of the Design and Diagnostics of Electronic Circuits and Systems Workshop, Szczyrk, Poland, September, 1998.
- [85] J. Mirkowski, "*Towards an effective Petri net based formal model for reactive embedded systems*", Proc. 2nd Workshop on Hardware Design and Petri Nets (HWPN'99) of the 20th Int. Conf. on Application and Theory of Petri Nets (PN'99), Williamsburg, VA, pp. 101-112, 1999.
- [86] K. Lautenbach, H. Schmid, "*Use of Petri nets for proving correctness of concurrent process systems*", Information Processing 1974 – North Holland Pub. Co., pp. 187-191, 1974.
- [87] R. David, H. Alla, "*Petri Nets and Grafcet – Tools for modelling discrete event systems*", Prentice Hall – ISBN 0-13-327537-X, 1992.
- [88] K. Varpaaniemi, "*On Stubborn Sets in the Verification of Linear Time Temporal Properties*", In: J. Desel, M. Silva (Eds.), Proceedings of the 19th International Conference on Application and Theory of Petri Nets, Lisbon, Portugal, June 1998. Lecture Notes in Computer Science, Vol. 1420, Springer-Verlag, pp. 124-143, 1998.

APÊNDICE I: TÓPICOS DA TEORIA DE GRAFOS

I.1. INTRODUÇÃO

Este apêndice apresenta os aspectos da teoria de grafos mais utilizados na teoria de redes de Petri.

Um **grafo** é uma estrutura $G = (V, E)$ onde V é um conjunto discreto finito e E é uma família cujos elementos e (não vazios) são definidos em função dos elementos v de V , em duas formas possíveis (figura I.1):

- se $E \subset P_2(V)$ (pares não ordenados), G será um **grafo não orientado** ou **grafo simples**;
- se $E \subset V^2$ (quadrado cartesiano), G será um **grafo orientado**.

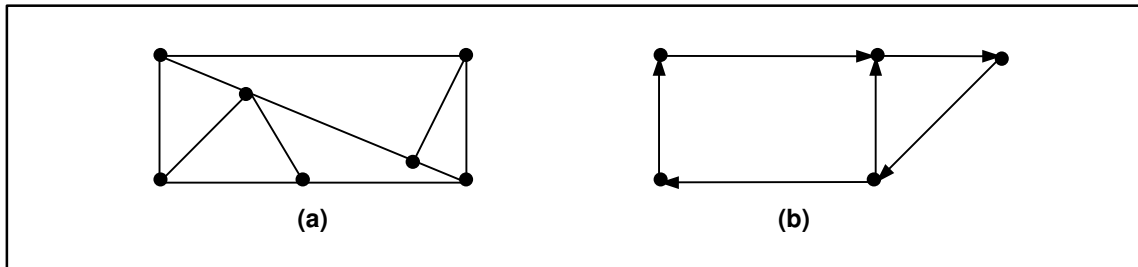


Figura I.1. Grafos não orientado (a) e orientado (b).

Os elementos de V são chamados **vértices**, **nós** ou **pontos** e o valor $n = |V|$ é a **ordem** da estrutura. A família E pode ser entendida como uma **relação ou conjunto de relações de adjacência** cujos elementos são chamados em geral **ligações**; em particular, nas estruturas não orientadas, os elementos e de E são conhecidos como **arestas** e, nas orientadas, como **arcos**. Dois vértices que participam de uma ligação são ditos adjacentes. O valor $m = |E|$ é chamado o **tamanho** do grafo. Se $m = 0$, o grafo é dito **trivial**.

Subestrutura de um grafo $G = (V, E)$ é um grafo $H = (X, Y)$ no qual $X \subseteq V$ e $Y \subseteq E \cap X^2$. Se $X = V$, mas $Y \subset E$, a subestrutura é chamada **grafo-parcial**. Por outro lado, se $X \subset V$, e $Y = E \cap X^2$, a subestrutura é chamada **subgrafo**. A figura I.2 mostra um grafo parcial e um subgrafo.

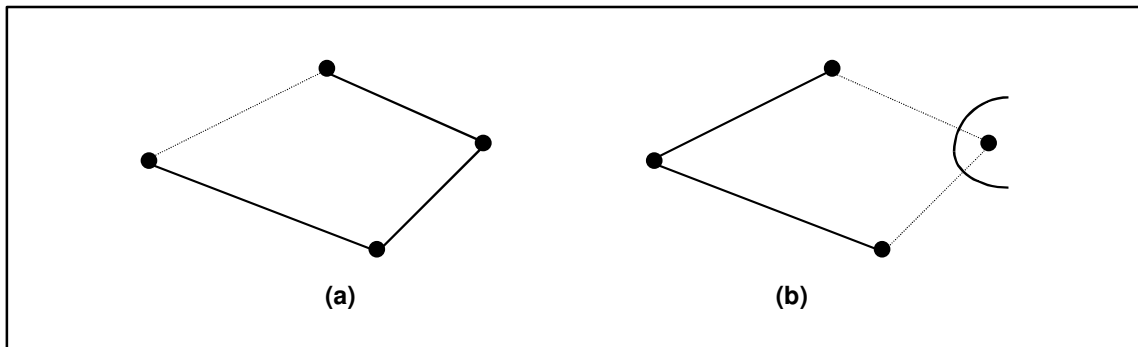


Figura I.2. Grafo parcial (a) e subgrafo (b).

O **esquema** de um grafo (habitualmente chamado **o grafo**) é obtido associando-se a cada vértice um ponto ou uma pequena área delimitada por uma fronteira e a cada ligação uma linha (ou seta, no caso de grafos orientados). A caracterização de V como um conjunto implica na identificação dos vértices. Neste caso, fala-se de **grafos rotulados**, nos quais cada vértice é identificado por um rótulo que é, geralmente, uma palavra ou número. A

identificação dos vértices por rótulos implica na identificação das ligações. Exemplos de grafos rotulados são dados na figura I.3.

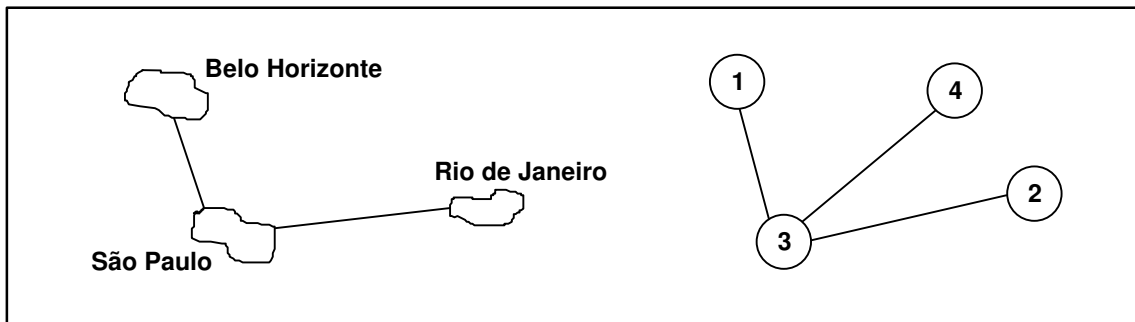


Figura I.3. Grafos rotulados.

Em muitas situações, como nas redes de Petri, há interesse no particionamento do conjunto de vértices de um grafo em subconjuntos que apresentam propriedades importantes para o estudo que se realiza. A mais utilizada das partições consiste em dividir o conjunto V em subconjuntos de vértices mutuamente não adjacentes.

Se existir uma partição P do conjunto de vértices de um grafo $G = (V, E)$, com cardinalidade k , de modo que existam ligações apenas entre vértices de diferentes elementos de P , o grafo é dito **k-partido**. Na figura I.4, as linhas pontilhadas ilustram a separação das diferentes partições.

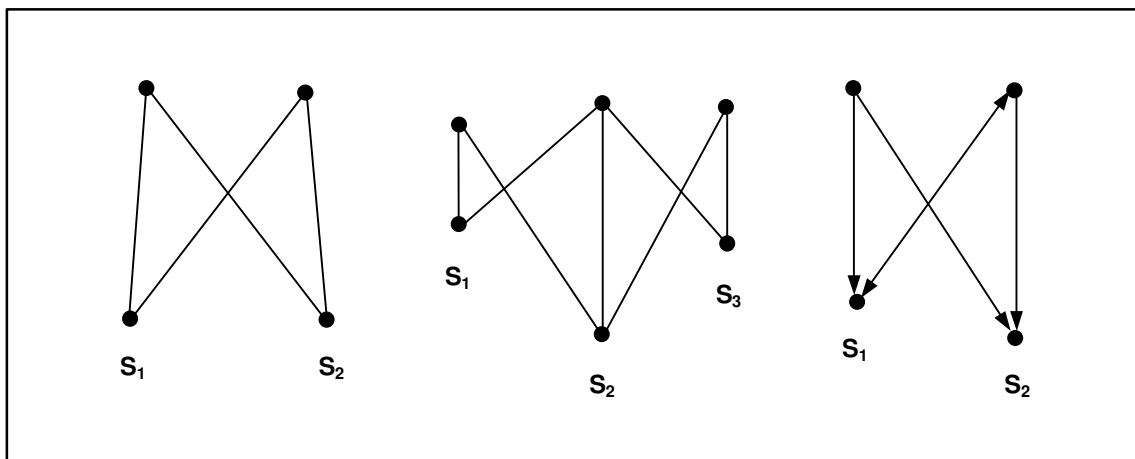


Figura I.4. Partição de grafos.

Ainda sobre partições, quando $k = 2$ o grafo é chamado **bipartido**. Este é o caso de uma rede de Petri. Nela, o conjunto de lugares e o conjunto de transições constituem duas partições do conjunto de vértices (conjunto formado pela união dos dois primeiros).

I.2. PERCURSOS EM UM GRAFO

Um **percurso**, ou **cadeia**, é uma família de **ligações sucessivamente adjacentes**. O percurso será **fechado** se a última ligação da sucessão for adjacente à primeira e, será **aberto**, em caso contrário. Na definição geral, despreza-se implicitamente a orientação das ligações, quando se trata de grafos orientados. A figura I.5 apresenta alguns exemplos de percursos.

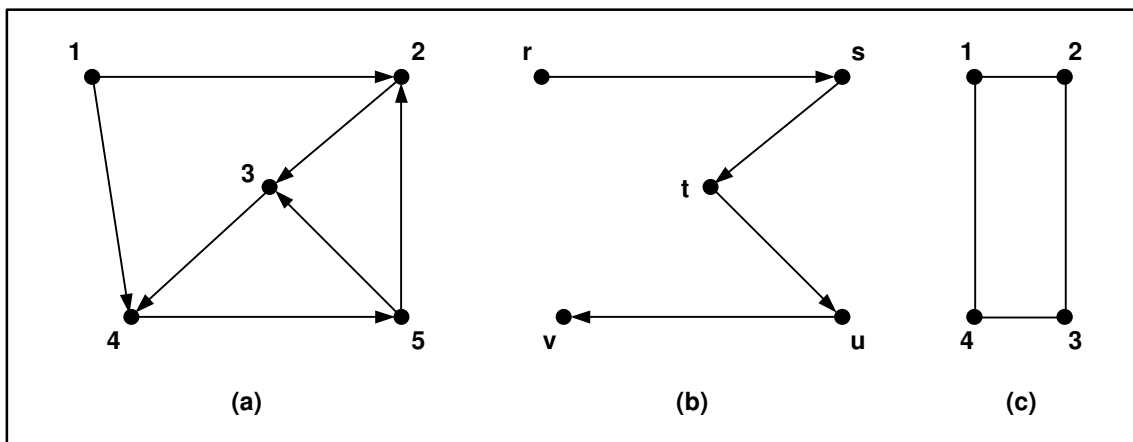


Figura I.5. Exemplos de percursos.

A notação é feita indicando-se a sucessão através dos vértices, das ligações ou de vértices e ligações alternados, ou apenas dos vértices inicial e final, quando isso for suficiente. Na figura I.5.a, são exemplos de percursos entre os vértices 1 e 3, as seguintes formas:

$p_{a1} = (1, 2, 3, 4, 5, 3),$
 $p_{a2} = ((1, 2), (2, 3), (3, 4), (4, 5), (5, 3)),$
 $p_{a3} = (1, (1, 2), 2, (2, 3), 3, (3, 4), 4, (4, 5), 5, (5, 3), 3),$
 $p_{a4} = ((1, 2), (2, 3), (3, 4), (4, 5), (5, 2), (2, 3))$ e
 $p_{a5} = p_{1-3},$

onde a última forma caracteriza qualquer percurso entre os vértices 1 e 3.

Um percurso é **simples** se não repetir ligações e é **elementar** se não repetir vértices. Por exemplo, na figura I.5.a, o percurso p_{a2} é simples, ao passo que o percurso p_{a4} não o é. Já o percurso $p_b = (r, s, t, u, v)$, na figura I.5.b, é elementar (além de simples, pois não há a repetição de ligações).

Um **ciclo** é uma cadeia simples e fechada. Um **caminho** é uma cadeia em um grafo orientado, na qual a orientação dos arcos é sempre a mesma, a partir do vértice inicial. Um **circuito** é um caminho simples e fechado em um grafo orientado. O percurso $p_c = (1, 2, 3, 4, 1)$, na figura I.5.c, é um ciclo. Já o percurso $p_b = (r, s, t, u, v)$, na figura I.5.b, é um caminho. Exemplo de circuito é o percurso $p_{a6} = (3, 4, 5, 3)$, na figura I.5.a.

I.3. CONEXIDADE EM UM GRAFO

A noção de conexidade está relacionada à **possibilidade de passagem** de um vértice a outro em um grafo através das ligações existentes. Ela traduz, portanto, o "estado da ligação" de um grafo e adquire aspectos diferentes conforme o grafo seja orientado ou não.

I.3.1. TIPOS DE CONEXIDADE

Um grafo qualquer (orientado ou não) é **não conexo**, ou **desconexo**, se nele existir ao menos um par de vértices não unidos por uma cadeia. Veja figura I.6.

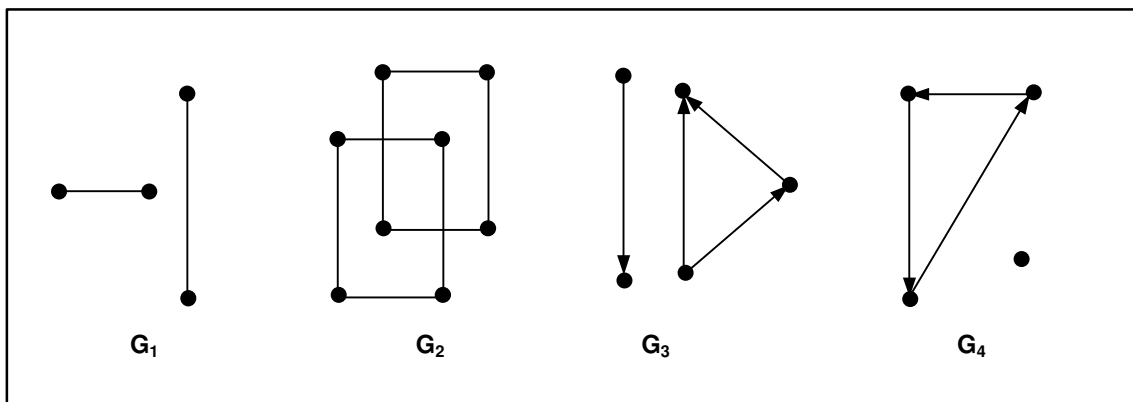


Figura I.6. Grafos não conexos.

Um grafo que não é desconexo – portanto, um grafo no qual todo par de vértices é unido por ao menos uma cadeia – é dito **conexo**. Trata-se de duas alternativas mutuamente exclusivas: um grafo não conexo pode sempre ser decomposto em subgrafos conexos, que são as suas **componentes conexas**.

Já no caso de grafos orientados, alguma distinção adicional precisa ser feita. Grafo **simplesmente conexo**, ou **s-conexo**, é um grafo no qual todo par de vértices é unido por ao menos uma cadeia. A definição é, portanto, a mesma do caso não orientado. A figura I.7 mostra alguns exemplos ilustrativos.

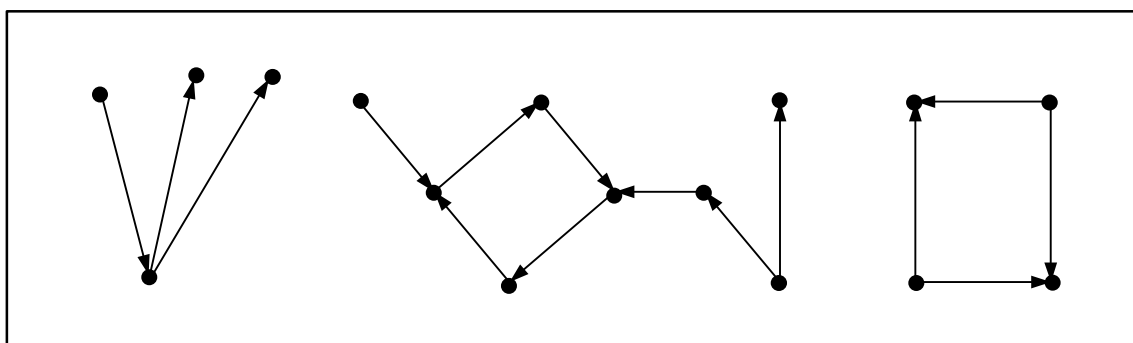


Figura I.7. Grafos s-conexos.

Grafo **semi-fortemente conexo**, ou **sf-conexo**, é um grafo no qual, em todo par de vértices, ao menos um é atingível a partir do outro (logo, entre eles existe um caminho em, ao menos, um dos dois sentidos possíveis). A figura I.8 apresenta alguns exemplos desses grafos.

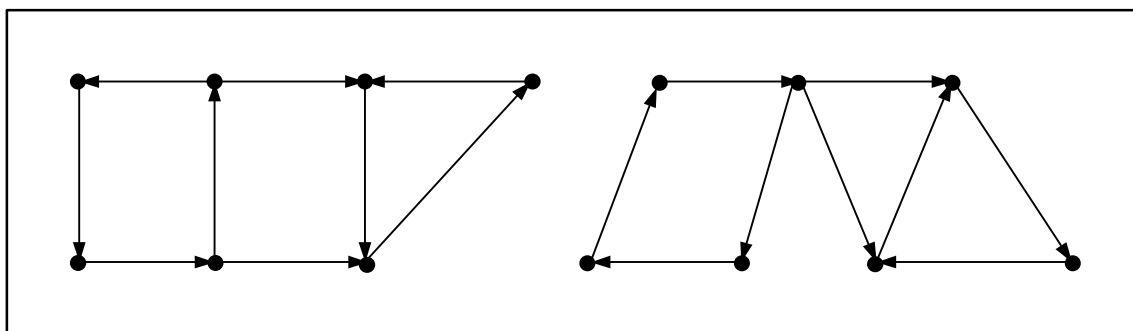


Figura I.8. Grafos sf-conexos.

Grafo **fortemente conexo**, ou **f-conexo**, é um grafo no qual dois vértices quaisquer são mutuamente atingíveis. Logo, a todo par de vértices está associado um par de caminhos de sentidos opostos. Em outras palavras, todo vértice é atingível a partir de um vértice fixado e todo vértice atinge qualquer vértice fixado. Exemplos desses grafos encontram-se na figura I.9.

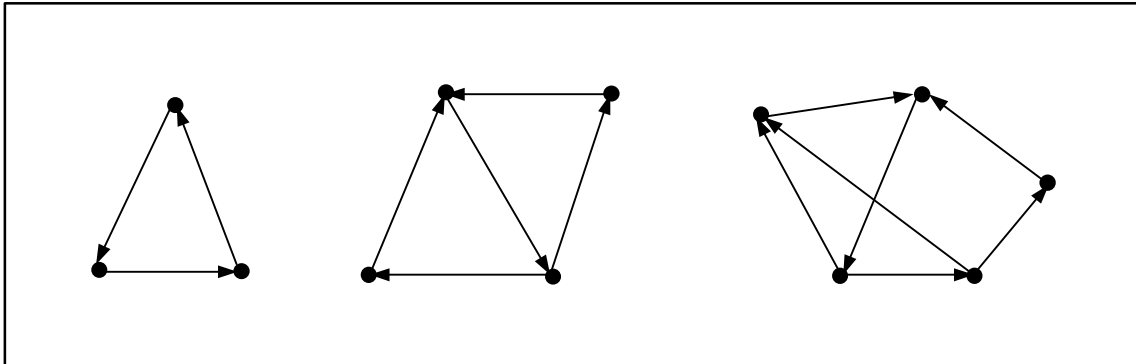


Figura I.9. Grafos f-conexos.

I.3.2. COMPONENTES FORTEMENTE CONEXAS (CFC)

Como foi visto, a f-conexidade implica em atingibilidade recíproca, o que se pode mostrar que corresponde a uma **relação de equivalência** sobre os vértices de um grafo $G = (V, E)$ orientado qualquer, o que implica em que ela define uma **partição** $S = \{S_i \mid S_i \subset V, S_i \cap S_j = \emptyset, i, j = 1, \dots, r\}$ do conjunto V de vértices. Os subgrafos correspondentes aos S_i são as **componentes f-conexas maximais** ou, de forma abreviada, **componentes f-conexas**. Se um grafo é f-conexo, a partição será apenas $S = \{V\}$.

A decomposição em componentes f-conexas pode ser observada na figura I.10. O grafo à direita é o mesmo que o da esquerda, porém com os vértices dispostos de modo a evidenciar as componentes f-conexas correspondentes.

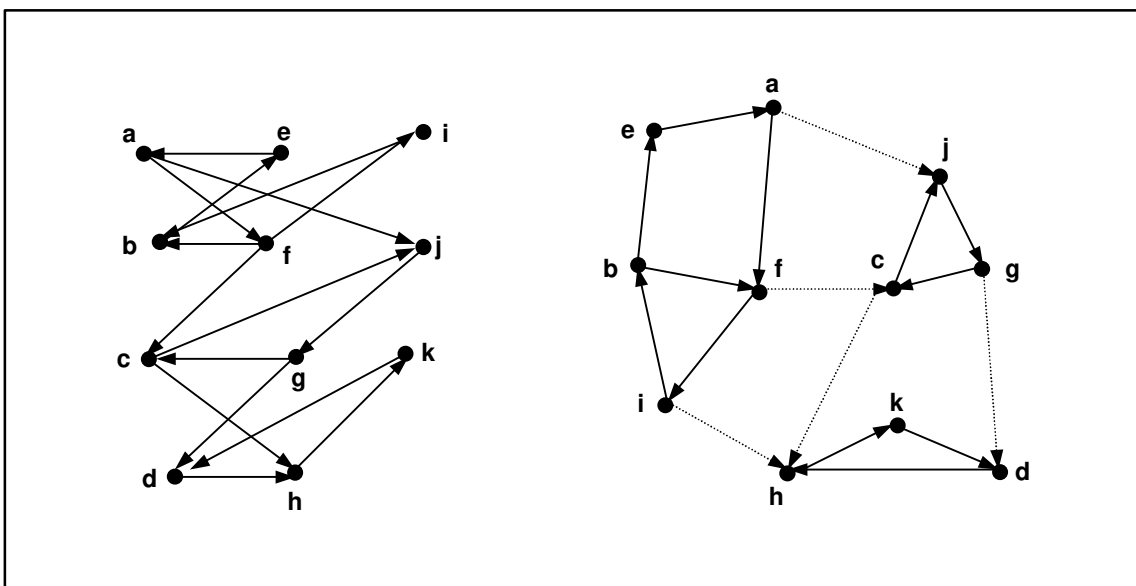


Figura I.10. Componentes f-conexas.

APÊNDICE II: VALIDAÇÃO DE RPSD COMO REDE DE PETRI

II.1. INTRODUÇÃO

Para validar RPSD como uma extensão legítima de rede de Petri, o capítulo 12 apresentou um conjunto de convenções e de modelos em redes de baixo nível, representando lugares, transições e ramos RPSD. Este apêndice apresenta a simulação de tais modelos de equivalência como forma de validá-los e, por conseguinte, validar RPSD como uma extensão de rede de Petri.

Os modelos de equivalência do capítulo 12 são descritos através de **redes de Petri Temporizadas de Merlin (RPTM)** – figuras 12.13 a 12.19. Conforme visto no capítulo 8, RPTM constitui-se numa extensão de redes **Lugar/Transição (RP L/T)**, ao agregar-lhes intervalos de disparo associados às transições.

Dentre os simuladores gratuitos de redes de Petri disponíveis na internet, com capacidade de simular modelos RPTM, destaca-se a ferramenta **ARP** (*Analisador de Redes de Petri*). Trata-se de um simulador/analizador desenvolvido pelo professor **Carlos Maziero**, em seu trabalho de mestrado na **Universidade Federal de Santa Catarina**. Escrito em **Turbo Pascal 6**, o **ARP** pode ser executado em máquinas com **MS-DOS** e **Windows**. Atualmente, o professor Maziero desenvolve uma versão gráfica da ferramenta **ARP**, ainda não disponível. Pode-se fazer o *download* da versão final da ferramenta **ARP** (versão 2.4) no endereço <http://www.ppgia.pucpr.br/~maziero/>.

Na ferramenta **ARP**, os modelos em redes de Petri são descritos usando-se de uma linguagem bastante simples, similar ao **Pascal**. Contudo, como os modelos do capítulo 12 são extensos, optou-se por outra ferramenta, complementar à ferramenta **ARP**, denominada **JARP**. Trata-se de um programa **Java**, desenvolvido, também, na **Universidade Federal de Santa Catarina**, composto por um editor gráfico e um simulador *token game* (interativo) para redes Lugar/Transição, capaz de exportar arquivos para os formatos **PN** (utilizado como entrada do **ARP**) e **PNML** (*Petri Net Markup Language* – baseada em **XML**), além dos formatos gráficos **GIF**, **JPEG**, **PNG** e **PPM**. O *download* da ferramenta **JARP** pode ser feito no endereço <http://sourceforge.net/projects/jarp/>. A versão utilizada foi a 1.1.13.

Desta forma, utilizou-se a interface gráfica do editor **JARP** para produzir arquivos para o simulador/analizador **ARP**. Além disso, como o **ARP** apresenta limitações quanto ao número de lugares, de transições e de estados gerados, houve a necessidade de utilizar-se a **simulação interativa** no **JARP**, para as redes correspondentes aos modelos descritos nas figuras 12.17 e 12.19.

A seção II.2 descreve a edição e a simulação interativa (*token game*) de modelos no **JARP**. A utilização da ferramenta **ARP** é descrita na seção II.3, juntamente com os resultados de simulação obtidos para as redes das figuras 12.13 a 12.19.

II.2. EDIÇÃO E SIMULAÇÃO INTERATIVA DE MODELOS NO JARP

A figura II.1 mostra a janela de edição de modelos do programa **JARP**. Na parte superior da janela encontra-se a barra de menus, cujos itens mais utilizados são acessíveis pelos botões dispostos em duas fileiras, uma horizontal e outra vertical, situados logo abaixo de tal barra, emoldurando a área de edição. A fileira horizontal de botões apresenta funções comuns a programas com interface padrão *Windows* (*novo arquivo / abrir arquivo / restaurar*

arquivo / salvar / imprimir etc.). Por outro lado, a fileira vertical de botões apresenta funções relativas à edição e à simulação de redes. O primeiro destes botões (com o ícone de uma seta preta, na diagonal) serve para retornar ao modo de edição, quando uma simulação está em execução. Na sequência, botões para a inserção de lugares, transições e comentários. Os três últimos botões são relativos à simulação: *iniciar / salvar marcação atual / desfazer o último disparo*.

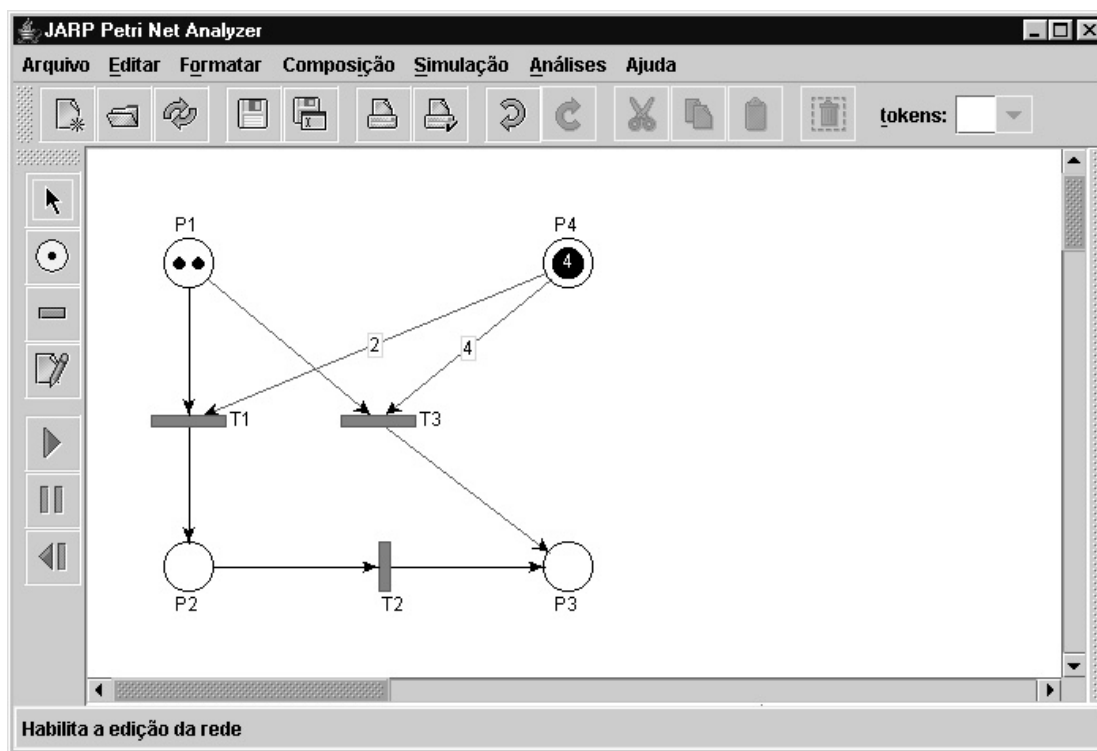


Figura II.1. Edição de uma rede no programa JARP.

Ao seleccionar um lugar na área de edição, pode-se definir o seu rótulo e a sua marcação. Esta última pode ser alterada com o uso de botões rotulados com os sinais de mais ('+') e de menos ('-'), os quais servem, respectivamente, para adicionar *tokens* ao lugar e para remover *tokens* do lugar (estes botões aparecem logo após a seleção do lugar). Até três *tokens* podem ser representados individualmente em um lugar. Acima deste valor, o total de *tokens* é representado por um número no interior de um círculo preto. Na figura II.1, observa-se que o lugar **P1** possui 2 *tokens*, os lugares **P2** e **P3** estão vazios e o lugar **P4** possui 4 *tokens*.

Para conectar um lugar a uma transição através de um ramo, deve-se, inicialmente, seleccionar o lugar, fazendo aparecer, desta forma, um círculo azul em seu interior. Posicionando o ponteiro do mouse sobre este círculo, deve-se dar um clique no botão esquerdo e, mantendo tal botão pressionado, deve-se arrastar o ramo que surge até a parte central da transição desejada. De forma análoga, pode-se conectar uma transição a um lugar.

A definição do peso associado a um ramo é feita de forma análoga à definição da marcação de um lugar: após a seleção do ramo pretendido, aparecerão botões rotulados com os sinais de mais e de menos, os quais permitirão, respectivamente, aumentar e diminuir o peso associado ao ramo. Na figura II.1, o lugar **P4** conecta-se às transições **T1** e **T3** com ramos de pesos 2 e 4, respectivamente.

Uma vez definida a rede, passa-se à sua simulação. A figura II.2 mostra a janela do programa **JARP**, imediatamente após o início da **simulação interativa**, a qual é acionada através do botão cujo ícone é uma seta para a direita.

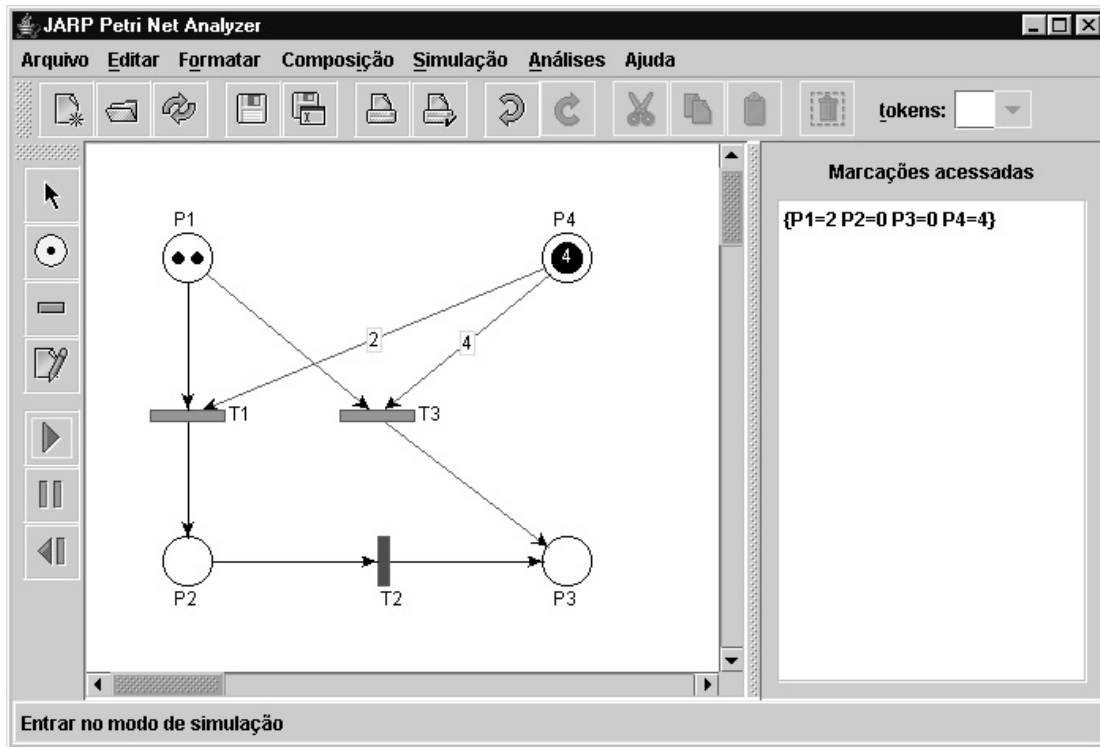


Figura II.2. Simulação de uma rede no programa JARP.

Comparando-se as figuras II.1 e II.2, pode-se observar o surgimento de uma área à direita, a qual descreve as marcações acessadas que foram salvas. Além disso, observa-se que as transições **T1** e **T3** apresentam coloração diferente. Esta diferença de coloração é utilizada para destacar as transições que estão habilitadas. Assim, logo no início da simulação, pode-se disparar tanto a transição **T1** quanto a **T3**: o disparo de **T1** promoverá a remoção de um *token* de **P1** e de 2 *tokens* de **P4**, gerando a marcação mostrada na figura II.3; por outro lado, o disparo de **T3** gerará a marcação ilustrada na figura II.4. Na marcação da figura II.3, as transições **T1** e **T2** encontram-se habilitadas. Já na marcação da figura II.4, nenhuma transição está habilitada.

Pelo exposto, observa-se que apesar de intuitiva, a simulação interativa deve ser usada com cautela, uma vez que a quantidade de possibilidades de disparo pode-se tornar grande, necessitando-se, desta forma, de um acompanhamento sistemático das seqüências de disparo que vão sendo geradas. Neste contexto, o **JARP** oferece a possibilidade de salvar o gráfico da rede, mostrando uma dada marcação e o respectivo conjunto de transições habilitadas. Este recurso foi utilizado na simulação dos modelos correspondentes às figuras 12.17 e 12.19.

Até o presente momento, a equipe responsável pelo desenvolvimento do programa **JARP** não disponibilizou uma versão que faça a simulação completa de modelos (simulação não interativa).

Outras informações sobre o funcionamento do **JARP**, assim como atualizações do mesmo, podem ser encontradas no endereço fornecido na introdução deste apêndice.

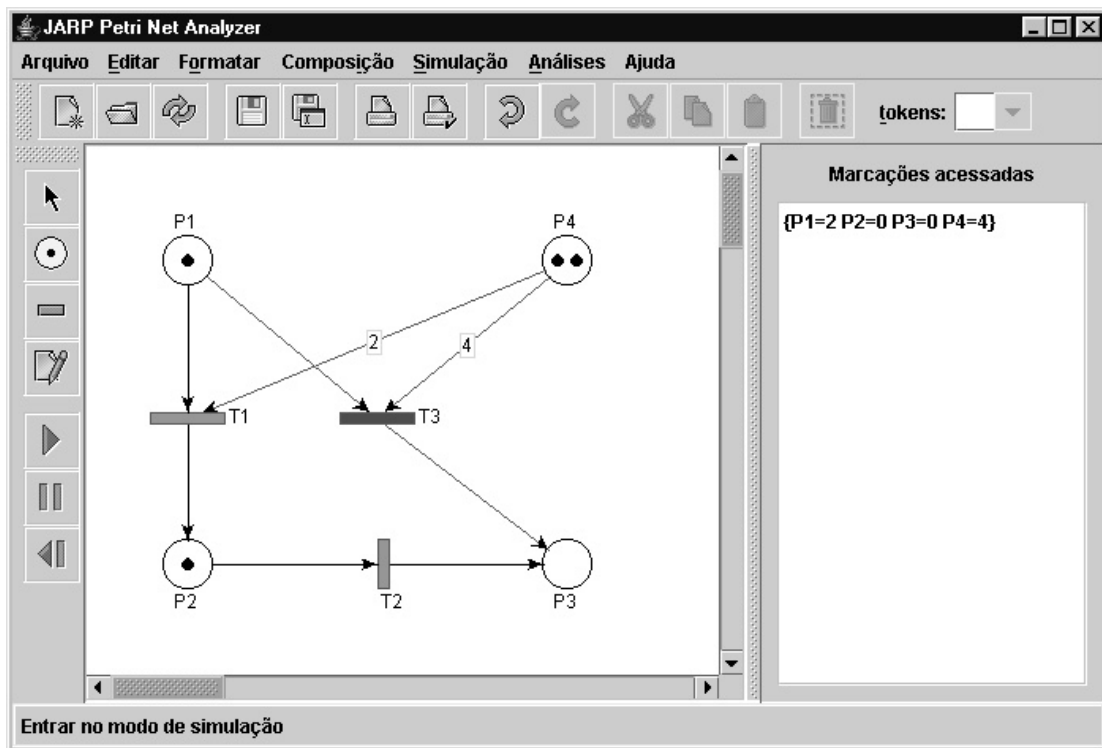


Figura II.3. Estado da rede da figura II.2 caso T1 dispare.

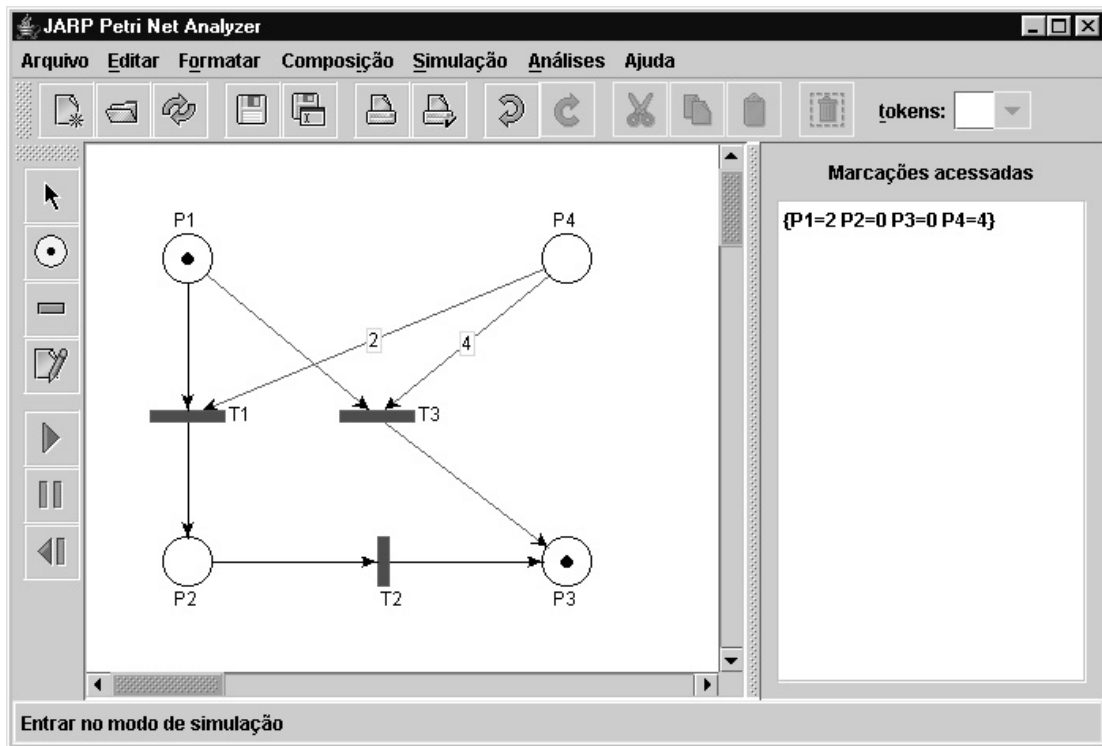


Figura II.4. Estado da rede da figura II.2 caso T3 dispare.

II.3. SIMULAÇÃO DOS MODELOS DE EQUIVALÊNCIA

Nesta seção, os modelos RPTM mostrados nas figuras 12.13 a 12.19 são simulados com o auxílio dos programas **JARP** e **ARP**. A utilização do **JARP** foi vista na seção anterior. A descrição da utilização do **ARP** será feita, nesta seção, usando-se, como exemplo, o modelo de equivalência da figura 12.13. A simulação dos modelos das figuras 12.14, 12.15, 12.16 e 12.18 segue o padrão de simulação do modelo da figura 12.13. Já a simulação dos modelos das figuras 12.17 e 12.19 não é possível de ser feita no **ARP**: a simulação da rede 12.17 trava ao estourar o limite de estados, enquanto a rede 12.19 excede o limite máximo de lugares e de transições permitido pelo **ARP**. Para estas redes, adotou-se a **simulação interativa** do **JARP**.

Como a edição de modelos no **ARP** mostrou-se um processo enfadonho e propenso a erros, o programa **JARP** foi utilizado para a edição de todos os modelos de equivalência. Os modelos editados no **JARP** foram, então, salvos em arquivos nos formatos **PN** e **GIF**, utilizando-se, para tal, da opção *Salvar como*, presente no **JARP**. O formato **PN** é apropriado para a leitura pelo **ARP**, enquanto o formato **GIF** é apropriado para a representação gráfica dos modelos **JARP** neste apêndice, tal como ocorre na figura II.5, a qual corresponde ao modelo **JARP** relacionado à figura 12.13 (lugares não conectados foram excluídos).

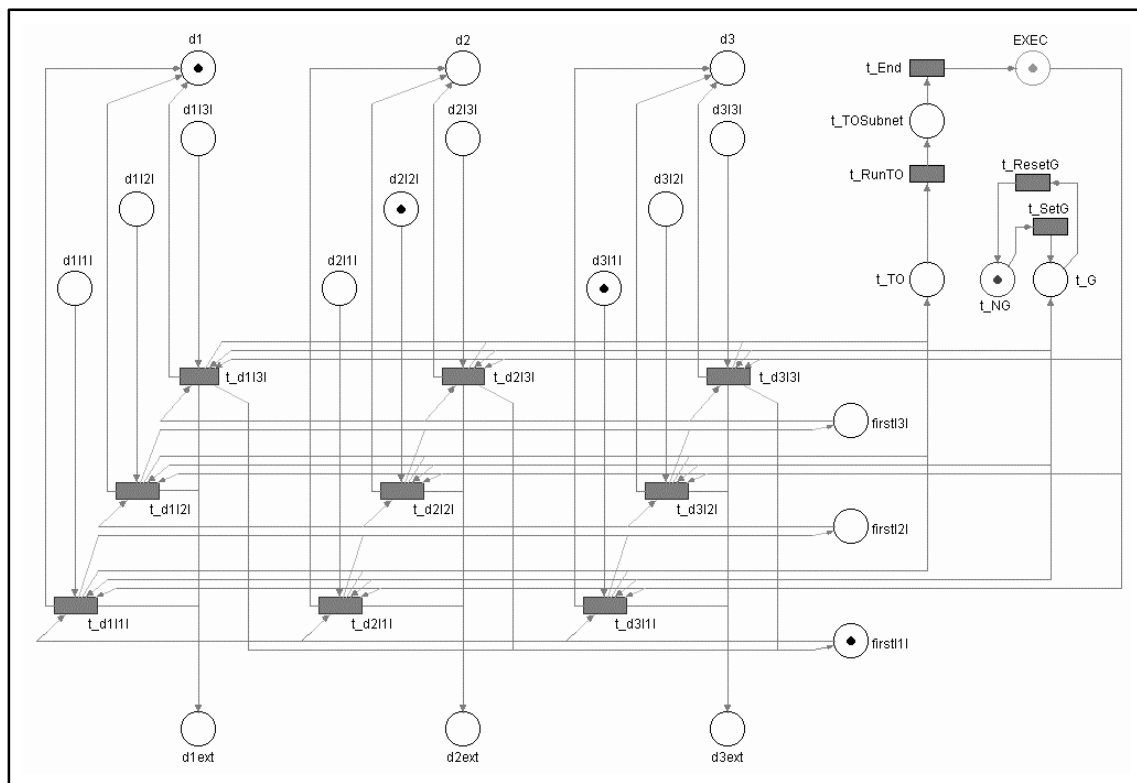


Figura II.5. Modelo JARP correspondente à RPTM da figura 12.13.

Comparando-se as figuras 12.13 e II.5, observa-se que o **JARP** não possui formas de atribuição de intervalos de disparo às transições. A adição de tais intervalos será feita somente no arquivo **PN** correspondente, para tratamento pelo programa **ARP**, conforme visto adiante. Outra observação refere-se ao acréscimo do lugar **t_{NG}** e das transições **t_{SetG}** e **t_{ResetG}**, à rede da figura II.5, necessários à simulação da guarda de **t**. O disparo de **t_{SetG}** promoverá a transferência do *token* presente em **t_{NG}** para **t_G**, tendo como significado a passagem da guarda **G(t)** de *falsa* para *verdadeira*. Já o disparo de **t_{ResetG}** terá efeito contrário.

Ao executar o programa **ARP**, surge a tela mostrada na figura II.6. As opções de menu ilustradas na figura serão tratadas oportunamente.



Figura II.6. Tela inicial do simulador ARP 2.4.

A primeira ação a ser feita consiste no carregamento do arquivo **PN**, gerado pelo programa **JARP**, correspondente à rede da figura 12.13. Acionando-se a opção **Load** (através da tecla 'L'), a tela do **ARP** apresenta a configuração mostrada na figura II.7, na qual deve-se selecionar o arquivo desejado (no caso, **R12-13.PN**).



Figura II.7. Carregamento da rede produzida pelo programa JARP.

Após o carregamento do arquivo, deve-se alterá-lo, para incluir os intervalos de disparo das transições. Para isso, faz-se necessário compreender a **sintaxe** da **linguagem de descrição** usada pelo **ARP**. Esta sintaxe – acessível através do menu **Help**, a partir da tela de edição do programa – encontra-se reproduzida na figura II.8.



Figura II.8. Instruções de edição.

Na figura II.8, observa-se que os **nós** da rede (**lugares** e **transições**) são descritos na seção intitulada **NODES** (o uso de caracteres maiúsculos e minúsculos é indiferente), enquanto as **associações** entre nós (**ramos**) são descritas na seção **STRUCTURE**. Na seção **NODES**, cada identificador de lugar é acompanhado da palavra **PLACE**, enquanto cada identificador de transição é seguido pela palavra **TRANSITION**. Um **lugar** pode ser **inicializado** com um ou mais *tokens*, bastando-se para isso discriminar tal quantidade, entre parênteses, após a palavra **PLACE** correspondente. Por sua vez, uma **transição** pode receber um **intervalo de disparo** através do acréscimo do mesmo, entre colchetes, após a palavra **TRANSITION** correspondente. Um **ramo** é especificado a partir do identificador da transição conectada, após o qual segue-se duas listas (entre parênteses, cada qual): a primeira relativa aos lugares de entrada da transição e, a segunda, relativa aos lugares de saída. Os pesos são indicados pela presença de um sinal de vezes (*), seguido do valor correspondente e do identificador do lugar a que se referem.

Retornando ao editor de textos do **ARP**, a figura II.9 mostra a adição de intervalos de disparo à listagem **R12-13.PN**, originalmente produzida pelo programa **JARP**, de forma a compatibilizá-la com a especificação da figura 12.13. A única transição a não receber intervalo é **t_SetG**, o que significa, para a extensão RPTM (veja capítulo 8), o mesmo que o intervalo $[0, \infty]$. Desta forma, pela regra de disparo, esta transição somente será disparada caso não exista outra transição habilitada com intervalo de disparo superior finito. Já o intervalo da transição **t_ResetG** foi definido por $[10, 10]$, para transpor o disparo de tal transição para depois do disparo das demais (obs.: não só no modelo **JARP** da figura II.5, mas também nos demais, o disparo de **t_ResetG** provocará apenas o acréscimo de ciclos redundantes no grafo de classes da rede modelada, os quais não interferem nas execuções em andamento na rede,

uma vez que os modelos de equivalência foram construídos de forma a garantir o disparo de uma transição como uma operação atômica).



Figura II.9. Edição da rede – adição de intervalos de disparo.

Após o acréscimo dos intervalos de disparo, o modelo **ARP** correspondente à figura 12.13 apresentará a descrição dada na listagem II.1. Observe que os caracteres '[' e ']' foram substituídos por 'I', uma vez que fazem parte da sintaxe da linguagem de descrição utilizada pelo **ARP**. Contudo, neste apêndice, serão utilizados os rótulos designados pelas figuras do capítulo 12. Por exemplo, a transição **t_d1I1I** da listagem II.1 será designada por **t_d1[1]** (o rótulo utilizado na figura 12.13).

Definida a rede, o programa **ARP** exige que o modelo seja compilado antes de sua simulação. Isto é feito através da ativação da opção **Compile**. Para o modelo **ARP** da listagem II.1, a compilação não detectou erros, conforme pode ser visto na figura II.10.

Após a compilação, a opção de simulação/análise mais completa é acionada através do menu **aNalysis** (tecla 'N'), a partir do qual pode-se fazer a análise de invariantes de lugar e de transição, além da enumeração de estados (geração do grafo de classes de estados, para modelos RPTM) – figura II.11. Escolhendo-se esta última opção (**State Enumeration**), a primeira ação do usuário corresponde à definição/alteração da marcação inicial. Na figura II.12, os lugares **d1**, **d2[2]**, **d3[1]**, **first[1]** e **t_NG** possuem, cada qual, um *token*.

Definida a marcação inicial, o programa **ARP** realiza a simulação e a análise de propriedades da rede. Inicialmente, é disponibilizada a análise de propriedades – figura II.13, a qual encontra-se transcrita na listagem II.2. Para compreender os termos utilizados na análise de propriedades, o **ARP** fornece uma tela de auxílio – figura II.14, acessada através da opção **Help**.

```

Net R12_13 ;

Nodes
  d1 : place (1) ;
  d2 : place ;
  d3 : place ;
  d1I1I : place ;
  d1I2I : place ;
  d1I3I : place ;
  d2I1I : place ;
  d2I2I : place (1) ;
  d2I3I : place ;
  d3I1I : place (1) ;
  d3I2I : place ;
  d3I3I : place ;
  d1ext : place ;
  d2ext : place ;
  d3ext : place ;
  firstI1I : place (1) ;
  firstI2I : place ;
  firstI3I : place ;
  t_G : place ;
  t_NG : place (1) ;
  t_TO : place ;
  t_TOSubnet : place ;
  EXEC : place (1) ;

  t_d1I1I : transition [5,7] ;
  t_d1I2I : transition [5,7] ;
  t_d1I3I : transition [5,7] ;
  t_d2I1I : transition [5,7] ;
  t_d2I2I : transition [5,7] ;
  t_d2I3I : transition [5,7] ;
  t_d3I1I : transition [5,7] ;
  t_d3I2I : transition [5,7] ;
  t_d3I3I : transition [5,7] ;
  t_SetG : transition ;
  t_ResetG : transition [10,10] ;
  t_RunTO : transition [0,0] ;
  t_End : transition [0,0] ;

Structure
  t_d1I1I : (d1I1I, EXEC, firstI1I, t_G) , (firstI2I, d1ext, d1, t_G, t_TO) ;
  t_d1I2I : (d1I2I, EXEC, firstI2I, t_G) , (firstI3I, d1ext, d1, t_G, t_TO) ;
  t_d1I3I : (d1I3I, EXEC, firstI3I, t_G) , (firstI1I, d1ext, d1, t_G, t_TO) ;
  t_d2I1I : (d2I1I, EXEC, firstI1I, t_G) , (firstI2I, d2ext, d2, t_G, t_TO) ;
  t_d2I2I : (d2I2I, EXEC, firstI2I, t_G) , (firstI3I, d2ext, d2, t_G, t_TO) ;
  t_d2I3I : (d2I3I, EXEC, firstI3I, t_G) , (firstI1I, d2ext, d2, t_G, t_TO) ;
  t_d3I1I : (d3I1I, EXEC, firstI1I, t_G) , (firstI2I, d3ext, d3, t_G, t_TO) ;
  t_d3I2I : (d3I2I, EXEC, firstI2I, t_G) , (firstI3I, d3ext, d3, t_G, t_TO) ;
  t_d3I3I : (d3I3I, EXEC, firstI3I, t_G) , (firstI1I, d3ext, d3, t_G, t_TO) ;
  t_End : (t_TOSubnet) , (EXEC) ;
  t_ResetG : (t_G) , (t_NG) ;
  t_RunTO : (t_TO) , (t_TOSubnet) ;
  t_SetG : (t_NG) , (t_G) ;

endNet.

```

Listagem II.1. Modelo ARP da figura II.5.



Figura II.10. Compilação da rede.



Figura II.11. Menu aNalysis.

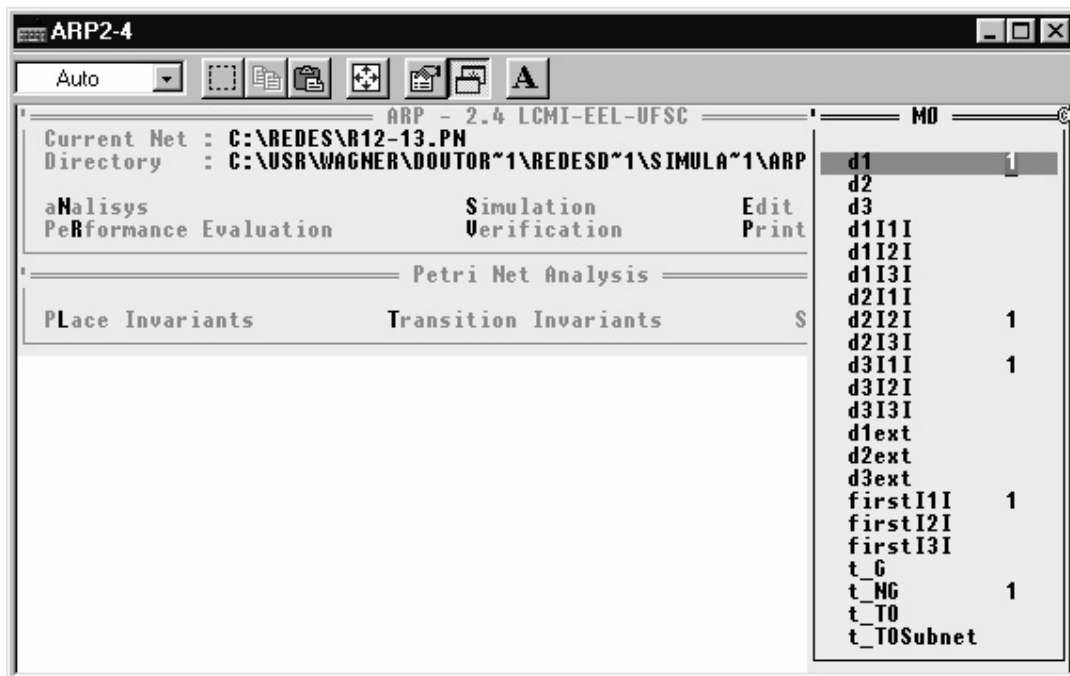


Figura II.12. Definição da marcação inicial M_0 .

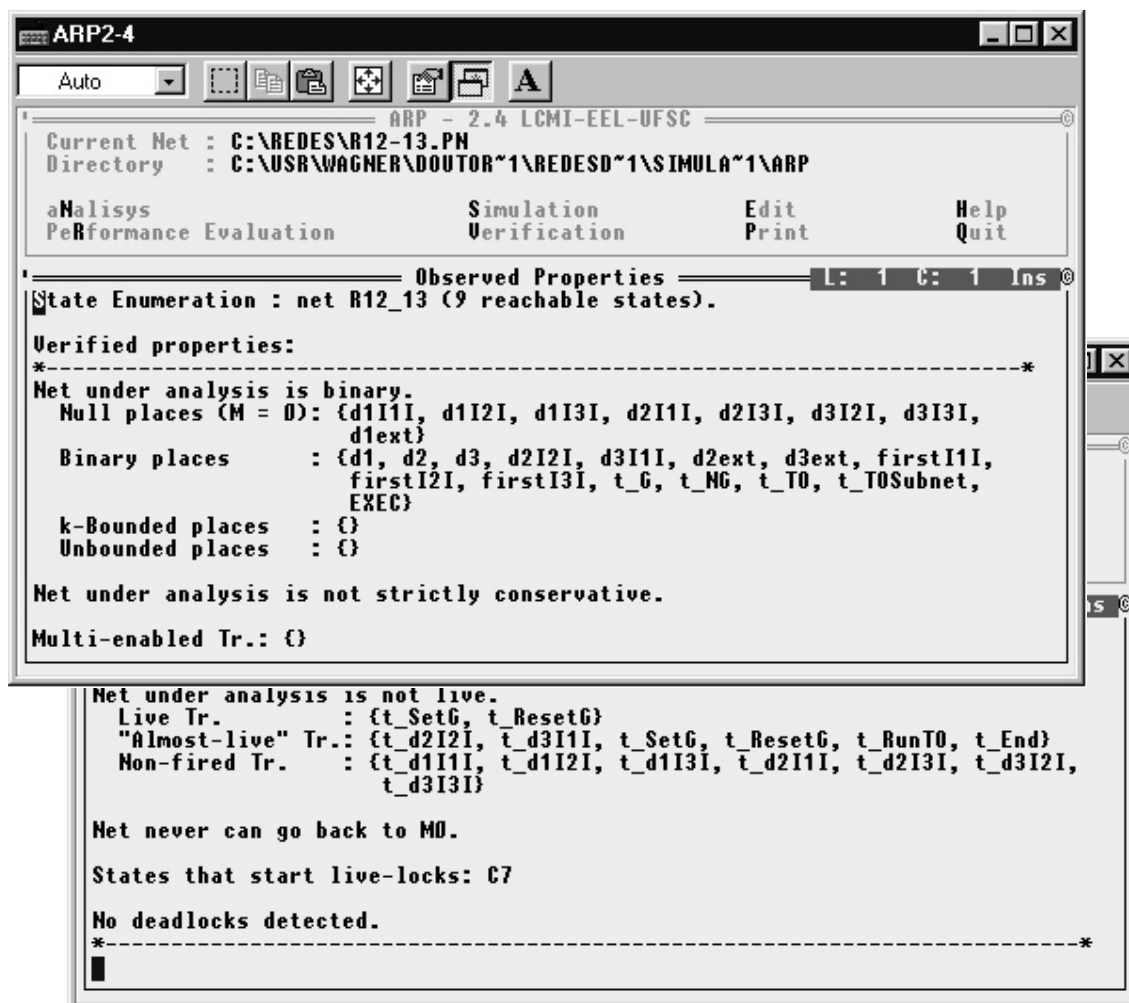


Figura II.13. Análise de propriedades para a rede R12-13.PN.

```

State Enumeration : net R12_13 (9 reachable states).

Verified properties:
*-----*
Net under analysis is binary.
  Null places (M = 0): {d1I1I, d1I2I, d1I3I, d2I1I, d2I3I, d3I2I, d3I3I,
                        d1ext}
  Binary places      : {d1, d2, d3, d2I2I, d3I1I, d2ext, d3ext, firstI1I,
                        firstI2I, firstI3I, t_G, t_NG, t_TO, t_TOSubnet,
                        EXEC}
  k-Bounded places   : {}
  Unbounded places   : {}

Net under analysis is not strictly conservative.

Multi-enabled Tr.: {}

Net under analysis is not live.
  Live Tr.           : {t_SetG, t_ResetG}
  "Almost-live" Tr.: {t_d2I2I, t_d3I1I, t_SetG, t_ResetG, t_RunTO, t_End}
  Non-fired Tr.      : {t_d1I1I, t_d1I2I, t_d1I3I, t_d2I1I, t_d2I3I, t_d3I2I,
                        t_d3I3I}

Net never can go back to M0.

States that start live-locks: C7

No deadlocks detected.
*-----*

```

Listagem II.2. Análise de propriedades: transcrição da figura II.13.

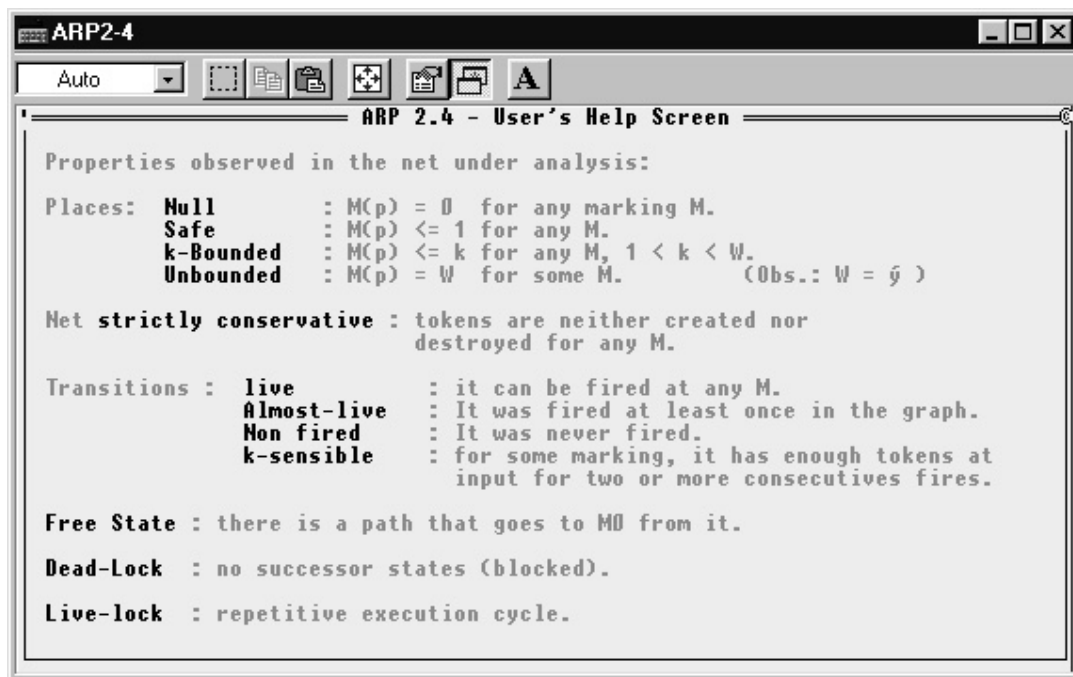


Figura II.14. Auxílio para compreensão dos termos utilizados.

As propriedades analisadas pelo **ARP** referem-se aos lugares e transições da rede, à própria rede e aos estados (marcações) gerados pela simulação – veja figura II.14. Considerando-se tais grupos de elementos, as propriedades analisadas são:

a) **Lugares.**

- **nulo:** um lugar p é nulo se $M(p) = 0$, para qualquer marcação M decorrente de M_0 (marcação inicial), onde $M(p)$ corresponde ao número de *tokens* em p ;
- **seguro:** um lugar p é seguro se $M(p) \leq 1$, para qualquer M decorrente de M_0 ;
- **k-limitado:** um lugar p é **k-limitado** se $M(p) \leq k$ (k finito), para qualquer marcação M decorrente de M_0 ;
- **ilimitado:** um lugar p é ilimitado se existir alguma marcação M , decorrente de M_0 , tal que $M(p) = W$, onde $W \rightarrow \infty$.

b) **Transições.**

- **viva:** uma transição t é **viva** se ela puder ser disparada em qualquer marcação M decorrente de M_0 ;
- **quase-viva:** uma transição t é **quase-viva** se ela for disparada ao menos uma vez em alguma marcação M decorrente de M_0 ;
- **não disparada:** quando uma transição t não é disparada em marcação alguma;
- **k-sensível:** uma transição t é **k-sensível** se existir uma marcação M , decorrente de M_0 , a partir da qual t possa ser disparada k vezes, consecutivamente.

c) **Rede.**

- **estritamente conservativa:** rede na qual a quantidade de *tokens* é fixa, considerando-se todos os lugares, em qualquer marcação M decorrente de M_0 .

d) **Estados (marcações).**

- **livre:** um estado (marcação) M , decorrente de M_0 , é livre se existir um caminho no grafo de estados (máquina de senhas) que conduza de M a M_0 , isto é, se a partir de M for possível retornar ao estado inicial da rede;
- **deadlock:** um estado M , decorrente de M_0 , está bloqueado (em *deadlock*) se ele não possuir estados (marcações) sucessores;
- **livelock:** um conjunto de estados $C_M = \{M_1, \dots, M_n\}$, decorrente de M_0 , encontra-se em *livelock* se os estados sucessores de qualquer estado M_i ($i = 1, \dots, n$) também pertencerem a C_M , configurando, desta forma, um ciclo repetitivo de execução.

Considerando-se as convenções adotadas para a construção da figura 12.13, a análise de propriedades gerada pelo programa **ARP** (listagem II.2) conduz às seguintes conclusões:

- i) A simulação produziu 9 estados: Tratando-se de um modelo RPTM, os estados são denominados **classes**, cada qual composta por uma **marcação** e um **domínio de tempo**, conforme visto no capítulo 8. Adiante, as classes geradas serão rotuladas de C_0 a C_8 ;
- ii) A rede é binária: Isto significa que o número máximo de *tokens* em qualquer lugar será 1, em qualquer marcação de qualquer estado decorrente de C_0 (estado inicial), o que é compatível com as convenções adotadas, as quais determinam que cada lugar da rede, em qualquer marcação, possuirá ou 0 ou 1 *token*;
- iii) Os lugares $d_1[1]$, $d_1[2]$, $d_1[3]$, $d_2[1]$, $d_2[3]$, $d_3[2]$, $d_3[3]$ e d_{1ext} são nulos: O fato de d_{1ext} permanecer nulo, ao longo da execução da rede, denota que o *token* d_1 não é retirado de P , pelo disparo de t , em momento algum da execução, o que está em conformidade com a RPSD da figura 12.13, uma vez que d_1 não está presente no *repositório* de P . Por outro lado, o fato dos demais lugares listados serem nulos, denota que o lugar P não recebe novos *tokens* (estes lugares estavam nulos na marcação inicial e assim permaneceram), o que também está em conformidade com a RPSD da figura 12.13, na qual não há transições capazes de inserir *tokens* em P ;

- iv) Os lugares **d₁**, **d₂**, **d₃**, **d₂[2]**, **d₃[1]**, **d₂ext**, **d₃ext**, **first[1]**, **first[2]**, **first[3]**, **t_G**, **t_{NG}**, **t_{To}**, **t_{ToSubnet}** e **EXEC** são binários: Isto significa que, em algum momento da simulação (isto é, em alguma marcação), estes lugares chegam a possuir 1 *token*. Na marcação inicial, presente na figura 12.13, os lugares **d₁**, **d₂[2]**, **d₃[1]**, **first[1]**, **t_G** e **EXEC** possuem 1 *token* cada (obs.: o lugar **next[3]** também possui 1 *token*, contudo, por estar desconectado, não foi representado nos modelos **JARP** e **ARP**). Assim, para serem qualificados como binários, os demais lugares da lista recebem 1 *token* em algum momento da execução, conforme pode ser constatado adiante, na descrição do grafo de classes da rede;
- v) A rede não é estritamente conservativa: A rede seria estritamente conservativa caso fossem desconsiderados os lugares **d₁ext**, **d₂ext** e **d₃ext**, uma vez que os demais lugares formam 6 grupos, cada qual contendo apenas 1 *token* (grupos {**d₁**, **d₁[1]**, **d₁[2]**, **d₁[3]**}; {**d₂**, **d₂[1]**, **d₂[2]**, **d₂[3]**}; {**d₃**, **d₃[1]**, **d₃[2]**, **d₃[3]**}; {**first[1]**, **first[2]**, **first[3]**}; {**EXEC**, **t_{To}**, **t_{ToSubnet}**}; e {**t_G**, **t_{NG}**});
- vi) Não há transições **k**-sensíveis: Desta forma, uma mesma transição da RPTM não pode ser disparada mais de uma vez seguida, o que é compatível com a figura 12.13. Apesar da transição **t** da RPSD poder ser disparada duas vezes consecutivas (há dois *tokens* em **P**), as transições disparadas na RPTM, representantes de **t**, serão distintas, devido ao fato de se remover *tokens* distintos de **P** (**d₃** no primeiro disparo e **d₂** no segundo);
- vii) A rede não é viva: Esta análise, em conjunto com as três seguintes, reflete o fato da transição **t** da RPSD poder ser disparada somente duas vezes, após o que somente as transições referentes à guarda de **t** (**t_{ResetG}** e **t_{SetG}**) poderão ser disparadas, alternadamente (obs.: uma rede é viva somente quando todas as suas transições são vivas);
- viii) As transições **t_{SetG}** e **t_{ResetG}** são vivas;
- ix) As transições **t_{d₂[2]}**, **t_{d₃[1]}**, **t_{SetG}**, **t_{ResetG}**, **t_{RunTo}** e **t_{End}** são quase-vivas: O disparo de **t_{d₃[1]}** representa a remoção do *token* **d₃** de **P**, enquanto o disparo de **t_{d₂[2]}** representa a remoção do *token* **d₂**. Ambos disparos são acompanhados pelo disparo consecutivo das transições **t_{RunTo}** e **t_{End}**. Como as transições **t_{SetG}** e **t_{ResetG}** são vivas, elas são também quase-vivas;
- x) As transições **t_{d₁[1]}**, **t_{d₁[2]}**, **t_{d₁[3]}**, **t_{d₂[1]}**, **t_{d₂[3]}**, **t_{d₃[2]}** e **t_{d₃[3]}** não disparam durante a simulação: Este fato se deve à configuração do *repositório* de **P**;
- xi) A rede não retorna ao estado original **C₀**: Esta análise está em conformidade com a figura 12.13, uma vez que não existe, na RPSD dessa figura, uma transição que restitua a **P** os *tokens* removidos por **t** (**d₃** e **d₂**);
- xii) O estado **C₇** inicia a execução repetitiva (*livelock*): Conforme visto adiante, na construção do grafo de classes, ao atingir o estado (classe) **C₇**, a rede ficará em *loop* (execução cíclica), alternando entre os estados **C₇** e **C₈** (devido ao disparo alternado das transições **t_{ResetG}** e **t_{SetG}**); e
- xiii) A rede não entra em *deadlock*: Atendo-se somente à figura 12.13, a rede deveria ficar bloqueada após a remoção dos dois *tokens* de **P**. Contudo, tal bloqueio é removido ao acrescentar-se a transição **t_{ResetG}**, no modelo **JARP** relacionado à figura 12.13 (veja a figura II.5). A transição **t_{ResetG}** possibilita a existência do *loop* descrito no item anterior.

Após a análise de propriedades, o programa **ARP** disponibiliza um menu que permite a visualização dos **estados gerados** (denominados aqui de **classes geradas**) e do **grafo de alcançabilidade** (denominado de **grafo de classes**) do modelo simulado. O grafo de classes encontra-se na figura II.15, enquanto as classes geradas estão na figura II.16. As informações de ambas figuras são agregadas na figura II.17.

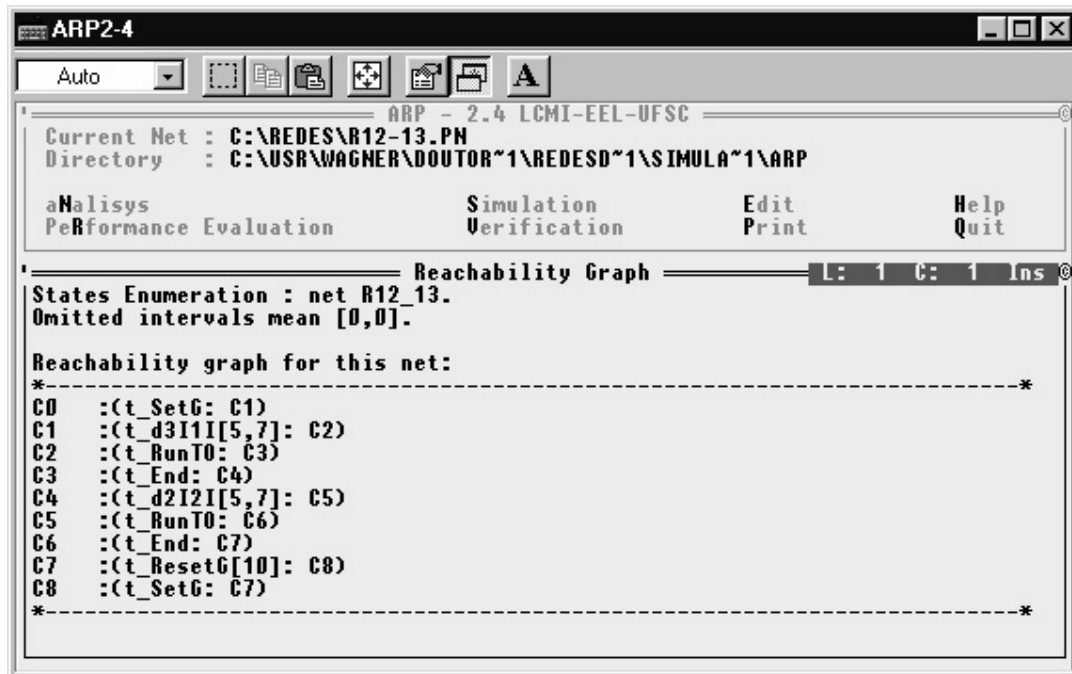


Figura II.15. Informações para a construção do grafo de classes.

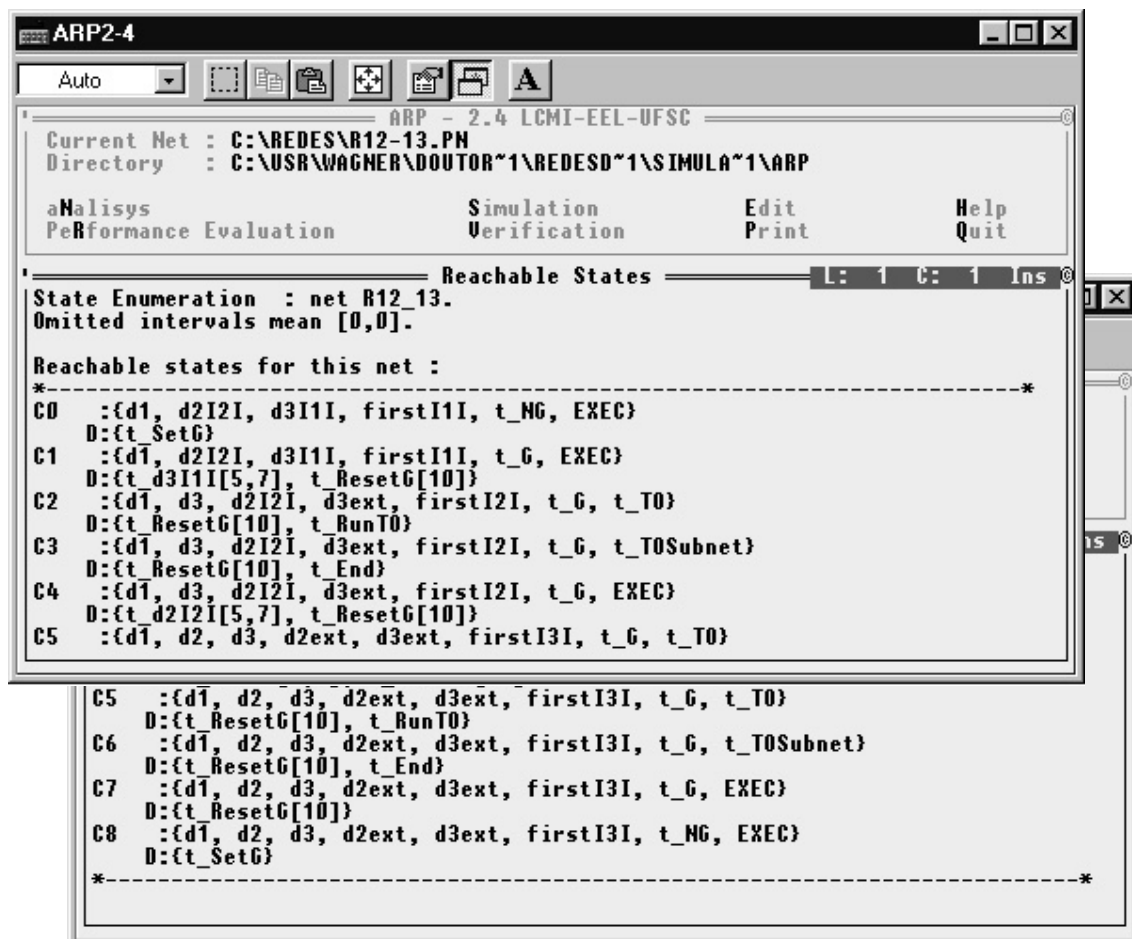


Figura II.16. Classes geradas.

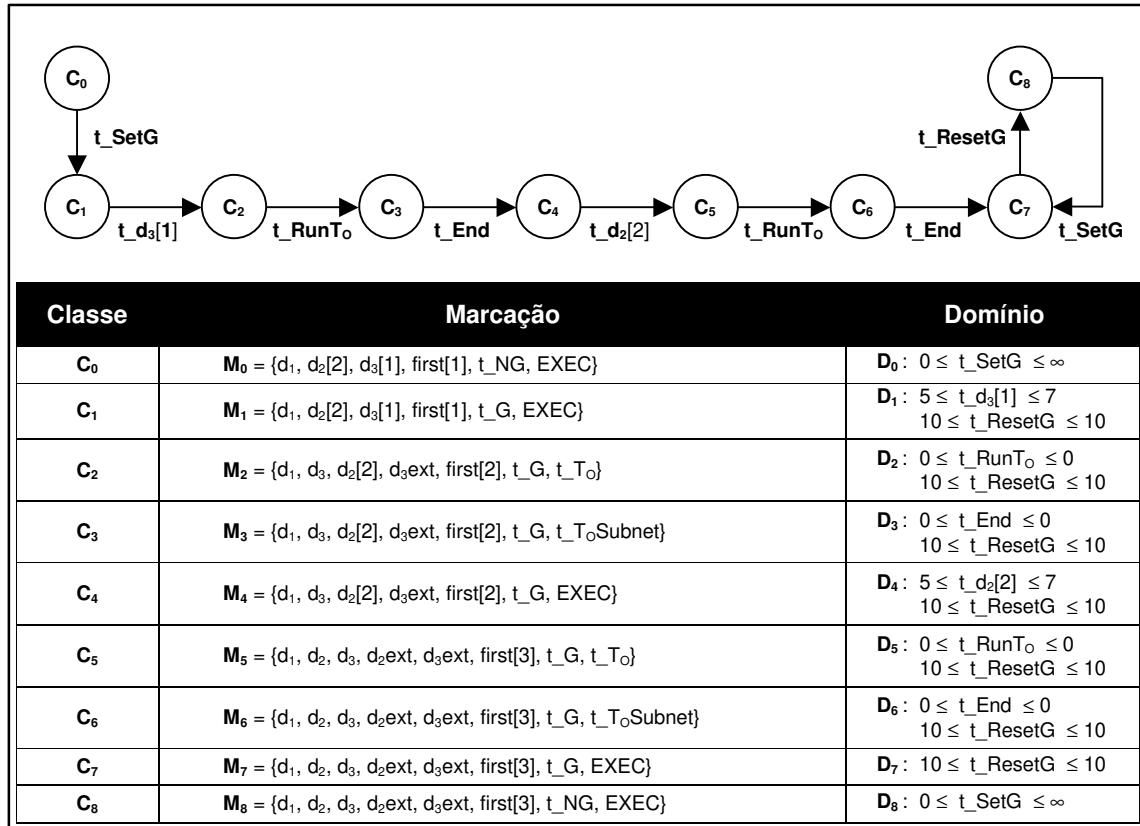


Figura II.17. Grafo de classes para a rede das figuras 12.13 / II.5.

Para cada classe da figura II.16, foi definido um nó na figura II.17. O relacionamento entre os nós é definido pelas informações da figura II.15. Por exemplo, a linha

C₁ : (t_d3I1I[5, 7] : C₂)

significa que a transição **t_d3[1]** está habilitada na classe **C₁**, tem o intervalo de disparo [5,7] e, ao disparar, conduzirá a rede da classe **C₁** à classe **C₂**. Observe que o intervalo [10] na linha

C₇ : (t_ResetG[10] : C₈)

é uma abreviação do intervalo [10,10]. Por outro lado, o **ARP** informa que intervalos omitidos equivalem ao intervalo [0,0]. Esta afirmação é válida somente para as linhas encabeçadas por **c2, c3, c5 e c6**. Para as linhas encabeçadas por **c0 e c8**, a omissão corresponde ao intervalo [0,∞], uma vez que não foi definido intervalo de disparo para a transição **t_SetG** (conforme visto no capítulo 8, a omissão de um intervalo de disparo equivale ao intervalo [0,∞]). Em casos assim, para poder fazer a simulação, o **ARP** força a adoção do intervalo [0,0], desde que outras transições, com intervalo definido, não estejam habilitadas. Finalmente, observa-se, na figura II.17, a adoção do identificador da transição disparada como o rótulo do ramo que conduz de uma classe a outra.

O grafo de classes da rede da figura 12.13, ilustrado na figura II.17, apresenta natureza linear, possuindo apenas um ciclo (formado pelas classes **C₇** e **C₈**). Este fato demonstra que a rede passa por cada estado apenas uma vez, excetuando-se a execução repetitiva representada pela alternância de disparos envolvendo o valor da guarda da transição, a qual faz parte da **memória** da RPSD. Acompanhando a passagem de uma classe para outra, desde **C₀**, tem-se a seguinte sequência de disparo de transições:

- i) **t_SetG**: representando a atribuição de *verdadeiro* à guarda de **t**;
- ii) **t_d3[1]**: representando a remoção do *token d3* da posição 1 do *repositório* de **P** (obs.: esta posição é relativa ao lugar **first** marcado – na classe **C1**, como o lugar **first** marcado é **first[1]**, a presença de um *token d3[1]* determina que o *token d3* ocupa a primeira posição do *repositório*);
- iii) **t_RunT0**: representando o início da execução das *operações* associadas à transição **t**;
- iv) **t_End**: representando o término da execução das *operações* associadas à transição **t**;
- v) **t_d2[2]**: representando a remoção do *token d2* da posição 1 do *repositório* de **P** (obs.: a posição é relativa ao lugar **first** marcado – na classe **C4**, o lugar **first** marcado é **first[2]**, cujo índice é igual ao da transição a disparar, isto é, **t_d3[2]**);
- vi) **t_RunT0**: vide item **iii**);
- vii) **t_End**: vide item **iv**);
- viii) **t_ResetG**: representando a atribuição de *falso* à guarda de **t**; e
- ix) **t_SetG / t_ResetG**: representando a alternância de valores à guarda de **t** (devido à inexistência de outras ações possíveis de ocorrer).

Pelo exposto, os itens de **ii**) a **iv**) representam o primeiro disparo de **t**, responsável pela remoção do *token d3* de **P**. Do mesmo modo, os itens de **v**) a **vii**) representam o segundo disparo de **t**, responsável pela remoção do *token d2* de **P**. Observe que, caso o *token d1* estivesse presente em **P**, a sua remoção ocorreria da mesma forma que a remoção dos *tokens d3* e *d2*. Além disso, observe que a representação do disparo de **t** constitui-se numa operação atômica. Isto se deve ao fato de que, uma vez *verdadeira* a guarda de **t**, uma, e somente uma, das transições representantes de **t** (iniciadas pelo rótulo **t_**) irá disparar (respeitando o intervalo de disparo de **t**, isto é, se neste ínterim a guarda de **t** passar para *falso*, a transição **t_** habilitada será desabilitada); após o disparo de **t_**, segue-se a sequência de disparo **t_RunT0** e **t_End**, formada por transições que não sofrem a interferência do valor da guarda de **t**; por último, não é possível um disparo sobreposto de outra transição **t_**, enquanto o lugar **EXEC** estiver vazio (**EXEC** só receberá um *token* após a finalização do disparo em curso, isto é, após o disparo de **t_End**).

Assim, de acordo com as considerações anteriores, a RPTM da figura 12.13 apresenta funcionamento compatível com as convenções e definições adotadas em sua concepção, validando, conseqüentemente, o modelo RPSD à qual é equivalente (presente na figura 12.13.a).

Na sequência, são analisadas as representações RPTM presentes nas figuras 12.14 a 12.19, através de modelos **JARP** e **ARP**. As redes das figuras 12.17 e 12.19 são tratadas através da **simulação interativa** do **JARP**, enquanto as demais são tratadas através de análise semelhante a feita para a rede 12.13.

A figura II.18 mostra o modelo **JARP** relacionado à figura 12.14, o qual se assemelha ao modelo **JARP** da figura 12.13 (figura II.5). Tal como foi feito para a listagem **R12-13.PN**, faz-se necessário o acréscimo de intervalos de disparo à listagem **R12-14.PN**, originalmente produzida pelo **JARP**, para tratamento pelo **ARP**. Outro acréscimo refere-se ao lugar **t_NG** e às transições **t_SetG** e **t_ResetG**, necessários à simulação da *guarda* de **t** (este acréscimo é feito em todos os modelos **JARP** deste apêndice). Após tais acréscimos, o modelo **ARP** correspondente à figura 12.14 apresentará a descrição dada na listagem II.3.

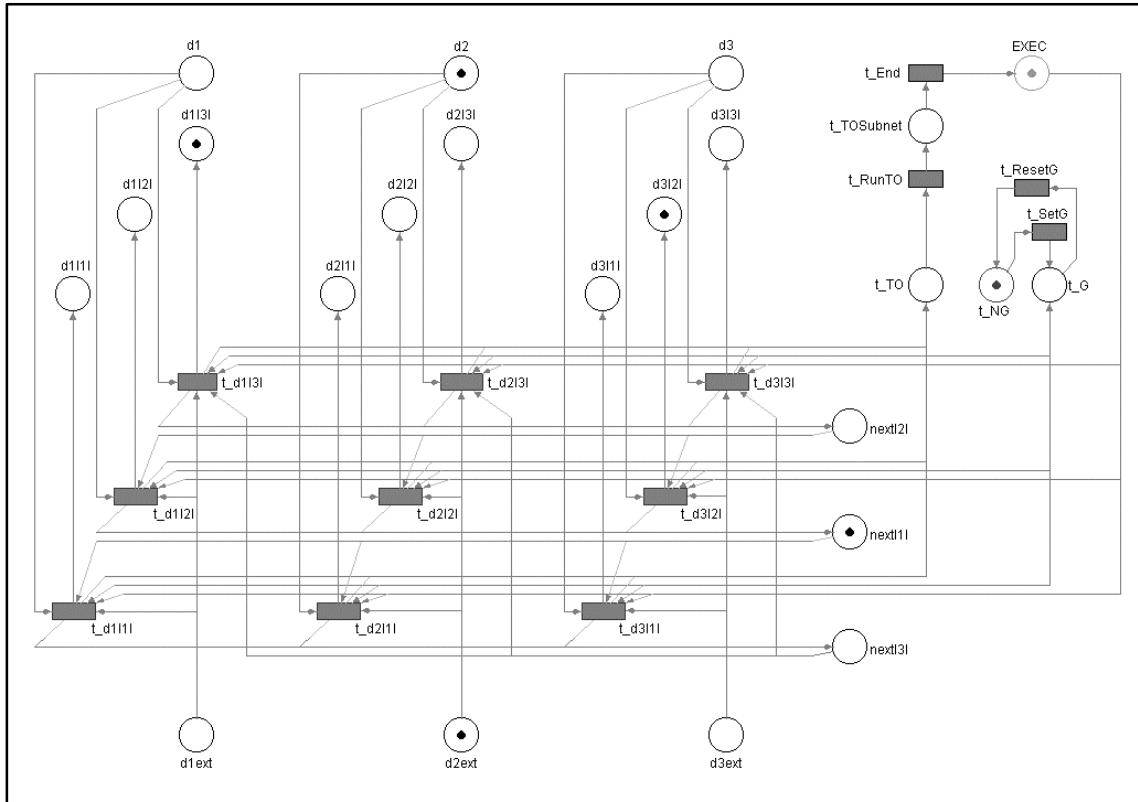


Figura II.18. Modelo JARP correspondente à RPTM da figura 12.14.

Após as etapas de compilação e definição de marcação inicial, a simulação e a análise de propriedades da rede, realizada pelo **ARP**, resultou na análise de propriedades transcrita na listagem II.4, assim como em informações que permitiram a construção do grafo de classes ilustrado na figura II.19.

Levando-se em conta as convenções adotadas na construção da figura 12.14, a análise de propriedades gerada pelo programa **ARP** (listagem II.4) conduz às seguintes conclusões:

- i) A simulação produziu 6 estados (classes): Adiante, as classes serão rotuladas de **C₀** a **C₅**;
- ii) A rede é binária: Isto significa que o número máximo de *tokens* em qualquer lugar será 1, em qualquer marcação de qualquer estado decorrente de **C₀** (estado inicial), o que é compatível com as convenções adotadas, as quais determinam que cada lugar da rede, em qualquer marcação, possuirá ou 0 ou 1 *token*;
- iii) Os lugares **d₁**, **d₃**, **d₁[1]**, **d₁[2]**, **d₂[2]**, **d₂[3]**, **d₃[1]**, **d₃[3]**, **d₁ext**, **d₃ext** e **next[2]** são nulos: O fato de **d₁ext** e **d₃ext** permanecerem nulos, ao longo da execução da rede, denota que não houve a disponibilização dos *tokens* **d₁** e **d₃** para inserção em **P**, pelo disparo de **t**, em momento algum da execução. Mesmo que houvesse tal disponibilização (isto foi testado alterando-se a marcação inicial dos lugares **d₁ext** e **d₃ext**), nenhuma das

transições responsáveis pela inserção de tais *tokens* (transições $t_{d_1[1]}$, $t_{d_1[2]}$ e $t_{d_1[3]}$, para inserção de d_1 , e $t_{d_3[1]}$, $t_{d_3[2]}$ e $t_{d_3[3]}$, para inserção de d_3) ficaria habilitada (veja item **x**)), uma vez que os lugares d_1 e d_3 permaneceram nulos ao longo da simulação – esta impossibilidade de disparo está de acordo com a RPSD da figura 12.14, pois os *tokens* d_1 e d_3 já se encontram presentes no *repositório* de **P**, inviabilizando, desta forma, a inserção de novas cópias dos mesmos. O fato dos lugares $d_1[1]$, $d_1[2]$, $d_3[1]$ e $d_3[3]$ permanecerem nulos, deve-se ao não disparo das transições capazes de inserir os *tokens* d_1 e d_3 em **P**. Em complemento, os lugares $d_2[2]$ e $d_2[3]$ permanecem nulos devido à transição disparada, para inserção do *token* d_2 em **P**, ser $t_{d_2[1]}$ (esta transição é selecionada por causa da presença de um *token* em $next[1]$). Como não há retirada de *tokens* de **P**, os lugares d_1 e d_3 permanecem nulos. Finalmente, o lugar $next[2]$ permanece nulo devido à possibilidade de uma única inserção no lugar **P** da RPSD da figura 12.14 (o disparo de $t_{d_2[1]}$ promove a transferência do *token* presente em $next[1]$ para $next[3]$ – somente o disparo de $t_{d_1[3]}$, $t_{d_2[3]}$ ou $t_{d_3[3]}$ seria capaz de transferir o *token* presente em $next[3]$ para $next[2]$);

- iv) Os lugares d_2 , $d_1[3]$, $d_2[1]$, $d_3[2]$, d_2ext , $next[1]$, $next[3]$, t_G , t_{NG} , t_{T_0} , $t_{T_0Subnet}$ e **EXEC** são binários: Na marcação inicial, presente na figura 12.14, os lugares d_2 , $d_1[3]$, $d_3[2]$, d_2ext , $next[1]$, t_G e **EXEC** possuem 1 *token* cada (obs.: o lugar $first[2]$ também possui 1 *token*, contudo, por estar desconectado, não foi representado nos modelos **JARP** e **ARP**). Assim, para serem qualificados como binários, os demais lugares da lista recebem 1 *token* em algum momento da execução, conforme pode ser visto adiante, na descrição do grafo de classes da rede;
- v) A rede não é estritamente conservativa: A rede seria estritamente conservativa caso fossem desconsiderados os lugares d_1ext , d_2ext e d_3ext , uma vez que os demais lugares formam 6 grupos, cada qual contendo apenas 1 *token* (grupos $\{d_1, d_1[1], d_1[2], d_1[3]\}$; $\{d_2, d_2[1], d_2[2], d_2[3]\}$; $\{d_3, d_3[1], d_3[2], d_3[3]\}$; $\{next[1], next[2], next[3]\}$; $\{EXEC, t_{T_0}, t_{T_0Subnet}\}$; e $\{t_G, t_{NG}\}$);
- vi) Não há transições **k**-sensíveis: Isto significa que uma mesma transição da RPTM não pode ser disparada mais de uma vez consecutiva, o que é compatível com a figura 12.14;
- vii) A rede não é viva: Esta análise, em conjunto com as três seguintes, reflete o fato da transição **t** da RPSD poder ser disparada somente uma vez, após o que somente as transições referentes à guarda de **t** (t_{ResetG} e t_{SetG}) poderão ser disparadas, alternadamente;
- viii) As transições t_{SetG} e t_{ResetG} são vivas;
- ix) As transições $t_{d_2[1]}$, t_{SetG} , t_{ResetG} , t_{RunT_0} e t_{End} são quase-vivas: O disparo de $t_{d_2[1]}$ representa a inserção do *token* d_2 em **P**. Este disparo é acompanhado pelo disparo consecutivo das transições t_{RunT_0} e t_{End} . Como as transições t_{SetG} e t_{ResetG} são vivas, elas são também quase-vivas;
- x) As transições $t_{d_1[1]}$, $t_{d_1[2]}$, $t_{d_1[3]}$, $t_{d_2[2]}$, $t_{d_2[3]}$, $t_{d_3[1]}$, $t_{d_3[2]}$ e $t_{d_3[3]}$ não dispararam durante a simulação: Este fato se deve à configuração do *repositório* de **P**;
- xi) A rede não retorna ao estado original C_0 : Esta análise está em conformidade com a RPSD da figura 12.14, na qual inexistem transições capazes de restaurar **P** ao estado C_0 ;
- xii) O estado C_4 inicia a execução repetitiva (*livelock*): Conforme visto adiante, no grafo de classes, ao atingir o estado C_4 , a rede ficará em *loop*, alternando entre os estados C_4 e C_5 (disparo alternado das transições t_{ResetG} e t_{SetG}); e
- xiii) A rede não entra em *deadlock*: Atendo-se somente à figura 12.14, a rede deveria ficar bloqueada após a inserção do *token* d_2 em **P**. Entretanto, este bloqueio é removido com o acréscimo da transição t_{ResetG} (ao modelo **JARP** correspondente à figura 12.14 – veja a figura II.18), a qual possibilita a existência do *loop* descrito no item anterior.

```

Net R12_14 ;

Nodes
  d1 : place ;
  d2 : place (1) ;
  d3 : place ;
  d1I1I : place ;
  d1I2I : place ;
  d1I3I : place (1) ;
  d2I1I : place ;
  d2I2I : place ;
  d2I3I : place ;
  d3I1I : place ;
  d3I2I : place (1) ;
  d3I3I : place ;
  dlext : place ;
  d2ext : place (1) ;
  d3ext : place ;
  nextI1I : place (1) ;
  nextI2I : place ;
  nextI3I : place ;
  t_G : place ;
  t_NG : place (1) ;
  t_TO : place ;
  t_TOSubnet : place ;
  EXEC : place (1) ;

  t_d1I1I : transition [5,7] ;
  t_d1I2I : transition [5,7] ;
  t_d1I3I : transition [5,7] ;
  t_d2I1I : transition [5,7] ;
  t_d2I2I : transition [5,7] ;
  t_d2I3I : transition [5,7] ;
  t_d3I1I : transition [5,7] ;
  t_d3I2I : transition [5,7] ;
  t_d3I3I : transition [5,7] ;
  t_SetG : transition ;
  t_ResetG : transition [10,10] ;
  t_RunTO : transition [0,0] ;
  t_End : transition [0,0] ;

Structure
  t_d1I1I : (d1, dlext, EXEC, nextI1I, t_G) , (d1I1I, nextI3I, t_G, t_TO) ;
  t_d1I2I : (d1, dlext, EXEC, nextI2I, t_G) , (d1I2I, nextI1I, t_G, t_TO) ;
  t_d1I3I : (d1, dlext, EXEC, nextI3I, t_G) , (d1I3I, nextI2I, t_G, t_TO) ;
  t_d2I1I : (d2, d2ext, EXEC, nextI1I, t_G) , (d2I1I, nextI3I, t_G, t_TO) ;
  t_d2I2I : (d2, d2ext, EXEC, nextI2I, t_G) , (d2I2I, nextI1I, t_G, t_TO) ;
  t_d2I3I : (d2, d2ext, EXEC, nextI3I, t_G) , (d2I3I, nextI2I, t_G, t_TO) ;
  t_d3I1I : (d3, d3ext, EXEC, nextI1I, t_G) , (d3I1I, nextI3I, t_G, t_TO) ;
  t_d3I2I : (d3, d3ext, EXEC, nextI2I, t_G) , (d3I2I, nextI1I, t_G, t_TO) ;
  t_d3I3I : (d3, d3ext, EXEC, nextI3I, t_G) , (d3I3I, nextI2I, t_G, t_TO) ;
  t_End : (t_TOSubnet) , (EXEC) ;
  t_ResetG : (t_G) , (t_NG) ;
  t_RunTO : (t_TO) , (t_TOSubnet) ;
  t_SetG : (t_NG) , (t_G) ;

endNet.

```

Listagem II.3. Modelo ARP da figura II.18.

```

State Enumeration : net R12_14 (6 reachable states).

Verified properties:
*-----*
Net under analysis is binary.
Null places (M = 0): {d1, d3, d1I1I, d1I2I, d2I2I, d2I3I, d3I1I, d3I3I,
d1ext, d3ext, nextI2I}
Binary places      : {d2, d1I3I, d2I1I, d3I2I, d2ext, nextI1I, nextI3I,
t_G, t_NG, t_TO, t_TOSubnet, EXEC}
k-Bounded places  : {}
Unbounded places  : {}

Net under analysis is not strictly conservative.

Multi-enabled Tr.: {}

Net under analysis is not live.
Live Tr.          : {t_SetG, t_ResetG}
"Almost-live" Tr.: {t_d2I1I, t_SetG, t_ResetG, t_RunTO, t_End}
Non-fired Tr.     : {t_d1I1I, t_d1I2I, t_d1I3I, t_d2I2I, t_d2I3I, t_d3I1I,
t_d3I2I, t_d3I3I}

Net never can go back to M0.

States that start live-locks: C4

No deadlocks detected.
*-----*

```

Listagem II.4. Análise de propriedades para a rede da figura II.18.

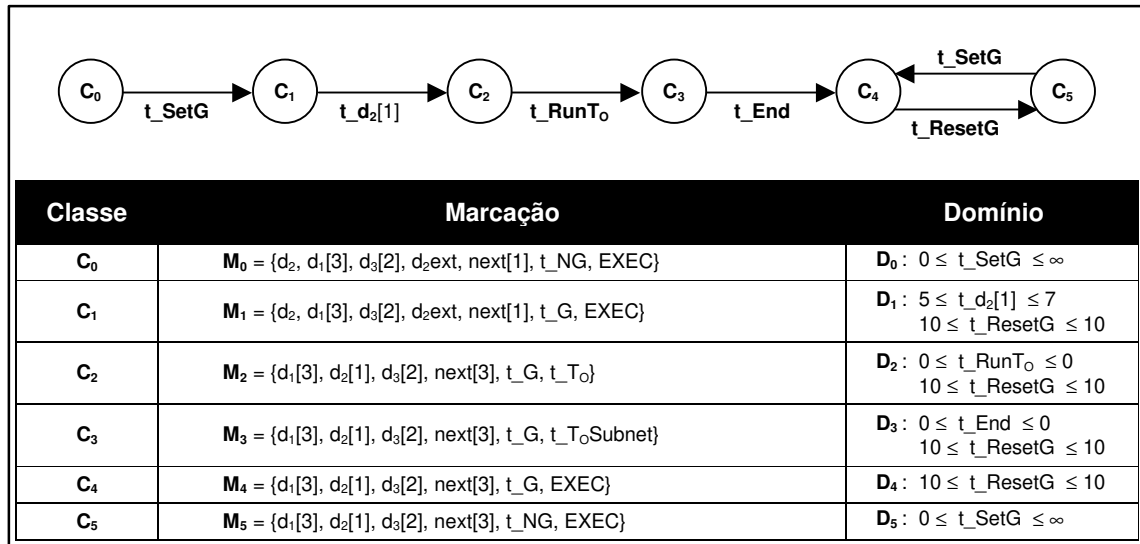


Figura II.19. Grafo de classes para a rede das figuras 12.14 / II.18.

Assim como o grafo de classes do modelo anterior, o grafo correspondente à figura 12.14, mostrado na figura II.19, apresenta natureza linear, possuindo apenas um ciclo (constituído por C_4 e C_5). Desde C_0 , tem-se a seguinte sequência de disparo de transições:

- i) **t_SetG**: representando a atribuição de *verdadeiro* à guarda de **t**;
- ii) **t_d₂[1]**: representando a inserção do *token* **d₂** na posição 3 do *repositório* de **P** (obs.: a fileira de lugares RPTM na qual o *token* será inserido dependerá do lugar **next** marcado – na classe C_1 , como o lugar **next** marcado é **next[1]**, a inserção do *token* **d₂** em **P** será representada pela inserção de um *token* em **d₂[1]**). Tal como ocorre com a rede da figura 12.13, a ordem entre as fileiras de lugares RPTM, representantes do *repositório* de **P**, é determinada pelos lugares **first** (veja as convenções de representação de lugares do tipo I, na seção 12.2.1);
- iii) **t_RunT₀**: representando o início da execução das *operações* associadas à transição **t**;
- iv) **t_End**: representando o término da execução das *operações* associadas à transição **t**;
- v) **t_ResetG**: representando a atribuição de *falso* à guarda de **t**; e
- vi) **t_SetG / t_ResetG**: representando a alternância de valores à guarda de **t** (devido à inexistência de outras ações possíveis de ocorrer).

Os itens de ii) a iv) representam o disparo de **t**, responsável pela inserção do *token* **d₂** em **P**. Observe que, caso os *tokens* **d₁** e **d₃** não estivessem presentes em **P** e, além disso, estivessem disponíveis para inserção em **P** (o que seria representado pela presença de um *token* em ambos os lugares **d₁ext** e **d₃ext**), a sua inserção ocorreria da mesma forma que a inserção do *token* **d₂**. Além disso, a representação RPTM do disparo de **t** constitui-se numa operação atômica. Isto se deve ao fato de que, uma vez *verdadeira* a guarda de **t**, havendo uma ou mais transições representantes de **t** habilitadas (iniciadas pelo rótulo **t_**), somente uma delas irá disparar (respeitando o intervalo de disparo de **t**, isto é, se neste íterim a guarda de **t** passar para *falso*, toda transição **t_** habilitada será desabilitada). A habilitação de uma transição **t_d_i[j]** ($i, j = 1, 2, 3$) dependerá dos lugares **d_iext**, **d_i**, **d_i[j]**, **EXEC** e **t_G** estarem marcados. Assim, mais de uma transição **t_** poderá ficar habilitada em um dado momento, mas somente uma delas irá disparar, pois todas compartilham o lugar **EXEC**. Após o disparo de **t_**, segue-se a sequência de disparo **t_RunT₀** e **t_End**, formada por transições que não sofrem a interferência do valor da guarda de **t**.

Outras marcações iniciais foram testadas para a rede analisada, gerando resultados análogos aos apresentados. Assim, pelo exposto, a RPTM presente na figura 12.14 é válida, uma vez que apresenta funcionamento compatível com as convenções e definições utilizadas em sua construção. Conseqüentemente, o modelo RPSD à qual é equivalente, presente na figura 12.14.a, também é válido.

A figura II.20 mostra o modelo **JARP** relacionado à figura 12.15. Comparando-se ambas as figuras, observa-se, na figura II.20, o acréscimo, além do lugar **t_NG** e das transições **t_SetG** e **t_ResetG**, de transições e lugares com rótulo iniciado por **Empty_**, os quais servem para simular o consumo dos *tokens* representados pelos *contêineres* **t₁** e **t₂**, impedindo, desta forma, que o grafo de classes do modelo seja ilimitado (este acréscimo não seria necessário, caso a RPSD da figura 12.15 apresentasse um lugar conectado à transição **t** por um *ramo de saída*). O uso do peso 2, nos ramos que conectam a transição **t_End** ao lugar **Empty_ext** e o lugar **Empty_ext_OK** à transição **Empty_OK**, justifica-se pela presença de dois resultados externos (presentes nos lugares representantes de **t₁** e **t₂**, cujos rótulos terminam com **ext**).


```

Net R12_15 ;

Nodes
    dfltI1I : place ;                // dflt[1]
    dfltI2I : place (1) ;
    dfltI3I : place ;
    dfltInI : place (1) ;
    d1I1I : place ;                  // d1[1]
    d1I2I : place ;
    d1I3I : place (1) ;
    d1InI : place ;
    d2I1I : place (1) ;
    d2I2I : place ;
    d2I3I : place ;
    d2InI : place ;
    dkI1I : place ;
    dkI2I : place ;
    dkI3I : place ;
    dkInI : place ;
    expr1_a0 : place (1) ;           // expr1=a0
    expr1_a1 : place ;
    expr1_a2 : place ;
    expr1_anminus1 : place ;         // expr1=an-1
    expr2_a0 : place ;
    expr2_a1 : place ;
    expr2_a2 : place (1) ;
    expr2_anminus1 : place ;
    rw : place (1) ;                 // r/w
    t1_read : place ;                // t1_read
    t2_read : place ;
    t1_dfltext : place ;             // t1_dfltext
    t1_d1ext : place ;               // t1_d1ext
    t1_d2ext : place ;
    t1_dkext : place ;
    t2_dfltext : place ;
    t2_d1ext : place ;
    t2_d2ext : place ;
    t2_dkext : place ;
    t_NG : place (1) ;
    t_G : place ;
    t_TO : place ;                   // tT0
    t_TOSubnet : place ;             // tT0Subnet
    EXEC : place (1) ;
    Empty_ext : place ;
    Empty_ext_OK : place ;

    t1dfltI1I : transition [2,2] ;   // t1_dflt[1]
    t1dfltI2I : transition [2,2] ;
    t1dfltI3I : transition [2,2] ;
    t1dfltInI : transition [2,2] ;
    t1d1I1I : transition [2,2] ;     // t1_d1[1]
    t1d1I2I : transition [2,2] ;
    t1d1I3I : transition [2,2] ;
    t1d1InI : transition [2,2] ;
    t1d2I1I : transition [2,2] ;
    t1d2I2I : transition [2,2] ;
    t1d2I3I : transition [2,2] ;
    t1d2InI : transition [2,2] ;
    t1dkI1I : transition [2,2] ;
    t1dkI2I : transition [2,2] ;
    t1dkI3I : transition [2,2] ;
    t1dkInI : transition [2,2] ;
    t2dfltI1I : transition [2,2] ;
    t2dfltI2I : transition [2,2] ;

```

Listagem II.5. Modelo ARP da figura II.20.

```

t2dfltI3I : transition [2,2] ;
t2dfltInI : transition [2,2] ;
t2d1I1I : transition [2,2] ;
t2d1I2I : transition [2,2] ;
t2d1I3I : transition [2,2] ;
t2d1InI : transition [2,2] ;
t2d2I1I : transition [2,2] ;
t2d2I2I : transition [2,2] ;
t2d2I3I : transition [2,2] ;
t2d2InI : transition [2,2] ;
t2dkI1I : transition [2,2] ;
t2dkI2I : transition [2,2] ;
t2dkI3I : transition [2,2] ;
t2dkInI : transition [2,2] ;
t_0_0 : transition [5,7] ; // t_a0a0
t_0_1 : transition [5,7] ;
t_0_2 : transition [5,7] ;
t_0_nminus1 : transition [5,7] ; // t_a0an-1
t_1_0 : transition [5,7] ;
t_1_1 : transition [5,7] ;
t_1_2 : transition [5,7] ;
t_1_nminus1 : transition [5,7] ;
t_2_0 : transition [5,7] ;
t_2_1 : transition [5,7] ;
t_2_2 : transition [5,7] ;
t_2_nminus1 : transition [5,7] ;
t_nminus1_0 : transition [5,7] ; // t_an-1a0
t_nminus1_1 : transition [5,7] ;
t_nminus1_2 : transition [5,7] ;
t_nminus1_nminus1 : transition [5,7] ; // t_an-1an-1
t_SetG : transition ;
t_ResetG : transition [10,10] ;
t_RunT0 : transition [0,0] ; // t_RunT0
t_End : transition [0,0] ;
Empty_t1_dfltext : transition [0,0] ;
Empty_t1_dlex : transition [0,0] ;
Empty_t1_d2ext : transition [0,0] ;
Empty_t1_dkext : transition [0,0] ;
Empty_t2_dfltext : transition [0,0] ;
Empty_t2_dlex : transition [0,0] ;
Empty_t2_d2ext : transition [0,0] ;
Empty_t2_dkext : transition [0,0] ;
Empty_OK : transition [0,0] ;

```

Structure

```

t1dfltI1I : (dfltI1I, expr1_a0, t1_read) ,
            (dfltI1I, t2_read, t1_dfltext, expr1_a0) ;
t1dfltI2I : (dfltI2I, expr1_a1, t1_read) ,
            (dfltI2I, t2_read, t1_dfltext, expr1_a1) ;
t1dfltI3I : (dfltI3I, expr1_a2, t1_read) ,
            (dfltI3I, t2_read, t1_dfltext, expr1_a2) ;
t1dfltInI : (dfltInI, expr1_anminus1, t1_read) ,
            (dfltInI, t2_read, t1_dfltext, expr1_anminus1) ;
t1d1I1I : (d1I1I, expr1_a0, t1_read) ,
            (d1I1I, t2_read, t1_dlex, expr1_a0) ;
t1d1I2I : (d1I2I, expr1_a1, t1_read) ,
            (d1I2I, t2_read, t1_dlex, expr1_a1) ;
t1d1I3I : (d1I3I, expr1_a2, t1_read) ,
            (d1I3I, t2_read, t1_dlex, expr1_a2) ;
t1d1InI : (d1InI, expr1_anminus1, t1_read) ,
            (d1InI, t2_read, t1_dlex, expr1_anminus1) ;
t1d2I1I : (d2I1I, expr1_a0, t1_read) ,
            (d2I1I, t2_read, t1_d2ext, expr1_a0) ;

```

Continuação da listagem II.5.

```

t1d2I2I : (d2I2I, expr1_a1, t1_read) ,
           (d2I2I, t2_read, t1_d2ext, expr1_a1) ;
t1d2I3I : (d2I3I, expr1_a2, t1_read) ,
           (d2I3I, t2_read, t1_d2ext, expr1_a2) ;
t1d2InI : (d2InI, expr1_anminus1, t1_read) ,
           (d2InI, t2_read, t1_d2ext, expr1_anminus1) ;
t1dkI1I : (dkI1I, expr1_a0, t1_read) ,
           (dkI1I, t2_read, t1_dkext, expr1_a0) ;
t1dkI2I : (dkI2I, expr1_a1, t1_read) ,
           (dkI2I, t2_read, t1_dkext, expr1_a1) ;
t1dkI3I : (dkI3I, expr1_a2, t1_read) ,
           (dkI3I, t2_read, t1_dkext, expr1_a2) ;
t1dkInI : (dkInI, expr1_anminus1, t1_read) ,
           (dkInI, t2_read, t1_dkext, expr1_anminus1) ;
t2dfltI1I : (dfltI1I, t2_read, expr2_a0) ,
            (dfltI1I, rw, t2_dfltext, t_TO, expr2_a0) ;
t2dfltI2I : (dfltI2I, t2_read, expr2_a1) ,
            (dfltI2I, rw, t2_dfltext, t_TO, expr2_a1) ;
t2dfltI3I : (dfltI3I, t2_read, expr2_a2) ,
            (dfltI3I, rw, t2_dfltext, t_TO, expr2_a2) ;
t2dfltInI : (dfltInI, t2_read, expr2_anminus1) ,
            (dfltInI, rw, t2_dfltext, t_TO, expr2_anminus1) ;
t2d1I1I : (d1I1I, t2_read, expr2_a0) ,
           (d1I1I, rw, t2_d1ext, t_TO, expr2_a0) ;
t2d1I2I : (d1I2I, t2_read, expr2_a1) ,
           (d1I2I, rw, t2_d1ext, t_TO, expr2_a1) ;
t2d1I3I : (d1I3I, t2_read, expr2_a2) ,
           (d1I3I, rw, t2_d1ext, t_TO, expr2_a2) ;
t2d1InI : (d1InI, t2_read, expr2_anminus1) ,
           (d1InI, rw, t2_d1ext, t_TO, expr2_anminus1) ;
t2d2I1I : (d2I1I, t2_read, expr2_a0) ,
           (d2I1I, rw, t2_d2ext, t_TO, expr2_a0) ;
t2d2I2I : (d2I2I, t2_read, expr2_a1) ,
           (d2I2I, rw, t2_d2ext, t_TO, expr2_a1) ;
t2d2I3I : (d2I3I, t2_read, expr2_a2) ,
           (d2I3I, rw, t2_d2ext, t_TO, expr2_a2) ;
t2d2InI : (d2InI, t2_read, expr2_anminus1) ,
           (d2InI, rw, t2_d2ext, t_TO, expr2_anminus1) ;
t2dkI1I : (dkI1I, t2_read, expr2_a0) ,
           (dkI1I, rw, t2_dkext, t_TO, expr2_a0) ;
t2dkI2I : (dkI2I, t2_read, expr2_a1) ,
           (dkI2I, rw, t2_dkext, t_TO, expr2_a1) ;
t2dkI3I : (dkI3I, t2_read, expr2_a2) ,
           (dkI3I, rw, t2_dkext, t_TO, expr2_a2) ;
t2dkInI : (dkInI, t2_read, expr2_anminus1) ,
           (dkInI, rw, t2_dkext, t_TO, expr2_anminus1) ;
t_0_0 : (expr1_a0, expr2_a0, t_G, rw, EXEC) ,
        (expr1_a0, expr2_a0, t_G, t1_read) ;
t_0_1 : (expr1_a0, expr2_a1, t_G, rw, EXEC) ,
        (expr1_a0, expr2_a1, t_G, t1_read) ;
t_0_2 : (expr1_a0, expr2_a2, t_G, rw, EXEC) ,
        (expr1_a0, expr2_a2, t_G, t1_read) ;
t_0_nminus1 : (expr1_a0, expr2_anminus1, t_G, rw, EXEC) ,
              (expr1_a0, expr2_anminus1, t_G, t1_read) ;
t_1_0 : (expr1_a1, expr2_a0, t_G, rw, EXEC) ,
        (expr1_a1, expr2_a0, t_G, t1_read) ;
t_1_1 : (expr1_a1, expr2_a1, t_G, rw, EXEC) ,
        (expr1_a1, expr2_a1, t_G, t1_read) ;
t_1_2 : (expr1_a1, expr2_a2, t_G, rw, EXEC) ,
        (expr1_a1, expr2_a2, t_G, t1_read) ;
t_1_nminus1 : (expr1_a1, expr2_anminus1, t_G, rw, EXEC) ,
              (expr1_a1, expr2_anminus1, t_G, t1_read) ;
t_2_0 : (expr1_a2, expr2_a0, t_G, rw, EXEC) ,
        (expr1_a2, expr2_a0, t_G, t1_read) ;

```

Continuação da listagem II.5.

```

t_2_1 : (expr1_a2, expr2_a1, t_G, rw, EXEC) ,
        (expr1_a2, expr2_a1, t_G, t1_read) ;
t_2_2 : (expr1_a2, expr2_a2, t_G, rw, EXEC) ,
        (expr1_a2, expr2_a2, t_G, t1_read) ;
t_2_nminus1 : (expr1_a2, expr2_anminus1, t_G, rw, EXEC) ,
               (expr1_a2, expr2_anminus1, t_G, t1_read) ;
t_nminus1_0 : (expr1_anminus1, expr2_a0, t_G, rw, EXEC) ,
               (expr1_anminus1, expr2_a0, t_G, t1_read) ;
t_nminus1_1 : (expr1_anminus1, expr2_a1, t_G, rw, EXEC) ,
               (expr1_anminus1, expr2_a1, t_G, t1_read) ;
t_nminus1_2 : (expr1_anminus1, expr2_a2, t_G, rw, EXEC) ,
               (expr1_anminus1, expr2_a2, t_G, t1_read) ;
t_nminus1_nminus1 : (expr1_anminus1, expr2_anminus1, t_G, rw, EXEC) ,
                    (expr1_anminus1, expr2_anminus1, t_G, t1_read) ;
t_SetG : (t_NG) , (t_G) ;
t_ResetG : (t_G) , (t_NG) ;
t_RunTO : (t_TO) , (t_TOSubnet) ;
t_End : (t_TOSubnet) , (2 * Empty_ext) ;
Empty_t1_dfltext : (t1_dfltext, Empty_ext) , (Empty_ext_OK) ;
Empty_t1_d1ext : (t1_d1ext, Empty_ext) , (Empty_ext_OK) ;
Empty_t1_d2ext : (t1_d2ext, Empty_ext) , (Empty_ext_OK) ;
Empty_t1_dkext : (t1_dkext, Empty_ext) , (Empty_ext_OK) ;
Empty_t2_dfltext : (t2_dfltext, Empty_ext) , (Empty_ext_OK) ;
Empty_t2_d1ext : (t2_d1ext, Empty_ext) , (Empty_ext_OK) ;
Empty_t2_d2ext : (t2_d2ext, Empty_ext) , (Empty_ext_OK) ;
Empty_t2_dkext : (t2_dkext, Empty_ext) , (Empty_ext_OK) ;
Empty_OK : (2 * Empty_ext_OK) , (EXEC) ;

endNet.

```

Continuação da listagem II.5.

Levando-se em conta as convenções adotadas na construção da figura 12.15, a análise de propriedades gerada pelo programa **ARP** (transcrita na listagem II.6) conduz às seguintes conclusões:

- i) A simulação produziu 11 estados (classes): Estas classes serão rotuladas de **C₀** a **C₁₀**;
- ii) A rede é limitada: O número máximo de *tokens* em qualquer lugar será 2, conforme pode ser constatado nos três itens seguintes, o que é compatível com as convenções adotadas;
- iii) Os lugares pertencentes aos conjuntos

$$S_1 = \{dfl[1], dfl[3], d_1[1], d_1[2], d_1[n], d_2[2], d_2[3], d_2[n], d_k[1], d_k[2], d_k[3], d_k[n]\},$$

$$S_2 = \{expr_1=a_1, expr_1=a_2, expr_1=a_{n-1}, expr_2=a_0, expr_2=a_1, expr_2=a_{n-1}\} \text{ e}$$

$$S_3 = \{t_1\_dfltext, t_1\_d1ext, t_1\_dkext, t_2\_dfltext, t_2\_d2ext, t_2\_dkext\}$$

são nulos: O fato dos lugares do conjunto **S₁** serem nulos, ao longo da simulação da rede, denota que as *células de armazenamento* de **P** não foram alteradas (sua configuração original, presente na classe **C₀**, foi preservada). Em relação ao conjunto **S₂**, seus lugares permaneceram nulos devido à não alteração das expressões utilizadas na *expressão de leitura*, associada ao ramo da RPSD da figura 12.15 (o controle da *expressão de leitura* não faz parte da RPTM representante da conexão envolvendo o lugar **P** e a transição **t**, e, sim, faz parte da *memória* do modelo RPSD – entretanto, para garantir a atomicidade de **t**, enquanto o lugar **EXEC** (associado a **P**) estiver vazio, a marcação dos lugares **expr₁=a_i** e **expr₂=a_j**, **i, j = 1, ..., n-1**, somente poderá ser alterada por transições pertencentes à subrede representada por **t_{TOSubnet}**). Finalmente, em relação ao conjunto **S₃**, seus lugares permaneceram nulos devido ao fato de não representarem os *tokens* **d₂** e **d₁**, os quais correspondem aos *tokens* armazenados nos contêineres **t₁** e **t₂**, lidos de **P** em decorrência do disparo de **t**;

```

State Enumeration : net R12_15 (11 reachable states).

Verified properties:
*-----*
Net under analysis is limited.
  Null places (M = 0): {dfltI1I, dfltI3I, d1I1I, d1I2I, d1InI, d2I2I, d2I3I,
                        d2InI, dkI1I, dkI2I, dkI3I, dkInI, expr1_a1,
                        expr1_a2, expr1_anminus1, expr2_a0, expr2_a1,
                        expr2_anminus1, t1_dfltext, t1_dlex, t1_dkext,
                        t2_dfltext, t2_d2ext, t2_dkext}
  Binary places      : {dfltI2I, dfltInI, d1I3I, d2I1I, expr1_a0, expr2_a2,
                        rw, t1_read, t2_read, t1_d2ext, t2_dlex, t_NG, t_G,
                        t_TO, t_TOSubnet, EXEC}
  k-Bounded places  : {2* Empty_ext, 2* Empty_ext_OK}
  Unbounded places  : {}

Net under analysis is not strictly conservative.

Multi-enabled Tr.: {}

Net under analysis is not live.
  Live Tr.          : {t1d2I1I, t2d1I3I, t_0_2, t_SetG, t_ResetG, t_RunTO,
                        t_End, Empty_t1_d2ext, Empty_t2_dlex, Empty_OK}
  "Almost-live" Tr.: {t1d2I1I, t2d1I3I, t_0_2, t_SetG, t_ResetG, t_RunTO,
                        t_End, Empty_t1_d2ext, Empty_t2_dlex, Empty_OK}
  Non-fired Tr.     : {t1dfltI1I, t1dfltI2I, t1dfltI3I, t1dfltInI, t1d1I1I,
                        t1d1I2I, t1d1I3I, t1d1InI, t1d2I2I, t1d2I3I, t1d2InI,
                        t1dkI1I, t1dkI2I, t1dkI3I, t1dkInI, t2dfltI1I,
                        t2dfltI2I, t2dfltI3I, t2dfltInI, t2d1I1I, t2d1I2I,
                        t2d1InI, t2d2I1I, t2d2I2I, t2d2I3I, t2d2InI, t2dkI1I,
                        t2dkI2I, t2dkI3I, t2dkInI, t_0_0, t_0_1, t_0_nminus1,
                        t_1_0, t_1_1, t_1_2, t_1_nminus1, t_2_0, t_2_1, t_2_2,
                        t_2_nminus1, t_nminus1_0, t_nminus1_1, t_nminus1_2,
                        t_nminus1_nminus1, Empty_t1_dfltext, Empty_t1_dlex,
                        Empty_t1_dkext, Empty_t2_dfltext, Empty_t2_d2ext,
                        Empty_t2_dkext}

Net can always go back to M0.

No live-locks detected.

No deadlocks detected.
*-----*

```

Listagem II.6. Análise de propriedades para a rede da figura II.20.

- iv) Os lugares **dflt[2]**, **dflt[n]**, **d1[3]**, **d2[1]**, **expr1=a0**, **expr2=a2**, **r/w**, **t1_read**, **t2_read**, **t1_d2ext**, **t2_d1ext**, **t_NG**, **t_G**, **t_TO**, **t_TOSubnet** e **EXEC** são binários: Na marcação inicial, presente na figura 12.15, os lugares **dflt[2]**, **dflt[n]**, **d1[3]**, **d2[1]**, **expr1=a0**, **expr2=a2**, **r/w**, **t_G** (**t_NG** no modelo **JARP**) e **EXEC** possuem 1 *token* cada. Assim, para serem qualificados como binários, os demais lugares da lista recebem 1 *token* em alguma classe, conforme pode ser visto adiante, na descrição do grafo de classes da rede;
- v) Os lugares **Empty_ext** e **Empty_ext_OK** são limitados a 2 *tokens*: Na marcação inicial, presente na figura 12.15, os lugares **dflt[2]**, **dflt[n]**, **d1[3]**, **d2[1]**, **expr1=a0**, **expr2=a2**, **r/w**, **t_G** (**t_NG** no modelo **JARP**) e **EXEC** possuem 1 *token* cada. Assim, para serem qualificados como binários, os demais lugares da lista recebem 1 *token* em alguma classe, conforme pode ser visto adiante, na descrição do grafo de classes da rede;
- vi) A rede não é estritamente conservativa: Três grupos de lugares são conservativos para 1 *token*, quais sejam, **{r/w, t1_read, t2_read}**, **{t_TO, t_TOSubnet}** e **{t_G, t_NG}**. O

- grupo de lugares $\{\text{expr}_1=\mathbf{a}_i, \text{expr}_2=\mathbf{a}_j \mid i, j = 1, \dots, n-1\}$ é conservativo para 2 *tokens* (a menos que a subrede representada pelo lugar $\mathbf{t\_T0Subnet}$ altere a marcação de algum destes lugares). O grupo de lugares $\{\mathbf{dflt}[j], \mathbf{d}_i[j] \mid i=1, \dots, k; j = 1, \dots, n\}$ é conservativo para n *tokens*. A rede não é conservativa para grupos envolvendo os demais lugares;
- vii) Não há transições k -sensíveis: Fato compatível com a figura 12.15;
 - viii) A rede não é viva: Esta análise, em conjunto com as três seguintes, reflete o fato da transição $\mathbf{t}$ da RPSD poder ser disparada somente para a leitura de duas células específicas, uma vez que as expressões expr_1 e expr_2 não estão sendo alteradas, na porção da RPSD mostrada na figura 12.15;
 - ix) As transições $\mathbf{t}_1\mathbf{d}_2[1]$, $\mathbf{t}_2\mathbf{d}_1[3]$, $\mathbf{t\_a0a2}$, $\mathbf{Empty\_t1\_d2ext}$, $\mathbf{Empty\_t2\_d1ext}$, $\mathbf{Empty\_OK}$, $\mathbf{t\_SetG}$, $\mathbf{t\_ResetG}$, $\mathbf{t\_RunT0}$ e $\mathbf{t\_End}$ são vivas/quase-vivas: O disparo destas transições é comentado adiante;
 - x) As demais transições não disparam durante a simulação: Este fato se deve à marcação constante dos lugares representes de $\mathbf{P}$ e da **memória da rede**, além de somente dois dos lugares representantes de $\mathbf{t}_1$ e $\mathbf{t}_2$ receberem *token* ao longo da simulação (habilitando, portanto, somente duas das transições relativas ao esvaziamento dos contêineres);
 - xi) A rede pode retornar ao estado original $\mathbf{C}_0$: Isto ocorre após a remoção dos *tokens* presentes nos contêineres $\mathbf{t}_1$ e $\mathbf{t}_2$, caso a transição $\mathbf{t\_ResetG}$ dispare, conforme pode-se constatar no grafo de classes da figura II.21. Caso a subrede descritora das **operações** associadas a $\mathbf{t}$, representada pelo lugar $\mathbf{t\_T0Subnet}$, altere a marcação de $\text{expr}_1=\mathbf{a}_0$ ou de $\text{expr}_2=\mathbf{a}_2$, não será possível a rede retornar ao estado $\mathbf{C}_0$;
 - xii) A rede não possui *livelocks*: Apesar de possuir ciclos no grafo de classes, a rede não ficará em *loop* infinito, uma vez que há a possibilidade de sair de tais ciclos; e
 - xiii) A rede não entra em *deadlock*: Como o fluxo de execução pode restaurar a rede ao seu estado inicial, não há possibilidade da rede ficar bloqueada.

O grafo de classes da rede correspondente à figura 12.15, mostrado na figura II.21, apresenta natureza circular. Destaca-se que, na classe $\mathbf{C}_9$, o fluxo de execução pode conduzir a rede à classe inicial (através do disparo de $\mathbf{t\_ResetG}$, tornando *falsa* a guarda de $\mathbf{t}$) ou à classe $\mathbf{C}_2$ (mediante o disparo de $\mathbf{t\_a0a2}$, o qual ocorre caso a guarda de $\mathbf{t}$ continue *verdadeira*). Além disso, na classe $\mathbf{C}_6$, a ordem de disparo das transições $\mathbf{Empty\_t1\_d2ext}$ e $\mathbf{Empty\_t2\_d1ext}$, determina o caminho a ser tomado na bifurcação representada pelas classes $\mathbf{C}_7$ e $\mathbf{C}_{10}$, a qual encerra-se na classe $\mathbf{C}_8$. Desde $\mathbf{C}_0$, tem-se a seguinte sequência de disparo de transições, considerando-se as bifurcações possíveis:

- i) $\mathbf{t\_SetG}$: representando a atribuição de *verdadeiro* à guarda de $\mathbf{t}$;
- ii) $\mathbf{t\_a0a2}$: representando o início do disparo de $\mathbf{t}$, através da remoção dos *tokens* presentes em $\mathbf{r/w}$ (sinalizando que $\mathbf{P}$ está realizando uma operação) e **EXEC** (denotando que $\mathbf{t}$ tem *acesso exclusivo* a $\mathbf{P}$), assim como da inserção de um *token* em $\mathbf{t1\_read}$ (habilitando a leitura a ser armazenada em $\mathbf{t}_1$);
- iii) $\mathbf{t1\_d2}[1]$: representando as seguintes ações: efetivação da leitura da primeira **célula de armazenamento** (célula $\mathbf{a}_0$), cujo conteúdo ($\mathbf{d}_2$) é copiado para o contêiner $\mathbf{t}_1$ (através da inserção de um *token* em $\mathbf{t1\_d2ext}$); sinalização de conclusão da leitura referente ao *token* $\mathbf{t}_1$ (através da remoção do *token* presente em $\mathbf{t1\_read}$); e habilitação da leitura cujo resultado será armazenado no *token* $\mathbf{t}_2$ (através da inserção de um *token* em $\mathbf{t2\_read}$);
- iv) $\mathbf{t2\_d1}[3]$: representando as seguintes ações: efetivação da leitura da terceira **célula de armazenamento** (célula $\mathbf{a}_2$), cujo conteúdo ($\mathbf{d}_1$) é copiado para o contêiner $\mathbf{t}_2$ (através da inserção de um *token* em $\mathbf{t2\_d1ext}$); sinalização de conclusão da leitura referente ao *token* $\mathbf{t}_2$ (através da remoção do *token* presente em $\mathbf{t2\_read}$); e liberação do lugar $\mathbf{P}$ para uma nova operação de leitura/escrita (através da inserção de um *token* em $\mathbf{r/w}$);

- v) **t_RunT₀**: representando o início da execução das *operações* associadas à transição **t**;
- vi) **t_End**: representando o término da execução das *operações* associadas à transição **t**;
- vii) Duas seqüências de disparo possíveis:
 - vii-1) **Empty_t₁_d₂ext / Empty_t₂_d₁ext**: representando o consumo dos *tokens* contidos nos contêineres **t₁** e **t₂**, respectivamente; ou
 - vii-2) **Empty_t₂_d₁ext / Empty_t₁_d₂ext**: representando o consumo dos *tokens* contidos nos contêineres **t₂** e **t₁**, respectivamente;
- viii) **Empty_OK**: representando que os contêineres **t₁** e **t₂** estão vazios; e
- ix) Dois disparos possíveis:
 - ix-1) **t_a₀a₂**: representando o início de um novo disparo de **t**, através dos procedimentos descritos no item ii), conduzindo a rede ao estado **C₂**; ou
 - ix-2) **t_ResetG**: representando a atribuição de *falso* à guarda de **t**, conduzindo a rede ao seu estado inicial **C₀**.

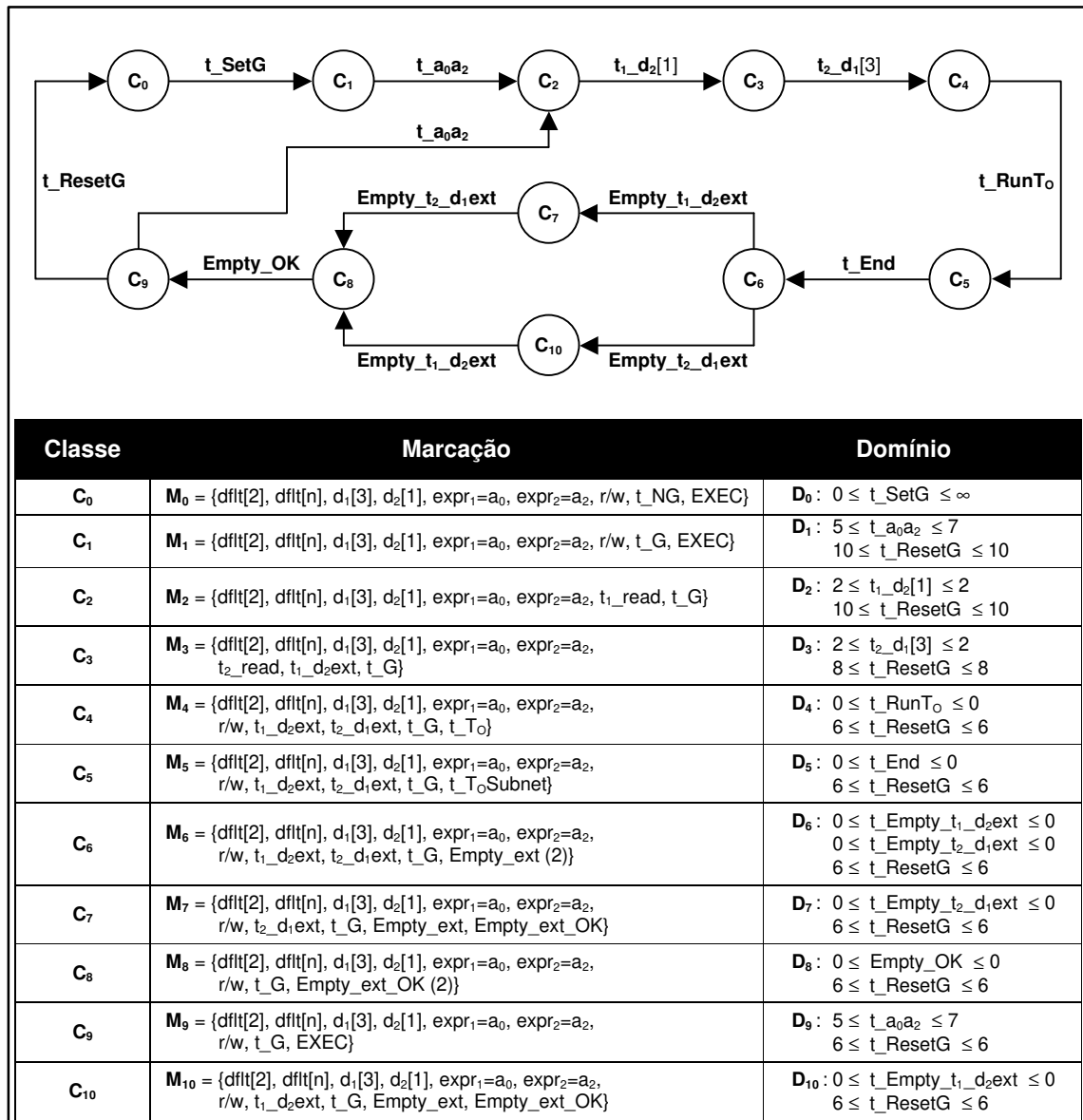


Figura II.21. Grafo de classes para a rede das figuras 12.15 / II.20.

Na listagem **R12-16.PN**, gerada pelo **JARP** para o modelo da figura II.22, faz-se necessário o acréscimo de intervalos de disparo. Considerando-se que as *latências de leitura* e de *escrita* de **P** sejam 2 e 3, respectivamente, e que o *intervalo de disparo* de **t** seja definido por [5,7], a listagem II.7 transcreve o modelo **ARP** correspondente à figura 12.16, após os acréscimos necessários.

Servindo de guia para a compreensão das conexões entre os lugares e as transições da rede da figura II.22, a listagem II.7 apresenta alguns rótulos que diferem um pouco dos utilizados na figura 12.16, devido às limitações para especificação de identificadores no **ARP**. Desta forma, tal como foi feito na listagem II.5, a listagem II.7 traz, ao final de algumas linhas, após duas barras ("/") e em negrito, o padrão de rotulagem utilizado na figura 12.16, para o grupo de lugares ou de transições em questão. Além disso, como o **ARP** não admite que um mesmo identificador seja utilizado para nomear dois ou mais nós (independentemente de serem lugares e/ou transições), o programa **JARP** acrescenta, na geração da seção **Nodes** de uma listagem **PN**, o caracter sublinha ('_') acompanhado de um número (a partir de zero), após a ocorrência de um identificador já utilizado na seção. Esta distinção pode ser observada na listagem II.7, na qual os identificadores **clr₁[i]** e **clr₂[i]** (**i** = 1, 2, 3, **n**) são utilizados tanto para lugares quanto para transições.

```
Net R12_16 ;

Nodes
  clr1I1I : place ;                // clr1[1]
  clr1I2I : place ;
  clr1I3I : place ;
  clr1InI : place ;
  clr2I1I : place ;
  clr2I2I : place ;
  clr2I3I : place ;
  clr2InI : place ;
  clr1OKI1I : place ;              // clr1[1]OK
  clr1OKI2I : place ;
  clr1OKI3I : place ;
  clr1OKInI : place ;
  clr2OKI1I : place ;
  clr2OKI2I : place ;
  clr2OKI3I : place ;
  clr2OKInI : place ;
  dfltI1I : place ;                // dflt[1]
  dfltI2I : place (1) ;
  dfltI3I : place ;
  dfltInI : place (1) ;
  d1I1I : place ;                  // d1[1]
  d1I2I : place ;
  d1I3I : place (1) ;
  d1InI : place ;
  d2I1I : place (1) ;
  d2I2I : place ;
  d2I3I : place ;
  d2InI : place ;
  dkI1I : place ;
  dkI2I : place ;
  dkI3I : place ;
  dkInI : place ;
  expr1_a0 : place (1) ;           // expr1=a0
  expr1_a1 : place ;
  expr1_a2 : place ;
  expr1_anminus1 : place ;         // expr1=an-1
```

Listagem II.7. Modelo ARP da figura II.22.

```

expr2_a0 : place ;
expr2_a1 : place ;
expr2_a2 : place (1) ;
expr2_anminus1 : place ;
rw : place (1) ;
t1_write : place ;
t2_write : place ;
t1_dfltext : place (1) ;
t1_d1ext : place ;
t1_d2ext : place ;
t1_dkext : place ;
t2_dfltext : place ;
t2_d1ext : place ;
t2_d2ext : place ;
t2_dkext : place (1) ;
t_G : place ;
t_NG : place (1) ;
t_TO : place ;
t_TOSubnet : place ;
EXEC : place (1) ;

c1dfltI1I : transition [0,0] ;
c1dfltI2I : transition [0,0] ;
c1dfltI3I : transition [0,0] ;
c1dfltInI : transition [0,0] ;
c1d1I1I : transition [0,0] ;
c1d1I2I : transition [0,0] ;
c1d1I3I : transition [0,0] ;
c1d1InI : transition [0,0] ;
c1d2I1I : transition [0,0] ;
c1d2I2I : transition [0,0] ;
c1d2I3I : transition [0,0] ;
c1d2InI : transition [0,0] ;
c1dkI1I : transition [0,0] ;
c1dkI2I : transition [0,0] ;
c1dkI3I : transition [0,0] ;
c1dkInI : transition [0,0] ;
c2dfltI1I : transition [0,0] ;
c2dfltI2I : transition [0,0] ;
c2dfltI3I : transition [0,0] ;
c2dfltInI : transition [0,0] ;
c2d1I1I : transition [0,0] ;
c2d1I2I : transition [0,0] ;
c2d1I3I : transition [0,0] ;
c2d1InI : transition [0,0] ;
c2d2I1I : transition [0,0] ;
c2d2I2I : transition [0,0] ;
c2d2I3I : transition [0,0] ;
c2d2InI : transition [0,0] ;
c2dkI1I : transition [0,0] ;
c2dkI2I : transition [0,0] ;
c2dkI3I : transition [0,0] ;
c2dkInI : transition [0,0] ;
clr1I1I_0 : transition [0,0] ;
clr1I2I_0 : transition [0,0] ;
clr1I3I_0 : transition [0,0] ;
clr1InI_0 : transition [0,0] ;
clr2I1I_0 : transition [0,0] ;
clr2I2I_0 : transition [0,0] ;
clr2I3I_0 : transition [0,0] ;
clr2InI_0 : transition [0,0] ;
t1dfltI1I : transition [3,3] ;
t1dfltI2I : transition [3,3] ;
t1dfltI3I : transition [3,3] ;

```

```

// r/w
// t1_write
// t1_dfltext
// t1_d1ext
// t_T0
// t_T0Subnet
// c1_dflt[1]
// c1_d1[1]
// clr1[1]
// t1_dflt[1]

```

Continuação da listagem II.7.

```

t1dfltInI : transition [3,3] ;
t1d1I1I : transition [3,3] ;           // t1-d1[1]
t1d1I2I : transition [3,3] ;
t1d1I3I : transition [3,3] ;
t1d1InI : transition [3,3] ;
t1d2I1I : transition [3,3] ;
t1d2I2I : transition [3,3] ;
t1d2I3I : transition [3,3] ;
t1d2InI : transition [3,3] ;
t1dkI1I : transition [3,3] ;
t1dkI2I : transition [3,3] ;
t1dkI3I : transition [3,3] ;
t1dkInI : transition [3,3] ;
t2dfltI1I : transition [3,3] ;
t2dfltI2I : transition [3,3] ;
t2dfltI3I : transition [3,3] ;
t2dfltInI : transition [3,3] ;
t2d1I1I : transition [3,3] ;
t2d1I2I : transition [3,3] ;
t2d1I3I : transition [3,3] ;
t2d1InI : transition [3,3] ;
t2d2I1I : transition [3,3] ;
t2d2I2I : transition [3,3] ;
t2d2I3I : transition [3,3] ;
t2d2InI : transition [3,3] ;
t2dkI1I : transition [3,3] ;
t2dkI2I : transition [3,3] ;
t2dkI3I : transition [3,3] ;
t2dkInI : transition [3,3] ;
t_dfltd200 : transition [5,7] ;       // t_dfltd2a0a0
t_dfltdk00 : transition [5,7] ;
t_dfltd202 : transition [5,7] ;
t_dfltdk02 : transition [5,7] ;
t_dfltd220 : transition [5,7] ;
t_dfltdk20 : transition [5,7] ;
t_dfltd222 : transition [5,7] ;
t_dfltdk22 : transition [5,7] ;
t_dld200 : transition [5,7] ;         // t_d1d2a0a0
t_dld202 : transition [5,7] ;
t_dld220 : transition [5,7] ;
t_dld222 : transition [5,7] ;
t_dldk00 : transition [5,7] ;
t_dldk02 : transition [5,7] ;
t_dldk20 : transition [5,7] ;
t_dldk22 : transition [5,7] ;
t_SetG : transition ;
t_ResetG : transition [10,10] ;
t_RunTO : transition [0,0] ;
t_End : transition [0,0] ;

```

Structure

```

c1dfltI1I : (dfltI1I, clr1I1I) , (clr1OKI1I) ;
c1dfltI2I : (dfltI2I, clr1I2I) , (clr1OKI2I) ;
c1dfltI3I : (dfltI3I, clr1I3I) , (clr1OKI3I) ;
c1dfltInI : (dfltInI, clr1InI) , (clr1OKInI) ;
c1d1I1I : (d1I1I, clr1I1I) , (clr1OKI1I) ;
c1d1I2I : (d1I2I, clr1I2I) , (clr1OKI2I) ;
c1d1I3I : (d1I3I, clr1I3I) , (clr1OKI3I) ;
c1d1InI : (d1InI, clr1InI) , (clr1OKInI) ;
c1d2I1I : (d2I1I, clr1I1I) , (clr1OKI1I) ;
c1d2I2I : (d2I2I, clr1I2I) , (clr1OKI2I) ;
c1d2I3I : (d2I3I, clr1I3I) , (clr1OKI3I) ;
c1d2InI : (d2InI, clr1InI) , (clr1OKInI) ;
c1dkI1I : (dkI1I, clr1I1I) , (clr1OKI1I) ;

```

Continuação da listagem II.7.

```

c1dkI2I : (dkI2I, clr1I2I) , (clr1OKI2I) ;
c1dkI3I : (dkI3I, clr1I3I) , (clr1OKI3I) ;
c1dkInI : (dkInI, clr1InI) , (clr1OKInI) ;
c2dfltI1I : (dfltI1I, clr2I1I) , (clr2OKI1I) ;
c2dfltI2I : (dfltI2I, clr2I2I) , (clr2OKI2I) ;
c2dfltI3I : (dfltI3I, clr2I3I) , (clr2OKI3I) ;
c2dfltInI : (dfltInI, clr2InI) , (clr2OKInI) ;
c2dI1I : (dI1I, clr2I1I) , (clr2OKI1I) ;
c2dI2I : (dI2I, clr2I2I) , (clr2OKI2I) ;
c2dI3I : (dI3I, clr2I3I) , (clr2OKI3I) ;
c2dInI : (dInI, clr2InI) , (clr2OKInI) ;
c2dkI1I : (dkI1I, clr2I1I) , (clr2OKI1I) ;
c2dkI2I : (dkI2I, clr2I2I) , (clr2OKI2I) ;
c2dkI3I : (dkI3I, clr2I3I) , (clr2OKI3I) ;
c2dkInI : (dkInI, clr2InI) , (clr2OKInI) ;
clr1I1I_0 : (expr1_a0, t1_write) , (expr1_a0, clr1I1I) ;
clr1I2I_0 : (expr1_a1, t1_write) , (expr1_a1, clr1I2I) ;
clr1I3I_0 : (expr1_a2, t1_write) , (expr1_a2, clr1I3I) ;
clr1InI_0 : (expr1_anminus1, t1_write) , (expr1_anminus1, clr1InI) ;
clr2I1I_0 : (expr2_a0, t2_write) , (expr2_a0, clr2I1I) ;
clr2I2I_0 : (expr2_a1, t2_write) , (expr2_a1, clr2I2I) ;
clr2I3I_0 : (expr2_a2, t2_write) , (expr2_a2, clr2I3I) ;
clr2InI_0 : (expr2_anminus1, t2_write) , (expr2_anminus1, clr2InI) ;
t1dfltI1I : (t1_dfltext, clr1OKI1I) , (dfltI1I, t2_write) ;
t1dfltI2I : (t1_dfltext, clr1OKI2I) , (dfltI2I, t2_write) ;
t1dfltI3I : (t1_dfltext, clr1OKI3I) , (dfltI3I, t2_write) ;
t1dfltInI : (t1_dfltext, clr1OKInI) , (dfltInI, t2_write) ;
t1dI1I : (t1_d1ext, clr1OKI1I) , (dI1I, t2_write) ;
t1dI2I : (t1_d1ext, clr1OKI2I) , (dI2I, t2_write) ;
t1dI3I : (t1_d1ext, clr1OKI3I) , (dI3I, t2_write) ;
t1dInI : (t1_d1ext, clr1OKInI) , (dInI, t2_write) ;
t1d2I1I : (t1_d2ext, clr1OKI1I) , (d2I1I, t2_write) ;
t1d2I2I : (t1_d2ext, clr1OKI2I) , (d2I2I, t2_write) ;
t1d2I3I : (t1_d2ext, clr1OKI3I) , (d2I3I, t2_write) ;
t1d2InI : (t1_d2ext, clr1OKInI) , (d2InI, t2_write) ;
t1dkI1I : (t1_dkext, clr1OKI1I) , (dkI1I, t2_write) ;
t1dkI2I : (t1_dkext, clr1OKI2I) , (dkI2I, t2_write) ;
t1dkI3I : (t1_dkext, clr1OKI3I) , (dkI3I, t2_write) ;
t1dkInI : (t1_dkext, clr1OKInI) , (dkInI, t2_write) ;
t2dfltI1I : (t2_dfltext, clr2OKI1I) , (dfltI1I, t_TO, rw) ;
t2dfltI2I : (t2_dfltext, clr2OKI2I) , (dfltI2I, t_TO, rw) ;
t2dfltI3I : (t2_dfltext, clr2OKI3I) , (dfltI3I, t_TO, rw) ;
t2dfltInI : (t2_dfltext, clr2OKInI) , (dfltInI, t_TO, rw) ;
t2dI1I : (t2_d1ext, clr2OKI1I) , (dI1I, t_TO, rw) ;
t2dI2I : (t2_d1ext, clr2OKI2I) , (dI2I, t_TO, rw) ;
t2dI3I : (t2_d1ext, clr2OKI3I) , (dI3I, t_TO, rw) ;
t2dInI : (t2_d1ext, clr2OKInI) , (dInI, t_TO, rw) ;
t2d2I1I : (t2_d2ext, clr2OKI1I) , (d2I1I, t_TO, rw) ;
t2d2I2I : (t2_d2ext, clr2OKI2I) , (d2I2I, t_TO, rw) ;
t2d2I3I : (t2_d2ext, clr2OKI3I) , (d2I3I, t_TO, rw) ;
t2d2InI : (t2_d2ext, clr2OKInI) , (d2InI, t_TO, rw) ;
t2dkI1I : (t2_dkext, clr2OKI1I) , (dkI1I, t_TO, rw) ;
t2dkI2I : (t2_dkext, clr2OKI2I) , (dkI2I, t_TO, rw) ;
t2dkI3I : (t2_dkext, clr2OKI3I) , (dkI3I, t_TO, rw) ;
t2dkInI : (t2_dkext, clr2OKInI) , (dkInI, t_TO, rw) ;
t_dfltd200 : (t_G, expr1_a0, expr2_a0, rw, t1_dfltext, t2_d2ext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_d2ext, expr1_a0, expr2_a0) ;
t_dfltdk00 : (t_G, t2_dkext, expr1_a0, expr2_a0, rw, t1_dfltext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_dkext, expr1_a0, expr2_a0) ;

```

Continuação da listagem II.7.

```

t_dfltd202 : (t_G, t2_d2ext, expr1_a0, expr2_a2, rw, t1_dfltext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_d2ext, expr1_a0, expr2_a2) ;
t_dfltdk02 : (t_G, t2_dkext, expr1_a0, expr2_a2, rw, t1_dfltext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_dkext, expr1_a0, expr2_a2) ;
t_dfltd220 : (t_G, t2_d2ext, expr1_a2, expr2_a0, rw, t1_dfltext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_d2ext, expr2_a0, expr1_a2) ;
t_dfltdk20 : (t_G, t2_dkext, expr1_a2, expr2_a0, rw, t1_dfltext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_dkext, expr2_a0, expr1_a2) ;
t_dfltd222 : (t_G, t2_d2ext, expr1_a2, expr2_a2, rw, t1_dfltext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_d2ext, expr1_a2, expr2_a2) ;
t_dfltdk22 : (t_G, t2_dkext, expr1_a2, expr2_a2, rw, t1_dfltext, EXEC) ,
              (t_G, t1_write, t1_dfltext, t2_dkext, expr1_a2, expr2_a2) ;
t_dld200 : (t_G, t2_d2ext, expr1_a0, expr2_a0, rw, t1_dlex, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_d2ext, expr1_a0, expr2_a0) ;
t_dld202 : (t_G, t1_dlex, t2_d2ext, expr1_a0, expr2_a2, rw, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_d2ext, expr1_a0, expr2_a2) ;
t_dld220 : (t_G, t1_dlex, t2_d2ext, expr1_a2, expr2_a0, rw, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_d2ext, expr2_a0, expr1_a2) ;
t_dld222 : (t_G, t1_dlex, t2_d2ext, expr1_a2, expr2_a2, rw, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_d2ext, expr1_a2, expr2_a2) ;
t_dldk00 : (t_G, t1_dlex, t2_dkext, expr1_a0, expr2_a0, rw, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_dkext, expr1_a0, expr2_a0) ;
t_dldk02 : (t_G, t1_dlex, t2_dkext, expr1_a0, expr2_a2, rw, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_dkext, expr1_a0, expr2_a2) ;
t_dldk20 : (t_G, t1_dlex, t2_dkext, expr1_a2, expr2_a0, rw, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_dkext, expr2_a0, expr1_a2) ;
t_dldk22 : (t_G, t1_dlex, t2_dkext, expr1_a2, expr2_a2, rw, EXEC) ,
            (t_G, t1_write, t1_dlex, t2_dkext, expr1_a2, expr2_a2) ;
t_SetG : (t_NG) , (t_G) ;
t_ResetG : (t_G) , (t_NG) ;
t_RunTO : (t_TO) , (t_TOSubnet) ;
t_End : (t_TOSubnet) , (EXEC) ;

endNet.

```

Continuação da listagem II.7.

A análise de propriedades da rede correspondente à figura 12.16, realizada pelo **ARP**, encontra-se transcrita na listagem II.8, enquanto o grafo de classes de tal rede, também gerado pelo **ARP**, está presente na figura II.23.

Levando-se em conta as convenções adotadas na construção da figura 12.16, a análise de propriedades gerada pelo programa **ARP** (listagem II.8) conduz às seguintes conclusões:

- i) A simulação produziu 13 estados (classes): Estas classes serão rotuladas de **C₀** a **C₁₂**;
- ii) A rede é binária: O número máximo de *tokens* em qualquer lugar será 1, conforme pode ser constatado nos três itens seguintes, o que é compatível com as convenções adotadas;
- iii) Os lugares pertencentes aos conjuntos

$$S_1 = \{\mathbf{dfl}[3], \mathbf{d}_1[1], \mathbf{d}_1[2], \mathbf{d}_1[n], \mathbf{d}_2[2], \mathbf{d}_2[3], \mathbf{d}_2[n], \mathbf{d}_k[1], \mathbf{d}_k[2], \mathbf{d}_k[n]\},$$

$$S_2 = \{\mathbf{expr}_1=\mathbf{a}_1, \mathbf{expr}_1=\mathbf{a}_2, \mathbf{expr}_1=\mathbf{a}_{n-1}, \mathbf{expr}_2=\mathbf{a}_0, \mathbf{expr}_2=\mathbf{a}_1, \mathbf{expr}_2=\mathbf{a}_{n-1}\},$$

$$S_3 = \{\mathbf{t}_1\text{-}\mathbf{d}_1\text{ext}, \mathbf{t}_1\text{-}\mathbf{d}_2\text{ext}, \mathbf{t}_1\text{-}\mathbf{d}_k\text{ext}, \mathbf{t}_2\text{-}\mathbf{dfl}\text{ext}, \mathbf{t}_2\text{-}\mathbf{d}_1\text{ext}, \mathbf{t}_2\text{-}\mathbf{d}_2\text{ext}\} \text{ e}$$

$$S_4 = \{\mathbf{clr}_1[2], \mathbf{clr}_1[3], \mathbf{clr}_1[n], \mathbf{clr}_2[1], \mathbf{clr}_2[2], \mathbf{clr}_2[n],$$

$$\mathbf{clr}_1[2]\text{OK}, \mathbf{clr}_1[3]\text{OK}, \mathbf{clr}_1[n]\text{OK}, \mathbf{clr}_2[1]\text{OK}, \mathbf{clr}_2[2]\text{OK}, \mathbf{clr}_2[n]\text{OK}\}$$

são nulos: Os lugares do conjunto **S₁** correspondem a valores não assumidos pelas *células de armazenamento* de **P**, ao longo da simulação (sua configuração original, presente na classe **C₀**, foi preservada). Considerando-se o conjunto de lugares representantes das células de armazenamento de **P**, três conjuntos complementam **S₁**, quais sejam, **S_{1A}** = {**dfl**[2], **dfl**[n]}, **S_{1B}** = {**d₂**[1], **d₁**[3]} e **S_{1C}** = {**dfl**[1], **d_k**[3]}. O conjunto **S_{1A}**

representa valores de células de armazenamento, vigentes na classe C_0 , que foram preservados. Por outro lado, o conjunto S_{1B} representa valores para as células a_0 e a_2 que foram substituídos pelos valores representados pelos lugares do conjunto S_{1C} . Observe, no item seguinte – iv), que os lugares pertencentes a S_{1A} , S_{1B} e S_{1C} são binários. Em relação ao conjunto S_2 , seus lugares permaneceram nulos devido à não alteração das expressões utilizadas na *expressão de escrita*, associada ao ramo da RPSD da figura 12.16 (o controle da marcação dos lugares relativos a esta expressão não está representado na RPTM da figura 12.16, uma vez que faz parte da **memória** do modelo RPSD – no entanto, para garantir a atomicidade de t , enquanto o lugar **EXEC** (associado a P) estiver vazio, a marcação dos lugares $\text{expr}_1=a_i$ e $\text{expr}_2=a_j$, $i, j = 1, \dots, n-1$, somente poderá ser alterada por transições pertencentes à subrede representada por $t_{T_0\text{Subnet}}$). Os lugares pertencentes ao conjunto S_3 permaneceram nulos devido ao fato de não representarem os *tokens* d_{flt} e d_k , os quais correspondem aos *tokens* armazenados nos contêineres t_1 e t_2 , a serem escritos nas células a_0 e a_2 de P , em decorrência do disparo de t . Finalmente, em relação ao conjunto S_4 , seus lugares permaneceram nulos devido ao fato de não estarem associados aos lugares do conjunto S_{1B} , os únicos que precisaram ser esvaziados, ao longo da simulação, para que as células que representavam recebessem novos valores;

- iv) Os lugares $\text{clr}_1[1]$, $\text{clr}_2[3]$, $\text{clr}_1[1]_{OK}$, $\text{clr}_2[3]_{OK}$, $d_{flt}[1]$, $d_{flt}[2]$, $d_{flt}[n]$, $d_1[3]$, $d_2[1]$, $d_k[3]$, $\text{expr}_1=a_0$, $\text{expr}_2=a_2$, r/w , $t_1\text{\_write}$, $t_2\text{\_write}$, $t_1\text{\_dflt}$, $t_2\text{\_d}_k$, t_{NG} , t_G , t_{T_0} , $t_{T_0\text{Subnet}}$ e **EXEC** são binários: Na marcação inicial, presente na figura 12.16, os lugares $d_{flt}[2]$, $d_{flt}[n]$, $d_1[3]$, $d_2[1]$, $\text{expr}_1=a_0$, $\text{expr}_2=a_2$, r/w , $t_1\text{\_dflt}$, $t_2\text{\_d}_k$, t_G (t_{NG} no modelo JARP) e **EXEC** possuem 1 *token* cada. Assim, para serem qualificados como binários, os demais lugares da lista recebem 1 *token* em alguma classe, conforme pode ser visto adiante, na descrição do grafo de classes da rede;
- v) A rede não é estritamente conservativa: Três grupos de lugares são conservativos para 1 *token*, quais sejam, $\{r/w, t_1\text{\_write}, t_2\text{\_write}\}$, $\{t_{T_0}, t_{T_0\text{Subnet}}\}$ e $\{t_G, t_{NG}\}$. O grupo de lugares $\{\text{expr}_1=a_i, \text{expr}_2=a_j \mid i, j = 1, \dots, n-1\}$ é conservativo para 2 *tokens* (a menos que a subrede representada pelo lugar $t_{T_0\text{Subnet}}$ altere a marcação de algum destes lugares). Ao contrário do modelo da figura 12.15, a rede não é conservativa para o grupo de lugares $\{d_{flt}[j], d_i[j] \mid i = 1, \dots, k; j = 1, \dots, n\}$, uma vez que, imediatamente após o disparo de uma das **transições de esvaziamento** (as com rótulo iniciado por c_1 ou c_2), ocorrerá o decréscimo de um *token* no grupo, o qual somente será repostado após o disparo de uma das **transições de escrita** (rótulo iniciado por t_1 ou t_2). Finalmente, a rede não é conservativa para grupos envolvendo os demais lugares;
- vi) Não há transições k -sensíveis: Fato compatível com a figura 12.16;
- vii) A rede não é viva: Esta análise, em conjunto com as três seguintes, reflete o fato da transição t poder ser disparada somente uma vez, para a atualização de duas células específicas, o que está de acordo com a porção da RPSD mostrada na figura 12.16, na qual não há o reabastecimento dos contêineres de *token* t_1 e t_2 (através do disparo de uma transição distinta de t);
- viii) As transições t_{SetG} e t_{ResetG} são vivas;
- ix) As transições $c_1d_2[1]$, $c_2d_1[3]$, $\text{clr}_1[1]$, $\text{clr}_2[3]$, $t_1d_{flt}[1]$, $t_2d_k[3]$, $t_d_{flt}d_ka_0a_2$, t_{SetG} , t_{ResetG} , t_{RunT_0} e t_{End} são quase-vivas: O disparo destas transições é comentado adiante;
- x) As demais transições não disparam durante a simulação: Este fato se deve ao conteúdo das células de armazenamento de P , ao valor das expressões utilizadas na *expressão de escrita* associada ao ramo e , também, ao não reabastecimento dos contêineres de *token* t_1 e t_2 ;

- xi) A rede não pode retornar ao estado original C_0 : Esta análise está em conformidade com a RPSD da figura 12.16, na qual inexistem transições capazes de fornecer dados para os contêineres t_1 e t_2 ;
- xii) O estado C_{11} inicia a execução repetitiva (*livelock*): Conforme pode-se constatar no grafo de classes (figura II.23), ao atingir o estado C_{11} , a rede ficará em *loop*, alternando entre os estados C_{11} e C_{12} (disparo alternado das transições t_ResetG e t_SetG); e
- xiii) A rede não entra em *deadlock*: Considerando-se somente a figura 12.16, a rede deveria ficar bloqueada após a atualização da *célula de armazenamento a₂*, em **P**. Entretanto, este bloqueio é removido com o acréscimo da transição t_ResetG (presente no modelo **JARP**), a qual possibilita a existência do *loop* descrito no item anterior.

```

State Enumeration : net R12_16 (13 reachable states).

Verified properties:
*-----*
Net under analysis is binary.
  Null places (M = 0): {clr1I2I, clr1I3I, clr1InI, clr2I1I, clr2I2I,
                        clr2InI, clr1OKI2I, clr1OKI3I, clr1OKInI, clr2OKI1I,
                        clr2OKI2I, clr2OKInI, dfltI3I, d1I1I, d1I2I, d1InI,
                        d2I2I, d2I3I, d2InI, dkI1I, dkI2I, dkInI, expr1_a1,
                        expr1_a2, expr1_anminus1, expr2_a0, expr2_a1,
                        expr2_anminus1, t1_d1ext, t1_d2ext, t1_dkext,
                        t2_dfltext, t2_d1ext, t2_d2ext}

  Binary places      : {clr1I1I, clr2I3I, clr1OKI1I, clr2OKI3I, dfltI1I,
                        dfltI2I, dfltInI, d1I3I, d2I1I, dkI3I, expr1_a0,
                        expr2_a2, rw, t1_write, t2_write, t1_dfltext,
                        t2_dkext, t_G, t_NG, t_TO, t_TOSubnet, EXEC}

  k-Bounded places  : {}
  Unbounded places  : {}

Net under analysis is not strictly conservative.

Multi-enabled Tr.: {}

Net under analysis is not live.
  Live Tr.          : {t_SetG, t_ResetG}
  "Almost-live" Tr.: {c1d2I1I, c2d1I3I, clr1I1I_0, clr2I3I_0, t1dfltI1I,
                        t2dkI3I, t_dfltdk02, t_SetG, t_ResetG, t_RunTO, t_End,
                        }
  Non-fired Tr.     : {c1dfltI1I, c1dfltI2I, c1dfltI3I, c1dfltInI, c1d1I1I,
                        c1d1I2I, c1d1I3I, c1d1InI, c1d2I2I, c1d2I3I, c1d2InI,
                        c1dkI1I, c1dkI2I, c1dkI3I, c1dkInI, c2dfltI1I,
                        c2dfltI2I, c2dfltI3I, c2dfltInI, c2d1I1I, c2d1I2I,
                        c2d1InI, c2d2I1I, c2d2I2I, c2d2I3I, c2d2InI, c2dkI1I,
                        c2dkI2I, c2dkI3I, c2dkInI, clr1I2I_0, clr1I3I_0,
                        clr1InI_0, clr2I1I_0, clr2I2I_0, clr2InI_0, t1dfltI2I,
                        t1dfltI3I, t1dfltInI, t1d1I1I, t1d1I2I, t1d1I3I,
                        t1d1InI, t1d2I1I, t1d2I2I, t1d2I3I, t1d2InI, t1dkI1I,
                        t1dkI2I, t1dkI3I, t1dkInI, t2dfltI1I, t2dfltI2I,
                        t2dfltI3I, t2dfltInI, t2d1I1I, t2d1I2I, t2d1I3I,
                        t2d1InI, t2d2I1I, t2d2I2I, t2d2I3I, t2d2InI, t2dkI1I,
                        t2dkI2I, t2dkInI, t_dfltd200, t_dfltdk00, t_dfltd202,
                        t_dfltd220, t_dfltdk20, t_dfltd222, t_dfltdk22,
                        t_d1d200, t_d1d202, t_d1d220, t_d1d222, t_d1dk00,
                        t_d1dk02, t_d1dk20, t_d1dk22}

Net never can go back to M0.

States that start live-locks: C11

No deadlocks detected.
*-----*

```

Listagem II.8. Análise de propriedades para a rede da figura II.22.

O grafo de classes da rede correspondente à figura 12.16, mostrado na figura II.23, apresenta natureza linear. Desde C_0 , tem-se a seguinte seqüência de disparo de transições:

- i) t_SetG : representando a atribuição de *verdadeiro* à guarda de t ;
- ii) $t\_dfld_k a_0 a_2$: representando o início do disparo de t , através da remoção dos *tokens* presentes em r/w (sinalizando que P está realizando uma operação) e $EXEC$ (denotando que t tem *acesso exclusivo* a P), assim como da inserção de um *token* em t_1_write (habilitando o processo de atualização da célula a_0). O rótulo $t\_dfld_k a_0 a_2$ indica que o disparo de t promoverá a atualização das *células de armazenamento* a_0 e a_2 , pertencentes a P , para os valores $dfld$ e d_k , respectivamente;
- iii) $clr_1[1]$: representando o início da limpeza (*reset*) da célula a_0 ;
- iv) $c_1 d_2[1]$: representando a limpeza da célula a_0 , removendo o seu conteúdo d_2 (através do esvaziamento do lugar $d_2[1]$);
- v) $t_1\_dfld[1]$: representando as seguintes ações: efetivação da atualização da primeira *célula de armazenamento* (célula a_0), cujo novo conteúdo ($dfld$) é copiado a partir do contêiner t_1 (através da remoção do *token* presente em $t_1\_dfldext$, assim como da inserção de um *token* em $dfld[1]$); e habilitação da atualização a ser feita a partir dos valores armazenados em t_2 e $expr_2$ (através da inserção de um *token* em t_2_write);
- vi) $clr_2[3]$: representando o início da limpeza (*reset*) da célula a_2 ;
- vii) $c_2 d_1[3]$: representando a limpeza da célula a_2 , removendo o seu conteúdo d_1 (através do esvaziamento do lugar $d_1[3]$);
- viii) $t_2\_d_k[3]$: representando as seguintes ações: efetivação da atualização da terceira *célula de armazenamento* (célula a_0), cujo novo conteúdo (d_k) é copiado a partir do contêiner t_2 (através da remoção do *token* presente em $t_2\_d_kext$, assim como da inserção de um *token* em $d_k[3]$); e liberação do lugar P para uma nova operação de leitura/escrita (através da inserção de um *token* em r/w);
- ix) t_RunT_O : representando o início da execução das *operações* associadas à transição t ;
- x) t_End : representando o término da execução das *operações* associadas à transição t ;
- xi) t_ResetG : representando a atribuição de *falso* à guarda de t ; e
- xii) t_SetG / t_ResetG : representando a alternância de valores à guarda de t (devido à impossibilidade de ocorrência de outras ações).

Os itens de ii) a x) representam o disparo de t , responsável pela atualização das *células de armazenamento* a_0 e a_2 , cujos novos valores encontram-se, no estado inicial, armazenados nos contêineres de *token* t_1 e t_2 . Antes de cada atualização, é realizada a remoção do conteúdo presente na célula correspondente, de forma a garantir as convenções adotadas para a representação de lugares de armazenamento (seção 12.2.2), as quais exigem a presença de um único *token* em cada uma das fileiras representantes das células de armazenamento de um lugar RPSD do tipo II. Encerradas as duas atualizações, são realizadas as *operações* associadas a t . Somente após a execução destas operações, ocorrerá a restituição de um *token* ao lugar $EXEC$ (associado a P), denotando o encerramento do disparo da transição t .

Tal qual para os demais modelos deste apêndice, outras marcações iniciais foram testadas para a rede analisada, gerando resultados análogos aos apresentados. Desta forma, pelo exposto anteriormente, a RPTM presente na figura 12.16 é válida, uma vez que apresenta funcionamento compatível com as convenções e definições adotadas em sua construção, fornecidas nas seções 12.2.2 e 12.3.3. Portanto, a RPSD associada, presente na figura 12.16.a, também é válida.

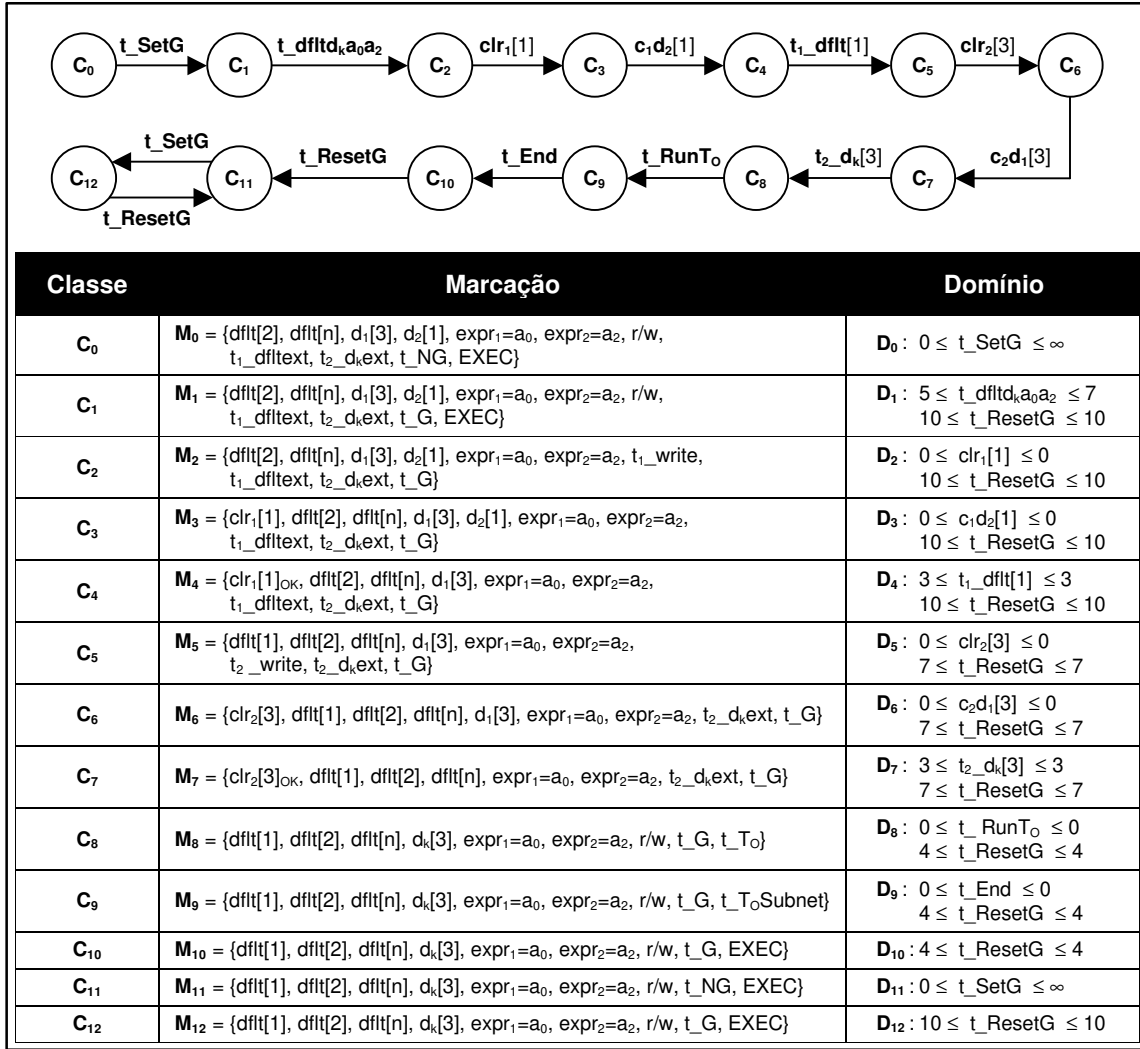


Figura II.23. Grafo de classes para a rede das figuras 12.16 / II.22.

A figura II.24 mostra o modelo **JARP** correlato à rede da figura 12.17, na qual um lugar funcional **P** é conectado a uma transição **t**, através de um *ramo de entrada coletor*, capaz de remover dois resultados de **P** de uma só vez, armazenando-os nos contêineres **t₁** e **t₂**. Comparando-se ambas as figuras, observa-se, na figura II.24, o acréscimo do lugar **t_{NG}** e das transições **t_{SetG}** e **t_{ResetG}**, necessários à simulação da guarda de **t**. Além disso, foram realizadas simplificações nos *repositórios* e no *comportamento B(P)*, para não sobrecarregar a rede da figura II.24. Estas simplificações não comprometem a simulação, uma vez que RPSD permite grande flexibilidade, tanto na definição dos tipos utilizados nos repositórios, quanto na definição da subrede descritora de um comportamento. No entanto, preservou-se a existência de dois argumentos de entrada e a geração de dois resultados de saída, tal como ocorre na rede da figura 12.17. Neste contexto, os *repositórios* de *entrada* e de *saída* de **P** foram alterados, na figura II.24, respectivamente, dos tipos **A** e **B** para **X** e **Y**, e dos tipos **C** e **D** para **F** e **G**, onde **X** = {**x₁**}, **Y** = {**y₁**}, **F** = {**f₁**} e **G** = {**g₁**}. Para acompanhar esta alteração, as redes representantes das *instâncias* do *comportamento B(P)* foram reduzidas, cada qual, a uma só transição **Bi**, conectada aos lugares **positivos Aix, Aiy, Cif e Dig**, e seus **negativos** correspondentes, **NAix, NAiy, NCif e NDig** (**i** = 1, 2, 3). Os rótulos destes lugares foram simplificados para não sobrecarregar a figura II.24. Entretanto, para padronizá-los com os rótulos da figura 12.17, estes lugares seriam rotulados por **x_{1Bi}, y_{1Bi}, f_{1Bi} e g_{1Bi}** – os **positivos**,


```

Net R12_17 ;

Nodes
  A1x : place ;           //  $x_{1B1}$ 
  A2x : place ;           //  $x_{1B2}$ 
  A3x : place ;           //  $x_{1B3}$ 
  A1y : place ;           //  $y_{1B1}$ 
  A2y : place ;           //  $y_{1B2}$ 
  A3y : place ;           //  $y_{1B3}$ 
  B1Active : place ;      //  $B_1Active$ 
  B2Active : place ;
  B3Active : place ;
  B1Clrd : place ;        //  $B_1Clrd$ 
  B2Clrd : place ;
  B3Clrd : place ;
  B1Idle : place (1) ;     //  $B_1Idle$ 
  B2Idle : place (1) ;
  B3Idle : place (1) ;
  ClearB1 : place ;       //  $ClearB_1$ 
  ClearB2 : place ;
  ClearB3 : place ;
  C1f : place ;           //  $f_{1B1}$ 
  C2f : place ;           //  $f_{1B2}$ 
  C3f : place ;           //  $f_{1B3}$ 
  C1g : place ;           //  $g_{1B1}$ 
  C2g : place ;           //  $g_{1B2}$ 
  C3g : place ;           //  $g_{1B3}$ 
  ENABLE : place (1) ;
  EXEC : place (1) ;
  ExecS1B1 : place ;      //  $ExecS_1B_1$ 
  ExecS1B2 : place ;
  ExecS1B3 : place ;
  ExecS2B1 : place ;
  ExecS2B2 : place ;
  ExecS2B3 : place ;
  ExecS3B1 : place ;
  ExecS3B2 : place ;
  ExecS3B3 : place ;
  fullI1I : place (1) ;   //  $full[1]$ 
  fullI2I : place (1) ;
  flext : place ;         //  $f_{1ext}$ 
  flI1_1I : place ;      //  $f_1[1,1]$ 
  flI2_1I : place ;
  flI3_1I : place ;
  glext : place ;         //  $g_{1ext}$ 
  glI1_2I : place ;      //  $g_1[1,2]$ 
  glI2_2I : place ;
  glI3_2I : place ;
  insI1_1I : place ;      //  $ins[1,1]$ 
  insI1_2I : place ;
  insI2_1I : place ;
  insI2_2I : place ;
  insI3_1I : place ;
  insI3_2I : place ;
  NA1x : place (1) ;      //  $Nx_{1B1}$ 
  NA2x : place (1) ;      //  $Nx_{1B2}$ 
  NA3x : place (1) ;      //  $Nx_{1B3}$ 
  NA1y : place (1) ;      //  $Ny_{1B1}$ 
  NA2y : place (1) ;      //  $Ny_{1B2}$ 
  NA3y : place (1) ;      //  $Ny_{1B3}$ 
  NC1f : place (1) ;      //  $Nf_{1B1}$ 
  NC2f : place (1) ;      //  $Nf_{1B2}$ 
  NC3f : place (1) ;      //  $Nf_{1B3}$ 

```

Listagem II.9. Modelo ARP da figura II.24.

```

NC1g : place (1) ; // Ng1B1
NC2g : place (1) ; // Ng1B2
NC3g : place (1) ; // Ng1B3
NfullI1I : place ; // Nfull[1]
NfullI2I : place ;
Nf1I1_1I : place (1) ; // Nf1[1,1]
Nf1I2_1I : place (1) ;
Nf1I3_1I : place (1) ;
Ng1I1_2I : place (1) ; // Ng1[1,2]
Ng1I2_2I : place (1) ;
Ng1I3_2I : place (1) ;
NinsI1_1I : place (1) ; // Nins[1,1]
NinsI1_2I : place (1) ;
NinsI2_1I : place (1) ;
NinsI2_2I : place (1) ;
NinsI3_1I : place (1) ;
NinsI3_2I : place (1) ;
NremI1I : place (1) ; // Nrem[1]
NremI2I : place (1) ;
NremI3I : place (1) ;
NShifting : place (1) ;
Nx1I1_1I : place ; // Nx1[1,1]
Nx1I2_1I : place ;
Nx1I3_1I : place ;
Ny1I1_2I : place ;
Ny1I2_2I : place ;
Ny1I3_2I : place ;
remI1I : place ; // rem[1]
remI2I : place ;
remI3I : place ;
ResetFull : place ;
Reset_OK : place ; // ResetOK
SampleI1I : place ; // Sample[1]
SampleI2I : place ;
SampleI3I : place ;
Shifting : place ;
Shift2to1 : place ;
Shift2to1_OK : place ; // Shift2to1OK
Shift3to2 : place ;
Shift3to2_OK : place ;
t_G : place ;
t_NG : place (1) ;
t_TO : place ; // t_T0
t_TOSubnet : place ; // t_T0Subnet
x1I1_1I : place (1) ; // x1[1,1]
x1I2_1I : place (1) ;
x1I3_1I : place (1) ;
y1I1_2I : place (1) ;
y1I2_2I : place (1) ;
y1I3_2I : place (1) ;

A1x_0 : transition [0,0] ; // x1B1
A2x_0 : transition [0,0] ; // x1B2
A3x_0 : transition [0,0] ; // x1B3
A1y_0 : transition [0,0] ; // y1B1
A2y_0 : transition [0,0] ; // y1B2
A3y_0 : transition [0,0] ; // y1B3
B1 : transition [10,20] ; // BB1
B2 : transition [10,20] ; // BB2
B3 : transition [10,20] ; // BB3
B1S2toS1 : transition [0,0] ; // B1S2toS1
B1S3toS2 : transition [0,0] ;
B2S2toS1 : transition [0,0] ;
B2S3toS2 : transition [0,0] ;

```

Continuação da listagem II.9.

```

B3S2toS1 : transition [0,0] ;
B3S3toS2 : transition [0,0] ;
C1f_0 : transition [0,0] ; //  $f_{1B1}$ 
C2f_0 : transition [0,0] ; //  $f_{1B2}$ 
C3f_0 : transition [0,0] ; //  $f_{1B3}$ 
C1g_0 : transition [0,0] ; //  $g_{1B1}$ 
C2g_0 : transition [0,0] ; //  $g_{1B2}$ 
C3g_0 : transition [0,0] ; //  $g_{1B3}$ 
EndShifting : transition [0,0] ;
NA1x_0 : transition [0,0] ; //  $Nx_{1B1}$ 
NA2x_0 : transition [0,0] ; //  $Nx_{1B2}$ 
NA3x_0 : transition [0,0] ; //  $Nx_{1B3}$ 
NA1y_0 : transition [0,0] ; //  $Ny_{1B1}$ 
NA2y_0 : transition [0,0] ; //  $Ny_{1B2}$ 
NA3y_0 : transition [0,0] ; //  $Ny_{1B3}$ 
NC1f_0 : transition [0,0] ; //  $Nf_{1B1}$ 
NC2f_0 : transition [0,0] ; //  $Nf_{1B2}$ 
NC3f_0 : transition [0,0] ; //  $Nf_{1B3}$ 
NC1g_0 : transition [0,0] ; //  $Ng_{1B1}$ 
NC2g_0 : transition [0,0] ; //  $Ng_{1B2}$ 
NC3g_0 : transition [0,0] ; //  $Ng_{1B3}$ 
Reset : transition [0,0] ;
RfullI1I : transition [0,0] ; //  $Rfull[1]$ 
RfullI2I : transition [0,0] ;
RNfullI1I : transition [0,0] ; //  $RNfull[1]$ 
RNfullI2I : transition [0,0] ;
RstB1 : transition [0,0] ; //  $RstB_1$ 
RstB2 : transition [0,0] ;
RstB3 : transition [0,0] ;
RunS1B1 : transition [0,0] ; //  $RunS_1B_1(x_1; y_1)$  - figura 12.12
RunS1B2 : transition [0,0] ; //  $RunS_1B_2(x_1; y_1)$ 
RunS1B3 : transition [0,0] ; //  $RunS_1B_3(x_1; y_1)$ 
RunS2B1 : transition [0,0] ; //  $RunS_2B_1(x_1; y_1)$ 
RunS2B2 : transition [0,0] ; //  $RunS_2B_2(x_1; y_1)$ 
RunS2B3 : transition [0,0] ; //  $RunS_2B_3(x_1; y_1)$ 
RunS3B1 : transition [0,0] ; //  $RunS_3B_1(x_1; y_1)$ 
RunS3B2 : transition [0,0] ; //  $RunS_3B_2(x_1; y_1)$ 
RunS3B3 : transition [0,0] ; //  $RunS_3B_3(x_1; y_1)$ 
Sf1I2_1I : transition [0,0] ; //  $Sf_1[2,1]$ 
Sf1I3_1I : transition [0,0] ;
Sg1I2_2I : transition [0,0] ;
Sg1I3_2I : transition [0,0] ;
Shift3to2_0 : transition [0,0] ; //  $Shift3to2$ 
SinsI2_1I : transition [0,0] ; //  $Sins[2,1]$ 
SinsI2_2I : transition [0,0] ;
SinsI3_1I : transition [0,0] ;
SinsI3_2I : transition [0,0] ;
SNf1I2_1I : transition [0,0] ; //  $SNf_1[2,1]$ 
SNf1I3_1I : transition [0,0] ;
Sng1I2_2I : transition [0,0] ;
Sng1I3_2I : transition [0,0] ;
SNinsI2_1I : transition [0,0] ; //  $SNins[2,1]$ 
SNinsI2_2I : transition [0,0] ;
SNinsI3_1I : transition [0,0] ;
SNinsI3_2I : transition [0,0] ;
SNremI2I : transition [0,0] ; //  $SNrem[2]$ 
SNremI3I : transition [0,0] ;
SNx1I2_1I : transition [0,0] ; //  $SNx_1[2,1]$ 
SNx1I3_1I : transition [0,0] ;
SNy1I2_2I : transition [0,0] ;
SNy1I3_2I : transition [0,0] ;
SremI2I : transition [0,0] ; //  $Srem[2]$ 
SremI3I : transition [0,0] ;

```

Continuação da listagem II.9.

```

StoreB1O1 : transition [0,0] ;           // StoreB1O1(f1;g1) - figura 12.12
StoreB1O2 : transition [0,0] ;           // StoreB1O2(f1;g1)
StoreB1O3 : transition [0,0] ;           // StoreB1O3(f1;g1)
StoreB2O1 : transition [0,0] ;           // StoreB2O1(f1;g1)
StoreB2O2 : transition [0,0] ;           // StoreB2O2(f1;g1)
StoreB2O3 : transition [0,0] ;           // StoreB2O3(f1;g1)
StoreB3O1 : transition [0,0] ;           // StoreB3O1(f1;g1)
StoreB3O2 : transition [0,0] ;           // StoreB3O2(f1;g1)
StoreB3O3 : transition [0,0] ;           // StoreB3O3(f1;g1)
Sx1I2_1I : transition [0,0] ;           // Sx1[2,1]
Sx1I3_1I : transition [0,0] ;
SylI2_2I : transition [0,0] ;
SylI3_2I : transition [0,0] ;
tB1O1 : transition [5,7] ;               // tB1O1(x1;y1)
tB2O1 : transition [5,7] ;               // tB2O1(x1;y1)
tB3O1 : transition [5,7] ;               // tB3O1(x1;y1)
tB1O2 : transition [5,7] ;               // tB1O2(x1;y1)
tB2O2 : transition [5,7] ;               // tB2O2(x1;y1)
tB3O2 : transition [5,7] ;               // tB3O2(x1;y1)
tB1O3 : transition [5,7] ;               // tB1O3(x1;y1)
tB2O3 : transition [5,7] ;               // tB2O3(x1;y1)
tB3O3 : transition [5,7] ;               // tB3O3(x1;y1)
t_End : transition [0,0] ;
t_ResetG : transition ;
t_RunTO : transition [0,0] ;             // t_RunT0
t_SetG : transition ;

Structure
A1x_0 : (ClearB1, A1x) , (B1Clrd) ;
A2x_0 : (ClearB2, A2x) , (B2Clrd) ;
A3x_0 : (ClearB3, A3x) , (B3Clrd) ;
A1y_0 : (ClearB1, A1y) , (B1Clrd) ;
A2y_0 : (ClearB2, A2y) , (B2Clrd) ;
A3y_0 : (ClearB3, A3y) , (B3Clrd) ;
B1 : (A1x, A1y, NC1f, NC1g, ENABLE) , (C1f, C1g, NA1x, NA1y, ENABLE) ;
B2 : (A2x, A2y, NC2f, NC2g, ENABLE) , (C2f, C2g, NA2x, NA2y, ENABLE) ;
B3 : (A3x, A3y, NC3f, NC3g, ENABLE) , (C3f, C3g, NA3x, NA3y, ENABLE) ;
B1S2toS1 : (4 * Shift2to1, SampleI2I, 3 * Shift2to1_OK, ExecS2B1) ,
(7 * Shift2to1_OK, SampleI1I, ExecS1B1) ;
B1S3toS2 : (4 * Shift3to2, 3 * Shift3to2_OK, SampleI3I, ExecS3B1) ,
(7 * Shift3to2_OK, SampleI2I, ExecS2B1) ;
B2S2toS1 : (ExecS2B2, SampleI2I, 4 * Shift2to1, 3 * Shift2to1_OK) ,
(ExecS1B2, SampleI1I, 7 * Shift2to1_OK) ;
B2S3toS2 : (ExecS3B2, SampleI3I, 4 * Shift3to2, 3 * Shift3to2_OK) ,
(ExecS2B2, SampleI2I, 7 * Shift3to2_OK) ;
B3S2toS1 : (ExecS2B3, SampleI2I, 4 * Shift2to1, 3 * Shift2to1_OK) ,
(ExecS1B3, SampleI1I, 7 * Shift2to1_OK) ;
B3S3toS2 : (ExecS3B3, SampleI3I, 4 * Shift3to2, 3 * Shift3to2_OK) ,
(ExecS2B3, SampleI2I, 7 * Shift3to2_OK) ;
C1f_0 : (ClearB1, C1f) , (B1Clrd) ;
C2f_0 : (ClearB2, C2f) , (B2Clrd) ;
C3f_0 : (ClearB3, C3f) , (B3Clrd) ;
C1g_0 : (ClearB1, C1g) , (B1Clrd) ;
C2g_0 : (ClearB2, C2g) , (B2Clrd) ;
C3g_0 : (ClearB3, C3g) , (B3Clrd) ;
EndShifting : (Shifting, 2 * Reset_OK) ,
(NfullI1I, NfullI2I, NShifting, t_TO) ;
NA1x_0 : (ClearB1, NA1x) , (B1Clrd) ;
NA2x_0 : (ClearB2, NA2x) , (B2Clrd) ;
NA3x_0 : (ClearB3, NA3x) , (B3Clrd) ;
NA1y_0 : (ClearB1, NA1y) , (B1Clrd) ;
NA2y_0 : (ClearB2, NA2y) , (B2Clrd) ;
NA3y_0 : (ClearB3, NA3y) , (B3Clrd) ;

```

Continuação da listagem II.9.

```

NC1f_0 : (ClearB1, NC1f) , (B1Clrd) ;
NC2f_0 : (ClearB2, NC2f) , (B2Clrd) ;
NC3f_0 : (ClearB3, NC3f) , (B3Clrd) ;
NC1g_0 : (ClearB1, NC1g) , (B1Clrd) ;
NC2g_0 : (ClearB2, NC2g) , (B2Clrd) ;
NC3g_0 : (ClearB3, NC3g) , (B3Clrd) ;
Reset : (7 * Shift3to2_OK) ,
        (Nx1I3_1I, Ny1I3_2I, 2 * ResetFull, Nf1I3_1I, Ng1I3_2I, NremI3I) ;
RfullI1I : (ResetFull, fullI1I) , (Reset_OK, insI3_1I) ;
RfullI2I : (ResetFull, fullI2I) , (Reset_OK, insI3_2I) ;
RNfullI1I : (NfullI1I, ResetFull) , (Reset_OK, NinsI3_1I) ;
RNfullI2I : (ResetFull, NfullI2I) , (Reset_OK, NinsI3_2I) ;
RstB1 : (4 * B1Clrd) , (B1Idle, NA1x, NA1y, NC1f, NC1g) ;
RstB2 : (4 * B2Clrd) , (B2Idle, NA2x, NA2y, NC2f, NC2g) ;
RstB3 : (4 * B3Clrd) , (B3Idle, NA3x, NA3y, NC3f, NC3g) ;
RunS1B1 : (x1I1_1I, y1I1_2I, B1Idle, NinsI1_1I, NinsI1_2I, NA1x, NA1y) ,
        (A1x, A1y, B1Active, SampleI1I, ExecS1B1) ;
RunS1B2 : (x1I1_1I, y1I1_2I, B2Idle, B1Active,
        NinsI1_1I, NinsI1_2I, NA2x, NA2y) ,
        (A2x, A2y, B1Active, B2Active, SampleI1I, ExecS1B2) ;
RunS1B3 : (x1I1_1I, y1I1_2I, B3Idle, B1Active, B2Active,
        NinsI1_1I, NinsI1_2I, NA3x, NA3y) ,
        (A3x, A3y, B1Active, B2Active, B3Active, SampleI1I, ExecS1B3) ;
RunS2B1 : (x1I2_1I, y1I2_2I, B1Idle, SampleI1I,
        NinsI2_1I, NinsI2_2I, NA1x, NA1y) ,
        (A1x, A1y, B1Active, SampleI1I, SampleI2I, ExecS2B1) ;
RunS2B2 : (x1I2_1I, y1I2_2I, B2Idle, B1Active,
        SampleI1I, NinsI2_1I, NinsI2_2I, NA2x, NA2y) ,
        (A2x, A2y, B1Active, B2Active, SampleI1I, SampleI2I, ExecS2B2) ;
RunS2B3 : (x1I2_1I, y1I2_2I, B3Idle, B1Active, B2Active,
        SampleI1I, NinsI2_1I, NinsI2_2I, NA3x, NA3y) ,
        (A3x, A3y, B1Active, B2Active, B3Active,
        SampleI1I, SampleI2I, ExecS2B3) ;
RunS3B1 : (x1I3_1I, y1I3_2I, B1Idle, SampleI2I,
        NinsI3_1I, NinsI3_2I, NA1x, NA1y) ,
        (A1x, A1y, B1Active, SampleI2I, SampleI3I, ExecS3B1) ;
RunS3B2 : (x1I3_1I, y1I3_2I, B2Idle, B1Active,
        SampleI2I, NinsI3_1I, NinsI3_2I, NA2x, NA2y) ,
        (A2x, A2y, B1Active, B2Active, SampleI2I, SampleI3I, ExecS3B2) ;
RunS3B3 : (x1I3_1I, y1I3_2I, B3Idle, B1Active, B2Active,
        SampleI2I, NinsI3_1I, NinsI3_2I, NA3x, NA3y) ,
        (A3x, A3y, B1Active, B2Active, B3Active,
        SampleI2I, SampleI3I, ExecS3B3) ;
Sf1I2_1I : (Shift2to1, f1I2_1I) , (Shift2to1_OK, f1I1_1I) ;
Sf1I3_1I : (Shift3to2, f1I3_1I) , (Shift3to2_OK, f1I2_1I) ;
Sg1I2_2I : (Shift2to1, g1I2_2I) , (Shift2to1_OK, g1I1_2I) ;
Sg1I3_2I : (Shift3to2, g1I3_2I) , (Shift3to2_OK, g1I2_2I) ;
Shift3to2_0 : (7 * Shift2to1_OK) , (7 * Shift3to2) ;
SinsI2_1I : (insI2_1I, Shift2to1) , (insI1_1I, Shift2to1_OK) ;
SinsI2_2I : (insI2_2I, Shift2to1) , (insI1_2I, Shift2to1_OK) ;
SinsI3_1I : (insI3_1I, Shift3to2) , (insI2_1I, Shift3to2_OK) ;
SinsI3_2I : (insI3_2I, Shift3to2) , (insI2_2I, Shift3to2_OK) ;
SNf1I2_1I : (Shift2to1, Nf1I2_1I) , (Shift2to1_OK, Nf1I1_1I) ;
SNf1I3_1I : (Shift3to2, Nf1I3_1I) , (Shift3to2_OK, Nf1I2_1I) ;
SNg1I2_2I : (Shift2to1, Ng1I2_2I) , (Shift2to1_OK, Ng1I1_2I) ;
SNg1I3_2I : (Shift3to2, Ng1I3_2I) , (Shift3to2_OK, Ng1I2_2I) ;
SNinsI2_1I : (NinsI2_1I, Shift2to1) , (NinsI1_1I, Shift2to1_OK) ;
SNinsI2_2I : (NinsI2_2I, Shift2to1) , (NinsI1_2I, Shift2to1_OK) ;
SNinsI3_1I : (NinsI3_1I, Shift3to2) , (NinsI2_1I, Shift3to2_OK) ;
SNinsI3_2I : (NinsI3_2I, Shift3to2) , (NinsI2_2I, Shift3to2_OK) ;
SNremI2I : (Shift2to1, NremI2I) , (Shift2to1_OK, NremI1I) ;
SNremI3I : (Shift3to2, NremI3I) , (Shift3to2_OK, NremI2I) ;
SNx1I2_1I : (Nx1I2_1I, Shift2to1) , (Nx1I1_1I, Shift2to1_OK) ;
SNx1I3_1I : (Nx1I3_1I, Shift3to2) , (Nx1I2_1I, Shift3to2_OK) ;

```

Continuação da listagem II.9.


```

SNy1I2_2I : (Ny1I2_2I, Shift2to1) , (Ny1I1_2I, Shift2to1_OK) ;
SNy1I3_2I : (Ny1I3_2I, Shift3to2) , (Ny1I2_2I, Shift3to2_OK) ;
SremI2I : (Shift2to1, remI2I) , (Shift2to1_OK, remI1I) ;
SremI3I : (Shift3to2, remI3I) , (Shift3to2_OK, remI2I) ;
StoreB101 : (C1f, C1g, ExecS1B1, NremI1I, Nf1I1_1I, Ng1I1_2I, NShifting) ,
(f1I1_1I, g1I1_2I, remI1I, ExecS1B1, NShifting, NC1f, NC1g) ;
StoreB102 : (C1f, C1g, ExecS2B1, NremI2I, Nf1I2_1I, Ng1I2_2I, NShifting) ,
(f1I2_1I, g1I2_2I, remI2I, ExecS2B1, NShifting, NC1f, NC1g) ;
StoreB103 : (C1f, C1g, ExecS3B1, NremI3I, Nf1I3_1I, Ng1I3_2I, NShifting) ,
(f1I3_1I, g1I3_2I, remI3I, ExecS3B1, NShifting, NC1f, NC1g) ;
StoreB201 : (C2g, C2f, ExecS1B2, NremI1I, Nf1I1_1I, Ng1I1_2I, NShifting) ,
(g1I1_2I, f1I1_1I, remI1I, ExecS1B2, NShifting, NC2f, NC2g) ;
StoreB202 : (C2g, C2f, ExecS2B2, NremI2I, Nf1I2_1I, Ng1I2_2I, NShifting) ,
(g1I2_2I, f1I2_1I, remI2I, ExecS2B2, NShifting, NC2f, NC2g) ;
StoreB203 : (C2g, C2f, ExecS3B2, NremI3I, Nf1I3_1I, Ng1I3_2I, NShifting) ,
(g1I3_2I, f1I3_1I, remI3I, ExecS3B2, NShifting, NC2f, NC2g) ;
StoreB301 : (C3g, C3f, ExecS1B3, NremI1I, Nf1I1_1I, Ng1I1_2I, NShifting) ,
(f1I1_1I, g1I1_2I, remI1I, ExecS1B3, NShifting, NC3f, NC3g) ;
StoreB302 : (C3g, C3f, ExecS2B3, NremI2I, Nf1I2_1I, Ng1I2_2I, NShifting) ,
(f1I2_1I, g1I2_2I, remI2I, ExecS2B3, NShifting, NC3f, NC3g) ;
StoreB303 : (C3g, C3f, ExecS3B3, NremI3I, Nf1I3_1I, Ng1I3_2I, NShifting) ,
(f1I3_1I, g1I3_2I, remI3I, ExecS3B3, NShifting, NC3f, NC3g) ;
Sx1I2_1I : (x1I2_1I, Shift2to1) , (x1I1_1I, Shift2to1_OK) ;
Sx1I3_1I : (x1I3_1I, Shift3to2) , (x1I2_1I, Shift3to2_OK) ;
Sy1I2_2I : (y1I2_2I, Shift2to1) , (y1I1_2I, Shift2to1_OK) ;
Sy1I3_2I : (y1I3_2I, Shift3to2) , (y1I2_2I, Shift3to2_OK) ;
tB101 : (ExecS1B1, SampleI1I, B1Active, remI1I,
f1I1_1I, g1I1_2I, NShifting, t_G, EXEC) ,
(4 * ClearB1, 7 * Shift2to1, Shifting, flect, glect, t_G) ;
tB201 : (ExecS1B2, SampleI1I, B2Active, remI1I,
f1I1_1I, g1I1_2I, NShifting, t_G, EXEC) ,
(4 * ClearB2, 7 * Shift2to1, Shifting, flect, glect, t_G) ;
tB301 : (ExecS1B3, SampleI1I, B3Active, remI1I,
f1I1_1I, g1I1_2I, NShifting, t_G, EXEC) ,
(4 * ClearB3, 7 * Shift2to1, Shifting, flect, glect, t_G) ;
tB102 : (ExecS2B1, SampleI2I, B1Active, remI2I,
f1I2_1I, g1I2_2I, NremI1I, NShifting, t_G, EXEC) ,
(4 * ClearB1, NremI1I, Shifting, 7 * Shift3to2, flect, glect, t_G) ;
tB202 : (ExecS2B2, SampleI2I, B2Active, remI2I,
f1I2_1I, g1I2_2I, NremI1I, NShifting, t_G, EXEC) ,
(4 * ClearB2, NremI1I, Shifting, 7 * Shift3to2, flect, glect, t_G) ;
tB302 : (ExecS2B3, SampleI2I, B3Active, remI2I,
f1I2_1I, g1I2_2I, NremI1I, NShifting, t_G, EXEC) ,
(4 * ClearB3, NremI1I, Shifting, 7 * Shift3to2, flect, glect, t_G) ;
tB103 : (ExecS3B1, SampleI3I, B1Active, remI3I, f1I3_1I, g1I3_2I,
NremI1I, NremI2I, NShifting, t_G, EXEC) ,
(4 * ClearB1, NremI1I, NremI2I, Shifting, 7 * Shift3to2_OK,
flect, glect, t_G) ;
tB203 : (ExecS3B2, SampleI3I, B2Active, remI3I, f1I3_1I, g1I3_2I,
NremI1I, NremI2I, NShifting, t_G, EXEC) ,
(4 * ClearB2, NremI1I, NremI2I, Shifting, 7 * Shift3to2_OK,
flect, glect, t_G) ;
tB303 : (ExecS3B3, SampleI3I, B3Active, remI3I, f1I3_1I, g1I3_2I,
NremI1I, NremI2I, NShifting, t_G, EXEC) ,
(4 * ClearB3, NremI1I, NremI2I, Shifting, 7 * Shift3to2_OK,
flect, glect, t_G) ;
t_End : (t_TOSubnet) , (EXEC) ;
t_ResetG : (t_G) , (t_NG) ;
t_SetG : (t_NG) , (t_G) ;
t_RunTO : (t_TO) , (t_TOSubnet) ;

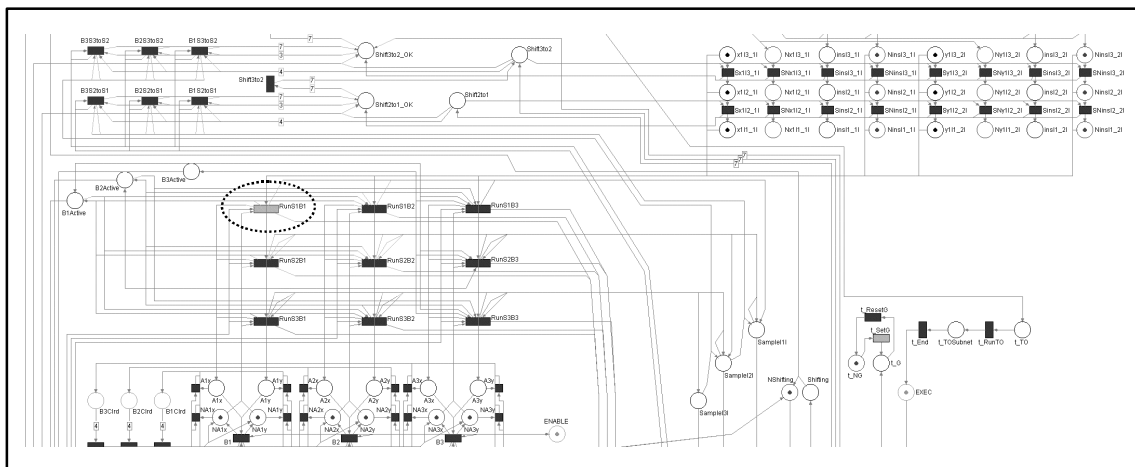
```

endNet.

Continuação da listagem II.9.

Desta forma, adotou-se a **simulação interativa** do programa **JARP**, procurando-se identificar grupos de transições nos quais diferentes seqüências de disparo resultassem em uma mesma classe. A temporização da rede da figura 12.17 facilita o uso da simulação interativa, uma vez que somente algumas transições possuem intervalos diferentes de [0,0], quais sejam, as transições representantes de **t** (as com rótulo iniciado por **t_**), as transições **B1**, **B2** e **B3** (pertencentes às instâncias do *comportamento*) e as transições **t_SetG** e **t_ResetG**, responsáveis pelo valor da guarda de **t**. Para estas transições, ao longo do processo de simulação, faz-se importante considerar diferentes escolhas de disparo, baseando-se em critérios temporais.

Na marcação da figura II.24, o valor da guarda de **t** é *falso* e as duas colunas do *repositório de entrada* estão cheias. Desta forma, há duas transições que podem ser disparadas, quais sejam, **t_SetG** e **RunS₁B₁**, conforme pode-se observar na figura II.25.



Na classe **C₀** (figura II.25), o disparo de **t_SetG** não habilita novas transições. Aliás, **t_SetG** somente habilitará novas transições quando houver o armazenamento de algum resultado no *repositório de saída* (através do disparo de uma das transições **Store**). Além disso, o disparo de **t_SetG** não desabilita outras transições. Desta forma, na classe **C₀**, opta-se pelo disparo de **RunS₁B₁**, o qual gera a classe **C₁**, mostrada na figura II.26.

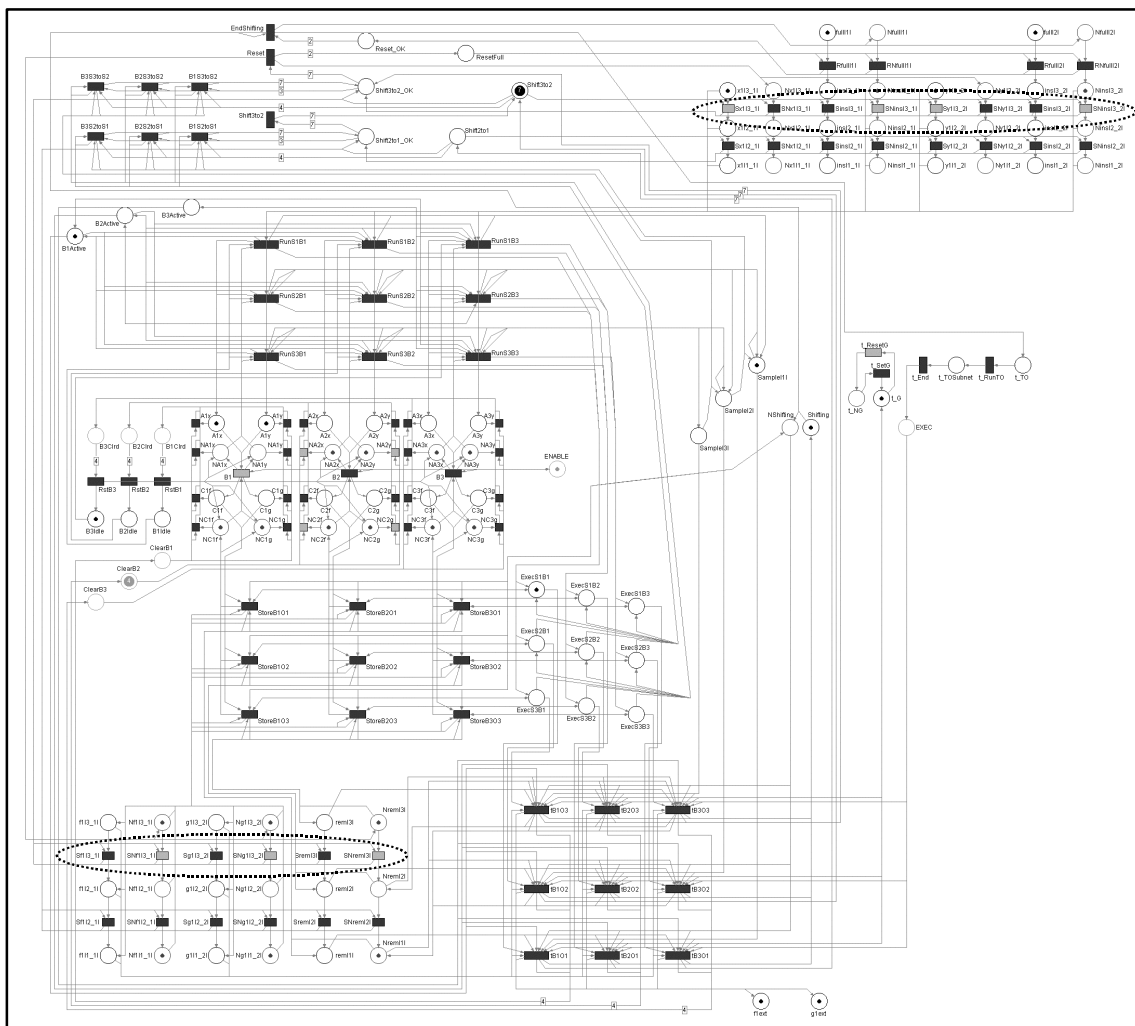


Figura II.31. Classe C₆.

O disparo de $t_{B_2O_2}$, na classe C₅, promove (compare as figuras II.30 e II.31):

- a remoção dos *tokens* presentes na segunda fileira de lugares do **repositório de saída**, representando o repasse dos resultados para um ou mais lugares RPSD conectados na saída de t (não mostrados na figura 12.17). Este repasse é representado, genericamente, pela inserção de *tokens* em f_{1ext} e g_{1ext} (caso os conjuntos F e G possuísem cardinalidade maior do que 1, poder-se-ia adotar outras relações de saída, em função dos argumentos de entrada, pertencentes aos conjuntos X e Y – contudo, esta extensão de cardinalidade dos conjuntos definidores dos **repositórios**, não interferiria no processo de simulação, apenas aumentaria a complexidade do modelo);
- a remoção do *token* presente em $B_2Active$, representando a desativação da instância B_2 (obs.: apesar de desativada, a instância B_2 não poderá ser associada a um novo *sample*, enquanto o lugar B_2Idle permanecer vazio);
- a remoção do *token* presente em $Sample[2]$, representando que o *sample* S_2 foi processado e removido de P ;
- a remoção do *token* presente em $ExecS_2B_2$, representando que a instância B_2 não está mais associada ao *sample* S_2 ;
- a transferência do *token* presente em $NShifting$ para $Shifting$, denotando que a operação de deslocamento de *tokens*, em ambos os **repositórios**, está em curso;
- a remoção do *token* presente em $EXEC$, denotando que t passa a ter *acesso exclusivo* a P ;

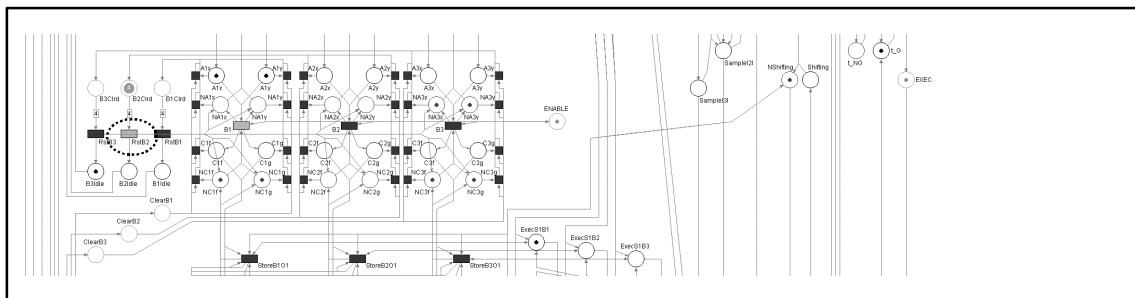


Figura II.39. Classe C_{14} .

Na classe C_{14} , a presença de 4 *tokens* em B_2Clrd habilita a transição $RstB_2$, a qual dispara, inserindo um *token* em cada um dos lugares **negativos** da subrede representante da instância B_2 , além de inserir um *token* no lugar B_2Idle , liberando a instância B_2 para a associação com um novo *sample*. A classe gerada, C_{15} , encontra-se ilustrada na figura II.40. Com a restituição do *token* ao lugar B_2Idle , a rede habilita novamente a admissão de argumentos pelas instâncias (através da habilitação de transições com rótulo inicial **Run**), suspensão desde o disparo da transição representante de **t**, isto é, desde a classe C_5 (figura II.30), na qual o disparo de $t_{B_2O_2}$ desabilitou a transição $RunS_3B_3$.

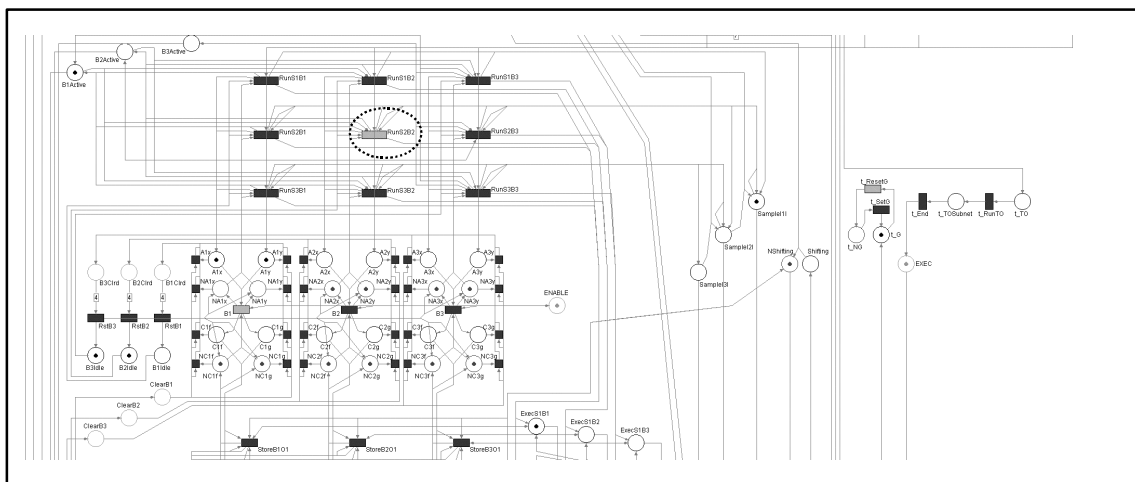


Figura II.40. Classe C_{15} .

No entanto, observa-se na classe C_{15} que, ao invés da transição $RunS_3B_3$ (habilitada na classe C_5), outra transição de admissão de argumentos torna-se habilitada, qual seja, $RunS_2B_2$. Isto se deve à disponibilidade da instância B_2 (a qual não estava disponível em C_5): de acordo com as convenções da seção 12.3.3, o *sample* admitido deverá sempre ser associado à instância disponível de menor número. Além disso, observa-se que o outrora *sample* S_3 (na classe C_5) passou a ser designado por S_2 , após a operação de deslocamento de *tokens* nos *repositórios*, provocada pelo disparo de **t**.

O disparo da transição $RunS_2B_2$, cujo efeito foi visto na descrição da classe C_1 (figura II.26), conduz a rede da classe C_{15} a C_{16} (figura II.41). Nesta nova classe, opções de disparo envolvem as transições **B1** e **B2**. O disparo de **B2** resultaria num processo de simulação semelhante àquele visto a partir da classe C_2 (figura II.27). Assim, opta-se pelo disparo da transição **B1**, representando a geração de resultados pela instância B_1 , o qual faz-se acompanhar do disparo de $StoreB_1O_1$, conduzindo a rede para a classe C_{17} (figura II.42).

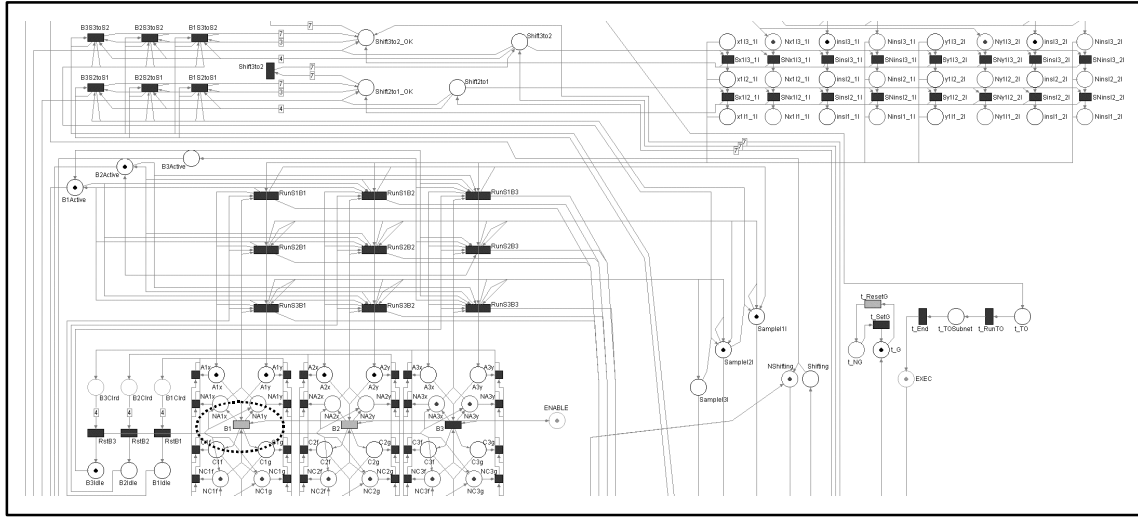


Figura II.41. Classe C₁₆.

Observa-se, na classe C₁₇, a inserção dos resultados do processamento de B_1 na linha inferior do *repositório de saída* (uma vez que a instância B_1 está associada ao *sample* S_1 , o qual refere-se à linha 1 de ambos os *repositórios*). Nesta classe, a transição $t_{B_1O_1}$ fica habilitada, possibilitando o disparo de t para a remoção dos *tokens* presentes na linha 1 do *repositório de saída*.

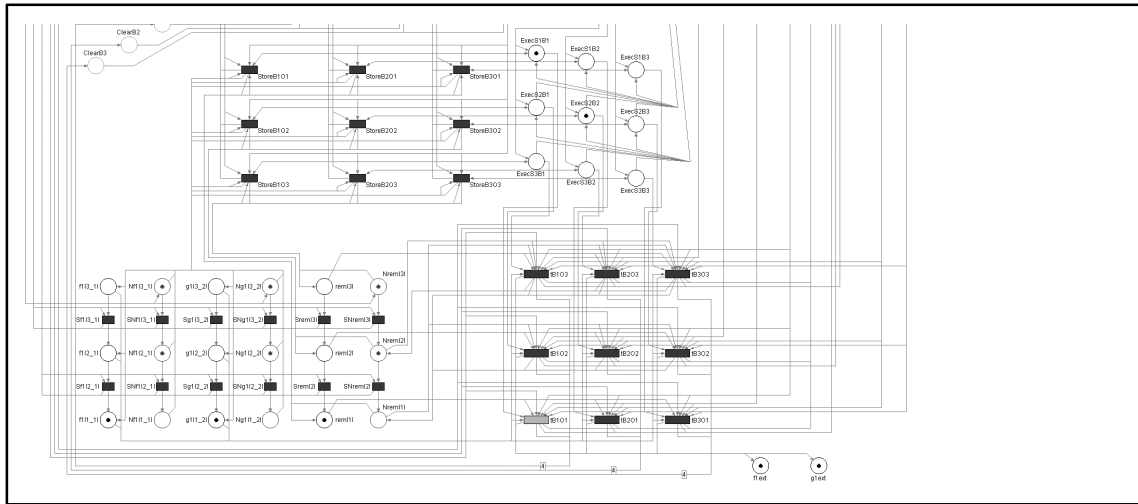
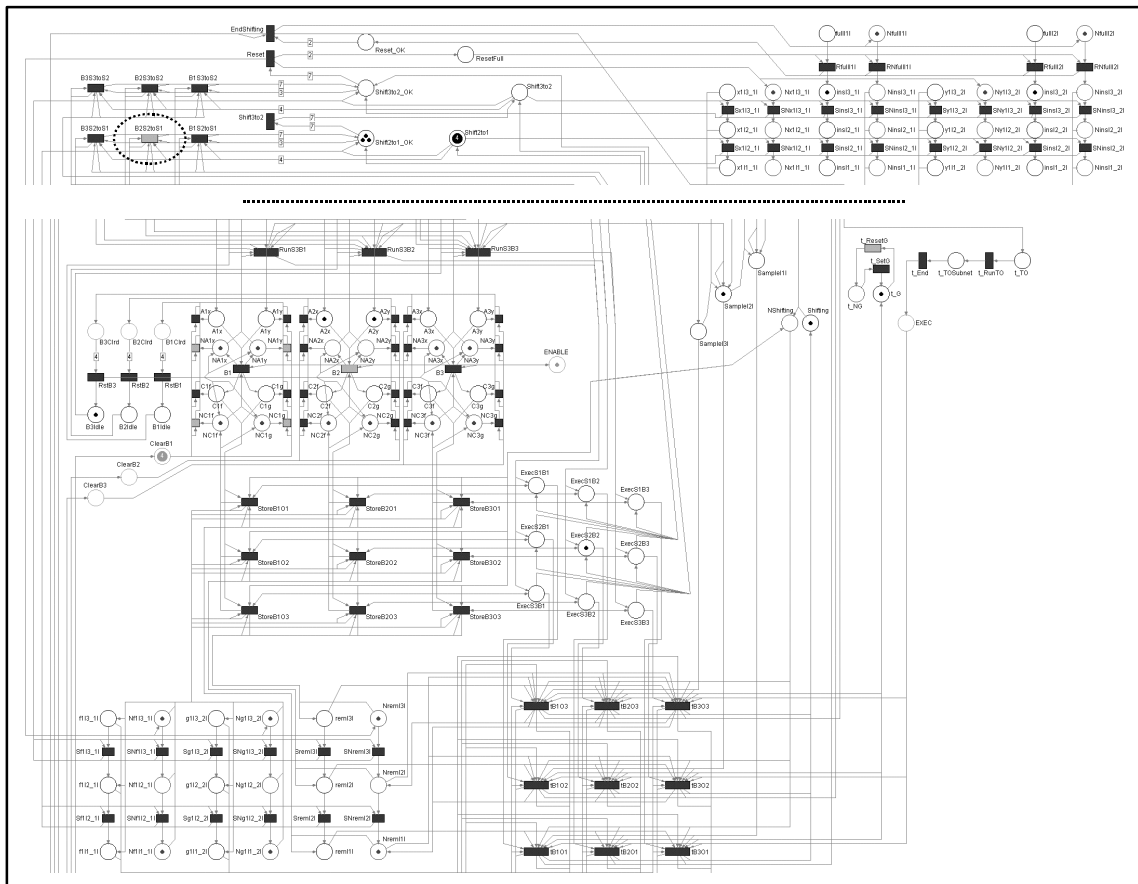


Figura II.42. Classe C₁₇.

Supondo-se o disparo de $t_{B_1O_1}$ na classe C₁₇, a rede passa para a classe C₁₈ (figura II.43), na qual observa-se que o disparo de $t_{B_1O_1}$ tem efeito semelhante ao do disparo de $t_{B_2O_2}$ na classe C₅ (compare as figuras II.30/II.31 com as figuras II.42/II.43). De forma resumida, o disparo de $t_{B_1O_1}$ promove:

- a remoção dos *tokens* presentes na primeira fileira de lugares do *repositório de saída*, representando o repasse dos resultados para um ou mais lugares RPSD conectados na saída de t . Este repasse é representado, genericamente, pela inserção de *tokens* em f_{1ext} e g_{1ext} (observe que estes lugares, na figura II.43, passam a armazenar 2 *tokens* cada, uma vez que não foram utilizadas transições para o consumo de tais *tokens* – esta abordagem serve para evidenciar a quantidade de disparos de t , não interferindo na simulação do modelo);
- a remoção do *token* presente em $B_1Active$, representando a desativação da instância B_1 ;

Após o disparo das transições de deslocamento $SNf_1[2,1]$, $SNg_1[2,2]$ e $SNrem[2]$, obtém-se a marcação ilustrada na classe denominada C_{19} (figura II.44). Observa-se, nesta classe, que as transições disparadas promoveram, além do deslocamento no *repositório de saída*, a transferência de 3 *tokens* do lugar $Shift2to1$ para o lugar $Shift2to1_{OK}$. Estes *tokens*, juntamente com os 4 *tokens* não consumidos de $Shift2to1$ (devido à não realização da operação de deslocamento no *repositório de entrada*), são responsáveis, em parte, pela habilitação da transição $B_2S_2toS_1$ – as outras condições de habilitação desta transição, inerentes ao fato da linha 2 do *repositório de entrada* estar vazia, correspondem à presença de um *token* em ambos os lugares $ExecS_2B_2$ e $Sample[2]$ (veja as convenções adotadas na seção 12.3.3). Desta forma, a transição $B_2S_2toS_1$ encontra-se habilitada na classe C_{19} .



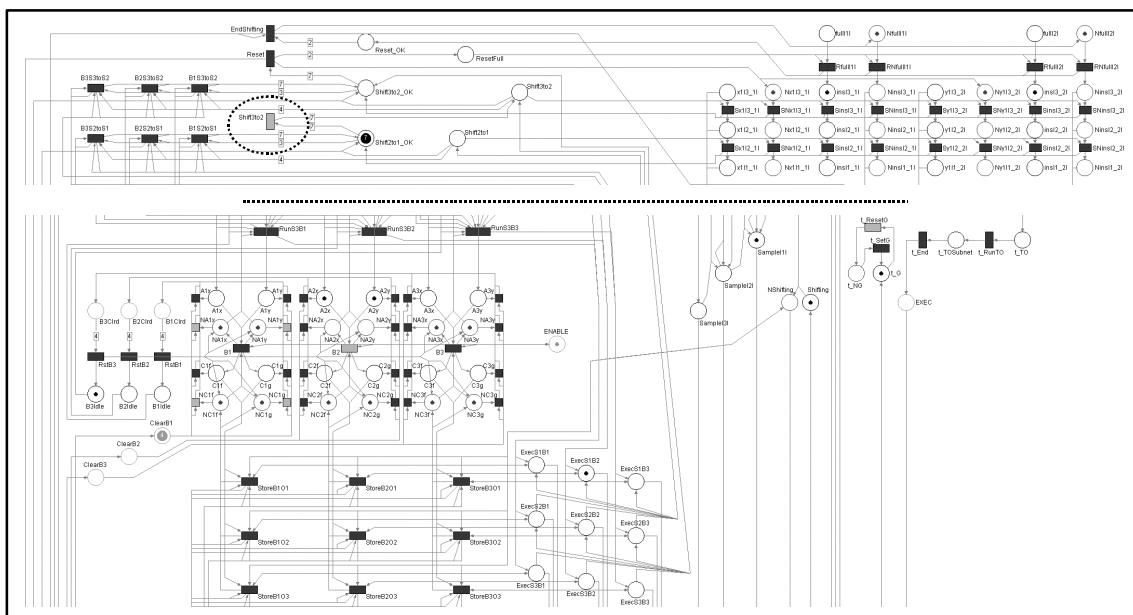


Figura II.45. Classe C₂₀.

Supondo-se que a transição **Shift3to2** dispare, a rede passa para a classe C₂₁, mostrada na figura II.46.

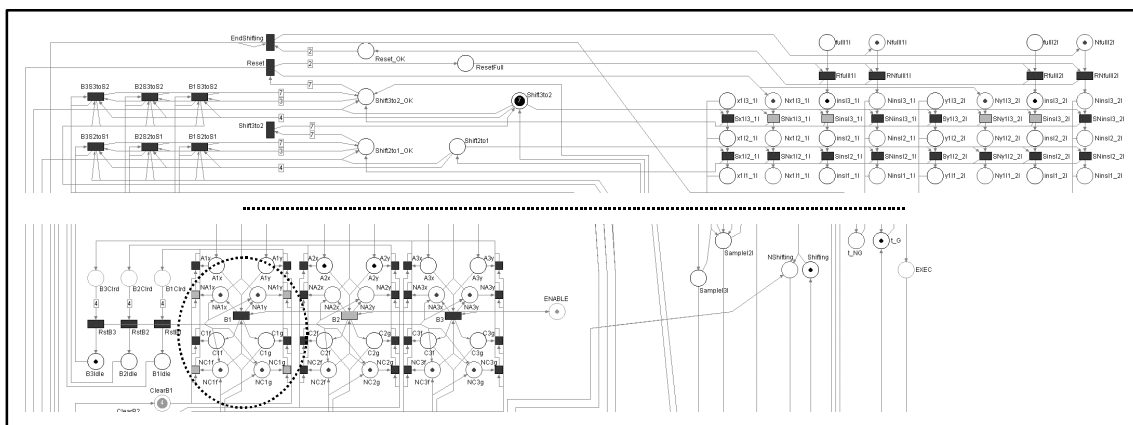


Figura II.46. Classe C₂₁.

Para explorar outras alternativas de simulação (e demonstrar que os resultados obtidos são os mesmos), na classe C₂₁, ao invés de prosseguir-se com a operação de deslocamento de *tokens* da linha 3 para a 2 (conforme foi feito a partir da classe C₆ – figura II.31), adotou-se o restauro da instância B₁, através do disparo das transições NA1x, NA1y, NC1f e NC1g, gerando, desta forma, a classe C₂₂ (figura II.47).

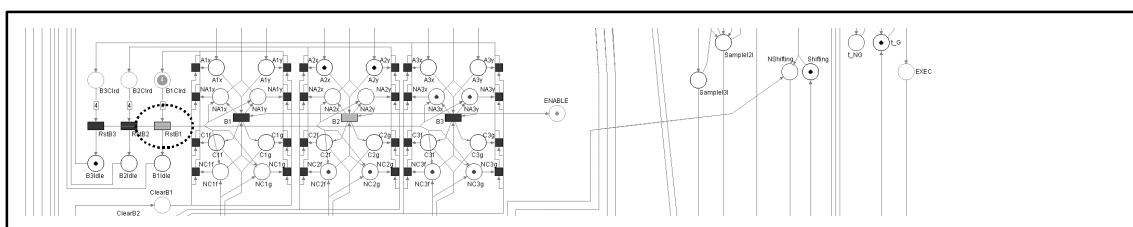
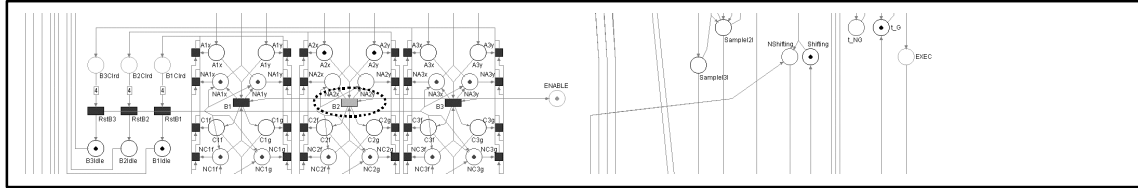


Figura II.47. Classe C₂₂.

Comparando-se as figuras II.46 e II.47, observa-se que o disparo das transições **NA2x**, **NA2y**, **NC2f** e **NC2g** promove o esvaziamento dos lugares com mesmo rótulo, assim como a transferência dos 4 *tokens* presentes em **ClearB₁** para **B₁Clrd**. A presença destes *tokens* em **B₁Clrd** habilita a transição **RstB₁**, cujo disparo promove a inserção de um *token* nos lugares **negativos** da subrede representante de **B₁**, assim como a inserção de um *token* em **B₁Idle**, liberando a instância **B₁**. A classe decorrente desse disparo, **C₂₃**, encontra-se na figura II.48.



Desta forma, na classe **C₂₄**, deve-se prosseguir com a operação de deslocamento de *tokens* da terceira para a segunda linha, em ambos os *repositórios*. Para demonstrar que as transições responsáveis por este deslocamento podem ser disparadas em qualquer seqüência, considere, inicialmente, o disparo de transições relativas ao *repositório de saída* (na classe **C₆**, figura II.31, em situação semelhante, optou-se pelo disparo das transições relativas ao *repositório de entrada*). Após o disparo das transições **SN_{f1}[3,1]** e **SN_{g1}[3,2]**, a rede apresenta a configuração mostrada na classe **C₂₅** (figura II.50).

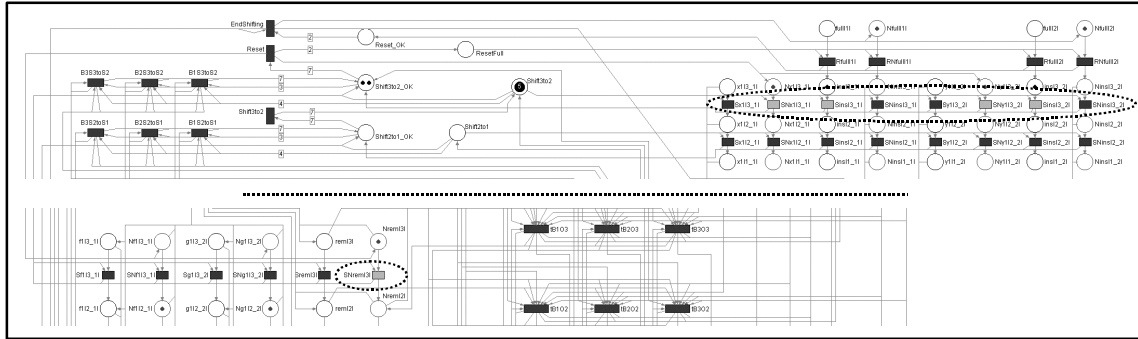


Figura II.50. Classe C₂₅.

Na classe **C₂₅**, os *tokens* presentes em **N_{f1}[3,1]** e **N_{g1}[3,2]** foram deslocados para os lugares **N_{f1}[2,1]** e **N_{g1}[2,2]**. O *token* presente em **N_{rem}[3]** não foi deslocado devido ao não disparo da transição **SN_{rem}[3]**. Além disso, observa-se que o lugar **Shift3to2** perdeu 2 *tokens*, os quais foram repassados para o lugar **Shift3to2OK**.

A classe **C₂₆** (figura II.51) é gerada a partir da **C₂₅**, após o disparo das transições **SN_{rem}[3]**, **SN_{x1}[3,1]** e **Sins[3,1]**, em qualquer ordem. Comparando-se ambas as classes, observa-se novas transferências de *tokens* entre os lugares **Shift3to2** e **Shift3to2OK**, assim como novos deslocamentos de *tokens* envolvendo ambos os *repositórios*.

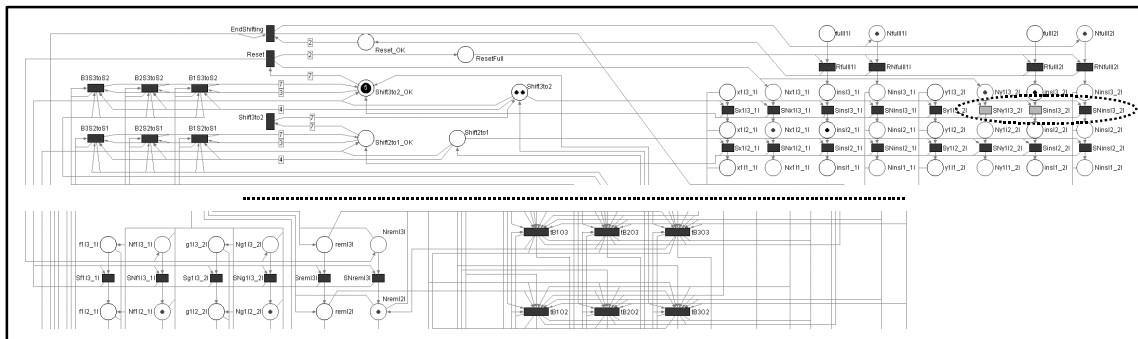


Figura II.51. Classe C₂₆.

Na classe **C₂₆**, resta fazer os deslocamentos envolvendo as transições **SN_{y1}[3,2]** e **Sins[3,2]**. O disparo destas transições conduz a rede à classe **C₂₇** (figura II.52).

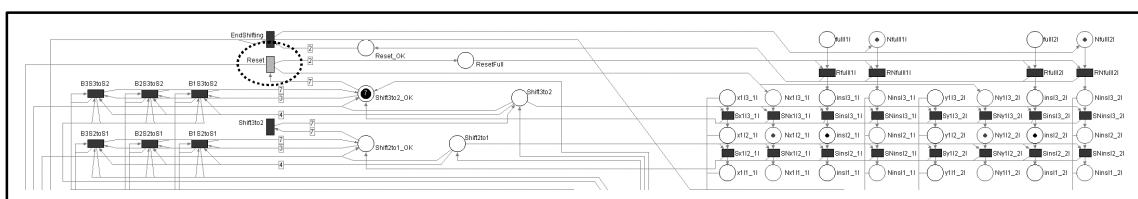


Figura II.52. Classe C₂₇.

habilitação da transição **StoreB₂O₁**. Esta transição é responsável pela passagem dos resultados gerados pela instância **B₂** para a linha 1 do *repositório de saída*. Contudo, para a geração da classe **C₃₁** (figura II.56), adotou-se o disparo da transição **t_RunT₀** (também habilitada na classe **C₃₀**), a qual simula o início da execução das operações associadas a **t**.

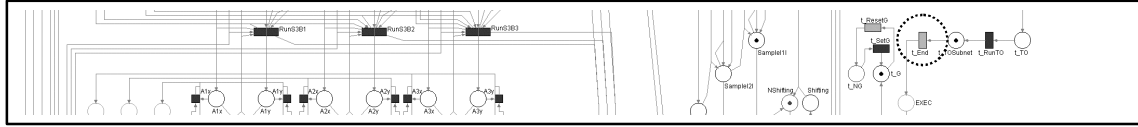


Figura II.56. Classe **C₃₁**.

O disparo de **t_End**, na classe **C₃₁**, sinaliza o término da execução das *operações* associadas a **t** e, conseqüentemente, o término do disparo de **t**. Na classe gerada por tal disparo – **C₃₂** (figura II.38), observa-se a restituição do *token* ao lugar **EXEC**, denotando que **t** não tem mais acesso exclusivo a **P**.

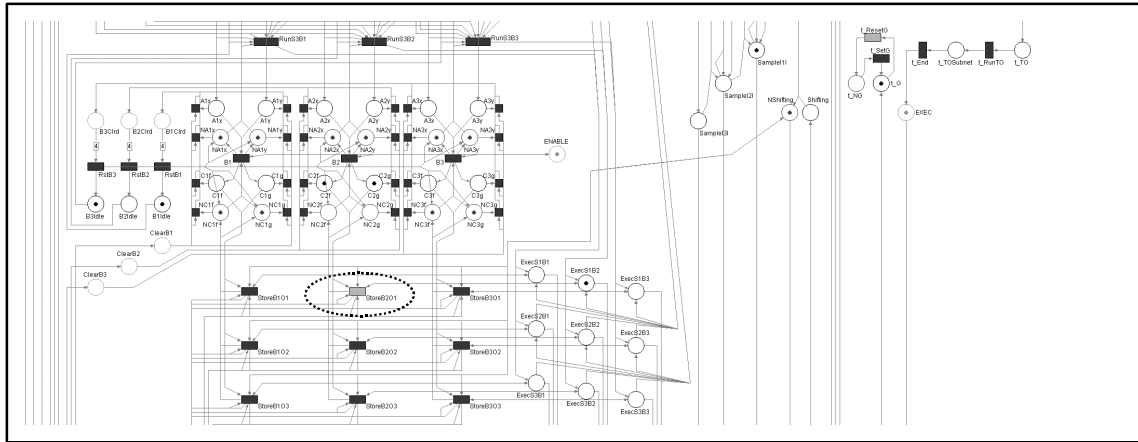


Figura II.57. Classe **C₃₂**.

A transição **StoreB₂O₁** é disparada na classe **C₃₂**, conduzindo a rede à classe **C₃₃** (figura II.58), na qual a transição **t_B₂O₁** é habilitada.

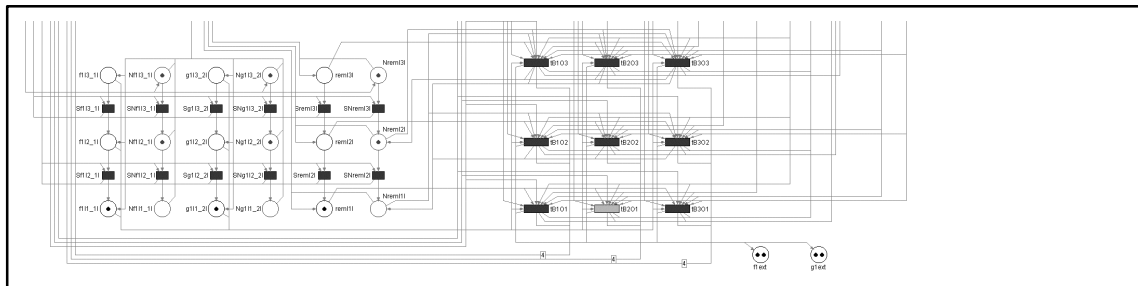


Figura II.58. Classe **C₃₃**.

Na classe **C₃₃**, a transição **t_B₂O₁** somente será desabilitada caso a transição **t_ResetG** dispare. Supondo-se que **t_ResetG** dispare, a habilitação de **t_B₂O₁** somente será possível caso **t_SetG** dispare.

Contudo, na classe **C₃₃**, suponha o disparo de **t_B₂O₁**, o qual leva à classe **C₃₄**, ilustrada na figura II.59. A partir da classe **C₃₄**, o disparo de **t_ResetG** não terá efeito no processo representante do disparo de **t** (o qual é iniciado pelo disparo de **t_B₂O₁**).

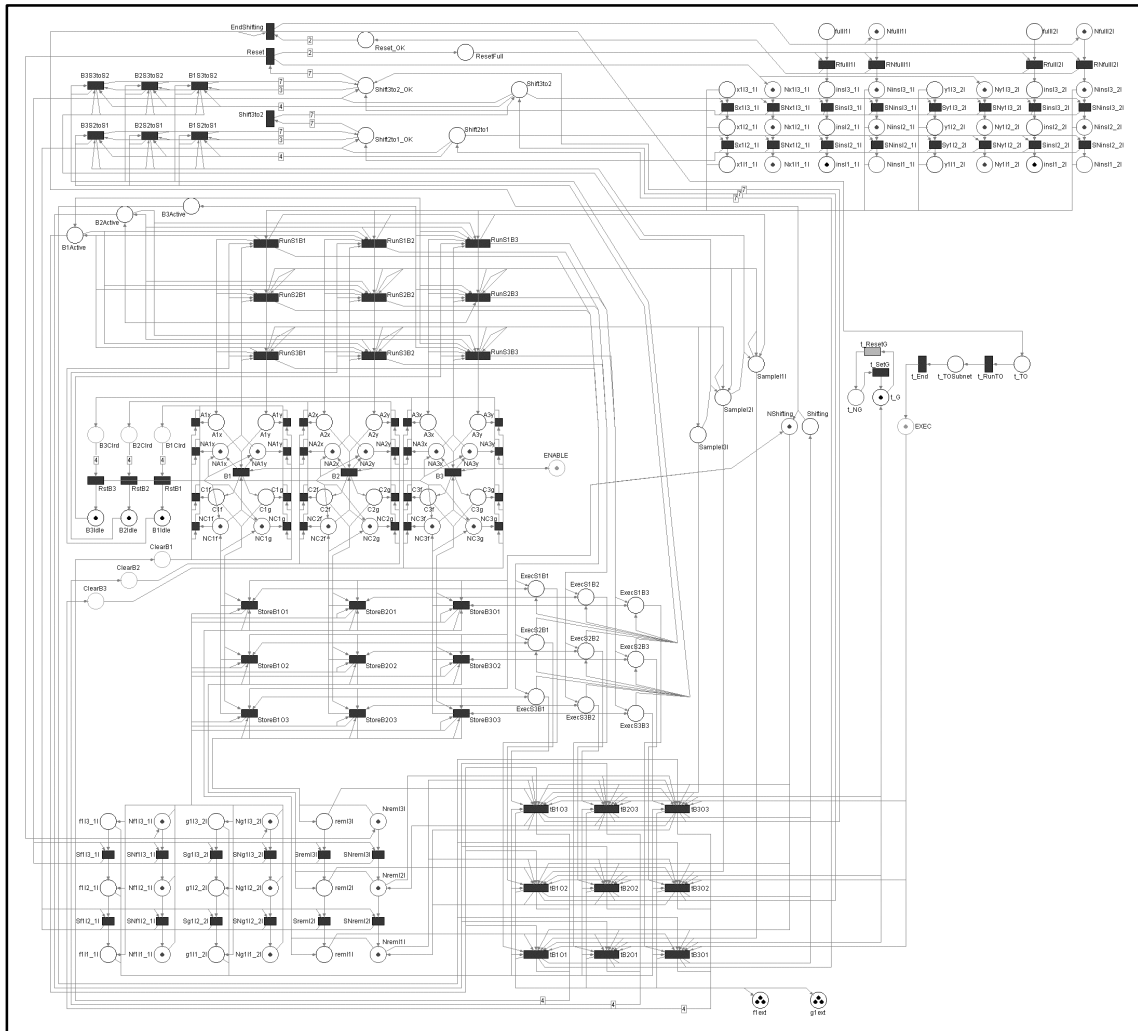


Figura II.60. Classe C35.


```

Net R12_18 ;

Nodes
  boolean1 : place ; // boolean1
  boolean2 : place (1) ;
  EXEC : place (1) ;
  fullI1I : place ; // full[1]
  fullI2I : place ;
  insI1_1I : place ; // ins[1,1]
  insI1_2I : place (1) ;
  insI2_1I : place ;
  insI2_2I : place ;
  insI3_1I : place (1) ;
  insI3_2I : place ;
  Nboolean1 : place (1) ; // Nboolean1
  {Nboolean2 : place ;} // lugar não utilizado
  NfullI1I : place (1) ; // Nfull[1]
  NfullI2I : place (1) ;
  NinsI1_1I : place (1) ; // Nins[1,1]
  NinsI2_1I : place (1) ;
  NinsI3_1I : place ;
  NinsI1_2I : place ;
  NinsI2_2I : place (1) ;
  NinsI3_2I : place (1) ;
  Nx1I1_1I : place ; // Nx1[1,1]
  Nx1I2_1I : place (1) ;
  Nx1I3_1I : place (1) ;
  {Nx2I1_1I : place (1) ;} // lugar não utilizado
  Nx2I2_1I : place ;
  Nx2I3_1I : place (1) ;
  {Ny1I1_2I : place (1) ;} // lugar não utilizado
  {Ny1I2_2I : place (1) ;} // lugar não utilizado
  {Ny1I3_2I : place (1) ;} // lugar não utilizado
  Ny2I1_2I : place (1) ;
  Ny2I2_2I : place (1) ;
  Ny2I3_2I : place (1) ;
  t_G : place ;
  t_NG : place (1) ;
  t_TO : place ; // t_T0
  t_TOSubnet : place ; // t_T0Subnet
  x1I1_1I : place (1) ; // x1[1,1]
  x1I2_1I : place ;
  x1I3_1I : place ;
  {x2I1_1I : place ;} // lugar não utilizado
  x2I2_1I : place (1) ;
  x2I3_1I : place ;
  {y1I1_2I : place ;} // lugar não utilizado
  {y1I2_2I : place ;} // lugar não utilizado
  {y1I3_2I : place ;} // lugar não utilizado
  y2I1_2I : place ;
  y2I2_2I : place ;
  y2I3_2I : place ;

  t_b1I1_2I : transition [5,7] ; // t_b1(1,2)
  t_b1I2_3I : transition [5,7] ;
  t_b2I1_1I : transition [5,7] ;
  t_b2I1_2I : transition [5,7] ;
  t_b2I1_3I : transition [5,7] ;
  t_b2I2_1I : transition [5,7] ;
  t_b2I2_2I : transition [5,7] ;
  t_b2I2_3I : transition [5,7] ;
  t_b2I3_1I : transition [5,7] ;
  t_b2I3_2I : transition [5,7] ;
  t_b2I3_3I : transition [5,7] ;

```

Listagem II.10. Modelo ARP da figura II.61.

```

t_End : transition [0,0] ;
t_SetG : transition ;
t_ResetG : transition [10,10] ;
t_RunTO : transition [0,0] ;          // t_RunT0

Structure
  t_b1I1_2I : (boolean1, Nx1I1_1I, Nx2I2_1I, insI1_1I,
              t_G, NinsI3_1I, EXEC) ,
              (boolean1, x1I1_1I, x2I2_1I, insI3_1I,
              t_G, t_TO, NinsI1_1I) ;
  t_b1I2_3I : (boolean1, Nx1I2_1I, Nx2I3_1I, insI2_1I,
              t_G, NfullI1I, EXEC) ,
              (boolean1, x1I2_1I, x2I3_1I, fullI1I,
              t_G, t_TO, NinsI2_1I) ;
  t_b2I1_1I : (insI1_1I, insI1_2I, Nx1I1_1I, Ny2I1_2I, boolean2,
              Nboolean1, t_G, NinsI2_2I, NinsI2_1I, EXEC) ,
              (x1I1_1I, y2I1_2I, insI2_2I, insI2_1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI1_2I, NinsI1_1I) ;
  t_b2I1_2I : (insI1_1I, insI2_2I, Nx1I1_1I, Ny2I2_2I, boolean2,
              Nboolean1, t_G, NinsI3_2I, NinsI2_1I, EXEC) ,
              (x1I1_1I, y2I2_2I, insI3_2I, insI2_1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI2_2I, NinsI1_1I) ;
  t_b2I1_3I : (insI1_1I, insI3_2I, Nx1I1_1I, Ny2I3_2I, boolean2,
              Nboolean1, t_G, NfullI2I, NinsI2_1I, EXEC) ,
              (x1I1_1I, y2I3_2I, fullI2I, insI2_1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI3_2I, NinsI1_1I) ;
  t_b2I2_1I : (insI2_1I, insI1_2I, Nx1I2_1I, Ny2I1_2I, boolean2,
              Nboolean1, t_G, NinsI2_2I, NinsI3_1I, EXEC) ,
              (y2I1_2I, x1I2_1I, insI2_2I, insI3_1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI1_2I, NinsI2_1I) ;
  t_b2I2_2I : (insI2_1I, insI2_2I, Nx1I2_1I, Ny2I2_2I, boolean2,
              Nboolean1, t_G, NinsI3_2I, NinsI3_1I, EXEC) ,
              (y2I2_2I, x1I2_1I, insI3_2I, insI3_1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI2_2I, NinsI2_1I) ;
  t_b2I2_3I : (insI2_1I, insI3_2I, Nx1I2_1I, Ny2I3_2I, boolean2,
              Nboolean1, t_G, NfullI2I, NinsI3_1I, EXEC) ,
              (y2I3_2I, x1I2_1I, fullI2I, insI3_1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI3_2I, NinsI2_1I) ;
  t_b2I3_1I : (insI3_1I, insI1_2I, Nx1I3_1I, Ny2I1_2I, boolean2,
              Nboolean1, t_G, NinsI2_2I, NfullI1I, EXEC) ,
              (y2I1_2I, x1I3_1I, insI2_2I, fullI1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI1_2I, NinsI3_1I) ;
  t_b2I3_2I : (insI3_1I, insI2_2I, Nx1I3_1I, Ny2I2_2I, boolean2,
              Nboolean1, t_G, NinsI3_2I, NfullI1I, EXEC) ,
              (y2I2_2I, x1I3_1I, insI3_2I, fullI1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI2_2I, NinsI3_1I) ;
  t_b2I3_3I : (insI3_1I, insI3_2I, Nx1I3_1I, Ny2I3_2I, boolean2,
              Nboolean1, t_G, NfullI2I, NfullI1I, EXEC) ,
              (y2I3_2I, x1I3_1I, fullI2I, fullI1I, boolean2,
              Nboolean1, t_G, t_TO, NinsI3_2I, NinsI3_1I) ;
  t_End : (t_TOSubnet) , (EXEC) ;
  t_SetG : (t_NG) , (t_G) ;
  t_ResetG : (t_G) , (t_NG) ;
  t_RunTO : (t_TO) , (t_TOSubnet) ;

endNet.

```

Continuação da listagem II.10.

A análise de propriedades e o grafo de classes da rede correspondente à figura 12.18, gerados pelo **ARP**, encontram-se, respectivamente, na listagem II.11 e na figura II.62.

```

State Enumeration : net R12_18 (6 reachable states).

Verified properties:
*-----*
Net under analysis is binary.
  Null places (M = 0): {boolean1, fullI2I, insI1_1I, insI2_1I,
                        insI3_2I, Nx1I1_1I, Nx2I2_1I, x1I2_1I, x2I3_1I,
                        y2I2_2I, y2I3_2I}
  Binary places      : {boolean2, EXEC, fullI1I, insI1_2I, insI2_2I,
                        insI3_1I, Nboolean1, NfullI1I, NfullI2I,
                        NinsI1_1I, NinsI2_1I, NinsI3_1I,
                        NinsI1_2I, NinsI2_2I, NinsI3_2I, Nx1I2_1I,
                        Nx1I3_1I, Nx2I3_1I, Ny2I1_2I, Ny2I2_2I, Ny2I3_2I,
                        t_G, t_NG, t_TO, t_TOSubnet, x1I1_1I, x1I3_1I,
                        x2I2_1I, y2I1_2I}
  k-Bounded places   : {}
  Unbounded places   : {}

Net under analysis is strictly conservative.

Multi-enabled Tr.: {}

Net under analysis is not live.
  Live Tr.           : {t_SetG, t_ResetG}
  "Almost-live" Tr.: {t_b2I3_1I, t_End, t_SetG, t_ResetG, t_RunTO}
  Non-fired Tr.      : {t_b1I1_2I, t_b1I2_3I, t_b2I1_1I, t_b2I1_2I, t_b2I1_3I,
                        t_b2I2_1I, t_b2I2_2I, t_b2I2_3I, t_b2I3_2I, t_b2I3_3I,
                        }

Net never can go back to M0.

States that start live-locks: C4

No deadlocks detected.
*-----*

```

Listagem II.11. Análise de propriedades para a rede da figura II.61.

Levando-se em conta as convenções adotadas na construção da figura 12.18, a análise de propriedades gerada pelo programa **ARP** (listagem II.11) conduz às seguintes conclusões:

- i) A simulação produziu 6 estados (classes): Estas classes serão rotuladas de C_0 a C_5 ;
- ii) Os lugares pertencentes aos conjuntos

$$S_1 = \{Nboolean_2\},$$

$$S_2 = \{x_2[1,1], Nx_2[1,1]\} \text{ e }$$

$$S_3 = \{y_1[3,2], Ny_1[3,2], y_1[2,2], Ny_1[2,2], y_1[1,2], Ny_1[1,2]\}$$
 não aparecem na listagem II.11: Embora presentes nas figuras 12.18 e II.61 (S_1 na **memória da rede**, S_2 na subrede representante da coluna **X** e S_3 na subrede representante da coluna **Y**), estes lugares não aparecem nos resultados gerados pelo **ARP** (listagem II.11/figura II.62), devido ao fato de terem sido isolados, através de comentários, no arquivo de entrada **R12-18.PN** (transcrito na listagem II.10). Conforme explanado anteriormente, estes lugares não podem ser compilados pelo **ARP**, por apresentarem-se desconectados (daí seu isolamento por meio de comentários);
- iii) A rede é binária: O número máximo de *tokens* em qualquer lugar será 1, conforme pode ser constatado nos três itens seguintes, o que é compatível com as convenções adotadas;
- iv) Os lugares pertencentes aos conjuntos

$$S_1 = \{boolean_1\},$$

$$S_2 = \{x_2[3,1], x_1[2,1], Nx_2[2,1], ins[2,1], Nx_1[1,1], ins[1,1]\} \text{ e }$$

$$S_3 = \{full[2], y_2[3,2], ins[3,2], y_2[2,2]\}$$

são nulos: Enquanto S_1 está contido na **memória da rede**, S_2 e S_3 estão contidos, respectivamente, nas subredes representantes das colunas **X** e **Y**. O lugar **boolean₁** permaneceu vazio, ao longo da simulação, porque a variável correspondente não teve o seu valor alterado. Os lugares pertencentes aos conjuntos S_2 e S_3 , referentes ao **repositório de entrada** de **P**, permaneceram vazios devido à transição representante de **t** disparada – **t_{b2}(3,1)** – não alterar a marcação dos mesmos, conforme pode ser constatado no grafo de classes, na figura II.62. Observa-se que a análise deste item não inclui lugares não compilados (listados no item ii));

- v) Os lugares pertencentes aos conjuntos

$S_1 = \{\text{EXEC}, \text{t\_G}, \text{t\_NG}, \text{Nboolean}_1, \text{boolean}_2, \text{t\_T}_0, \text{t\_T}_0\text{Subnet}\},$

$S_2 = \{\text{full}[1], \text{Nfull}[1], \text{x}_1[3,1], \text{Nx}_1[3,1], \text{Nx}_2[3,1], \text{ins}[3,1],$
 $\text{Nins}[3,1], \text{Nx}_1[2,1], \text{x}_2[2,1], \text{Nins}[2,1], \text{x}_1[1,1], \text{Nins}[1,1]\}$ e

$S_3 = \{\text{Nfull}[2], \text{Ny}_2[3,2], \text{Nins}[3,2], \text{Ny}_2[2,2], \text{ins}[2,2],$
 $\text{Nins}[2,2], \text{y}_2[1,2], \text{Ny}_2[1,2], \text{ins}[1,2], \text{Nins}[1,2]\}$

são binários: Novamente, S_1 está associado à **memória da rede**, S_2 à coluna **X** e S_3 à coluna **Y**. Na marcação inicial, presente na figura 12.18, possuem 1 *token* cada os lugares pertencentes aos conjuntos S_{1C0} , S_{2C0} e S_{3C0} (os quais são subconjuntos, respectivamente, de S_1 , S_2 e S_3), onde

$S_{1C0} = \{\text{EXEC}, \text{t\_G}, \text{Nboolean}_1, \text{boolean}_2\}$ ($\{\text{EXEC}, \text{t\_NG}, \text{Nboolean}_1, \text{boolean}_2\}$, no modelo **JARP** da figura II.61),

$S_{2C0} = \{\text{Nfull}[1], \text{Nx}_1[3,1], \text{Nx}_2[3,1], \text{ins}[3,1],$
 $\text{Nx}_1[2,1], \text{x}_2[2,1], \text{Nins}[2,1], \text{x}_1[1,1], \text{Nins}[1,1]\}$ e

$S_{3C0} = \{\text{Nfull}[2], \text{Ny}_2[3,2], \text{Nins}[3,2], \text{Ny}_2[2,2], \text{Nins}[2,2], \text{Ny}_2[1,2], \text{ins}[1,2]\}.$

Assim, para serem qualificados como binários, os lugares pertencentes aos subconjuntos complementares de S_{1C0} , S_{2C0} e S_{3C0} , quais sejam,

$S_1 \setminus S_{1C0} = \{\text{t\_NG}, \text{t\_T}_0, \text{t\_T}_0\text{Subnet}\}$ ($\{\text{t\_G}, \text{t\_T}_0, \text{t\_T}_0\text{Subnet}\}$, no modelo **JARP**),

$S_2 \setminus S_{2C0} = \{\text{full}[1], \text{x}_1[3,1], \text{Nins}[3,1]\}$ e

$S_3 \setminus S_{3C0} = \{\text{ins}[2,2], \text{y}_2[1,2], \text{Nins}[1,2]\},$

recebem 1 *token* em alguma classe, conforme pode ser visto adiante, na descrição do grafo de classes da rede. Observa-se que a análise do presente item não inclui lugares não compilados (listados no item ii));

- vi) A rede é estritamente conservativa: Isto significa que a rede pode ser particionada em conjuntos de lugares, conservativos para um número constante de *tokens*. Em relação à **memória da rede**, são conservativos, para 1 *token*, os conjuntos $\{\text{boolean}_1, \text{Nboolean}_1\}$, $\{\text{boolean}_2, \text{Nboolean}_2\}$, $\{\text{EXEC}, \text{t\_T}_0, \text{t\_T}_0\text{Subnet}\}$ e $\{\text{t\_G}, \text{t\_NG}\}$. Em relação ao **repositório de entrada**, cada dual (par de lugares, como, por exemplo, **full[1]** e **Nfull[1]**) é conservativo para 1 *token*. Caso estivessem representadas, na figura 12.18, as transições de admissão de *samples* (transições com rótulo inicial **Run**, descritas na explanação da figura 12.12), a rede deixaria de ser estritamente conservativa, pois o disparo de uma destas transições resulta no esvaziamento de uma das fileiras de lugares do **repositório de entrada**, conforme pode ser visto nas figuras II.25, II.26 e II.27 deste apêndice, referentes à simulação interativa envolvendo o lugar funcional da figura 12.17;
- vii) Não há transições **k**-sensíveis: Fato compatível com a figura 12.18;
- viii) A rede não é viva: Esta análise, em conjunto com as três seguintes, reflete o fato da transição **t** poder ser disparada somente uma vez, na marcação presente na figura 12.18. O disparo de **t** promoverá a inserção de **x₁** na coluna **X** e de **y₂** na coluna **Y**, fazendo com que a coluna **X** fique cheia (pois já existem dois *tokens* armazenados em **X**). Desta forma, um novo disparo de **t** somente será possível em dois casos, quais sejam: a variável **boolean₁** ser *verdadeira* e existir, pelo menos, duas posições livres na coluna **X**; ou a

variável *boolean₂* ser *verdadeira* e existir, pelo menos, uma posição livre em ambas as colunas do *repositório de entrada*. Estes dois casos são impossíveis de ocorrer na figura 12.18, uma vez que inexistem transições conectadas a **P** que faça o consumo de *tokens* do *repositório de saída* (o que liberaria linhas do *repositório de entrada*);

- ix) As transições **t_SetG** e **t_ResetG** são vivas;
- x) As transições **t_b₂[3,1]**, **t_SetG**, **t_ResetG**, **t_RunT₀** e **t_End** são quase-vivas: O disparo destas transições é comentado adiante;
- xi) As demais transições não disparam durante a simulação: Este fato se deve à possibilidade de **t** disparar somente uma vez, inviabilizando, portanto, o disparo das transições **t** diferentes de **t_b₂[3,1]**;
- xii) A rede não pode retornar ao estado original **C₀**: Esta análise está em conformidade com a RPSD da figura 12.18, na qual inexistem transições capazes de consumir dados presentes no *repositório de saída*, assim como inexistem transições que possibilitem a inserção de *tokens* somente na coluna **Y**. Da combinação destas transições, seria possível conceber uma rede que permitisse o retorno à marcação mostrada na figura 12.18;
- xiii) O estado **C₄** inicia a execução repetitiva (*livelock*): Conforme pode-se constatar no grafo de classes (figura II.62), ao atingir o estado **C₄**, a rede ficará em *loop*, alternando entre os estados **C₄** e **C₅** (disparo alternado das transições **t_ResetG** e **t_SetG**); e
- xiv) A rede não entra em *deadlock*: Considerando-se a rede da figura 12.18, a mesma deveria ficar bloqueada após a inserção de **x₁** e de **y₂**, respectivamente, nas colunas **X** e **Y** do *repositório de entrada* de **P**. Entretanto, este bloqueio é removido na rede **JARP**, com o acréscimo da transição **t_ResetG**, a qual possibilita a existência do *loop* descrito acima.

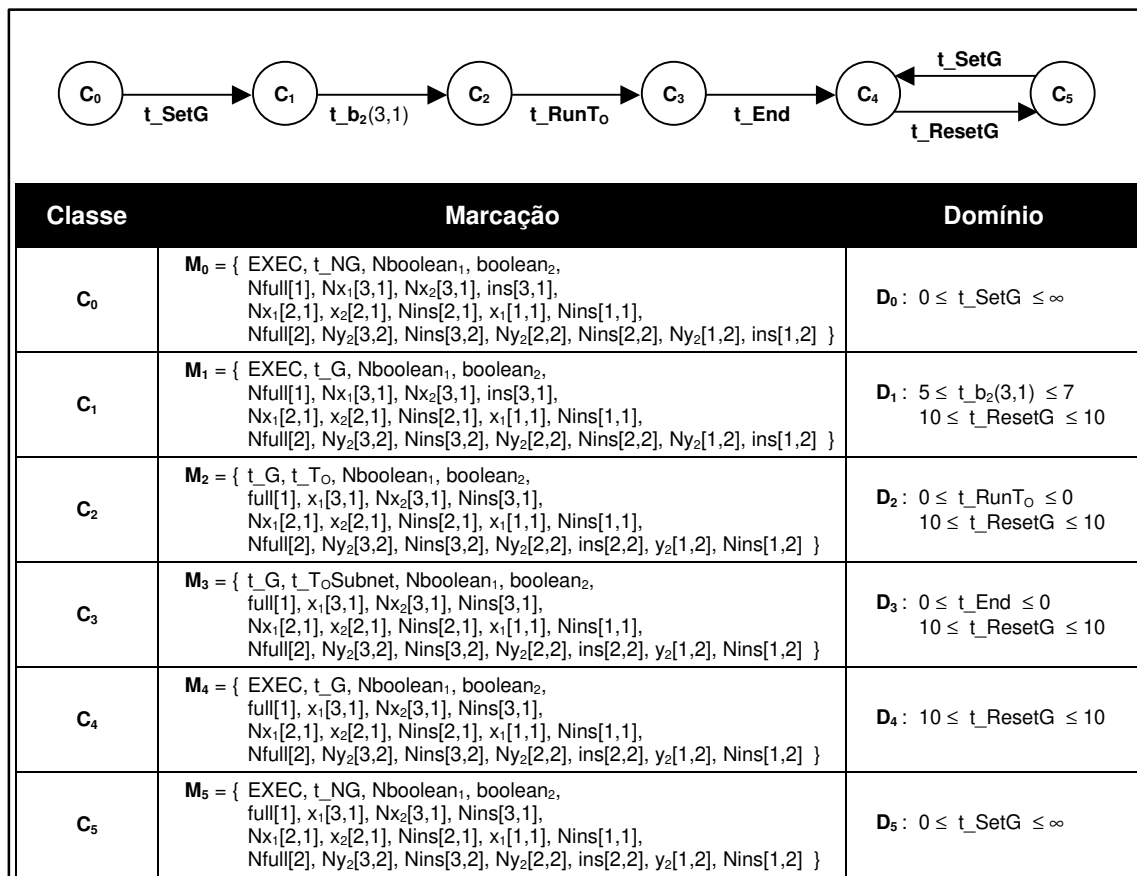


Figura II.62. Grafo de classes para a rede das figuras 12.18 / II.61.

O grafo de classes da rede correspondente à figura 12.18, mostrado na figura II.62, apresenta natureza linear. Desde C_0 , tem-se a seguinte seqüência de disparo de transições:

- i) **t_SetG**: representando a atribuição de *verdadeiro* à guarda de **t**;
- ii) **t_b2(3,1)**: a seleção da transição representante de **t** a disparar, dentre as transições **t_**, baseia-se no valor das variáveis booleanas *boolean₁* e *boolean₂*, assim como nos pontos de inserção, em ambas as colunas do *repositório de entrada*. Na rede da figura 12.18, como a variável *boolean₁* é *falsa* e a variável *boolean₂* é *verdadeira*, **t** irá disparar para inserir os *tokens* do conjunto $\{x_1, y_2\}$ no *repositório de entrada* de **P** (x_1 na coluna **X** e y_2 na coluna **Y**) – caso uma das colunas do *repositório de entrada* estivesse cheia, **t** não ficaria habilitada. Ainda na figura 12.18, a presença de um *token* no lugar **ins**[3,1] denota que a próxima inserção na coluna 1, correspondente a **X**, será na linha 3, enquanto a presença de um *token* em **ins**[1,2] determina que a próxima inserção na coluna 2 (**Y**) será na linha 1. Com a rede apresentando esta configuração, a transição representante de **t** selecionada é **t_b2(3,1)**;
- iii) **t_RunT_O**: representando o início da execução das *operações* associadas à transição **t**;
- iv) **t_End**: representando o término da execução das *operações* associadas à transição **t**;
- v) **t_ResetG**: representando a atribuição de *falso* à guarda de **t**; e
- vi) **t_SetG / t_ResetG**: representando a alternância de valores à guarda de **t** (devido à impossibilidade de ocorrência de outras ações).

Os itens de **ii)** a **iv)** representam o disparo de **t**, o qual promoverá, na RPSD, a inserção do *token* x_1 na posição [3,1] da matriz $R_I(P)$, a inserção do *token* y_2 na posição [1,2] da matriz $R_I(P)$ e, também, a ativação das operações associadas à transição **t**.

O disparo de **t_b2(3,1)** – item **ii)** – promove:

- a) a transferência do *token* presente em $Nx_1[3,1]$ para o seu dual $x_1[3,1]$, representando a inserção do *token* x_1 na posição [3,1] da matriz $R_I(P)$;
- b) a transferência do *token* presente em $Ny_2[1,2]$ para o seu dual $y_2[1,2]$, representando a inserção do *token* y_2 na posição [1,2] da matriz $R_I(P)$;
- c) a transferência dos *tokens* presentes em **ins**[3,1] e **ins**[1,2] para os seus duais **Nins**[3,1] e **Nins**[1,2], representando que as posições [3,1] e [1,2], da matriz $R_I(P)$, não estão mais livres;
- d) a transferência dos *tokens* presentes em **Nfull**[1] e **Nins**[2,2] para os seus duais **full**[1] e **ins**[2,2], representando que a primeira coluna da matriz $R_I(P)$ está cheia, ao passo que a linha 2 da segunda coluna está livre;
- e) a remoção do *token* presente em **EXEC** (associado a **P**), indicando que **t** tem acesso exclusivo a **P**;
- f) a inserção de um *token* em **t_To**, possibilitando o início da execução das *operações* associadas a **t** – item **iii)**.

Após a execução das operações associadas a **t**, ocorrerá a restituição de um *token* ao lugar **EXEC** (associado a **P**), denotando o encerramento do disparo da transição **t** – item **iv)**.

Outras marcações iniciais foram testadas para a rede da figura 12.18, cujas simulações apresentaram resultados análogos aos expostos nesta discussão. Assim, a RPTM presente na figura 12.18 é válida, uma vez que apresenta funcionamento compatível com as convenções e definições adotadas em sua concepção, as quais foram fornecidas nas seções 12.2.2 e 12.3.3, validando, conseqüentemente, a RPSD associada (presente na figura 12.18.a).

12.19, na montagem do modelo **JARP** da figura II.63, foram adicionados, sobre o modelo da figura II.24, os seguintes elementos de rede (lugares/transições):

- lugares **p_G** e **p_NG** / transições **p_SetG** e **p_ResetG**: para a modelagem da guarda de **p**;
- transição **p_Empty**: para a modelagem do início do processo de transformações referente ao disparo de **p**. Este processo promoverá o restauro, ao estado original (restituição à marcação *default*), das redes representantes do **repositório de entrada**, do **repositório de saída** e das instâncias. O processo de restauro envolve, também, os diversos lugares de controle da rede;
- lugares **ClearIR** e **IRcleared** / transições **rótulo_lugar_{IR}** (para cada lugar **rótulo_lugar_{IR}** pertencente à subrede representante do **repositório de entrada**): utilizados no processo de remoção dos *tokens* presentes nos lugares representantes do **repositório de entrada**;
- lugares **ClearOR** e **ORcleared** / transições **rótulo_lugar_{OR}** (para cada lugar **rótulo_lugar_{OR}** pertencente à subrede representante do **repositório de saída**): utilizados no processo de remoção dos *tokens* presentes nos lugares representantes do **repositório de saída**;
- transições **S_jB_i** (*i, j* = 1, 2, 3): utilizadas na finalização do processo de remoção de *tokens* do **repositório de entrada**, através da remoção de eventuais *tokens* excedentes em **ClearIR**;
- transição **p_EmptyOK**: utilizada no restauro dos lugares associados aos **repositórios** à sua marcação *default* (após o término do processo de remoção de *tokens* dos mesmos), através da inserção de um *token* em cada um dos lugares **negativos**, tanto na rede representante do **repositório de entrada** quanto na representante do **repositório de saída**; e
- lugares **p_To** e **p_ToSubnet** / transições **p_RunTo** e **p_End**: utilizados na modelagem das **operações** associadas à transição **p – To(p)**.

Na figura II.63 observa-se, na porção referente à **memória da rede**, que o lugar **EXEC** não foi duplicado, uma vez que ele é associado ao lugar **P**, e não às transições. Desta forma, somente uma das transições conectadas ao lugar **P** poderá ter acesso ao mesmo, por vez.

Na listagem **R12-19.PN**, gerada pelo **JARP** para o modelo da figura II.63, faz-se necessário o acréscimo de **intervalos de disparo**. Considerando-se que os intervalos de **t** e **p** sejam, respectivamente, [5,7] e [4,8], a listagem II.12 transcreve o modelo **ARP** final para a rede da figura 12.19. Esta listagem estende a listagem II.9 (relativa às figuras 12.17/II.24), ao incorporar os lugares e transições descritos acima, assim como os ramos relacionados (os elementos de rede agregados encontram-se destacados na listagem II.12). Além disso, a listagem II.12 permite a compreensão das conexões entre os lugares e as transições no modelo **JARP**. Devido às limitações para especificação de rótulos no **ARP**, a listagem II.12 apresenta, ao final de algumas linhas, o padrão de rotulagem utilizado na figura 12.19, para o grupo de lugares ou de transições em questão.

```
Net R12_19 ;

Nodes
  A1x : place ;           // x1B1
  A2x : place ;           // x1B2
  A3x : place ;           // x1B3
  A1y : place ;           // y1B1
  A2y : place ;           // y1B2
  A3y : place ;           // y1B3
  B1Active : place ;      // B1Active
  B2Active : place ;
  B3Active : place ;
```

Listagem II.12. Modelo ARP da figura II.63.

```

B1Clrd : place ;           // B1Clrd
B2Clrd : place ;
B3Clrd : place ;
B1Idle : place (1) ;       // B1Idle
B2Idle : place (1) ;
B3Idle : place (1) ;
ClearB1 : place ;          // ClearB1
ClearB2 : place ;
ClearB3 : place ;
C1f : place ;              // f1B1
C2f : place ;              // f1B2
C3f : place ;              // f1B3
C1g : place ;              // g1B1
C2g : place ;              // g1B2
C3g : place ;              // g1B3
ENABLE : place (1) ;
EXEC : place (1) ;
ExecS1B1 : place ;         // ExecS1B1
ExecS1B2 : place ;
ExecS1B3 : place ;
ExecS2B1 : place ;
ExecS2B2 : place ;
ExecS2B3 : place ;
ExecS3B1 : place ;
ExecS3B2 : place ;
ExecS3B3 : place ;
fullI1I : place (1) ;     // full[1]
fullI2I : place (1) ;
f1ext : place ;           // f1ext
f1I1_1I : place ;         // f1[1,1]
f1I2_1I : place ;
f1I3_1I : place ;
g1ext : place ;           // g1ext
g1I1_2I : place ;         // g1[1,2]
g1I2_2I : place ;
g1I3_2I : place ;
insI1_1I : place ;        // ins[1,1]
insI1_2I : place ;
insI2_1I : place ;
insI2_2I : place ;
insI3_1I : place ;
insI3_2I : place ;
NA1x : place (1) ;        // Nx1B1
NA2x : place (1) ;        // Nx1B2
NA3x : place (1) ;        // Nx1B3
NA1y : place (1) ;        // Ny1B1
NA2y : place (1) ;        // Ny1B2
NA3y : place (1) ;        // Ny1B3
NC1f : place (1) ;        // Nf1B1
NC2f : place (1) ;        // Nf1B2
NC3f : place (1) ;        // Nf1B3
NC1g : place (1) ;        // Ng1B1
NC2g : place (1) ;        // Ng1B2
NC3g : place (1) ;        // Ng1B3
NfullI1I : place ;        // Nfull[1]
NfullI2I : place ;
Nf1I1_1I : place (1) ;    // Nf1[1,1]
Nf1I2_1I : place (1) ;
Nf1I3_1I : place (1) ;
Ng1I1_2I : place (1) ;    // Ng1[1,2]
Ng1I2_2I : place (1) ;
Ng1I3_2I : place (1) ;
NinsI1_1I : place (1) ;   // Nins[1,1]
NinsI1_2I : place (1) ;

```

Continuação da listagem II.12.

```

NinsI2_1I : place (1) ;
NinsI2_2I : place (1) ;
NinsI3_1I : place (1) ;
NinsI3_2I : place (1) ;
NremI1I : place (1) ;           // Nrem[1]
NremI2I : place (1) ;
NremI3I : place (1) ;
NShifting : place (1) ;
Nx1I1_1I : place ;             // Nx1[1,1]
Nx1I2_1I : place ;
Nx1I3_1I : place ;
Ny1I1_2I : place ;
Ny1I2_2I : place ;
Ny1I3_2I : place ;
remI1I : place ;               // rem[1]
remI2I : place ;
remI3I : place ;
ResetFull : place ;
Reset_OK : place ;            // ResetOK
SampleI1I : place ;           // Sample[1]
SampleI2I : place ;
SampleI3I : place ;
Shifting : place ;
Shift2to1 : place ;
Shift2to1_OK : place ;        // Shift2to1OK
Shift3to2 : place ;
Shift3to2_OK : place ;
t_G : place ;
t_NG : place (1) ;
t_TO : place ;                // tT0
t_TOSubnet : place ;          // tT0Subnet
x1I1_1I : place (1) ;         // x1[1,1]
x1I2_1I : place (1) ;
x1I3_1I : place (1) ;
y1I1_2I : place (1) ;
y1I2_2I : place (1) ;
y1I3_2I : place (1) ;

```

// os lugares a seguir foram acrescentados à listagem II.9

```

ClearIR : place ;             // ClearIR
ClearOR : place ;            // ClearOR
IRcleared : place ;          // IRcleared
ORcleared : place ;          // ORcleared
p_G : place ;
p_NG : place (1) ;
p_TO : place ;
p_TOSubnet : place ;

A1x_0 : transition [0,0] ;     // x1B1
A2x_0 : transition [0,0] ;     // x1B2
A3x_0 : transition [0,0] ;     // x1B3
A1y_0 : transition [0,0] ;     // y1B1
A2y_0 : transition [0,0] ;     // y1B2
A3y_0 : transition [0,0] ;     // y1B3
B1 : transition [10,20] ;      // BB1
B2 : transition [10,20] ;      // BB2
B3 : transition [10,20] ;      // BB3
B1S2toS1 : transition [0,0] ; // B1S2toS1
B1S3toS2 : transition [0,0] ;
B2S2toS1 : transition [0,0] ;
B2S3toS2 : transition [0,0] ;
B3S2toS1 : transition [0,0] ;
B3S3toS2 : transition [0,0] ;

```

Continuação da listagem II.12.

```

C1f_0 : transition [0,0] ;           //  $f_{1B1}$ 
C2f_0 : transition [0,0] ;           //  $f_{1B2}$ 
C3f_0 : transition [0,0] ;           //  $f_{1B3}$ 
C1g_0 : transition [0,0] ;           //  $g_{1B1}$ 
C2g_0 : transition [0,0] ;           //  $g_{1B2}$ 
C3g_0 : transition [0,0] ;           //  $g_{1B3}$ 
EndShifting : transition [0,0] ;
NA1x_0 : transition [0,0] ;           //  $Nx_{1B1}$ 
NA2x_0 : transition [0,0] ;           //  $Nx_{1B2}$ 
NA3x_0 : transition [0,0] ;           //  $Nx_{1B3}$ 
NA1y_0 : transition [0,0] ;           //  $Ny_{1B1}$ 
NA2y_0 : transition [0,0] ;           //  $Ny_{1B2}$ 
NA3y_0 : transition [0,0] ;           //  $Ny_{1B3}$ 
NC1f_0 : transition [0,0] ;           //  $Nf_{1B1}$ 
NC2f_0 : transition [0,0] ;           //  $Nf_{1B2}$ 
NC3f_0 : transition [0,0] ;           //  $Nf_{1B3}$ 
NC1g_0 : transition [0,0] ;           //  $Ng_{1B1}$ 
NC2g_0 : transition [0,0] ;           //  $Ng_{1B2}$ 
NC3g_0 : transition [0,0] ;           //  $Ng_{1B3}$ 
Reset : transition [0,0] ;
RfullI1I : transition [0,0] ;         //  $Rfull[1]$ 
RfullI2I : transition [0,0] ;
RNfullI1I : transition [0,0] ;        //  $RNfull[1]$ 
RNfullI2I : transition [0,0] ;
RstB1 : transition [0,0] ;            //  $RstB_1$ 
RstB2 : transition [0,0] ;
RstB3 : transition [0,0] ;
RunS1B1 : transition [0,0] ;          //  $RunS_1B_1(x_1; y_1)$  - figura 12.12
RunS1B2 : transition [0,0] ;          //  $RunS_1B_2(x_1; y_1)$ 
RunS1B3 : transition [0,0] ;          //  $RunS_1B_3(x_1; y_1)$ 
RunS2B1 : transition [0,0] ;          //  $RunS_2B_1(x_1; y_1)$ 
RunS2B2 : transition [0,0] ;          //  $RunS_2B_2(x_1; y_1)$ 
RunS2B3 : transition [0,0] ;          //  $RunS_2B_3(x_1; y_1)$ 
RunS3B1 : transition [0,0] ;          //  $RunS_3B_1(x_1; y_1)$ 
RunS3B2 : transition [0,0] ;          //  $RunS_3B_2(x_1; y_1)$ 
RunS3B3 : transition [0,0] ;          //  $RunS_3B_3(x_1; y_1)$ 
Sf1I2_1I : transition [0,0] ;         //  $Sf_1[2,1]$ 
Sf1I3_1I : transition [0,0] ;
Sg1I2_2I : transition [0,0] ;
Sg1I3_2I : transition [0,0] ;
Shift3to2_0 : transition [0,0] ;      //  $Shift3to2$ 
SinsI2_1I : transition [0,0] ;        //  $Sins[2,1]$ 
SinsI2_2I : transition [0,0] ;
SinsI3_1I : transition [0,0] ;
SinsI3_2I : transition [0,0] ;
SNf1I2_1I : transition [0,0] ;        //  $SNf_1[2,1]$ 
SNf1I3_1I : transition [0,0] ;
SNG1I2_2I : transition [0,0] ;
SNG1I3_2I : transition [0,0] ;
SNinsI2_1I : transition [0,0] ;        //  $SNins[2,1]$ 
SNinsI2_2I : transition [0,0] ;
SNinsI3_1I : transition [0,0] ;
SNinsI3_2I : transition [0,0] ;
SNremI2I : transition [0,0] ;          //  $SNrem[2]$ 
SNremI3I : transition [0,0] ;
SNx1I2_1I : transition [0,0] ;        //  $SNx_1[2,1]$ 
SNx1I3_1I : transition [0,0] ;
SNy1I2_2I : transition [0,0] ;
SNy1I3_2I : transition [0,0] ;
SremI2I : transition [0,0] ;          //  $Srem[2]$ 
SremI3I : transition [0,0] ;
StoreB1O1 : transition [0,0] ;        //  $StoreB_1O_1(f_1; g_1)$  - figura 12.12
StoreB1O2 : transition [0,0] ;        //  $StoreB_1O_2(f_1; g_1)$ 
StoreB1O3 : transition [0,0] ;        //  $StoreB_1O_3(f_1; g_1)$ 

```

Continuação da listagem II.12.


```

StoreB2O1 : transition [0,0] ;           // StoreB2O1(f1;g1)
StoreB2O2 : transition [0,0] ;           // StoreB2O2(f1;g1)
StoreB2O3 : transition [0,0] ;           // StoreB2O3(f1;g1)
StoreB3O1 : transition [0,0] ;           // StoreB3O1(f1;g1)
StoreB3O2 : transition [0,0] ;           // StoreB3O2(f1;g1)
StoreB3O3 : transition [0,0] ;           // StoreB3O3(f1;g1)
Sx1I2_1I : transition [0,0] ;           // Sx1[2,1]
Sx1I3_1I : transition [0,0] ;
Sy1I2_2I : transition [0,0] ;
Sy1I3_2I : transition [0,0] ;
tB1O1 : transition [5,7] ;               // tB1O1(x1;y1)
tB2O1 : transition [5,7] ;               // tB2O1(x1;y1)
tB3O1 : transition [5,7] ;               // tB3O1(x1;y1)
tB1O2 : transition [5,7] ;               // tB1O2(x1;y1)
tB2O2 : transition [5,7] ;               // tB2O2(x1;y1)
tB3O2 : transition [5,7] ;               // tB3O2(x1;y1)
tB1O3 : transition [5,7] ;               // tB1O3(x1;y1)
tB2O3 : transition [5,7] ;               // tB2O3(x1;y1)
tB3O3 : transition [5,7] ;               // tB3O3(x1;y1)
t_End : transition [0,0] ;
t_ResetG : transition ;
t_RunTO : transition [0,0] ;
t_SetG : transition ;

// as transições a seguir foram acrescentadas à listagem II.9

fullI1I_0 : transition [0,0] ;
fullI2I_0 : transition [0,0] ;
flI1_1I_0 : transition [0,0] ;
flI2_1I_0 : transition [0,0] ;
flI3_1I_0 : transition [0,0] ;
glI1_2I_0 : transition [0,0] ;
glI2_2I_0 : transition [0,0] ;
glI3_2I_0 : transition [0,0] ;
insI1_1I_0 : transition [0,0] ;
insI1_2I_0 : transition [0,0] ;
insI2_1I_0 : transition [0,0] ;
insI2_2I_0 : transition [0,0] ;
insI3_1I_0 : transition [0,0] ;
insI3_2I_0 : transition [0,0] ;
NfullI1I_0 : transition [0,0] ;
NfullI2I_0 : transition [0,0] ;
NflI1_1I_0 : transition [0,0] ;
NflI2_1I_0 : transition [0,0] ;
NflI3_1I_0 : transition [0,0] ;
NglI1_2I_0 : transition [0,0] ;
NglI2_2I_0 : transition [0,0] ;
NglI3_2I_0 : transition [0,0] ;
NinsI1_1I_0 : transition [0,0] ;
NinsI1_2I_0 : transition [0,0] ;
NinsI2_1I_0 : transition [0,0] ;
NinsI2_2I_0 : transition [0,0] ;
NinsI3_2I_0 : transition [0,0] ;
NinsI3_1I_0 : transition [0,0] ;
NremI1I_0 : transition [0,0] ;
NremI2I_0 : transition [0,0] ;
NremI3I_0 : transition [0,0] ;
Nx1I1_1I_0 : transition [0,0] ;
Nx1I2_1I_0 : transition [0,0] ;
Nx1I3_1I_0 : transition [0,0] ;
NylI1_2I_0 : transition [0,0] ;
NylI2_2I_0 : transition [0,0] ;
NylI3_2I_0 : transition [0,0] ;
remI1I_0 : transition [0,0] ;

```

Continuação da listagem II.12.

```

remI2I_0 : transition [0,0] ;
remI3I_0 : transition [0,0] ;
S1B1 : transition [0,0] ;           // S1B1
S1B2 : transition [0,0] ;
S1B3 : transition [0,0] ;
S2B1 : transition [0,0] ;
S2B2 : transition [0,0] ;
S2B3 : transition [0,0] ;
S3B1 : transition [0,0] ;
S3B2 : transition [0,0] ;
S3B3 : transition [0,0] ;
x1I1_1I_0 : transition [0,0] ;
x1I2_1I_0 : transition [0,0] ;
x1I3_1I_0 : transition [0,0] ;
y1I1_2I_0 : transition [0,0] ;
y1I2_2I_0 : transition [0,0] ;
y1I3_2I_0 : transition [0,0] ;
p_Empty : transition [4,8] ;
p_Empty_OK : transition [0,0] ;
p_End : transition [0,0] ;
p_ResetG : transition ;
p_RunTO : transition [0,0] ;
p_SetG : transition ;

```

Structure

```

A1x_0 : (ClearB1, A1x) , (B1Clrd) ;
A2x_0 : (ClearB2, A2x) , (B2Clrd) ;
A3x_0 : (ClearB3, A3x) , (B3Clrd) ;
A1y_0 : (ClearB1, A1y) , (B1Clrd) ;
A2y_0 : (ClearB2, A2y) , (B2Clrd) ;
A3y_0 : (ClearB3, A3y) , (B3Clrd) ;
B1 : (A1x, A1y, NC1f, NC1g, ENABLE) , (C1f, C1g, NA1x, NA1y, ENABLE) ;
B2 : (A2x, A2y, NC2f, NC2g, ENABLE) , (C2f, C2g, NA2x, NA2y, ENABLE) ;
B3 : (A3x, A3y, NC3f, NC3g, ENABLE) , (C3f, C3g, NA3x, NA3y, ENABLE) ;
B1S2toS1 : (4 * Shift2to1, SampleI2I, 3 * Shift2to1_OK, ExecS2B1) ,
            (7 * Shift2to1_OK, SampleI1I, ExecS1B1) ;
B1S3toS2 : (4 * Shift3to2, 3 * Shift3to2_OK, SampleI3I, ExecS3B1) ,
            (7 * Shift3to2_OK, SampleI2I, ExecS2B1) ;
B2S2toS1 : (ExecS2B2, SampleI2I, 4 * Shift2to1, 3 * Shift2to1_OK) ,
            (ExecS1B2, SampleI1I, 7 * Shift2to1_OK) ;
B2S3toS2 : (ExecS3B2, SampleI3I, 4 * Shift3to2, 3 * Shift3to2_OK) ,
            (ExecS2B2, SampleI2I, 7 * Shift3to2_OK) ;
B3S2toS1 : (ExecS2B3, SampleI2I, 4 * Shift2to1, 3 * Shift2to1_OK) ,
            (ExecS1B3, SampleI1I, 7 * Shift2to1_OK) ;
B3S3toS2 : (ExecS3B3, SampleI3I, 4 * Shift3to2, 3 * Shift3to2_OK) ,
            (ExecS2B3, SampleI2I, 7 * Shift3to2_OK) ;
C1f_0 : (ClearB1, C1f) , (B1Clrd) ;
C2f_0 : (ClearB2, C2f) , (B2Clrd) ;
C3f_0 : (ClearB3, C3f) , (B3Clrd) ;
C1g_0 : (ClearB1, C1g) , (B1Clrd) ;
C2g_0 : (ClearB2, C2g) , (B2Clrd) ;
C3g_0 : (ClearB3, C3g) , (B3Clrd) ;
EndShifting : (Shifting, 2 * Reset_OK) ,
              (NfullI1I, NfullI2I, NShifting, t_TO) ;
NA1x_0 : (ClearB1, NA1x) , (B1Clrd) ;
NA2x_0 : (ClearB2, NA2x) , (B2Clrd) ;
NA3x_0 : (ClearB3, NA3x) , (B3Clrd) ;
NA1y_0 : (ClearB1, NA1y) , (B1Clrd) ;
NA2y_0 : (ClearB2, NA2y) , (B2Clrd) ;
NA3y_0 : (ClearB3, NA3y) , (B3Clrd) ;
NC1f_0 : (ClearB1, NC1f) , (B1Clrd) ;
NC2f_0 : (ClearB2, NC2f) , (B2Clrd) ;
NC3f_0 : (ClearB3, NC3f) , (B3Clrd) ;

```

Continuação da listagem II.12.

```

NC1g_0 : (ClearB1, NC1g) , (B1Clrd) ;
NC2g_0 : (ClearB2, NC2g) , (B2Clrd) ;
NC3g_0 : (ClearB3, NC3g) , (B3Clrd) ;
Reset : (7 * Shift3to2_OK) ,
        (Nx1I3_1I, Ny1I3_2I, 2 * ResetFull, Nf1I3_1I, Ng1I3_2I, NremI3I) ;
RfullI1I : (ResetFull, fullI1I) , (Reset_OK, insI3_1I) ;
RfullI2I : (ResetFull, fullI2I) , (Reset_OK, insI3_2I) ;
RNfullI1I : (NfullI1I, ResetFull) , (Reset_OK, NinsI3_1I) ;
RNfullI2I : (ResetFull, NfullI2I) , (Reset_OK, NinsI3_2I) ;
RstB1 : (4 * B1Clrd) , (B1Idle, NA1x, NA1y, NC1f, NC1g) ;
RstB2 : (4 * B2Clrd) , (B2Idle, NA2x, NA2y, NC2f, NC2g) ;
RstB3 : (4 * B3Clrd) , (B3Idle, NA3x, NA3y, NC3f, NC3g) ;
RunS1B1 : (x1I1_1I, y1I1_2I, B1Idle, NinsI1_1I, NinsI1_2I, NA1x, NA1y) ,
        (A1x, A1y, B1Active, SampleI1I, ExecS1B1) ;
RunS2B1 : (x1I2_1I, y1I2_2I, B1Idle,
        SampleI1I, NinsI2_1I, NinsI2_2I, NA1x, NA1y) ,
        (A1x, A1y, B1Active, SampleI1I, SampleI2I, ExecS2B1) ;
RunS3B1 : (x1I3_1I, y1I3_2I, B1Idle,
        SampleI2I, NinsI3_1I, NinsI3_2I, NA1x, NA1y) ,
        (A1x, A1y, B1Active, SampleI2I, SampleI3I, ExecS3B1) ;
RunS1B2 : (x1I1_1I, y1I1_2I, B2Idle,
        B1Active, NinsI1_1I, NinsI1_2I, NA2x, NA2y) ,
        (A2x, A2y, B1Active, B2Active, SampleI1I, ExecS1B2) ;
RunS2B2 : (x1I2_1I, y1I2_2I, B2Idle, B1Active, SampleI1I,
        NinsI2_1I, NinsI2_2I, NA2x, NA2y) ,
        (A2x, A2y, B1Active, B2Active, SampleI1I, SampleI2I, ExecS2B2) ;
RunS3B2 : (x1I3_1I, y1I3_2I, B2Idle, B1Active, SampleI2I,
        NinsI3_1I, NinsI3_2I, NA2x, NA2y) ,
        (A2x, A2y, B1Active, B2Active, SampleI2I, SampleI3I, ExecS3B2) ;
RunS1B3 : (x1I1_1I, y1I1_2I, B3Idle, B1Active, B2Active,
        NinsI1_1I, NinsI1_2I, NA3x, NA3y) ,
        (A3x, A3y, B1Active, B2Active, B3Active, SampleI1I, ExecS1B3) ;
RunS2B3 : (x1I2_1I, y1I2_2I, B3Idle, B1Active, B2Active, SampleI1I,
        NinsI2_1I, NinsI2_2I, NA3x, NA3y) ,
        (A3x, A3y, B1Active, B2Active, B3Active,
        SampleI1I, SampleI2I, ExecS2B3) ;
RunS3B3 : (x1I3_1I, y1I3_2I, B3Idle, B1Active, B2Active, SampleI2I,
        NinsI3_1I, NinsI3_2I, NA3x, NA3y) ,
        (A3x, A3y, B1Active, B2Active, B3Active,
        SampleI2I, SampleI3I, ExecS3B3) ;
Sf1I2_1I : (Shift2to1, f1I2_1I) , (Shift2to1_OK, f1I1_1I) ;
Sf1I3_1I : (Shift3to2, f1I3_1I) , (Shift3to2_OK, f1I2_1I) ;
Sg1I2_2I : (Shift2to1, g1I2_2I) , (Shift2to1_OK, g1I1_2I) ;
Sg1I3_2I : (Shift3to2, g1I3_2I) , (Shift3to2_OK, g1I2_2I) ;
Shift3to2_0 : (7 * Shift2to1_OK) , (7 * Shift3to2) ;
SinsI2_1I : (insI2_1I, Shift2to1) , (insI1_1I, Shift2to1_OK) ;
SinsI2_2I : (insI2_2I, Shift2to1) , (insI1_2I, Shift2to1_OK) ;
SinsI3_1I : (insI3_1I, Shift3to2) , (insI2_1I, Shift3to2_OK) ;
SinsI3_2I : (insI3_2I, Shift3to2) , (insI2_2I, Shift3to2_OK) ;
SNf1I2_1I : (Shift2to1, Nf1I2_1I) , (Shift2to1_OK, Nf1I1_1I) ;
SNf1I3_1I : (Shift3to2, Nf1I3_1I) , (Shift3to2_OK, Nf1I2_1I) ;
SNg1I2_2I : (Shift2to1, Ng1I2_2I) , (Shift2to1_OK, Ng1I1_2I) ;
SNg1I3_2I : (Shift3to2, Ng1I3_2I) , (Shift3to2_OK, Ng1I2_2I) ;
SNinsI2_1I : (NinsI2_1I, Shift2to1) , (NinsI1_1I, Shift2to1_OK) ;
SNinsI2_2I : (NinsI2_2I, Shift2to1) , (NinsI1_2I, Shift2to1_OK) ;
SNinsI3_1I : (NinsI3_1I, Shift3to2) , (NinsI2_1I, Shift3to2_OK) ;
SNinsI3_2I : (NinsI3_2I, Shift3to2) , (NinsI2_2I, Shift3to2_OK) ;
SNremI2I : (Shift2to1, NremI2I) , (Shift2to1_OK, NremI1I) ;
SNremI3I : (Shift3to2, NremI3I) , (Shift3to2_OK, NremI2I) ;
SNx1I2_1I : (Nx1I2_1I, Shift2to1) , (Nx1I1_1I, Shift2to1_OK) ;
SNx1I3_1I : (Nx1I3_1I, Shift3to2) , (Nx1I2_1I, Shift3to2_OK) ;
SNy1I2_2I : (Ny1I2_2I, Shift2to1) , (Ny1I1_2I, Shift2to1_OK) ;
SNy1I3_2I : (Ny1I3_2I, Shift3to2) , (Ny1I2_2I, Shift3to2_OK) ;
SremI2I : (Shift2to1, remI2I) , (Shift2to1_OK, remI1I) ;

```

Continuação da listagem II.12.

```

SremI3I : (Shift3to2, remI3I) , (Shift3to2_OK, remI2I) ;
StoreB101 : (C1f, C1g, ExecS1B1, NremI1I, Nf1I1_1I, Ng1I1_2I, NShifting) ,
(f1I1_1I, g1I1_2I, remI1I, ExecS1B1, NShifting, NC1f, NC1g) ;
StoreB102 : (C1f, C1g, ExecS2B1, NremI2I, Nf1I2_1I, Ng1I2_2I, NShifting) ,
(f1I2_1I, g1I2_2I, remI2I, ExecS2B1, NShifting, NC1f, NC1g) ;
StoreB103 : (C1f, C1g, ExecS3B1, NremI3I, Nf1I3_1I, Ng1I3_2I, NShifting) ,
(f1I3_1I, g1I3_2I, remI3I, ExecS3B1, NShifting, NC1f, NC1g) ;
StoreB201 : (C2g, C2f, ExecS1B2, NremI1I, Nf1I1_1I, Ng1I1_2I, NShifting) ,
(g1I1_2I, f1I1_1I, remI1I, ExecS1B2, NShifting, NC2f, NC2g) ;
StoreB202 : (C2g, C2f, ExecS2B2, NremI2I, Nf1I2_1I, Ng1I2_2I, NShifting) ,
(g1I2_2I, f1I2_1I, remI2I, ExecS2B2, NShifting, NC2f, NC2g) ;
StoreB203 : (C2g, C2f, ExecS3B2, NremI3I, Nf1I3_1I, Ng1I3_2I, NShifting) ,
(g1I3_2I, f1I3_1I, remI3I, ExecS3B2, NShifting, NC2f, NC2g) ;
StoreB301 : (C3g, C3f, ExecS1B3, NremI1I, Nf1I1_1I, Ng1I1_2I, NShifting) ,
(f1I1_1I, g1I1_2I, remI1I, ExecS1B3, NShifting, NC3f, NC3g) ;
StoreB302 : (C3g, C3f, ExecS2B3, NremI2I, Nf1I2_1I, Ng1I2_2I, NShifting) ,
(f1I2_1I, g1I2_2I, remI2I, ExecS2B3, NShifting, NC3f, NC3g) ;
StoreB303 : (C3g, C3f, ExecS3B3, NremI3I, Nf1I3_1I, Ng1I3_2I, NShifting) ,
(f1I3_1I, g1I3_2I, remI3I, ExecS3B3, NShifting, NC3f, NC3g) ;
Sx1I2_1I : (x1I2_1I, Shift2to1) , (x1I1_1I, Shift2to1_OK) ;
Sx1I3_1I : (x1I3_1I, Shift3to2) , (x1I2_1I, Shift3to2_OK) ;
Sy1I2_2I : (y1I2_2I, Shift2to1) , (y1I1_2I, Shift2to1_OK) ;
Sy1I3_2I : (y1I3_2I, Shift3to2) , (y1I2_2I, Shift3to2_OK) ;
tB101 : (ExecS1B1, SampleI1I, B1Active, remI1I,
f1I1_1I, g1I1_2I, NShifting, t_G, EXEC) ,
(4 * ClearB1, 7 * Shift2to1, Shifting, flect, glect, t_G) ;
tB201 : (ExecS1B2, SampleI1I, B2Active, remI1I,
f1I1_1I, g1I1_2I, NShifting, t_G, EXEC) ,
(4 * ClearB2, 7 * Shift2to1, Shifting, flect, glect, t_G) ;
tB301 : (ExecS1B3, SampleI1I, B3Active, remI1I,
f1I1_1I, g1I1_2I, NShifting, t_G, EXEC) ,
(4 * ClearB3, 7 * Shift2to1, Shifting, flect, glect, t_G) ;
tB102 : (ExecS2B1, SampleI2I, B1Active, remI2I,
f1I2_1I, g1I2_2I, NremI1I, NShifting, t_G, EXEC) ,
(4 * ClearB1, NremI1I, Shifting, 7 * Shift3to2,
flect, glect, t_G) ;
tB202 : (ExecS2B2, SampleI2I, B2Active, remI2I,
f1I2_1I, g1I2_2I, NremI1I, NShifting, t_G, EXEC) ,
(4 * ClearB2, NremI1I, Shifting, 7 * Shift3to2,
flect, glect, t_G) ;
tB302 : (ExecS2B3, SampleI2I, B3Active, remI2I,
f1I2_1I, g1I2_2I, NremI1I, NShifting, t_G, EXEC) ,
(4 * ClearB3, NremI1I, Shifting, 7 * Shift3to2,
flect, glect, t_G) ;
tB103 : (ExecS3B1, SampleI3I, B1Active, remI3I, f1I3_1I, g1I3_2I,
NremI1I, NremI2I, NShifting, t_G, EXEC) ,
(4 * ClearB1, NremI1I, NremI2I, Shifting, 7 * Shift3to2_OK,
flect, glect, t_G) ;
tB203 : (ExecS3B2, SampleI3I, B2Active, remI3I, f1I3_1I, g1I3_2I,
NremI1I, NremI2I, NShifting, t_G, EXEC) ,
(4 * ClearB2, NremI1I, NremI2I, Shifting, 7 * Shift3to2_OK,
flect, glect, t_G) ;
tB303 : (ExecS3B3, SampleI3I, B3Active, remI3I, f1I3_1I, g1I3_2I,
NremI1I, NremI2I, NShifting, t_G, EXEC) ,
(4 * ClearB3, NremI1I, NremI2I, Shifting, 7 * Shift3to2_OK,
flect, glect, t_G) ;
t_End : (t_TOSubnet) , (EXEC) ;
t_ResetG : (t_G) , (t_NG) ;
t_RunTO : (t_TO) , (t_TOSubnet) ;
t_SetG : (t_NG) , (t_G) ;

// os ramos a seguir foram acrescentados à listagem II.9

fullI1I_0 : (fullI1I, ClearIR) , (IRcleared) ;

```

Continuação da listagem II.12.

```

fullI2I_0 : (fullI2I, ClearIR) , (IRcleared) ;
f1I1_1I_0 : (f1I1_1I, ClearOR) , (ORcleared) ;
f1I2_1I_0 : (f1I2_1I, ClearOR) , (ORcleared) ;
f1I3_1I_0 : (f1I3_1I, ClearOR) , (ORcleared) ;
g1I1_2I_0 : (g1I1_2I, ClearOR) , (ORcleared) ;
g1I2_2I_0 : (g1I2_2I, ClearOR) , (ORcleared) ;
g1I3_2I_0 : (g1I3_2I, ClearOR) , (ORcleared) ;
insI1_1I_0 : (insI1_1I, ClearIR) , (IRcleared) ;
insI1_2I_0 : (insI1_2I, ClearIR) , (IRcleared) ;
insI2_1I_0 : (insI2_1I, ClearIR) , (IRcleared) ;
insI2_2I_0 : (insI2_2I, ClearIR) , (IRcleared) ;
insI3_1I_0 : (insI3_1I, ClearIR) , (IRcleared) ;
insI3_2I_0 : (insI3_2I, ClearIR) , (IRcleared) ;
NfullI1I_0 : (NfullI1I, ClearIR) , (IRcleared) ;
NfullI2I_0 : (NfullI2I, ClearIR) , (IRcleared) ;
Nf1I1_1I_0 : (Nf1I1_1I, ClearOR) , (ORcleared) ;
Nf1I2_1I_0 : (Nf1I2_1I, ClearOR) , (ORcleared) ;
Nf1I3_1I_0 : (Nf1I3_1I, ClearOR) , (ORcleared) ;
Ng1I1_2I_0 : (Ng1I1_2I, ClearOR) , (ORcleared) ;
Ng1I2_2I_0 : (Ng1I2_2I, ClearOR) , (ORcleared) ;
Ng1I3_2I_0 : (Ng1I3_2I, ClearOR) , (ORcleared) ;
NinsI1_1I_0 : (NinsI1_1I, ClearIR) , (IRcleared) ;
NinsI1_2I_0 : (NinsI1_2I, ClearIR) , (IRcleared) ;
NinsI2_1I_0 : (NinsI2_1I, ClearIR) , (IRcleared) ;
NinsI2_2I_0 : (NinsI2_2I, ClearIR) , (IRcleared) ;
NinsI3_1I_0 : (NinsI3_1I, ClearIR) , (IRcleared) ;
NinsI3_2I_0 : (NinsI3_2I, ClearIR) , (IRcleared) ;
NremI1I_0 : (NremI1I, ClearOR) , (ORcleared) ;
NremI2I_0 : (NremI2I, ClearOR) , (ORcleared) ;
NremI3I_0 : (NremI3I, ClearOR) , (ORcleared) ;
Nx1I1_1I_0 : (Nx1I1_1I, ClearIR) , (IRcleared) ;
Nx1I2_1I_0 : (Nx1I2_1I, ClearIR) , (IRcleared) ;
Nx1I3_1I_0 : (Nx1I3_1I, ClearIR) , (IRcleared) ;
Nyl1I1_2I_0 : (Nyl1I1_2I, ClearIR) , (IRcleared) ;
Nyl1I2_2I_0 : (Nyl1I2_2I, ClearIR) , (IRcleared) ;
Nyl1I3_2I_0 : (Nyl1I3_2I, ClearIR) , (IRcleared) ;
remI1I_0 : (remI1I, ClearOR) , (ORcleared) ;
remI2I_0 : (remI2I, ClearOR) , (ORcleared) ;
remI3I_0 : (remI3I, ClearOR) , (ORcleared) ;
S1B1 : (ExecS1B1, 4 * ClearIR, SampleI1I, B1Active) ,
      (4 * ClearB1, 4 * IRcleared) ;
S1B2 : (ExecS1B2, 4 * ClearIR, SampleI1I, B2Active) ,
      (4 * ClearB2, 4 * IRcleared) ;
S1B3 : (ExecS1B3, 4 * ClearIR, SampleI1I, B3Active) ,
      (4 * ClearB3, 4 * IRcleared) ;
S2B1 : (ExecS2B1, 4 * ClearIR, SampleI2I, B1Active) ,
      (4 * ClearB1, 4 * IRcleared) ;
S2B2 : (ExecS2B2, 4 * ClearIR, SampleI2I, B2Active) ,
      (4 * ClearB2, 4 * IRcleared) ;
S2B3 : (ExecS2B3, 4 * ClearIR, SampleI2I, B3Active) ,
      (4 * ClearB3, 4 * IRcleared) ;
S3B1 : (ExecS3B1, 4 * ClearIR, SampleI3I, B1Active) ,
      (4 * ClearB1, 4 * IRcleared) ;
S3B2 : (ExecS3B2, 4 * ClearIR, SampleI3I, B2Active) ,
      (4 * ClearB2, 4 * IRcleared) ;
S3B3 : (ExecS3B3, 4 * ClearIR, SampleI3I, B3Active) ,
      (4 * ClearB3, 4 * IRcleared) ;
x1I1_1I_0 : (x1I1_1I, ClearIR) , (IRcleared) ;
x1I2_1I_0 : (x1I2_1I, ClearIR) , (IRcleared) ;
x1I3_1I_0 : (x1I3_1I, ClearIR) , (IRcleared) ;
y1I1_2I_0 : (y1I1_2I, ClearIR) , (IRcleared) ;
y1I2_2I_0 : (y1I2_2I, ClearIR) , (IRcleared) ;
y1I3_2I_0 : (y1I3_2I, ClearIR) , (IRcleared) ;

```

Continuação da listagem II.12.

```

p_Empty : (p_G, EXEC, ENABLE) , (9 * ClearOR, 14 * ClearIR, p_G) ;
p_Empty_OK : (9 * ORcleared, 14 * IRcleared, B1Idle, B2Idle, B3Idle) ,
(p_TO, Nx1I1_1I, Nx1I2_1I, Nx1I3_1I, insI1_1I, NinsI2_1I,
NinsI3_1I, NfullI1I, Ny1I1_2I, Ny1I2_2I, Ny1I3_2I,
insI1_2I, NinsI2_2I, NinsI3_2I, NfullI2I, Nf1I1_1I,
Nf1I2_1I, Nf1I3_1I, Ng1I1_2I, Ng1I2_2I, Ng1I3_2I,
NremI1I, NremI2I, NremI3I, B1Idle, B2Idle, B3Idle, ENABLE) ;
p_End : (p_TOSubnet) , (EXEC) ;
p_ResetG : (p_G) , (p_NG) ;
p_RunTO : (p_TO) , (p_TOSubnet) ;
p_SetG : (p_NG) , (p_G) ;

endNet.

```

Continuação da listagem II.12.

Da mesma forma que o modelo **R12-17.PN**, a simulação do modelo **R12-19.PN** pelo **ARP** não foi possível. Inicialmente, o modelo não foi compilado, uma vez que possui 163 transições, excedendo o limite de 150 transições do **ARJ**. Após isolar algumas transições como comentários, a rede foi compilada. Contudo, por apresentar comportamento próximo ao da rede da figura 12.17, o simulador também travou durante a geração do grafo de classes, devido a um grande número de alternativas de disparo, envolvendo determinados grupos de transições concorrentes. Conforme explicado anteriormente, estes grupos propiciam diferentes seqüências de disparo, as quais resultam em bifurcações que convergem para um mesmo ponto (classe).

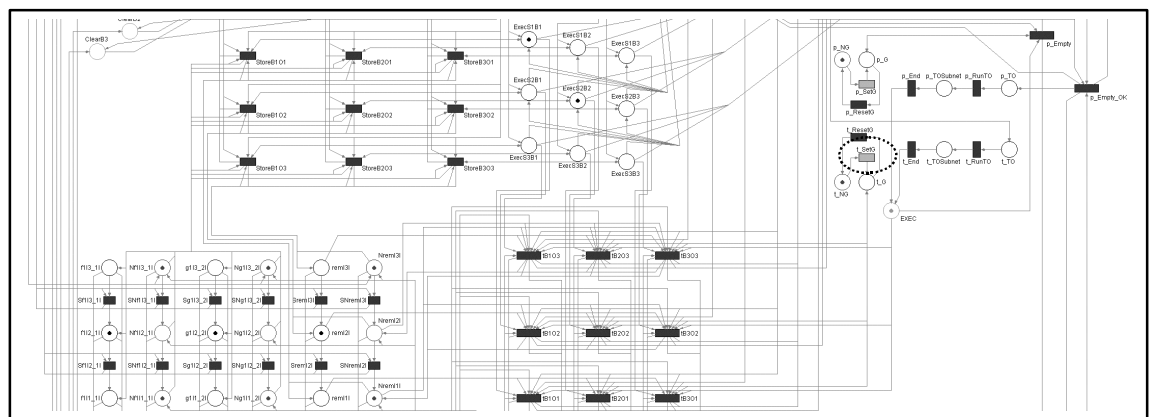
Assim, adotou-se a **simulação interativa** do **JARP**, procurando-se identificar grupos de transições nos quais diferentes seqüências de disparo resultassem em uma mesma classe, seguindo-se os procedimentos adotados para o modelo **JARP** da figura II.24.

A simulação interativa é desenvolvida nas figuras seguintes, tendo como ponto de partida a marcação presente na figura II.63. Cada uma destas figuras representa uma classe do modelo, na qual as transições habilitadas encontram-se em branco (na maioria das figuras, somente trechos com transições habilitadas são mostrados). Indica-se por uma elipse tracejada a transição (ou transições) escolhida(s) para disparo. Estas figuras foram geradas pelo **JARP**, o qual permite salvar, durante a simulação interativa, a marcação corrente mostrada na tela, em arquivo no formato gráfico.

Na marcação presente na figura II.63, as guardas de **t** e **p** são *falsas* e as duas colunas do **repositório de entrada** estão cheias. Assim, há três transições que podem ser disparadas, quais sejam, **p_SetG**, **t_SetG** e **RunS₁B₁**. De início, opta-se que transições relacionadas a **p** não sejam disparadas, pois ainda não há o que ser esvaziado. Como o modelo da figura II.24 está contido no modelo da figura II.63, inclusive no que se refere à marcação inicial dos lugares em comum, será utilizada, para "espalhar" alguns *tokens* pela rede, a seqüência inicial de disparo de transições adotada para a rede da figura II.24. Após esta seqüência inicial, a transição **p** será habilitada e disparada, promovendo as ações esperadas (remoção de todos os *tokens* presentes nos **repositórios de entrada** e de *saída*, assim como o restauro das instâncias à sua marcação *default*).

Para evitar confusão na designação das classes referentes aos modelos **JARP** das figuras II.24 e II.63, serão utilizados, deste ponto em diante, rótulos **C_i**, para designar classes decorrentes de **C₀** (a classe inicial da rede ilustrada na figura II.24), e rótulos **E_j**, para designar classes decorrentes de **E₀** (a classe inicial da rede mostrada na figura II.63).

O disparo da transição habilitada **StoreB₂O₂**, na classe **E₁**, conduz a rede à classe **E₂** (figura II.65), a qual é equivalente à classe **C₄** (figura II.29).



Até o momento (classe $\mathbf{E}_2$), temos que:

- i) dois *samples* foram removidos do **repositório de entrada**;
- ii) a instância $\mathbf{B}_1$, associada ao *sample* $\mathbf{S}_1$, encontra-se em execução; e
- iii) a instância $\mathbf{B}_2$, associada ao *sample* $\mathbf{S}_2$, foi executada e seus resultados foram armazenados na segunda linha do **repositório de saída** – contudo, estes resultados não podem ser removidos, uma vez que a guarda de $\mathbf{t}$ é *falsa*.

419

Finalmente, na classe **E₄**, ocorre o disparo de uma transição relativa a **p**, qual seja, **p_SetG**. Contudo, observa-se na classe gerada por este disparo – **E₅** (figura II.68) – que, independentemente do *intervalo de disparo* de **p**, a transição representante da mesma – **p_Empty** – não encontra-se habilitada, devido ao fato da transição **t** possuir *acesso exclusivo* ao lugar **P**, obtido com a remoção do *token* do lugar **EXEC** (obs.: a menção ao intervalo de disparo de **p** deve-se ao fato de, a operação de deslocamento de *tokens*, desencadeada pelo disparo de **t**, ser realizada em tempo zero – mesmo que **p** tivesse intervalo [0,0], ela não estaria habilitada, devido à ausência de *token* no lugar **EXEC** associado a **P**).

A classe **E₅**, a exemplo da **E₄**, também é equivalente à classe **C₆**, uma vez que estão sendo considerados apenas os lugares em comum, nos modelos **JARP** correspondentes às figuras 12.17 e 12.19.

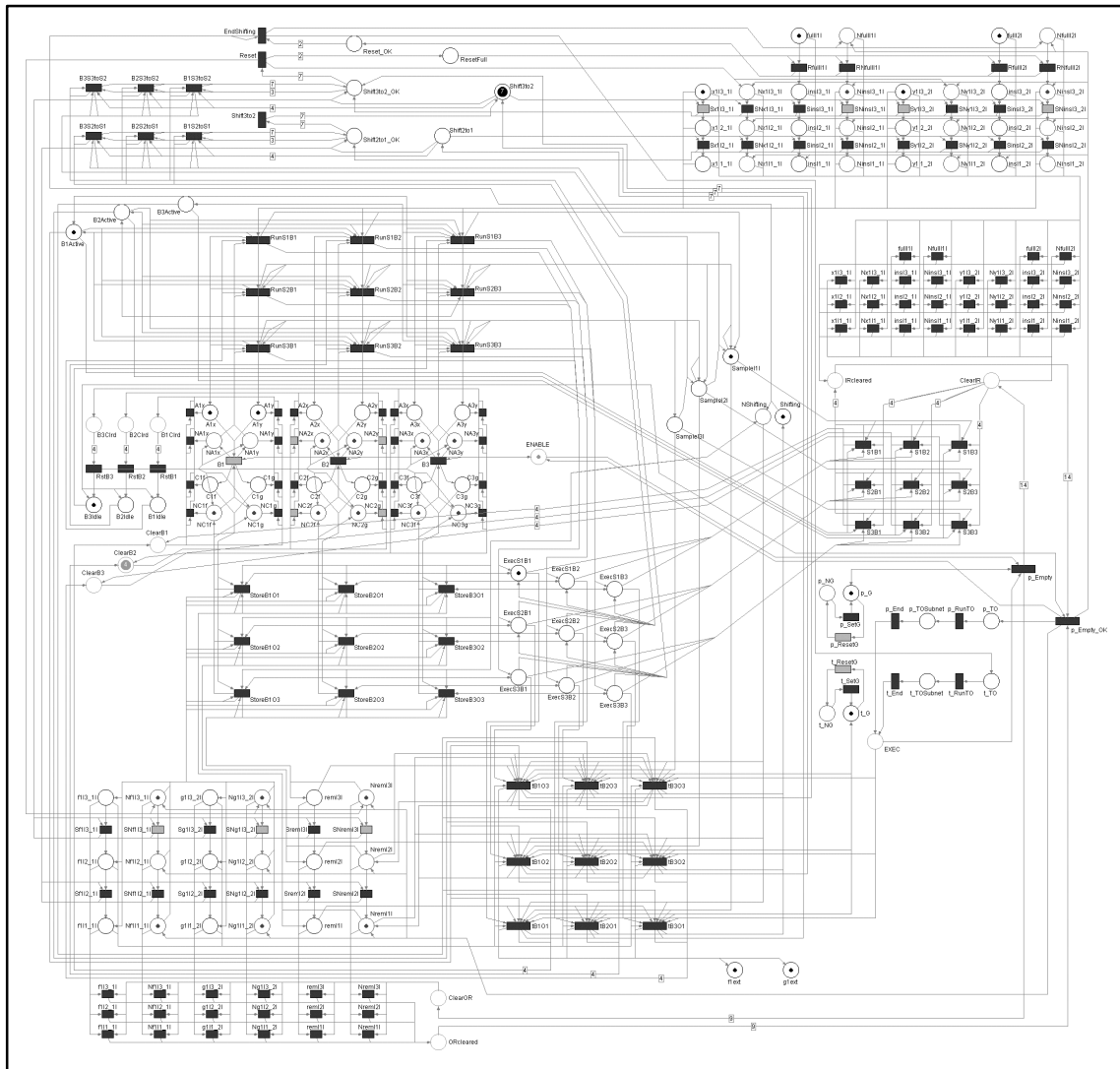


Figura II.68. Classe E₅ – equivalente à classe C₆ (figura II.31).

Para que a transição **p_Empty** fique habilitada, é necessário o encerramento do processo envolvendo o disparo de **t**, ou seja, é necessário que a transição **t_End** dispare, restituindo o *token* ao lugar **EXEC** associado a **P**. Isto é feito para garantir a atomicidade do disparo de uma transição.

Desta forma, deve-se prosseguir com a sequência de disparo utilizada para o modelo da figura II.24, até atingir-se a classe na qual a transição **t_End** fique habilitada. Isto ocorre na classe **C₁₂** (figura II.37), a qual é representada na figura II.69, sob a denominação de **E₆**.

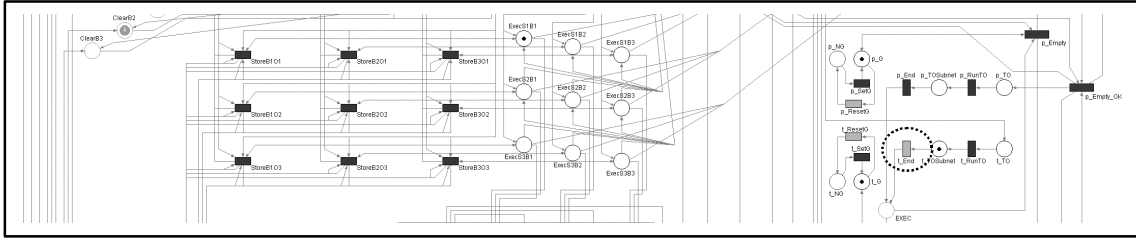


Figura II.69. Classe E₆ – equivalente à classe C₁₂ (figura II.37).

O disparo de **t_End** conduz a rede à classe **E₇**, na qual a transição **p_Empty** fica habilitada (devido à restituição do *token* a **EXEC**), conforme pode ser visto na figura II.70.

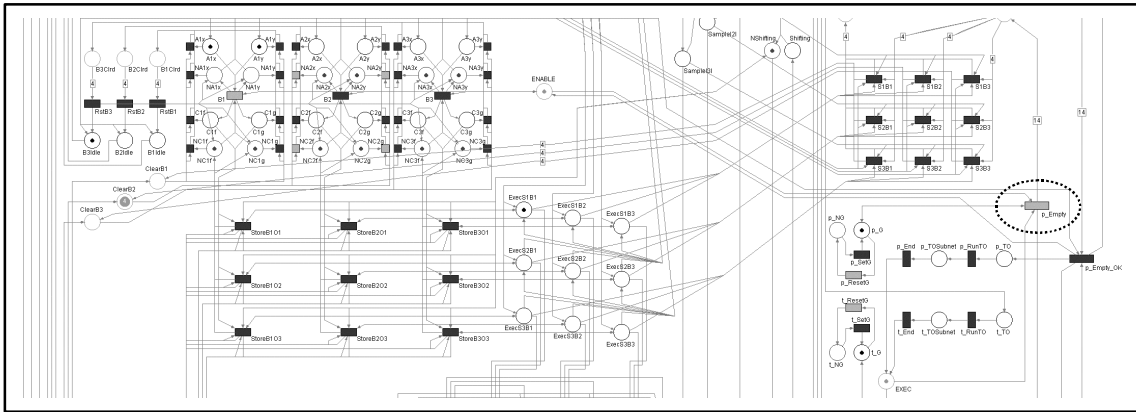


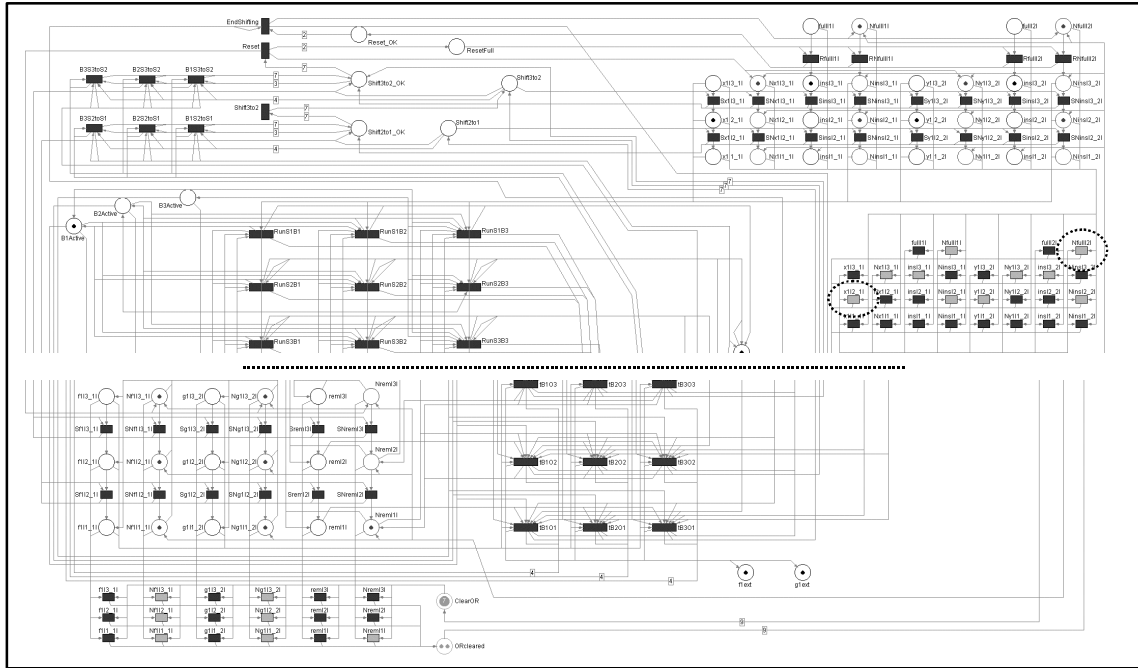
Figura II.70. Classe E₇.

Na classe **E₇**, a instância **B₁** encontra-se em execução, ao passo que a instância **B₂** ainda não foi restaurada à sua marcação *default*, o que poderia ser feito através do disparo das transições habilitadas **NA2x**, **NA2y**, **NC2f** e **NC2g**. O disparo destas transições conduziria a rede a uma classe equivalente a **C₁₄** (figura II.39). Da mesma forma, o disparo de **B₁**, encerrando a execução da instância **B₁**, conduziria a rede a uma marcação já analisada.

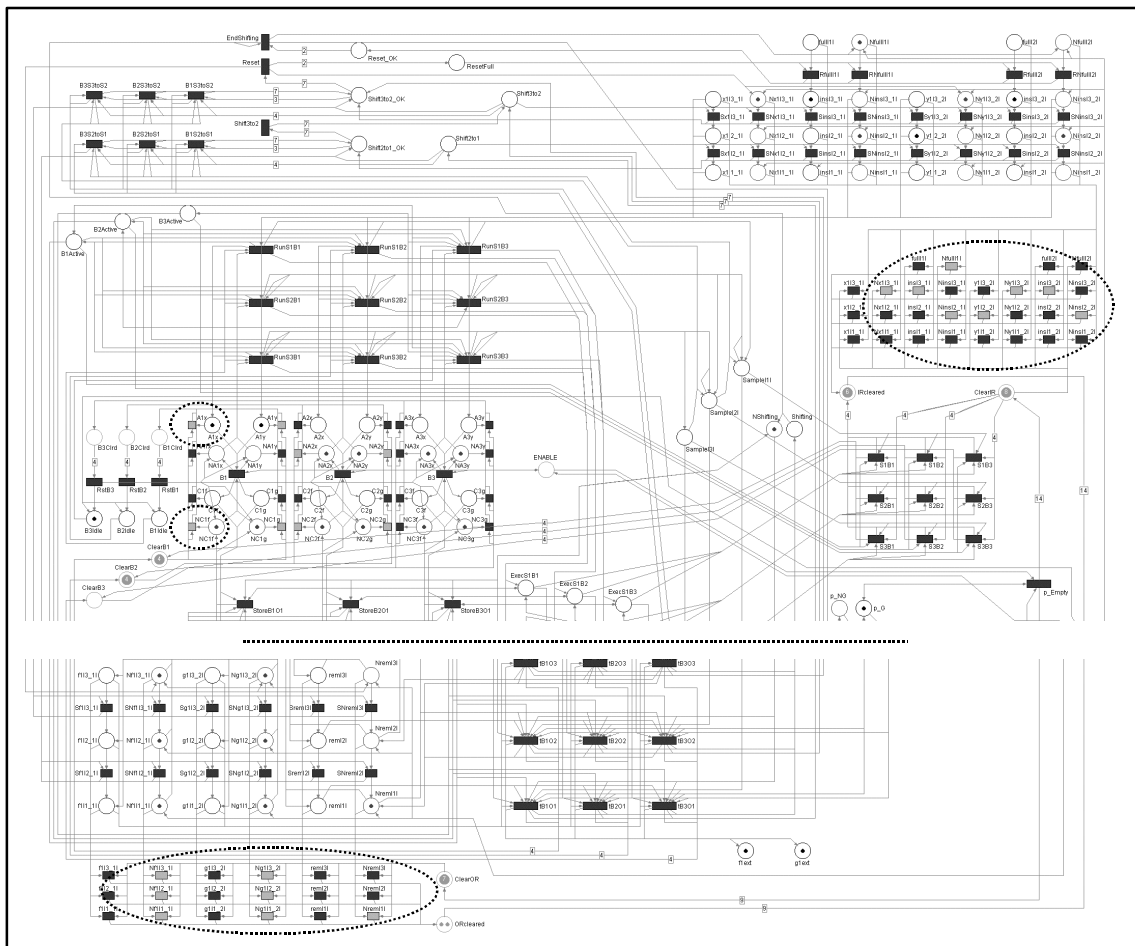
Assim, como a transição **p_Empty** encontra-se habilitada e, considerando-se que há *tokens* espalhados em diversos pontos da rede, opta-se por disparar tal transição, gerando a classe **E₈**, mostrada na figura II.71. Caso **p_Empty** não fosse disparada na classe **E₇**, ela poderia ser disparada em qualquer outra classe decorrente de **E₇**, desde que o lugar **EXEC** possuíse um *token* e a guarda de **p** fosse *verdadeira*. Isto significa que, enquanto não disparar uma transição representante de **t** ou a transição **p_ResetG**, **p_Empty** poderá ser disparada.

Observe, na figura II.71, que o disparo de **p_Empty** promoveu a remoção dos *tokens* presentes em **EXEC** e **ENABLE**, assim como a inserção de 14 *tokens* em **ClearI_R** e de 9 *tokens* em **ClearO_R**, o que está de acordo com as convenções adotadas na seção 12.3.3, para a construção da figura 12.19. De acordo com tais convenções, o disparo de **p_Empty** desencadeia o restauro, ao estado original (restituição à marcação *default*), das redes representantes do **repositório de entrada** (a partir da inserção de 14 *tokens* em **ClearI_R**), do **repositório de saída** (a partir da inserção de 9 *tokens* em **ClearO_R**) e das instâncias (nas quais o restauro do estado original se dará após a remoção de todos os *tokens* do **repositório de**

As transições habilitadas na classe E_8 podem ser disparadas em qualquer ordem, uma vez que são concorrentes. Assim, a classe E_9 (figura II.72) é decorrente da classe E_8 , mediante o disparo das transições $Nrem[2]$ e $Nrem[3]$. O disparo de $Nrem[2]$ é responsável pela remoção do *token* presente no lugar $Nrem[2]$, assim como pela transferência de um *token* de $ClearO_R$ para $O_Rcleared$. A transição $Nrem[3]$ apresenta efeito análogo.



- a) a remoção dos *tokens* presentes nos lugares de controle **Sample[1]**, **B₁Active** e **ExecS₁B₁**;
- b) a transferência de 4 *tokens* de **ClearI_R** para **I_Rcleared** (os quais ficariam excedentes, após a remoção de todos os *tokens* do **repositório de entrada**, devido ao fato de, quando do disparo de **p_Empty**, haver um *sample* associado a uma instância em execução – o que deixa uma das linhas do **repositório de entrada** vazia); e
- c) a inserção de 4 *tokens* no lugar **ClearB₁**, necessários à remoção dos *tokens* presentes na instância **B₁**.



A figura II.75 mostra a classe $\mathbf{E}_{12}$, decorrente de $\mathbf{E}_{11}$, através do disparo das transições destacadas na figura II.74, dentre as quais encontram-se as transições de remoção $\mathbf{A1x}$ e $\mathbf{NC1f}$, relativas à subrede representante de $\mathbf{B}_1$.

Para concluir a remoção de *tokens* da subrede representante da instância $\mathbf{B}_1$, faz-se necessário o disparo das transições $\mathbf{A1x}$ e $\mathbf{NC1g}$, o que ocorre na classe $\mathbf{E}_{12}$, gerando a classe $\mathbf{E}_{13}$ (figura II.76), na qual a presença de 4 *tokens* em $\mathbf{B1Clrd}$ habilita a transição $\mathbf{RstB1}$.

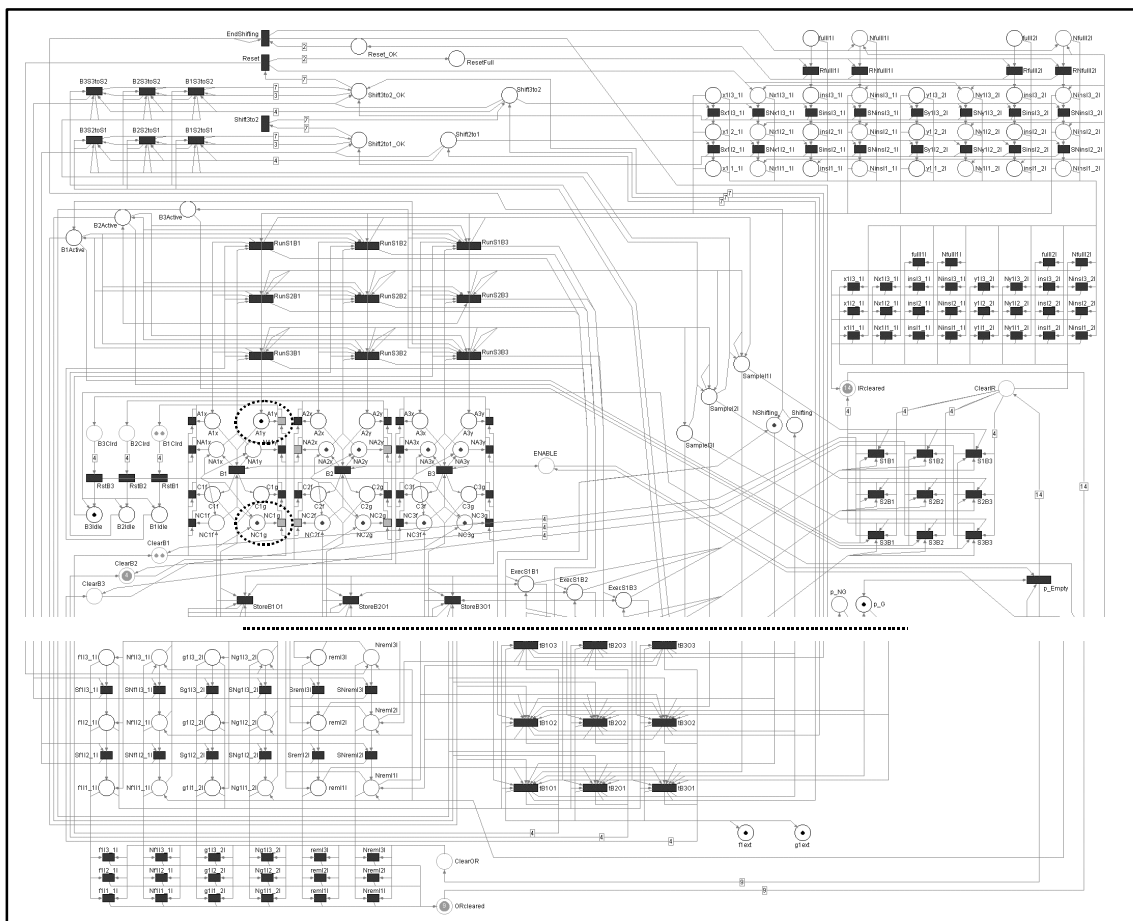


Figura II.75. Classe E₁₂.

Na classe E₁₃, ao invés do disparo de **RstB₁**, opta-se pelo disparo das transições relativas à remoção dos *tokens* da subrede correspondente à instância **B₂**, as quais já estavam habilitadas antes do disparo de **p_Empty**. A classe gerada, E₁₄, encontra-se na figura II.77.

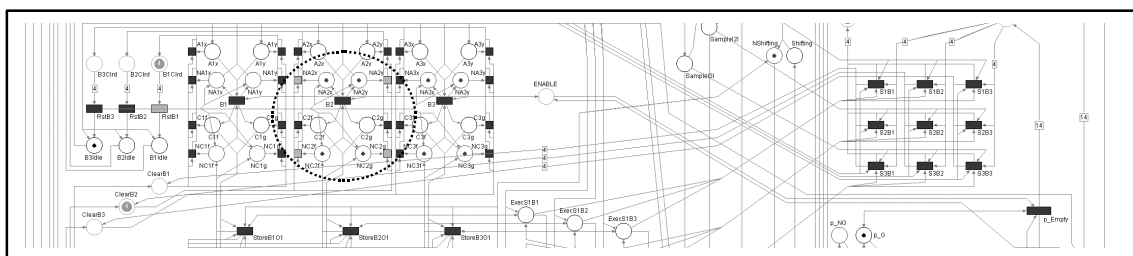


Figura II.76. Classe E₁₃.

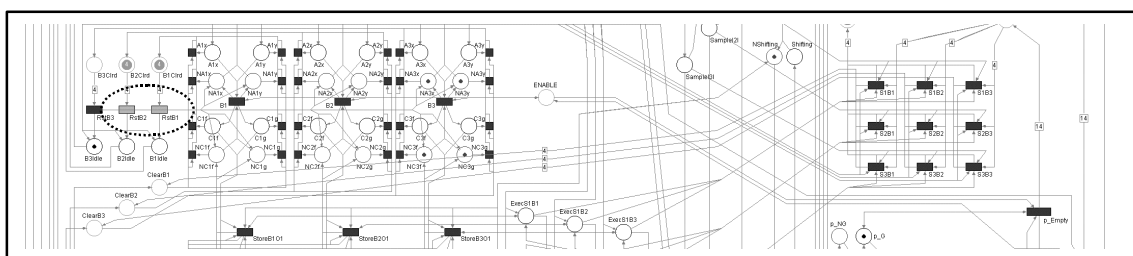
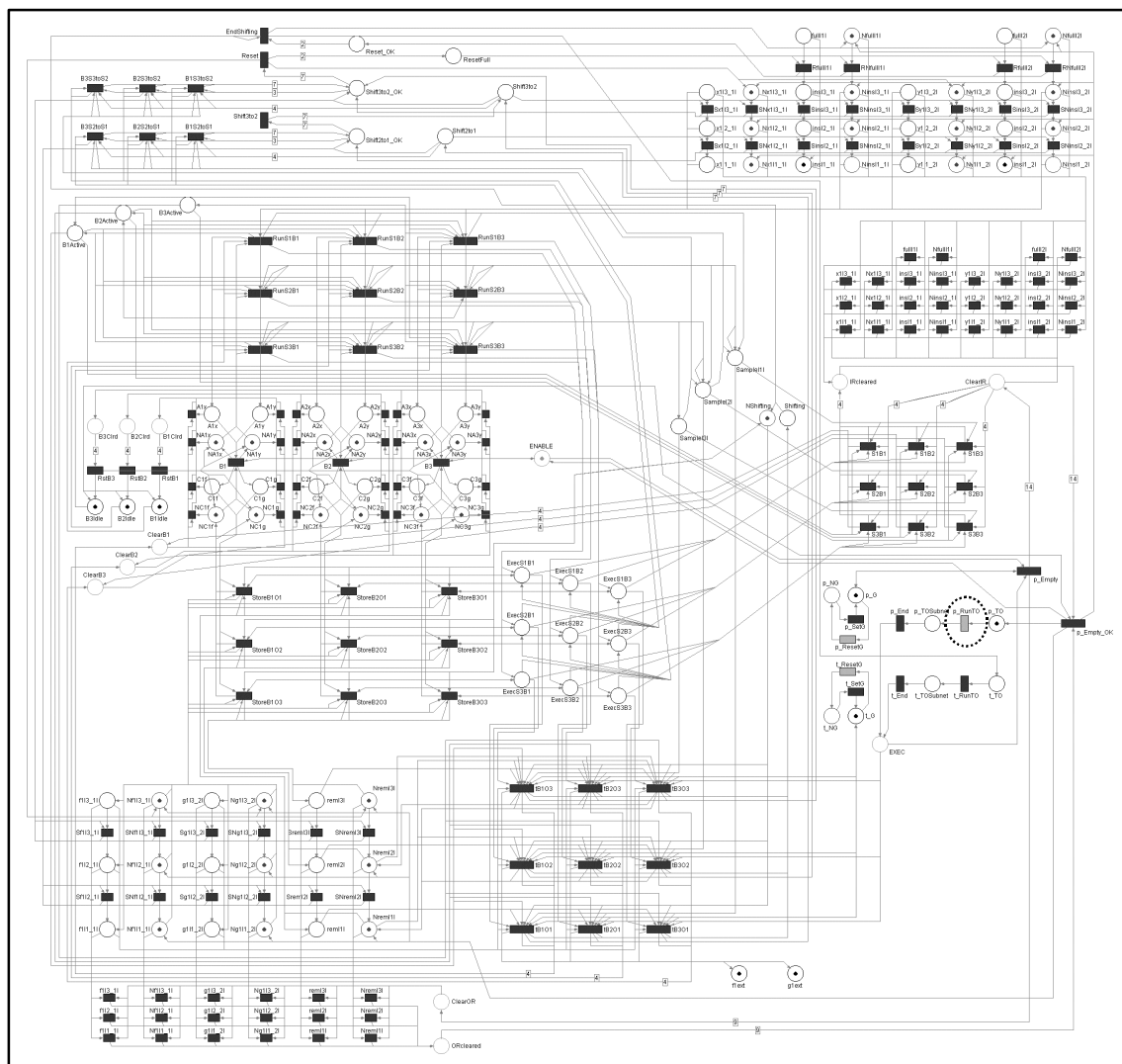


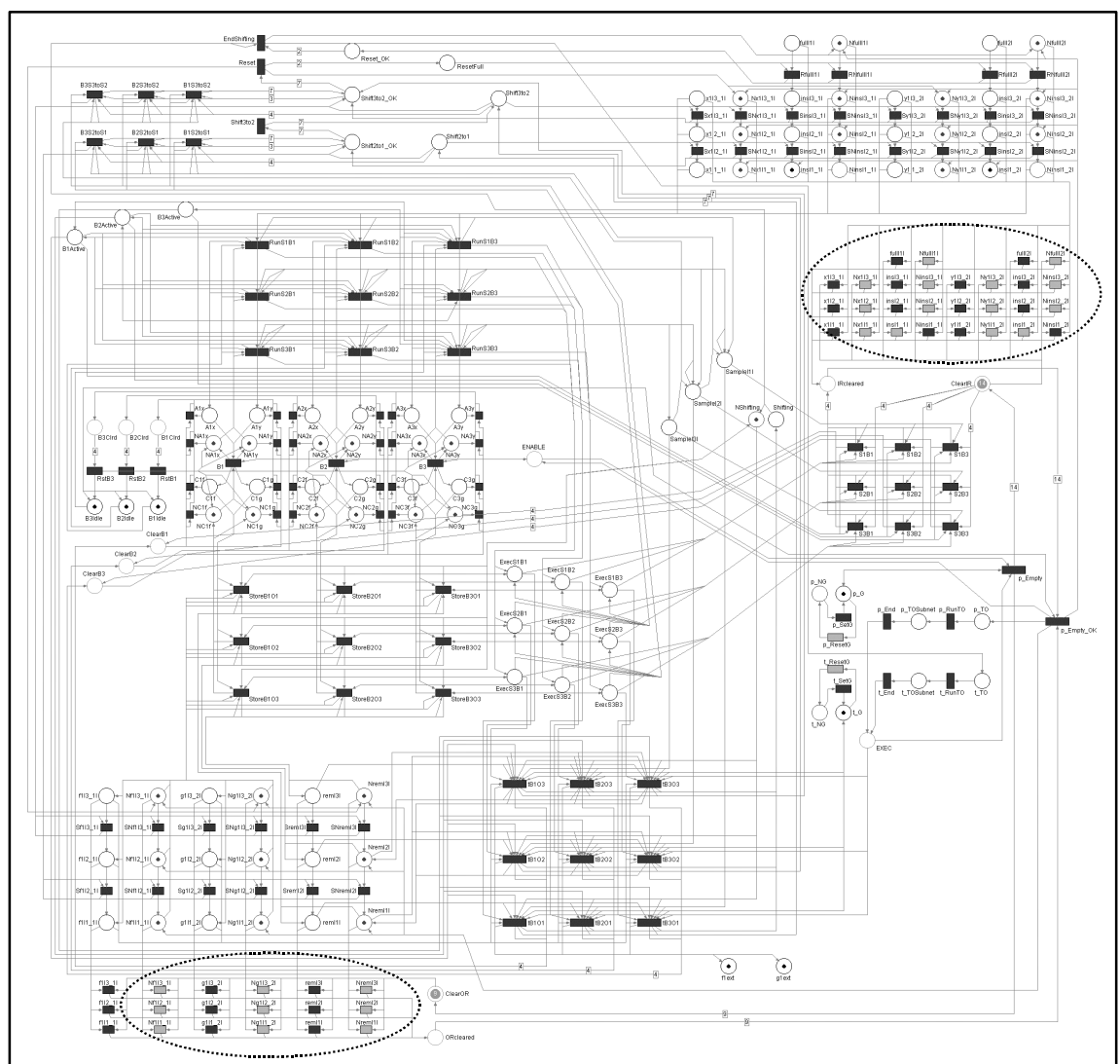
Figura II.77. Classe E₁₄.



Na classe **E₁₇**, o disparo de **p_End** sinaliza a conclusão das *operações* associadas a **p** e, por conseguinte, o término do disparo de **p**. Este disparo gera a classe **E₁₈** (figura II.81), na qual observa-se a restituição do *token* ao lugar **EXEC** (**p** não tem mais acesso exclusivo a **P**).

Observa-se, na classe **E₁₈**, que a transição **p_Empty** volta a ficar habilitada, uma vez que a guarda de **p** continua *verdadeira* (*token* em **p_G**). Na RPSD da figura 12.19, a *operação* **Error := 0**, associada à transição **p**, é responsável pela não habilitação consecutiva da transição **p**. Contudo, no modelo **JARP** da figura 12.19, a subrede representante das operações associadas a **p** é substituída por um único lugar, qual seja, **p_ToSubnet**. Disso resulta a não passagem da guarda de **p** para *falso*, o que deveria ter ocorrido com o disparo de **p_End** (ocorrido na classe **E₁₇**). Por outro lado, é possível, logo após o disparo de **p**, que alguma transição da RPSD, não presente na figura 12.19, atribua o valor **5** à variável **Error**,

Desta forma, a simulação admitirá que a guarda de **p** é *verdadeira*, para que se possa acompanhar o efeito do disparo de uma transição sobre um lugar funcional já vazio. Esta abordagem gera a classe **E₁₉** (figura II.82), a partir do disparo de **p_Empty** na classe **E₁₈**.



429

Embora o lugar funcional **P** se encontre sem *tokens* na classe **E₁₉**, a mesma constatação não é válida para a RPTM equivalente a **P**, devido às convenções adotadas (as quais estabelecem, como marcação *default*, a presença de um *token* em cada um dos lugares **negativos**, nas subredes representantes dos **repositórios** e das instâncias). Por esta razão, as transições associadas à remoção dos *tokens* dos lugares **negativos** das subredes representantes dos **repositórios**, ficam habilitadas. Em relação às instâncias, não há transições de remoção habilitadas, pois os lugares **B₁Idle**, **B₂Idle** e **B₃Idle** já possuem um *token* cada.

A classe **E₂₀** (figura II.83), gerada a partir da classe **E₁₉**, após o disparo das transições destacadas na figura II.82, é equivalente à classe **E₁₅** (figura II.78).

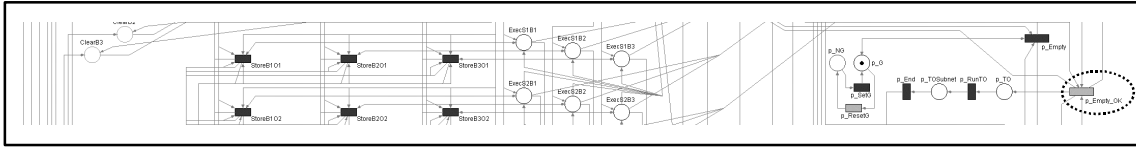


Figura II.83. Classe E₂₀ – equivalente à classe E₁₅ (figura II.78).

Após o disparo das transições **p_EmptyOK** e **p_RunTO**, atinge-se a classe **E₂₁** (figura II.84), a qual é equivalente à classe **E₁₇** (figura II.80).

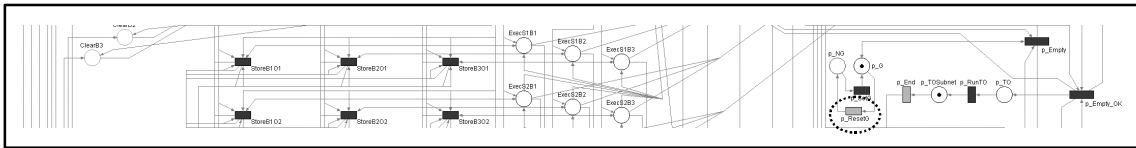


Figura II.84. Classe E₂₁ – equivalente à classe E₁₇ (figura II.80).

Na classe **E₂₁**, simula-se a execução da **operação** associada a **p** (qual seja **Error := 0**), a qual torna **falsa** a guarda de **p**, através do disparo da transição **p_ResetG**. Este disparo conduz a rede à classe **E₂₂**, mostrada na figura II.85.

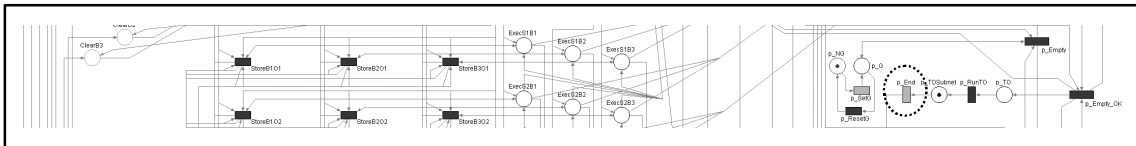


Figura II.85. Classe E₂₂.

Finalmente, na classe **E₂₂**, ocorre o disparo da transição **p_End**, o qual encerra o disparo de **p**, resultando na classe **E₂₃**, ilustrada na figura II.86.

Como não há outras alternativas a serem tratadas, que venham a agregar informações, encerra-se a simulação do modelo **JARP** da figura II.63. Pelo exposto nesta simulação, pode-se concluir que o funcionamento de tal modelo é compatível com as convenções dadas no capítulo 12, tanto no que se refere à representação de lugares funcionais, como no que se refere à representação do disparo de uma transição, conectada a um lugar funcional, por um **ramo de entrada preemptivo**. Na explanação feita para as classes compreendidas entre **E₀** (figura II.63) e **E₂₃** (figura II.86), observou-se a **garantia de atomicidade** do disparo de **p**, o **processo de remoção de tokens** em ambos os **repositórios** e nas **instâncias ativas** (e seus controles associados) e, também, o **restauração dos repositórios e das instâncias** à sua **marcação default**. Destaque foi dado a **grupos de transições concorrentes**, nos quais diferentes seqüências de disparo convergem a uma mesma classe (estado).

Assim, pelo exposto, o modelo correlato ao da figura II.63, descrito na figura 12.19, é funcional e está de acordo com as convenções utilizadas em sua concepção, sendo, portanto, um modelo válido.

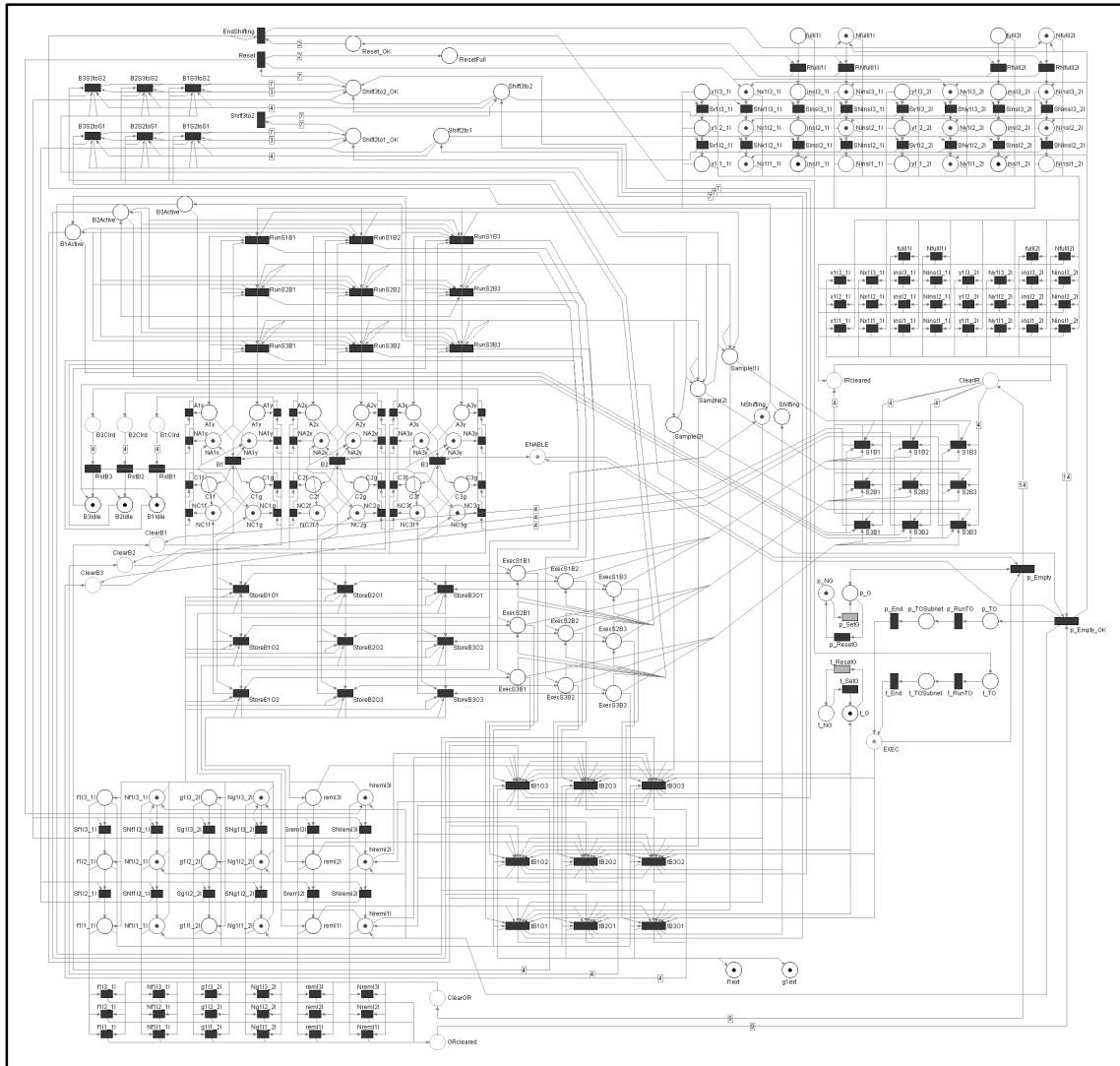


Figura II.86. Classe E₂₃.

As simulações realizadas neste apêndice permitem validar as representações, em **redes de Petri Temporizadas de Merlin (RPTM)**, dos modelos **RPSD** de conexões envolvendo transições e lugares dos tipos I, II e III. Tal validação decorre do fato de tais representações mostrarem-se compatíveis com o funcionamento esperado, o qual, por sua vez, é determinado tanto pela definição dos diferentes componentes de **RPSD** (lugares, transições, ramos e memória da rede), quanto pela adoção de um conjunto de convenções de representação.

Destarte, como os diferentes elementos de **RPSD** possuem representações funcionais em **RPTM** e, sendo **RPTM** uma extensão válida de **rede de Petri**, conclui-se que **RPSD** é, também, uma extensão válida de **rede de Petri**.