UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Desenvolvimento Metodológico de Sistemas Distribuídos Abertos

Maria Adela Romero Alvarez

Orientador: **Prof. Dr. Eleri Cardozo**

Dissertação de Mestrado

Este exemplar corresponde à redação final da defendida por Maria Adela Romero Alvarez e aprovada pela Comissão Julgadora em 19/12/1996

Eleri Cardozo
Orientador

Campinas - SP Brasil
1996

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

R664d

Romero Alvarez, Maria Adela
Desenvolvimento metodológico de sistemas distribuídos abertos / Maria Adela Romero Alvarez.--Campinas, SP: [s.n.], 1996.

Orientador: Eleri Cardozo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Software - Desenvolvimento. 2. SDL (Linguagem de programação de computador). 3. Processamento distribuído. 4. Sistemas abertos (Computadores). I. Cardozo, Eleri. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Desenvolvimento Metodológico de Sistemas Distribuídos Abertos

Maria Adela Romero Alvarez
Orientador: **Prof. Dr. Eleri Cardozo**

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Elétrica, Área de concentração: Automação Industrial

Campinas - SP Brasil
1996

Dedico este trabalho aos meus pais
Gustavo e Adela e aos meus irmãos Jorge
e Mily, pela sua presença sempre
alentadora apesar da distância. Saibam
que os admiro e amo profundamente.

Agradecimentos

Ao CNPq, pelo apoio financeiro recebido.

Aos meus pais pelo apoio dado às minhas iniciativas e por se preocuparem pela minha formação pessoal e profissional.

Ao Prof. Eleri, por acompanhar o desenvolvimento deste trabalho.

Ao Ricardo e à tia Juanita, por me animar a começar; a Ludvik, por confiar em mim, e pela sua ajuda desinteressada.

Ao Renato, quem me acompanhou na etapa mais difícil do trabalho e a quem devo o término deste trabalho. A sua ajuda foi fundamental. As suas perguntas, as correções dos textos, as discussões me ensinaram a pensar e analisar cada ponto com mais detalhe. Obrigada também pelas caronas, as sopinhas quentes! por me dar ânimo, força e por me ajudar a recuperar a confiança em mim mesma. Obrigada pela paciência, por me compreender e por me amar.

À Lucimara por ter a paciência de sentar ao meu lado e me ensinar. Os seus conhecimentos foram de grande ajuda.

Às pessoas das minhas repúblicas: Nilda e Marcelo, pela paciência que tiveram comigo; às Cris's, pelo ambiente legal que me brindaram e os grandes momentos que tivemos, pela sua amizade incondicional, e por estar comigo apesar de tudo.

Ao Sr. Ripper, Afira, Stella, Isolda e Bruna por me aceitar, acolher e me fazer sentir filha, irmã e tia.

Aos meus amigos Moi, Marcelo, Jugurta, Julimara, que me ajudaram no processo de adaptação.

Ao meus amigos do DCC, por me darem muitas alegrias, muitos incentivos e por contribuir a me sentir em casa.

Aos meus grandes amigos e companheiros do LCA, Silvinha, Paulinho, Claudinha Lucimara, Anderson, Affonso, Gonzaga, Jussara, Douglas, Well, Mauro, Lucci, Leo, Raquel, Marta, entre muitos outros, que fazem deste ambiente um lugar de encontro, amizade, força, calor humano e colaboração.

A Teté por estar "sempre da minha parte".

A ti Padre Amoroso, por me dar vida e saúde. Por receber de ti muita força valor e entusiasmo.

A satisfação está no esforço e
não apenas na realização final.
Ghandi.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

ROMERO, A. M. A., Desenvolvimento Metodológico de Sistemas Distribuídos Abertos.

Campinas: DCA/FEEC/UNICAMP, 1996. (Dissertação de Mestrado)

No desenvolvimento de Sistemas Distribuídos Abertos o uso de recursos que facilitem o entendimento do sistema, a verificação do seu comportamento, a implementação e a documentação são muito importantes.

O objetivo deste trabalho é empregar práticas de Engenharia de Software ao processo de desenvolvimento de Sistemas Distribuídos Abertos através da aplicação de métodos, técnicas e ferramentas adequadas. Um estudo de caso foi realizado abordando requisitos temporais e de comunicação, próprios de aplicações distribuídas. Foi utilizado o método OMT (Object Modeling Technique) na modelagem estática do sistema e a técnica SDL (Specification and Description Language) na parte dinâmica e funcional. Duas ferramentas foram utilizadas para edição dos modelos, simulação do comportamento do sistema, geração de código e documentação do sistema. São apresentados os resultados obtidos bem como uma avaliação dos métodos, técnicas e ferramentas utilizados. Conclui-se que a combinação de OMT e SDL e a utilização das ferramentas escolhidas beneficiam significativamente o processo de desenvolvimento de Sistemas Distribuídos Abertos.

Palavras-Chaves: Software - Desenvolvimento. SDL (Linguagem de programação de computador). Processamento Distribuído. Sistemas abertos (computadores).

Abstract

ROMERO, A. M. A., Metodological Development of Open Distributed Systems.

Campinas: DCA/FEEC/UNICAMP, 1996. (Masters)

The development of open distributed systems is better conducted when the activities related to overall system understanding, behavior verification, implementation, and documentation are well supported.

The objective of this work is to use Software Enginneering practices in the development of open distributed systems, proposing an appropriate combination of methods, techniques, and tools. A study was carried out taking into account temporal and communication requirements, characteristics very common in distributed applications. The combination mentioned above includes the OMT method for modeling the static aspects of the system, the SDL technique for modeling the dynamical and functional aspects, and CASE tools for graphical edition of the diagrams, simulation, code generation, and documentation. An evaluation of the methods, techniques and tools used is also presented.

One conclusion of this evaluation reveals that the proposed combination of methods improve several aspects of the open distributed systems development process, as reported in the thesis.

Key-words: Software - Development. SDL (computer programming language). Distributed Processing. Open Systems (computers).

Lista de Abreviações

SDL - Specification and Description Language

FDT - Formal Description Technique

ADT - Abstract Data Type

CORBA - Common Object Request Broker Architecture

IDL - Interface Definition Language

OMT - Object Modeling Technique

OMG - Object Management Group

SDA - Sistemas Distribuídos Abertos

ST - Serviço de Transações

MSC - Message Sequence Chart

ORB - Object Request Broker

EFSM - Extended Finite State Machine

CASE - Computer Aided Software Engineering

IPC - Inter Processes Comunication

1. Introdução

1.1. Objetivo

Atualmente, o desenvolvimento de aplicações na área de Sistemas Distribuídos Abertos (SDAs) carece de métodos, técnicas e ferramentas que visem tornar este processo mais efetivo, desde as fases iniciais até a geração de código. A efetividade do processo nos mais diferentes aspectos fica mais evidente devido à grande complexidade comum a esta área de aplicações.

Para resolver este problema, o presente trabalho visa avaliar o emprego de métodos e ferramentas "bastante sofisticados" no desenvolvimento de uma aplicação da área de SDA, denominado Serviço de Transações (ST). Uma conclusão desta avaliação também é apresentada.

1.2. Escopo do trabalho

No presente trabalho, foram aplicados o método OMT (Object Modeling Technique) a técnica de especificação formal SDL (Specification and Description Language), utilizando as ferramentas CASE (Computer Aided Software Engineering) disponíveis tais como LOV/OMT e GEODE, para auxiliar a modelagem e a implementação de um SDA como é o caso do ST.

As características importantes a serem modeladas em SDAs são principalmente as relacionadas com a dinâmica temporal e a comunicação.

O ST é especificado em termos de objetos. Sendo assim, a utilização do método OMT se deve à necessidade de uma abordagem de orientação a objetos para este sistema e sua ênfase à modelagem estática dos sistemas.

Os SDAs, e particularmente o ST, são sistemas cuja parte dinâmica é crítica. Possuem restrições temporais e muita interação entre as suas partes. Portanto, se faz necessário um método que permita modelar da melhor forma possível estas características. A utilização de SDL se deve principalmente à abordagem dada aos aspectos dinâmicos e à parte temporal dos sistemas, podendo-se modelar facilmente conceitos como: *time outs*, troca de sinais, concorrência, comunicação, etc. SDL facilita também a representação gráfica do comportamento através de diagramas de

transição de estado. SDL pode ser aplicada nas áreas de telecomunicações, sistemas de tempo real, e sistemas distribuídos[18]. É importante considerar que SDL é uma técnica padronizada pela ITU-T [7]. Sendo uma técnica formal bem próxima à implementação, SDL permite a simulação do modelo e a geração automática de código.

A ferramenta CASE GEODE que suporta SDL, permite a edição dos diagramas, a detecção automática de erros de sintaxe, a simulação da especificação, e a geração de código.

A ferramenta CASE LOV/OMT que suporta o método OMT permite principalmente a edição dos modelos e a construção de um dicionário de dados para auxiliar a documentação.

1.3. Motivação

Dada a complexidade de SDAs, se faz necessário um trabalho mais especializado desde as etapas iniciais do desenvolvimento. Para conseguir isto, seria necessário um trabalho mais formalizado para beneficiar a documentação, favorecer a continuidade dos diversos trabalhos nesta área, a criação de sistemas modulares e mais consistentes, além de introduzir o uso de métodos, técnicas e ferramentas no desenvolvimento de sistemas distribuído no nosso meio.

No inicio do trabalho, a idéia foi realizar a aplicação de técnicas de modelagem para o desenvolvimento de SDA com o uso de ferramentas CASE. As motivações para realizar este trabalho foram:

- No Departamento de Engenharia de Computação e Automação Industrial (DCA), da Faculdade de Engenharia Elétrica e de Computação tem-se desenvolvido aplicações diversas nas áreas de sistemas distribuídos e de tempo real que possuem como característica um nível alto de complexidade, onde os conceitos de tempo, concorrência, sincronização, são usados freqüentemente. A forma corrente de desenvolvimento não faz uso de metodologias para modelagem, sendo que não existem modelos que descrevem os sistemas desenvolvidos, nem uma formalização do processo de desenvolvimento.

  Visando melhorar o processo de desenvolvimento deste tipo de sistemas é necessário um estudo de técnicas e métodos para a modelagem desses sistemas.

- Além da falta de modelos, existe uma deficiência na documentação das aplicações e uma falta de experiência no uso de *software* de auxilio ao desenvolvimento. Dado que o DCA já possui ferramentas de auxilio ao desenvolvimento, é recomendável estudá-las e aplicá-las no processo de desenvolvimento, visando melhorar a documentação e auxiliar a manutenção dos sistemas criados.

- O desenvolvimento de aplicações na área de SDA utilizando modelos formais constitui uma nova opção à forma de desenvolvimento de aplicações a serem realizadas no departamento.
- O uso de ferramentas para especificação formal e simulação permitiriam validar os modelos facilitando a etapa de implementação.

1.4. Estrutura da dissertação

O presente trabalho está dividido em 5 capítulos. Esta divisão e a seqüência dos capítulos obedece ao próprio processo utilizado no desenvolvimento dos trabalhos.

Inicialmente, foram estudados alguns temas considerados como pré-requisitos: OMT, SDL, e CORBA (Common Object Request Broker Architecture). Uma visão geral dos métodos utilizados tais como OMT e SDL é dada no Capítulo 2. No Capítulo 2 é apresentada também uma visão global do padrão CORBA.

Em seguida deu-se inicio à etapa experimental. No Capítulo 3 é descrito como foi desenvolvido o caso de estudo denominado Serviço de Transações (ST). A avaliação dos resultados obtidos é apresentada no Capítulo 4.

Por último, no Capítulo 5 são feitas as considerações finais de todo o trabalho e propostas de futuras extensões que dão continuidade ao presente trabalho.

2. Revisão Bibliográfica

2.1. Introdução

Os sistemas de *software* necessitam ser especificados ou descritos de modo a capturar da melhor forma possível tanto as suas características quanto as suas restrições. Para alcançar este objetivo são utilizados métodos ou técnicas que ajudam a descrever os sistemas segundo suas particularidades. Nesta seção são descritos dois métodos utilizados na modelagem do ST. Primeiro é apresentado o Modelo de Classes do método OMT e em seguida é abordada a técnica de descrição formal SDL. No final do capítulo é apresentada uma breve revisão sobre o padrão CORBA.

2.2. OMT (Object Modeling Technique)

2.2.1. Introdução

OMT é um método de modelagem criado por Rumbaugh et.al, que tem por objetivo auxiliar o desenvolvimento de *software* através da construção de modelos que o representem. A técnica aborda todos os passos do ciclo de vida de desenvolvimento, tais como análise, projeto e implementação. OMT propõe a construção de três modelos: o Modelo de Classes, que captura as características estáticas do sistema, o Modelo Dinâmico, que captura o comportamento do sistema no tempo e o Modelo Funcional que determina a funcionalidade do sistema, permitindo dessa forma alcançar um melhor entendimento da aplicação a ser desenvolvida. Resumindo, a modelagem em OMT aborda os aspectos estático, dinâmico e funcional vinculados ao desenvolvimento de sistemas. O Modelo de Classes visa mostrar os elementos ou classes que fazem parte do sistema, vinculando uma classe a outra através de linhas de relacionamento entre elas. Cada elemento ou classe possui um nome, as suas características ou atributos e as operações ou serviços que esta classe pode fornecer. O Modelo Dinâmico mostra como os objetos interagem entre si e com o mundo externo, além das mudanças nos seus estados internos[1]. No Modelo Funcional são consideradas as funcionalidades que as classes devem fornecer durante a operação do

[1] Um estado é formado pelos valores dos atributos de um objeto em um determinado instante.

sistema, incluindo as transformações nos valores dos dados usando Diagramas de Fluxo de Dados.

Dado que na modelagem foi usado apenas o Modelo de Classes[2]., nos próximos parágrafos se faz uma descrição deste modelo dando uma visão geral dos aspectos mais importantes. Esta descrição inclui conceitos, notação e exemplos de cada construção. Informações sobre os modelos Dinâmico e Funcional podem ser encontrados em [1].

Uma avaliação sobre as vantagens e os problemas na aplicação do método é apresentada no Capítulo 4. A aplicação do Modelo de Classes no caso de estudo (ST) deste trabalho assim como algumas decisões tomadas com relação ao modelo são apresentadas no Capítulo 3.

Para obter maiores detalhes sobre este método veja a referência [1].

2.2.2. O Modelo de Classes OMT

O início de desenvolvimento de um sistema parte de um estudo daquilo que será desenvolvido através de um modelo que o represente. Este modelo será completado, corrigido e melhorado a medida que o problema vai sendo abordado por outros pontos de vista.

OMT propõe a utilização de modelos em todas as etapas de desenvolvimento, cobrindo os aspectos estático, dinâmico e funcional. Em cada etapa (análise, projeto, ou implementação) são acrescentados mais detalhes até se obter um resultado mais próximo da implementação. Dependendo do nível de abstração enfocado, detalhes não essenciais a um determinado aspecto do problema são ignorados.

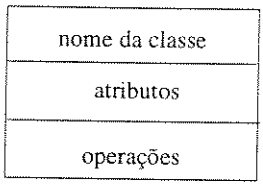
O Modelo de Classes é considerado o mais utilizado dentre os três modelos. Nele são mostrados **classes,** que definem um conjunto de **objetos**. A classe possui **atributos** e **operações** que a caraterizam, e relacionamentos entre si através de **associações**. Como resultado desta modelagem, obtém-se um diagrama de classes. Os conceitos básicos envolvidos na construção deste modelo (classes, objetos, atributos, operações, relacionamentos) são detalhados nos próximos parágrafos.

Classe: conjunto de objetos com as mesmas características (atributos), comportamento (operações) e relacionamentos. As instâncias de uma classe são chamadas de objeto.

As classes são importantes porque providenciam uma abstração do problema, uma visão geral do sistema, suas fronteiras e suas partes constituintes.

[2] Os modelos Dinâmico e funcional foram substituídos pela modelagem em SDL/MSC.

Segundo o diagrama de classes a notação gráfica para classe é a seguinte:



Exemplo: Na Figura 2-1 são apresentados alguns exemplos de classes. A classe arquivo com três atributos (nome_arq, tamanho_arq, ultima_atualiz) e duas operações (registrar, salvar); a classe pessoa com três atributos (nome, endereço, estado civil) sem operações e a classe conta com dois atributos (número, tipo) e uma operação (atualiza).

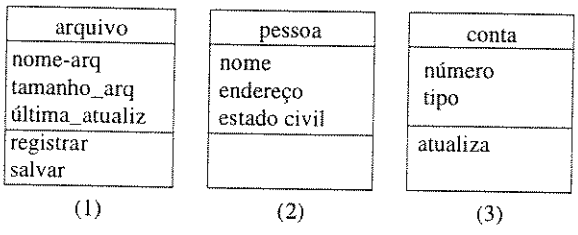
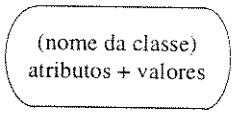


Figura 2-1: Exemplos de classes com a notação gráfica OMT. (1) Classe arquivo, (2) classe pessoa, (3) classe conta.

A identificação das classes é o primeiro passo na construção do diagrama de classes. Quando se identificam classes está-se definindo o conjunto de objetos com características comuns.

Objeto: representação de uma entidade do mundo real[3]. Cada objeto possui uma identificação única, atributos ou valores de dados e comportamento. É também uma instância da classe à qual pertence.

A notação gráfica para objeto é a seguinte:



Exemplo: Na Figura 2-2 são mostrados os objetos de classes definidas na Figura 2-1 com os seus valores de atributos.

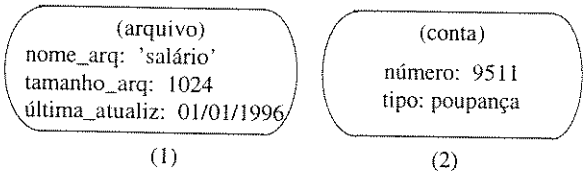


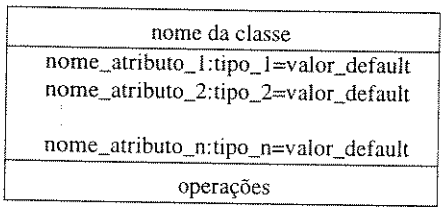
Figura 2-2: Exemplos de instâncias de classes. (1) Instância da classe arquivo, (2) instância da classe conta.

[3] Entidade do mundo real pode ser um conceito, uma coisa, uma abstração que tem sentido dentro do contexto do sistema.

Identificar objetos favorece o entendimento do mundo real e proporcionam uma base para a implementação. Eles podem ser representados utilizando diagramas de objetos ou de instâncias, que descrevem um conjunto particular de objetos e seus relacionamentos.

Atributo: propriedade do objeto descrita em uma classe. Cada objeto possui um valor para cada atributo. Em uma classe, os atributos são especificados por um nome, tipo de dado e um valor *default* (opcional).

A notação gráfica para atributos é mostrada a seguir:



Exemplo: Um exemplo de atributos é mostrado na Figura 2-3. A classe aluno tem como atributos o nome, a idade e o sexo cujo valor *default* é M.

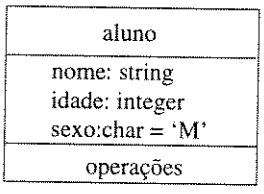


Figura 2-3: Exemplo de atributos de uma classe.

O conjunto de atributos define as características que cada objeto da classe possui.

A notação gráfica de objeto com atributos é mostrada na Figura 2-4, sendo (1) um objeto genérico, e (2) um objeto da classe aluno cujo nome é João, idade: 28 e sexo: 'M'.

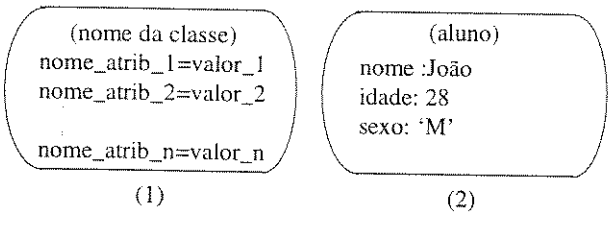


Figura 2-4: Atributos de objeto. (1) Objeto genérico, (2) objeto da classe aluno.

Tanto a descrição dos atributos quanto a descrição das operações fazem parte do Modelo de Classes. Todo objeto de uma classe controla e tem acesso aos valores de seus atributos através de operações.

Operação: funções ou transformações aplicadas ao objeto de uma classe. Todos os objetos de uma classe compartilham as mesmas operações.

Polimorfismo: significa que o mesmo método pode ter diferentes implementações, dependendo da classe em questão.

O Modelo de Classes representa as operações através de uma notação que fornece o nome da operação, uma lista de argumentos de entrada e o tipo de resultado que será gerado. Na Figura 2-5 é mostrada (1) a notação gráfica para operações e (2) um exemplo de aplicação mostrando a classe objeto geométrico com atributos e com as operações: mover cujo parâmetro de entrada é: "delta" do tipo *string*; selecionar com parâmetro de entrada: "p" do tipo "ponto" e com valor de retorno do tipo *boolean*; e a operação rotar com parâmetro de entrada "ângulo".

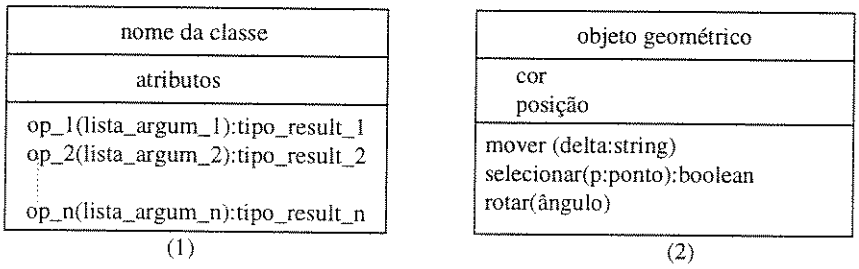


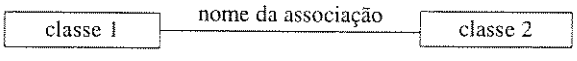
Figura 2-5: Notação e exemplo de operações. (1) notação gráfica para operações. (2) operações da classe objeto geométrico.

A implementação de uma operação em uma classe é chamada de método.

Os objetos que fazem parte de um sistema interagem entre si para fornecer determinados serviços. Por isso existe a necessidade de estabelecer relacionamentos ou associações.

Associação: descreve uma relação entre classes mostrando a existência de uma cooperação entre elas.

A instância de uma associação é denominada ***link***. Estabelece a conexão entre objetos específicos. A notação básica de uma associação está representada por uma linha que une as classes relacionadas e um nome que identifica o tipo de relação existente. Sua notação gráfica é mostrada a seguir:



Dependendo do número de classes relacionadas, as associações podem ser binárias, ternárias e assim por diante.

A seguir serão apresentados alguns exemplos deste conceito. No primeiro exemplo descreve-se uma associação binária entre a classe país e a classe cidade. No exemplo 2 é mostrada uma associação entre três classes diferentes denominada associação ternária. Este último tipo de associação poderia ser representado em função de relações binárias, mas as vezes isto causa perda de informação.

Exemplo 1: A associação binária, apresentada na Figura 2-6, é chamada associação de um-para-um. Representa a relação existente entre os objetos da classe

país e os objetos da classe cidade. Esta associação pode ser interpretada da seguinte forma: "Um país (por exemplo: Brasil) tem como capital uma cidade: Brasília".

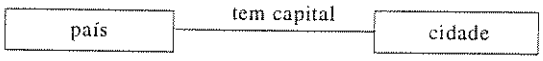


Figura 2-6: Associação binária.

Exemplo 2: A associação ternária mostrada na Figura 2-7, representa a associação existente entre as classes aluno, universidade e orientador. Esta associação pode ser interpretada da seguinte forma: "Um aluno tem um orientador em uma universidade".

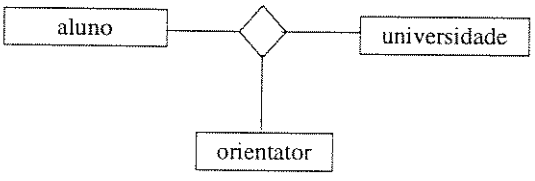


Figura 2-7: Associação ternária.

As associações podem possuir atributos que têm sentido somente na existência da associação (como mostrado no exemplo 3), ou estas associações podem se tornar classes com nome, atributos e operações (como ilustrado no exemplo 4).

Exemplo 3: A Figura 2-8 ilustra uma associação com atributos. Neste exemplo pode-se observar que o salário e o cargo são atributos dependentes da existência da relação "trabalha para" entre a pessoa e a companhia. Sem esta relação, a existência do salário e o cargo não tem sentido.

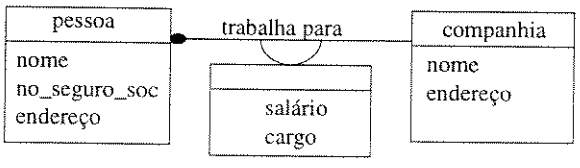


Figura 2-8: Associação com atributos.

Exemplo 4: Este exemplo mostra uma associação representada como uma classe conforme ilustrado na Figura 2-9. Considerando o mesmo exemplo da Figura 2-8 os atributos que pertencem à relação "trabalha para" podem ser modelados como uma classe, podendo ter operações a serem executadas sobre os valores dos atributos.

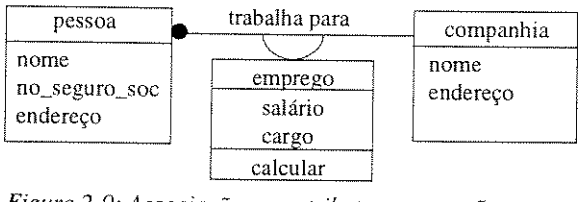


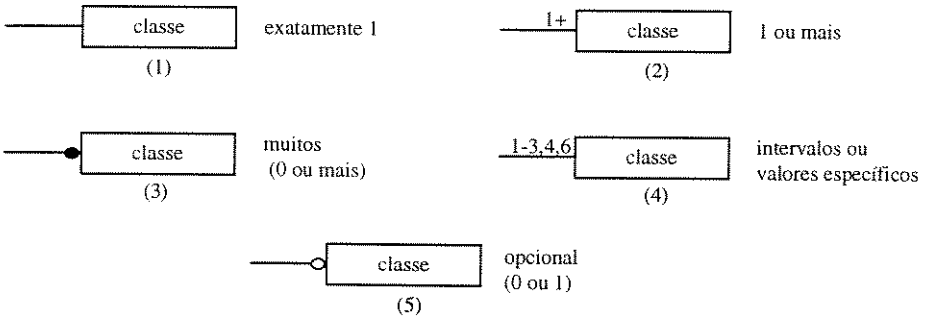
Figura 2-9: Associação com atributos e operações.

Identificar associações é importante para se conhecer o tipo de relação entre as classes. Se o objeto de uma classe é modificado, as suas associações mostram quais classes de objetos podem ser afetadas com esta alteração. As associações estabelecem uma certa dependência e cooperação entre as classes sem mostrar como isto é implementado. As associações completam a visão estática do sistema, mostrando a necessidade de vincular um objeto com outro.

Existe um outro elemento importante no Modelo de Classes que mostra quantos objetos de uma classe podem estar envolvidos em um determinado relacionamento. Este elemento é denominado **multiplicidade**.

Multiplicidade: especifica quantas instâncias de uma classe se relacionam com uma instância de uma classe associada, limitando o número de objetos relacionados.

OMT possui uma representação específica para as diferentes formas de multiplicidade. A notação gráfica de cada uma é mostrada a seguir:



(1) Significa que existirá um e somente um objeto da classe. (2) Denota um mínimo de instâncias da classe sem limitar o número máximo. Usa-se o operador "+" na notação. (3) Esta notação, utilizando um círculo preto, mostra inexistência ou existência de muitos objetos da classe. (4) Nesta notação os números especificam os intervalos e/ou quantidades específicas de instâncias da classe. (5) O círculo vazio significa que pode existir um objeto ou não pertencente à classe.

Os exemplos seguintes mostram associações binárias com diferentes multiplicidades.

O primeiro exemplo é de uma associação binária um-para-muitos. No segundo exemplo a associação é de multiplicidade um-para-um.

Exemplo 1: A Figura 2-10 mostra que em "Uma firma trabalham muitas pessoas".

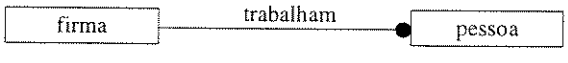


Figura 2-10: Multiplicidade: um-para-muitos.

Exemplo 2: A Figura 2-11 mostra que "Um piloto comanda um e somente um avião".

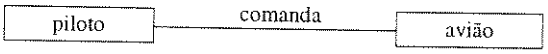
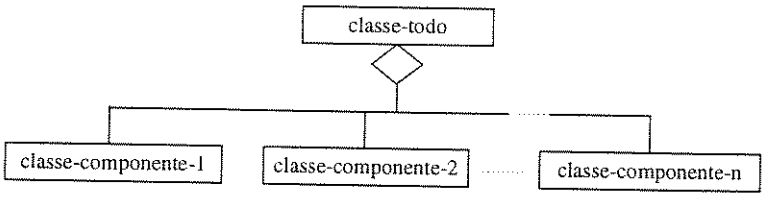


Figura 2-11: Multiplicidade: um-para- um.

Existem também duas formas especiais de associação: **Agregação** e **Generalização**.

Agregação: é uma forma de associação conhecida como "relação Todo-Parte", que relaciona um conjunto de classes com uma única classe chamada classe agregada, sendo cada classe do conjunto um componente da classe agregada.

Em OMT a associação de agregação é representada com a seguinte notação gráfica:



Agregação é importante porque possibilita a organização das classes componentes e facilita o entendimento da estrutura da classe agregada.

Nas próximas figuras são mostrados alguns exemplos que ilustram o uso da relação de agregação na modelagem de objetos, incluindo multiplicidade na relação. Na Figura 2-12 é apresentada uma parte de um Modelo de Classes para um processador de textos, onde um documento está composto de muitos parágrafos e cada um desses parágrafos consiste de muitas palavras.

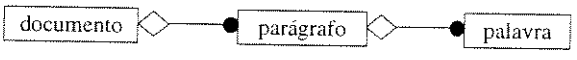


Figura 2-12: Agregação de classes.

A Figura 2-13 ilustra uma relação de agregação que mostra a classe carro e as suas partes.

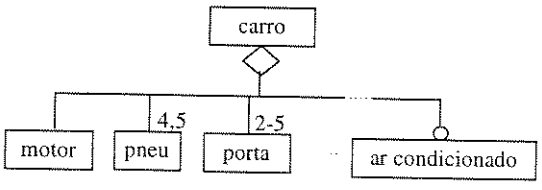


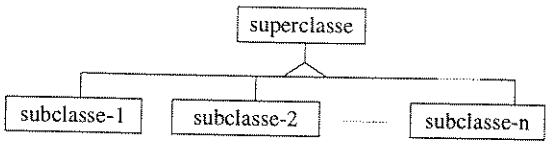
Figura 2-13: Agregação da classe carro.

Generalização: é um tipo de relacionamento entre uma classe denominada superclasse e uma ou mais versões refinadas chamadas subclasses.

Os atributos e operações comuns a um grupo de subclasses são definidos na superclasse e compartilhados por cada subclasse; esta característica é denominada **herança**.

Herança: é um mecanismo utilizado para compartilhar atributos e operações de uma superclasse com as subclasses, as quais herdam as características da sua superclasse. Cada subclasse pode adicionar atributos e operações particulares.

A notação gráfica para generalização é a seguinte:



A Figura 2-14 mostra um exemplo de generalização. Neste exemplo a superclasse "meio de transporte" é a generalização de: "transporte terrestre", "transporte fluvial" e "transporte aéreo". É possível definir subclasses para a classe "transporte terrestre", por exemplo "carro", "moto" ou "bicicleta", formando-se dessa forma uma hierarquia de classes.

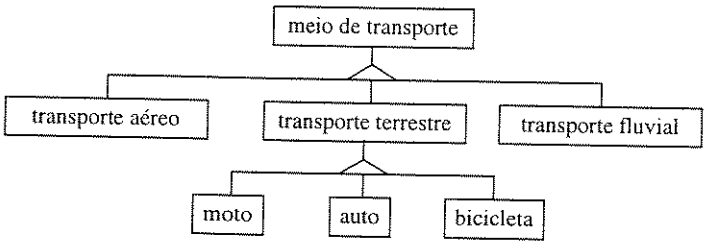


Figura 2-14: Relacionamento de generalização.

O compartilhamento das características comuns entre classes, através de generalização, é importante na etapa de implementação porque aumenta a possibilidade de reuso de código.

Os conceitos apresentados constituem os elementos básicos de OMT, mais especificamente do Modelo de Classes. Outros conceitos podem ser encontrados na referência [1].

No contexto do presente trabalho, foi construído um Modelo de Classes para representar a estrutura estática de uma aplicação na área de SDA, denominada Serviço de Transações (ST). A modelagem foi auxiliada pela ferramenta LOV/OMT [16]. O modelo resultante é apresentado no Capítulo 3.

Apenas o Modelo de Classes não é suficiente para o desenvolvimento de um sistema distribuído como o ST, os aspectos dinâmicos e funcionais também precisam ser contemplados. Para isto, foi utilizada a técnica de especificação formal chamada SDL. Mais detalhes sobre esta linguagem são apresentados na seção2.3.

2.3. SDL (Specification and Description Language)

2.3.1. Introdução

SDL é uma Técnica de Descrição Formal (FDT: Formal Description Technique) desenvolvida pela ITU-T (International Telecomunication Union[4]) com o objetivo de especificar sistemas para telecomunicações [7].

Uma FDT é uma notação que possui uma semântica definida e não ambígua que permite a verificação, simulação, validação e geração de código [3].

Geralmente uma FDT contém três partes: descrição da arquitetura do sistema (decomposição hierárquica), descrição do comportamento (em geral baseada em máquinas de estados finito) e gerenciamento dos dados.

SDL pode ser representada na forma textual (PR), através da escrita dos comandos, ou gráfica (GR), através de representações gráficas dos comandos. Um sistema pode ser descrito usando tanto o SDL-GR quanto o SDL-PR (Figura 2-15).

A Figura 2-15 ilustra a representação gráfica (GR) do bloco "exemplo" (1) e a sua representação textual (PR) equivalente, note a inclusão da palavra chave BLOCK (2). É mostrado também a declaração de variáveis (Sintaxe textual comum).

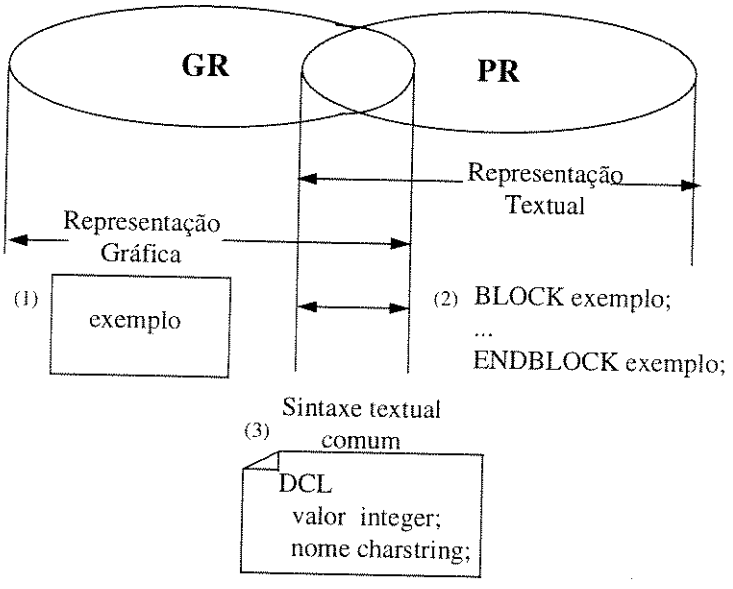


Figura 2-15: Formas de representação em SDL-GR e SDL-PR.

Normalmente as ferramentas CASE (Computer Aided Software Engineering) existentes que suportam SDL trabalham com a forma gráfica (SDL-GR) e geram código em SDL-PR.

[4]Setor de Padronização das Telecomunicações, antiga CCITT.

Desde a primeira publicação da recomendação em 1976 [7], esta linguagem vem sendo revisada a cada quatro anos. A última versão, do ano 1992, passou a incluir conceitos de orientação a objetos.

SDL é importante porque é uma técnica de descrição formal (FDT) e como tal possibilita a edição, simulação, verificação e geração de código. É importante também por ser uma linguagem para especificação que possui um certo nível de abstração que o diferencia de uma linguagem de programação convencional. Também é importante porque possui duas formas de representação, uma textual (SDL-PR) e uma gráfica (SDL-GR) que facilita a representação gráfica do comportamento através de diagramas de transição de estado.

Aspectos gerais de uma descrição baseada em SDL

Uma aplicação especificada em SDL possui uma estrutura hierárquica composta de **blocos** que se comunicam através de **canais**. O nível mais baixo de decomposição é formado por **processos**, cujo comportamento é descrito por meio de máquinas de estado finito estendido (EFSM - Extended Finite State Machine), incluindo a comunicação entre elas. Os processos são concorrentes e a comunicação é assíncrona. A informação processada ou trocada é descrita por Tipos Abstratos de Dados (ADT-Abstract Data Types), que serão descritos mais adiante (seção 2.3.5).

Uma máquina de estado finito possui entidades que processam dados ou ficam disponíveis para processá-los. Quando os dados ficam disponíveis, a máquina inicia o seu processamento e continua até que todos os possíveis processos tenham sido completados, permanecendo no mesmo ou mudando de estado [3].

O processamento executado depende do estado em que a máquina se encontra e da disponibilidade dos dados. Se o processamento depender de condições locais, a máquina é chamada máquina de estado finito estendido (EFSM).

A descrição de um sistema em SDL é composta de quatro partes (Figura 2-16):

- Estrutura do sistema (Hierarquia).
- Comportamento do sistema (EFSM).
- Comunicação.
- Definição de dados (ADT).

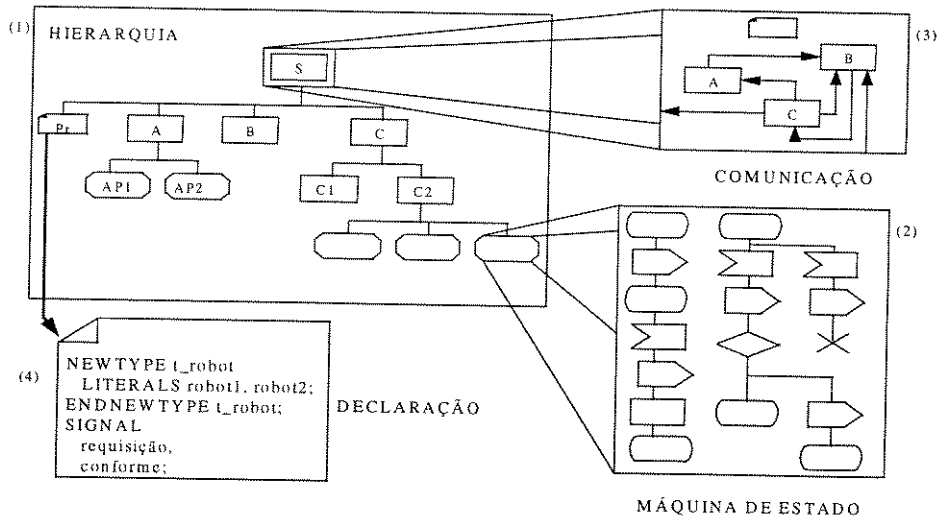


Figura 2-16: Partes de uma descrição em SDL.

2.3.2. Estrutura do Sistema

A estrutura do sistema especificado em SDL é hierárquica. É constituída pelos elementos representados na Figura 2-17: O sistema (SYSTEM) (S), que é a entidade de mais alto nível, contém uma ou mais estruturas, chamadas blocos (BLOCK) (B1, B2, B3) e/ou subestruturas (SUBSTRUCTURE). Um bloco é constituído por estruturas de tipo processo (PROCESS) (P11, P12, P21, P31)) e/ou por subestruturas (BS_1), que por sua vez podem conter um ou mais blocos (BS_11, BS_12).

Um processo é a unidade básica da descrição. Processos são concorrentes e independentes; o seu comportamento é descrito por EFSM. Um processo pode usar procedimentos (PROCEDURE) (Pr111, Pr112), ou serviços (SERVICE) (S1, S2), os quais são também representados na hierarquia. Serviços podem também fazer uso de procedimentos (Pr121).

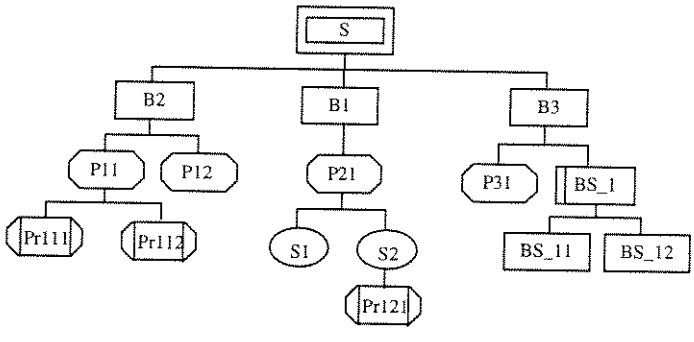


Figura 2-17: Hierarquia com Blocos, Processos, Subestruturas e Serviços.

Cada processo contém estados (STATES) e transições. Os estados são ativados pelo recebimento de sinais (INPUT) e influenciam na mudança do estado do

processo. É nas transições que se realiza o processamento dos dados através de tarefas (TASK), decisões (DECISION), envio de sinais (OUTPUT) entre outras ações executadas em forma seqüencial.

A comunicação entre processos é feita através de rotas de sinal (SIGNALROUTE) e a comunicação entre blocos é feita por meio de canais (CHANNEL). Devido a estas construções torna-se possível realizar o transporte de sinais (SIGNAL).

O diagrama de interconexão da Figura 2-18 representa o transporte de sinais; os canais são representados como C1, C2, C3, C4, e C5, enquanto que as rotas de sinal dentro do bloco B3 são representadas como r1, r2, r3, e r4. Maiores informações encontram-se na seção 2.3.4 deste capítulo e em [3].

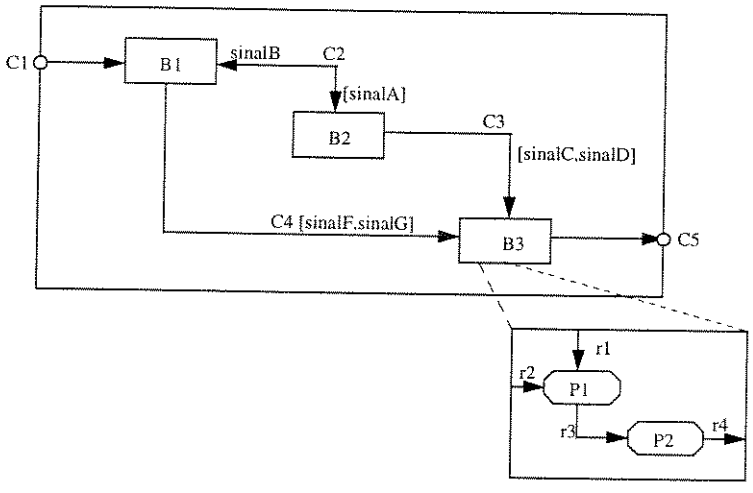


Figura 2-18: Diagrama de interconexão.

2.3.3. Comportamento do Sistema

Um processo é a construção básica para se modelar o comportamento de um sistema. Constitui o elemento comum entre a parte estática (nível de Bloco) e dinâmica (nível de Processo) da descrição de um sistema em SDL (Figura 2-19). Cada processo é representado por uma EFSM, que reage a estímulos externos (sinais). O tratamento de um sinal normalmente implica na execução de ações seqüenciais definidas em uma transição. Por exemplo na Figura 2-19 o processo P1 ao receber um sinal S1 dá início à transição T1 e à execução das ações A1, ... An.

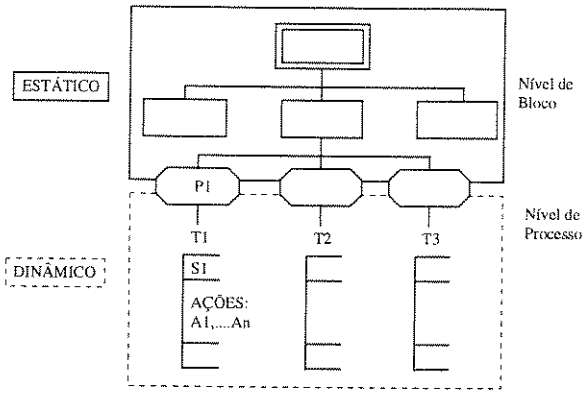


Figura 2-19: Níveis de descrição em SDL.

Os processos podem ser criados na inicialização do sistema, ou criados (e terminados) em tempo de execução (*run-time*). Pode existir mais de uma instância do processo e cada uma possui a sua própria identificação. Isto permite o envio de sinais para instâncias individuais de um determinado processo. Cada instância se comporta segundo o especificado pela sua máquina de estado.

2.3.3.1. EFSM (Máquina de Estado Finito Estendido)

O corpo de um processo é descrito utilizando uma EFSM.

Quando chega um sinal ao processo, é ativada uma transição. As transições também podem ser ativadas quando uma condição (representada pela palavra chave PROVIDED) é verdadeira. As ações dentro de cada transição são executadas em forma seqüencial e estas podem ser operações sobre dados, decisões, chamadas de procedimento, envios de sinal, entre outros.

2.3.3.1.1.Elementos Básicos de uma EFSM

A seguir são mostrados os principais elementos de uma EFSM:

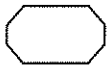
STATE:

É qualquer modo observável de comportamento de um sistema, por exemplo, estados para um sistema de controle e monitoramento podem ser: inativo, monitorando, alarme, etc.

INPUT SIGNAL:

Representa o sinal que causará uma transição. Um sinal de entrada (chamado também de evento), dispara a transição associada e possibilita utilizar a informação contida no sinal.

PROCESS:



É a construção básica para modelar o comportamento de um sistema. Qualquer comportamento é sempre referido a um

sistema e em particular a uma parte de um sistema. É nos processos que a informação é transformada.

OUTPUT:

É a ação de criar um sinal que pode conter informações e o caminho por onde será enviado.

CREATE:

Produz uma instância de um processo.

CONTINUOUS SIGNAL:

Um sinal contínuo permite iniciar uma transição dependendo da condição de uma variável visível ao contexto do processo.

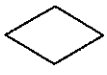
SAVE:

Usado quando se quer mostrar que alguns sinais de entrada devem ser mantidos na fila ou serão usados em um estado futuro.

TASK:

Usado para especificar operações sobre dados que são executadas em algum ponto de uma transição.

DECISION:

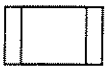


Permite selecionar uma seqüência de ações entre duas ou mais alternativas.

STOP:

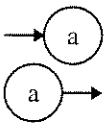
Ação que destrói uma instância de um processo.

PROCEDURE CALL:



Ação utilizada para executar um procedimento que encapsula um comportamento. Pode ser chamado repetidas vezes em um processo.

JOIN:



Ação que permite a transferência do fluxo de controle de uma parte da especificação gráfica do processo para uma outra.

O comportamento dos processos é descrito em função das mudanças dos seus estados. Uma vez inicializado, um processo fica no estado ativo, esperando um sinal (o tratamento dos sinais pode ser postergado ou estar condicionado). Quando um processo muda de estado devido a um estímulo, é ativada uma transição na qual são executadas diversas tarefas, tais como o envio de um sinal, a chamada de um procedimento, a criação de novas instâncias de processos e cálculos diversos.

Decisões podem ser tomadas durante a transição. Após a sua execução as instâncias dos processos podem ser desativadas.

A Figura 2-20 mostra um exemplo de uso das EFSMs. O processamento representado no exemplo é trivial cujo objetivo é mostrar uma máquina de estado em SDL e o uso dos elementos básicos.

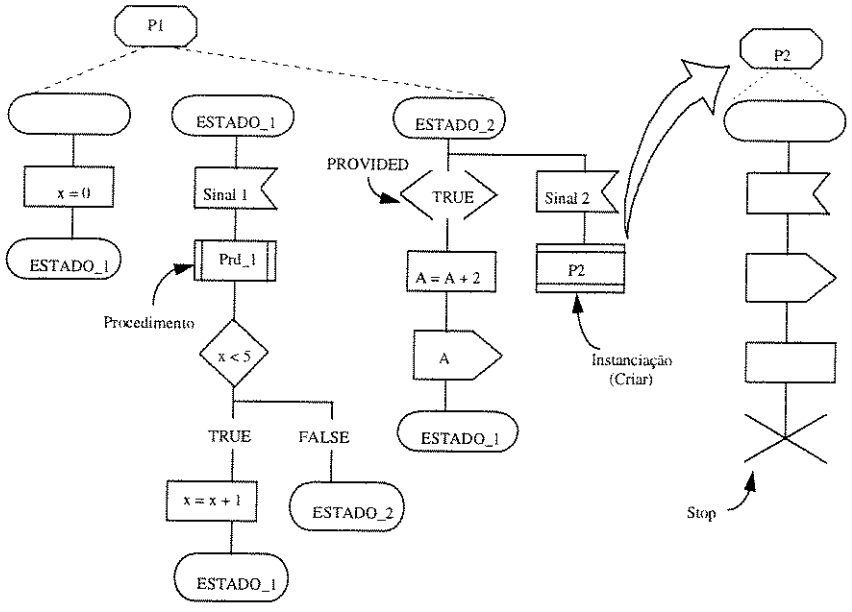


Figura 2-20: Elementos básicos de uma EFSM.

O processo P1 é inicializado atribuindo o valor 0 à variável X. A chegada do Sinal 1 no Estado_1 ativa a transição e a chamada de procedimento Prd_1 é executada. Se a condição seguinte (X<5) for verdadeira o valor da variável X será incrementado e o processo permanecerá no mesmo estado (Estado_1). Caso contrário, é produzida uma mudança para o Estado_2. Estado_2 pode ser mudado se uma determinada condição for verdadeira ou pela chegada do Sinal 2.

No caso da condição ser TRUE o valor da variável A é acrescido de 2 e enviado para um outro processo. Em seguida o processo P1 volta ao Estado_1.

Caso ocorra a chegada do Sinal 2, será executada a ação de criar uma instância do processo 2 (ação indicada pela seta maior no lado direito da Figura 2-20). O símbolo de *Stop* (X) significa que após a execução esta instância deixará de existir.

2.3.4. Comunicação

Comunicação é a troca de informação dentro dos limites de um sistema e do sistema com o seu ambiente, tendo sempre um produtor de informação e um ou mais receptores ou usuários dessa informação.

A comunicação em SDL baseia-se nas seguintes construções:

- Envio de um sinal a um receptor.
- Comunicação através de dados compartilhados.
- Comunicação através de ativação de instâncias.

Com isto é possível modelar diferentes formas de comunicação, como descrito no Anexo 1.

2.3.4.1.Envio de um sinal a um receptor (OUTPUT)

A comunicação entre processos por meio de sinais se dá dentro dos limites de um bloco ou entre processos que pertençam a diferentes blocos. O meio de comunicação fora dos limites dos blocos é o **canal** (CHANNEL) e dentro dos limites de um bloco a comunicação se dá através de **rotas de sinal** (SIGNALROUTE). Estas últimas não implicam em nenhum atraso na comunicação.

Na Figura 2-21, são ilustrados os principais elementos do envio de sinal a um receptor.

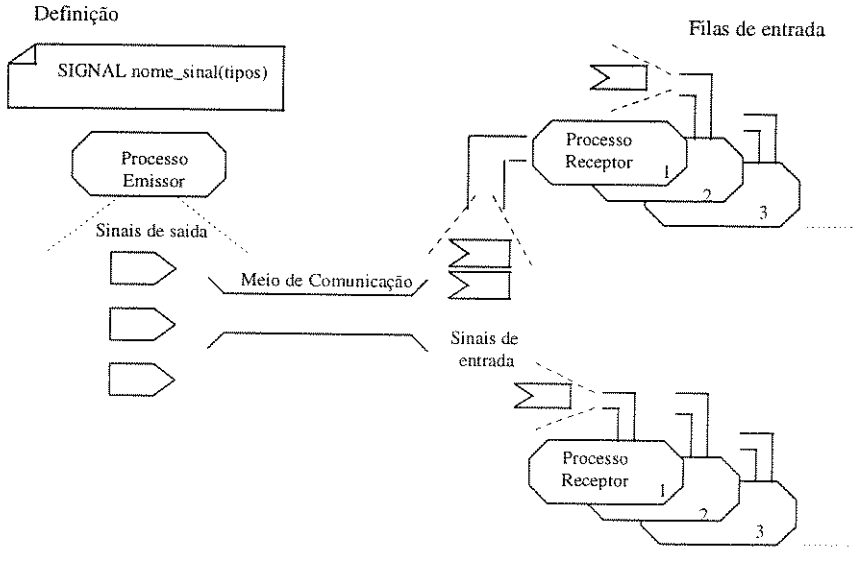


Figura 2-21: Comunicação através de troca de sinais.

Qualquer sinal deve ser estaticamente definido com um nome e os tipos das informações associadas. Quando o sinal é transmitido deve ser identificado o meio de comunicação por onde será transportado (uma rota de sinal ou um canal). Uma vez transportado, o sinal é colocado na(s) fila(s) de entrada do(s) processo(s) receptor(es). As filas são do tipo FIFO (First Input First Output).

A representação em SDL de um envio de sinal na forma textual (SDL_PR) é com o uso da palavra chave OUTPUT, como mostrado a seguir:

OUTPUT sinal_1(tipoA,tipoB,...) TO processo_receptor_id VIA canal_3

A representação na forma gráfica (SDL_GR) de um envio de sinal é mostrada na Figura 2-22; onde r1, r2, e r3 representam rotas de sinal e c3 um canal de comunicação.

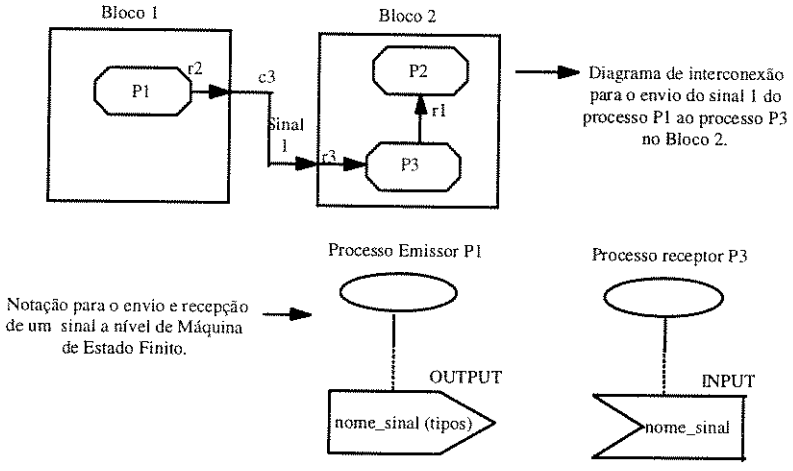


Figura 2-22: Envio e recepção de sinais em SDL.

Quando necessário, um canal pode ser representado em termos de outras construções SDL, por exemplo, blocos, processos, etc. Isto é útil para especificar atrasos máximos permitidos. Caso contrário, se o atraso é desprezível, não é necessário detalhar o comportamento do canal.

Um sinal é definido pela cláusula:

```
SIGNAL    nome_sinal (tipo1, tipo2, tipo3),
          nome_sinal (tipo2, tipo4),
          .....
          nome_sinal (tipoM, tipo1, tipoX,.....);
```

Quando um OUTPUT é interpretado, o sinal contém associado a seguinte informação: Nome do sinal, identificação do receptor, identificação do emissor e a informação associada (Figura 2-23).

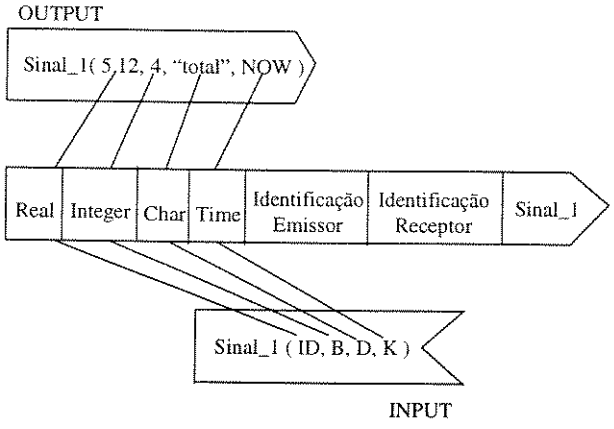


Figura 2-23: Conteúdo de um sinal no instante do Envio (OUTPUT).

2.3.4.2.Comunicação através de dados compartilhados

Os dados sempre pertencem a uma instância de um processo. Dados podem ser compartilhados entre diferentes instâncias de processos. Para isto, SDL dispõe de dois mecanismos:

- Mecanismo REVEALED-VIEW
- Mecanismo EXPORT-IMPORT

2.3.4.2.1.Mecanismo REVEALED-VIEW

Este mecanismo opera dentro dos limites de um bloco e não consome tempo. O processo que possui a informação declara uma variável compartilhada da seguinte forma:

DCL REVEALED var_nome_compart Tipo_var;

O processo que faz acesso à variável compartilhada declara uma variável que conterá o valor associado usando a palavra chave VIEWED, seguida pelo nome da variável e o seu tipo:

VIEWED var_nome_acesso Tipo_var;

O acesso ao valor da variável compartilhada (var_nome_compart) por outros processos é realizado durante uma transição usando a palavra chave VIEW:

VIEW (var_nome_compart , Processo_dono);

Exemplo de aplicação:

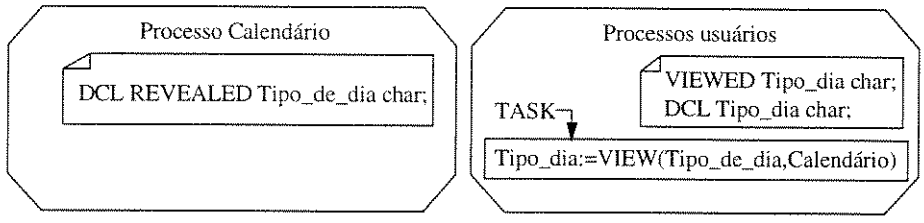


Figura 2-24: Mecanismo REVEALED-VIEW.

Na Figura 2-24, o processo calendário declara a variável "Tipo_de_dia" como revelada. Os processos usuários declaram uma variável receptora ("Tipo_dia"), podendo ter acesso à variável compartilhada na execução de uma tarefa usando VIEW.

2.3.4.2.2.Mecanismo EXPORT-IMPORT

Este mecanismo é utilizado dentro e fora dos limites de um bloco.

O processo dono da informação decide que variável será exportada, executando a ação de EXPORT dentro de uma transição. Esta ação produz uma cópia do valor da variável que permanece disponível para os processos importadores até a próxima exportação onde o valor da cópia é atualizado.

Exemplo de aplicação:

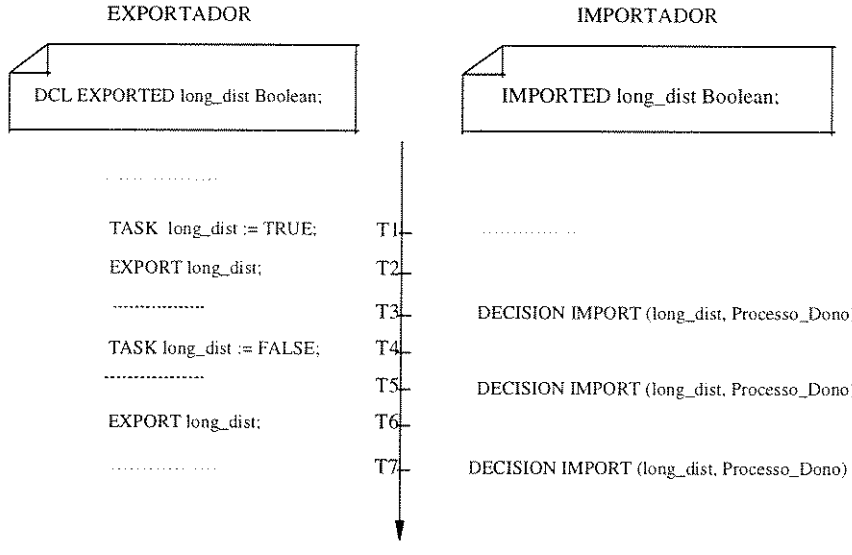


Figura 2-25: Mecanismo EXPORT/IMPORT.

No exemplo da Figura 2-25 pode se observar a exportação e importação da variável "long_dist" no decorrer do tempo:

⇒ T1 : declaração e inicialização com TRUE da variável a ser exportada e importada.

⇒ T2 : o valor da variável é exportado.

⇒ T3 : o processo usuário importa o valor da variável long_dist, cujo valor é TRUE, para tomar uma decisão.

⇒ T4 : o processo dono muda o valor de long_dist para FALSE (esta mudança não afeta o processo importador, que ainda vê o valor como TRUE).

⇒ T5 : o processo usuário importa o valor da variável long_dist (cujo valor ainda é TRUE, não o seu valor atual).

⇒ T6 : o novo valor da variável long_dist é exportado.

⇒ T7 : o processo usuário importa o valor exportado (agora com o valor FALSE).

O(s) importador(es) não pode(m) ter acesso ao valor atual da variável, a menos que este seja exportado pelo processo proprietário da informação.

2.3.4.3.Comunicação através de ativação de instâncias

Uma instância de um processo pode criar outras instâncias do mesmo processo, ou de processos diferentes dentro dos limites de um bloco.

A comunicação, utilizando este mecanismo, causa uma troca de informação entre a instância criadora e a instância criada.

Exemplo:

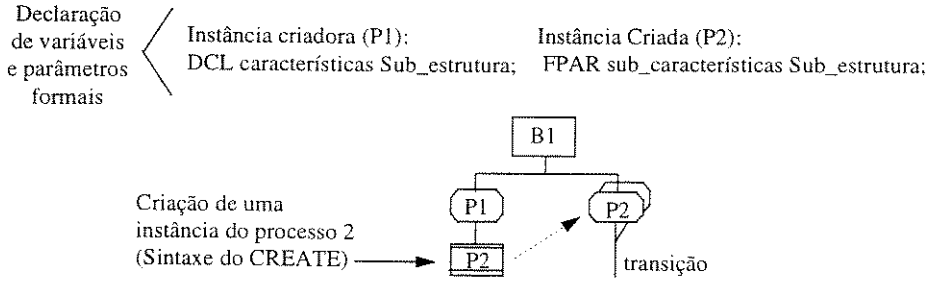


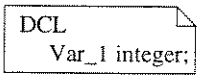
Figura 2-26: Comunicação através de ativação de instâncias.

Quando a sentença CREATE (Figura 2-26) é interpretada, uma nova instância do processo P2 é criada e a informação associada à variável "características" na instância criadora fica imediatamente disponível à instância criada na variável "sub_características", declarada como parâmetro formal (FPAR). Tanto a instância criada quanto a criadora trocam informações do mesmo tipo (tipo "Sub_estrutura").

2.3.5. Definição de Dados

Dados são entidades que contêm informações. Dados em SDL precisam ser definidos dentro da especificação do sistema através da declaração de variáveis (DCL).

A definição de uma variável especifica basicamente a sua identificação, por exemplo Var_1, e o tipo de variável, segundo o seguinte formato:



Uma variável pode ser de um tipo predefinido (*natural*, *integer*, *real*, *boolean*, *character*, *duration*, *time*, *pid*, *array*, *struct*, *powerset*, *charstring*) ou de um tipo definido pelo usuário podendo receber valores daquele tipo.

2.3.5.1. Tipos Abstratos de Dados

Todos os tipos de dados são definidos por um tipo abstrato de dado (ADT). Um ADT define um conjunto particular de valores, as possíveis operações sobre esses dados e os valores resultantes destas operações (Figura 2-27).

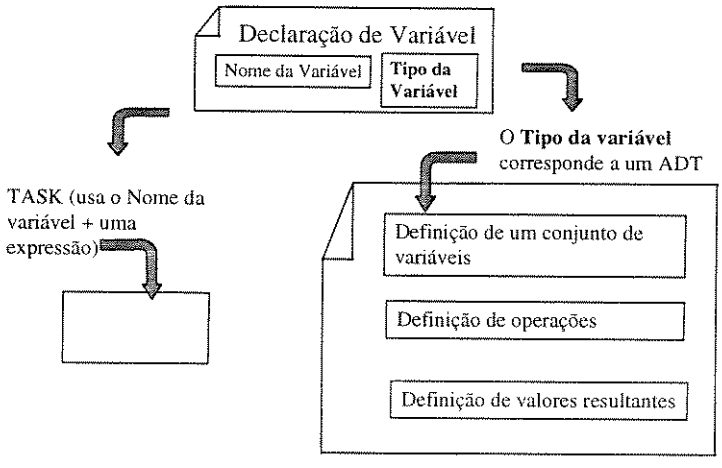


Figura 2-27: Tipos Abstratos de Dados (ADT).

A Figura 2-28 mostra um exemplo de declaração (NEWTYPE) e uso do Tipo "COR". São definidos os seus valores, como LITERALS, e a operação "misturar", que será aplicada sobre duas cores dando como resultado uma outra cor.

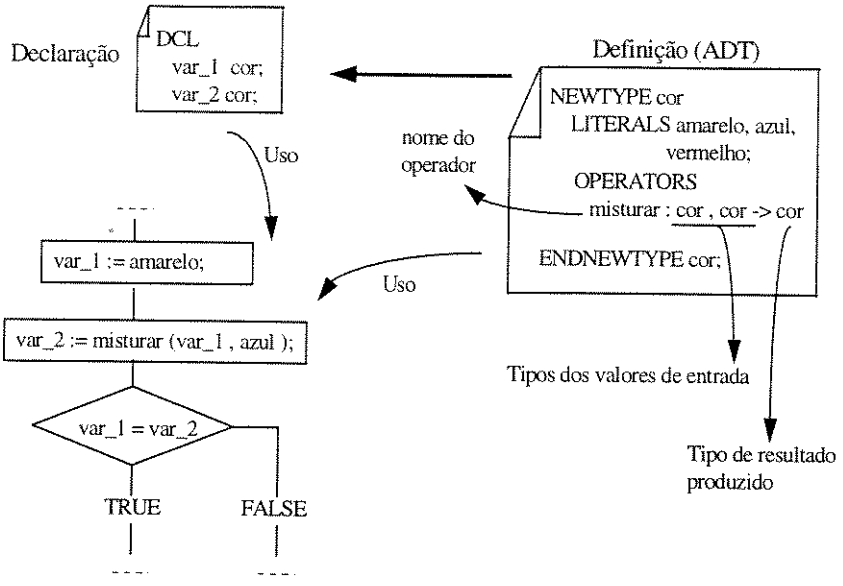


Figura 2-28: Declaração e uso de um ADT.

Os tipos abstratos de dados em SDL podem ser divididos em três categorias: Tipos simples, Tipos estruturados e Tipos geradores.

As três categorias mencionadas podem ser classificadas como [17]: tipos predefinidos e tipos definidos pelo usuário.

A Tabela 2-I mostra os tipos predefinidos disponíveis:

Tipos predefinidos	Nomes dos tipos
Tipos simples	Boolean, Character, Integer, Natural, Real, Duration, Time, Pid
Tipos estruturados	Charstring
Tipos geradores	String, Powerset, Array

Tabela 2-I: Tipos predefinidos em SDL.

Os tipos definidos pelo usuário são construídos em função dos predefinidos, como mostra a seguinte definição:

```
NEWTYPE t_descrição
  Struct
      nome charstring;
      número natural;
      modo charstring;
ENDNEWTYPE t_descrição;
.....
DCL minha_desc t_descrição;
```

Na definição acima, o tipo "t_descrição" consiste de 3 campos. A variável "minha_desc" é do tipo "t_descrição".

Dentro das declarações (DCL) são consideradas, além das variáveis e tipos, a declaração dos sinais (SIGNAL) que circulam no sistema, a declaração de constantes (CONSTANT), e parâmetros associados aos processos e/ou procedimentos (FPAR), como ilustrado na Figura 2-29.

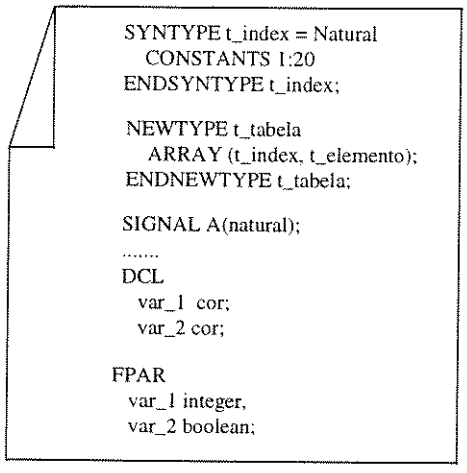


Figura 2-29: Declarações em SDL (constantes, tipos, sinais, variáveis, parâmetros formais).

Na Figura 2-29, t_index é um tipo que identifica números naturais entre 1 e 20, isto significa que uma variável do tipo t_index poderá assumir unicamente valores entre 1 e 20. SYNTYPE é utilizada para a declaração de valores constantes. t_tabela é um tipo de *array* definido pelo usuário e possui duas componentes, a primeira do tipo t_index que significa que o *array* terá 20 posições no máximo para serem preenchidas, a segunda componente do tipo t_elemento (a definição deste tipo definido pelo usuário não é mostrado na figura) que representa o tipo de valores que conterá o *array*.

2.3.6. O Tempo em SDL

Em SDL o tempo é modelado sobre o princípio de um *clock*, que é usado para especificar restrições temporais em sistemas de tempo real.

Em uma especificação SDL o tempo é consumido na transferência de um sinal, desde a sua criação (OUTPUT) até o seu consumo no destino final (INPUT), e durante a interpretação de uma transição.

A representação de tempo é realizada através de temporizadores (*Timer*), utilizando a expressão NOW, que provê o valor de tempo absoluto, usando variáveis do tipo *time* (para tempos absolutos) e do tipo *duration* (para tempos relativos).

Os temporizadores são controlados por duas construções:

- SET: ativa o *timer* por um intervalo de tempo especificado. Após este tempo, um sinal de *time out* é gerado e colocado na fila de entrada do processo;

- RESET: desativa o *timer* e tira o sinal de *time out* da fila de entrada caso exista.

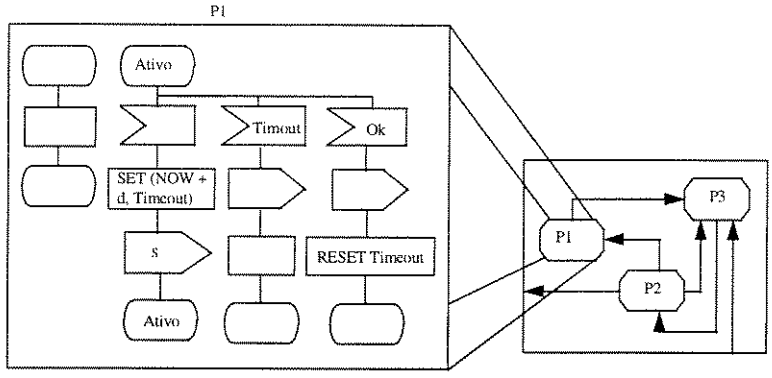


Figura 2-30: Controle de tempo usando time out.

Na Figura 2-30 é mostrado o uso de temporizadores para controlar o envio de um sinal. O processo emissor (P1) "seta" o relógio por um tempo "d", envia o sinal "S" e fica no estado "Ativo", à espera da confirmação de recebimento do sinal por parte do destinatário. A recepção de um sinal é confirmada com o envio de um sinal "OK" para o processo emissor. Se este sinal chegar antes do tempo previsto, o relógio é "resetado" (RESET). Caso contrário, após o tempo "d" o processo recebe um sinal de *time out*.

Como anteriormente mencionado, o tempo é consumido durante a interpretação de uma transição. Na Figura 2-31 é mostrada a forma como este tempo pode ser controlado.

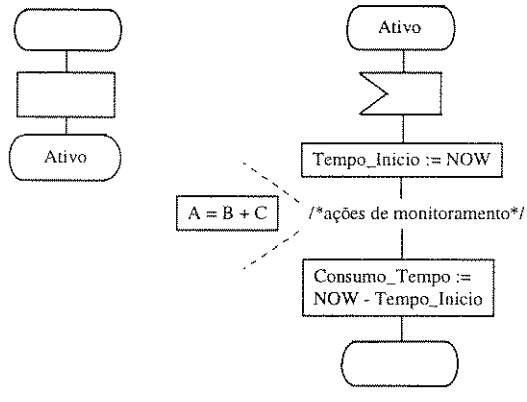


Figura 2-31: Controle do tempo consumido em uma transição.

Na Figura 2-31, o processo inicializa a variável "Tempo_Inicio" com o valor atual do relógio, através do operador NOW. Em seguida, podem ser executadas ações a serem controladas (A:=B+C, etc.). O tempo consumido pela execução destas ações, é armazenado na variável "Consumo_Tempo".

2.4. CORBA

2.4.1. Introdução

CORBA (Common Object Request Broker Architecture) [9] é um padrão da OMG (Object Management Group) , um consórcio internacional dedicado ao estabelecimento de padrões para ambientes orientados a objetos, que provê a comunicação entre objetos distribuídos em ambientes heterogêneos.

No padrão CORBA um objeto oferece um ou mais serviços que podem ser solicitados por clientes através da interface do objeto. Esta interface é especificada em IDL (Interface Definition Language) e contém a descrição das operações que o objeto fornece e a forma como as operações devem ser invocadas.

A comunicação entre cliente e servidor é realizada com o auxilio do ORB (Object Request Broker), que gerencia a troca de requisições e resultados inerentes à comunicação.

Nas seções seguintes são apresentados os principais elementos da arquitetura e o processo de geração de aplicações utilizando este padrão.

Para obter mais informações sobre CORBA ver as referências [9] e [12].

2.4.2. Elementos da arquitetura

Um pedido de serviço é realizado por um **cliente** através do **ORB** para um **objeto servidor** que implementa o serviço solicitado como mostrado naFigura 2-32.

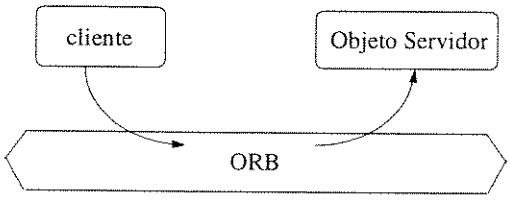


Figura 2-32:Comunicação através do ORB.

As definições dos elementos apresentados naFigura 2-32 é mostrada a seguir:

Cliente: é um programa, um processo ou um outro objeto servidor que solicita um serviço ou método de um objeto servidor.

O cliente tem acesso à referência do objeto através da qual identifica o objeto e invoca operações sobre ele. O conhecimento que o cliente tem do objeto servidor se limita à sua interface e é através dela que decide qual serviço solicitar.

Objeto servidor: são objetos que oferecem algum serviço, e possuem uma interface escrita em IDL (Interface Definition Language). Esta interface descreve os métodos que podem ser requisitados pelos clientes.

ORB: é o responsável por interceptar a chamada do cliente, encontrar o objeto servidor que implementa o método solicitado, enviar os parâmetros necessários, invocar o método do objeto e retornar o resultado para o cliente. O ORB mascara a localização do objeto (que pode ser local ou remoto), a linguagem na qual o objeto foi implementado, o mecanismo de acesso, etc. Esta característica é chamada de transparência.

Alguns elementos próprios do ORB são utilizados durante este processo de troca de requisições e respostas. A Figura 2-33 mostra estes elementos.

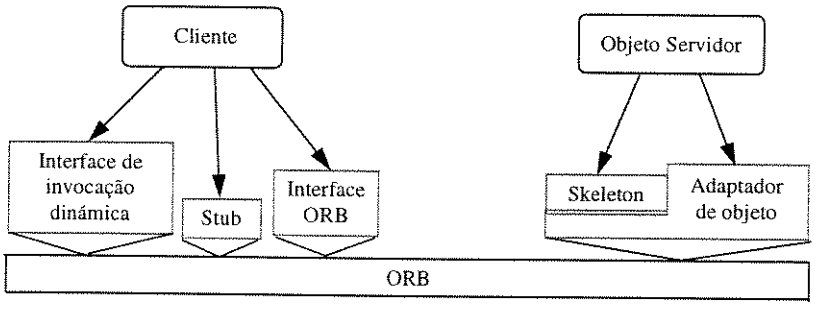


Figura 2-33: Principais elementos de um ORB.

Para fazer um pedido aos objetos servidores, o cliente pode utilizar duas formas de invocação: dinâmica (quando o cliente não conhece qual objeto pode realizar esse serviço, deixando para o ORB a procura de algum objeto que satisfaça aquele pedido) ou estática (quando o cliente sabe qual é o objeto que fornece o serviço). No primeiro caso o cliente faz uso de uma interface do ORB denominada Interface de Invocação Dinâmica (Figura 2-33), que faz acesso a um Repositório de Interfaces de objetos a procura de uma que satisfaça a requisição do cliente. No segundo caso a requisição do método de um determinado objeto é passada para o ***stub***. Um cliente pode interagir diretamente com o ORB através da Interface ORB para algumas funções.

Em ambos casos, o ORB localiza o código que implementa o serviço requisitado, transmite os parâmetros necessários transferindo o controle para a implementação do objeto através de um ***skeleton***. Quando o serviço for executado, é retornada a resposta para o cliente.

Uma implementação de objeto pode interagir diretamente com o ORB através do Adaptador de Objeto. As informações sobre as implementações de objetos são armazenadas em um Repositório de Implementação.

Stub: é a estrutura da interface do objeto servidor disponível para o cliente. É utilizado em invocações estáticas e a sua função é montar a requisição de um método. Quando o cliente executa uma chamada de um método de um objeto, é o *stub* quem se encarrega de montar a mensagem, que é passada para o ORB.

Skeleton: é o elemento do ORB encarregado de desmontar uma requisição recebida. Após isto, invoca o método do objeto passando os parâmetros necessários.

O processo que permite implementar os métodos e fazer as interfaces disponíveis para os clientes é apresentada na seção 2.4.3.

2.4.3. Processo de Geração de uma aplicação

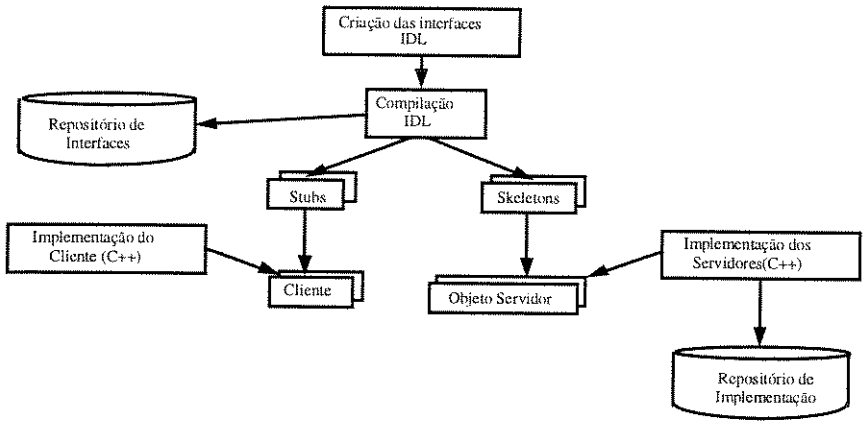


Figura 2-34: Processo de desenvolvimento em CORBA.

O primeiro passo no desenvolvimento em CORBA, como mostra a Figura 2-34, é a construção das interfaces dos objetos que fazem parte da aplicação a ser desenvolvida. Isto é importante para informar os clientes sobre os serviços oferecidos pelos objetos servidores e como devem ser invocados. Isto é feito com o auxilio de IDL.

A partir das interfaces em IDL é possível mapear os objetos CORBA para uma linguagem de programação específica como C++; portanto, as interfaces passam por um processo de compilação gerando *stubs* para os clientes e *skeletons* para os objetos servidores. As interfaces são armazenadas no Repositório de Interfaces.

Com o processo de compilação obtém-se também a estrutura básica dos objetos segundo a linguagem de programação utilizada, a partir da qual é realizada a implementação das funcionalidades de cada método dos objetos definidos. É necessário implementar também o código do cliente na mesma linguagem de programação. Após a implementação, é necessário registrar os servidores no Repositório de Implementação permitindo desta forma que o ORB localize e ative implementações de objetos.

No presente trabalho foi utilizada a ferramenta Orbix, uma das implementações baseada na especificação CORBA para a construção de objetos distribuídos. A implementação feita em Orbix faz parte do ambiente externo que interage com a aplicação desenvolvida em SDL. Os detalhes da implementação podem ser encontrados no Capítulo 3.

3. Implementação

Este capítulo descreve o processo de desenvolvimento do Serviço de Transações (ST) proposto como estudo de caso neste trabalho. Este processo parte da construção de modelos que descrevem as características estáticas, dinâmicas e funcionais do sistema, tendo como base um documento que especifica os requisitos do sistema. A partir desses modelos é realizada a simulação do sistema e finalmente a geração de código.

3.1. Introdução

Tradicionalmente, o processo de desenvolvimento de *software* deve começar com a especificação de requisitos do sistema [2]. Para este trabalho específico isto não foi necessário devido à existência de um documento de especificação do ST, definido pela OMG (Object Management Group) [10]. Neste documento são apresentadas as interfaces que o sistema deve possuir e as funcionalidades que o serviço deve fornecer.

O modelo estático foi construído usando o método OMT. Os modelos dinâmico e funcional foram construídos usando a técnica de descrição formal SDL. Foi realizada também a simulação do comportamento do sistema favorecendo a verificação, validação e testes do sistema, bem como a codificação do sistema com base nos modelos. Os resultados da modelagem são apresentados neste capítulo e no Capítulo 4.

Para se ter uma idéia do que é um ST, são apresentados na Seção 3.2 conceitos básicos como transação e gerenciamento de transações. Na Seção 3.3 são apresentados os modelos do ST, segundo o método OMT e a técnica SDL. A descrição do processo de simulação e verificação do modelo construído em SDL bem como os resultados obtidos são apresentados na Seção 3.4. A descrição do ambiente de simulação também pode ser encontrado na Seção 3.4. Finalmente, na Seção 3.5 é abordado o processo de geração de código.

3.2. Descrição do problema

Esta seção visa descrever o que é o ST. Antes porém, são descritos os conceitos básicos de transação, gerenciamento de transações e o protocolo *commit* em duas fases.

3.2.1. Transação e gerenciamento de transações

O gerenciamento de transações aborda os diferentes mecanismos para a prevenção de falhas e recuperação das informações após a sua ocorrência. As falhas podem ser causadas por erros na programação ou na operação do sistema, problemas com algum dispositivo, queda de energia, problemas na comunicação, no disco, etc., fazendo com que um programa termine de forma anormal. A recuperação após a ocorrência de uma falha visa recuperar as informações que foram alteradas dentro de uma transação, usando para isso mecanismos de recuperação.

Se as operações sobre os dados do escopo de uma transação forem executadas com sucesso, então a transação termina, tornando as alterações realizadas sobre os dados permanentes. Caso contrário, se ocorrer uma falha durante a execução, toda manipulação sobre os dados deve ser desfeita até a volta a um estado anterior consistente. Esta propriedade das transações é denominada **atomicidade**, que garante que todas as operações da transação serão executadas (passando o sistema a um próximo estado consistente) ou nenhuma delas será concretizada (voltando a um estado anterior consistente).

Exemplos de transações: Depósito/Retirada de dinheiro em uma conta bancária, reajuste no salário dos funcionários de uma empresa, reserva de passagens.

Em geral, toda transação começa com uma operação que marca o início da transação e termina com uma operação chamada ***commit*** (transação completada com sucesso) ou ***rollback*** (o trabalho realizado é desfeito).

Transação

> Uma transação é um conjunto de operações de uma aplicação específica executadas seqüencialmente, constituindo uma unidade de trabalho.

As operações que podem ser desfeitas dentro do escopo de uma transação são chamadas de operações recuperáveis e envolvem um ou mais recursos.

Quando em uma transação houver interação com mais de um recurso, é necessário utilizar um protocolo para o seu gerenciamento e garantir atomicidade da transação. Esta necessidade é maior quando se trata de sistemas distribuídos [6], nos quais os recursos estão localizados em diferentes pontos da rede, são autônomos e independentes. A falha em um nó em que a transação estiver interagindo pode afetar o resultado final.

3.2.2. Protocolo *commit* em duas fases (two-phase commit)

É um protocolo utilizado para o gerenciamento de transações em ambientes distribuídos.

Este protocolo é usado em transações que envolvem vários recursos, denominados participantes. Existe também o papel do gerenciador do encerramento da transação, chamado coordenador da transação. Em síntese, o protocolo funciona da seguinte maneira: o processo que inicia a transação envia uma requisição de *commit* para o coordenador da transação. Na recepção dessa requisição o coordenador da transação executa as duas fases seguintes:

- Primeira Fase: o coordenador da transação envia um sinal de *prepare* para todos os participantes da transação e espera as respostas de todos eles. Os participantes podem responder a este sinal com um sinal de OK, que indica que o recurso está preparado para fechar a transação ou com um sinal de NOT_OK, se ocorreu alguma falha. Um *time out* na espera equivale a uma resposta NOT_OK. Quando um participante recebe um sinal de *prepare*, significa que ele deve registrar todas as informações necessárias para desfazer as mudanças realizadas dentro da transação; se o participante tiver sucesso nesta ação, responde OK. Os registros servirão em caso de ocorrência de alguma falha em outro participante da transação.
- Segunda Fase: se todos os participantes responderam OK, o coordenador da transação envia um sinal de *commit* para cada um deles indicando que podem tornar permanentes todas as atualizações realizadas durante a transação. Caso contrário, o coordenador da transação envia um sinal de *rollback* para cada participante indicando que aconteceu uma falha e devem desfazer todas as alterações realizadas no escopo da transação.

Existem diversas propostas de implementação deste protocolo para ambientes distribuídos. Neste trabalho foi escolhida a proposta feita pela OMG, denominada Serviço de Transações (ST). Basicamente, o ST gerencia a execução do protocolo *commit* em duas fases para garantir atomicidade na execução da transação.

3.2.3. O Serviço de Transações

O Serviço de Transações (ST) abordado neste trabalho é parte das especificações da OMG [10]. A escolha foi feita porque ela adota a abordagem orientada a objetos e porque apresenta as interfaces necessárias para o suporte do serviço, facilitando a sua descrição e posterior implementação em ambiente Orbix. No documento da OMG encontra-se as definições das interfaces de múltiplos objetos distribuídos, escritas em IDL (Interface Definition Language), que irão cooperar para fornecer um ST para aplicações distribuídas que interagem com múltiplos objetos locais e remotos.

Um ST, segundo a especificação da OMG, deve controlar o escopo e a duração de uma transação e coordenar a sua finalização, considerando a existência de um ou

mais objetos distribuídos envolvidos em uma transação que é única e atômica, permitindo mudanças nos dados dos objetos envolvidos.

Basicamente, o ST está composto pelos elementos mostrados na Tabela 3-I a seguir:

Elementos	Função
Iniciador da transação (cliente) e servidor recuperável (fábrica de recursos).	Constituem o ambiente externo que interage com o ST.
Factory (fábrica) e *control* (controle).	Permitem o acesso aos objetos gerenciadores de uma transação.
Terminator (terminador), *coordinator* (coordenador), e *resource* (recurso).	Interagem para fornecer as funcionalidades relativas ao protocolo *commit* em duas fases.
Recoverycoordinator (coordenador de recuperação).	Gerencia o processo de recuperação quando necessário.

Tabela 3-I: Elementos básicos do ST.

O serviço é fornecido ao cliente através das seguintes interfaces:

- Interface *factory* (ou fábrica): inicia uma nova transação a ser gerenciada a pedido de um cliente.
- Interface *control* (ou controle): permite o acesso aos objetos *terminator* e *coordinator*, que fornecem as funcionalidades básicas do ST (registrar recursos e gerenciar a finalização da transação).
- Interface *terminator* (ou terminador): gerencia o término da transação, que pode ser bem sucedido ou não. Para isto, o terminador usa o protocolo *commit* em duas fases garantindo através dele a atomicidade da transação, fundamental em ambientes distribuídos [10]. Para executar este protocolo, o terminador precisa interagir com cada recurso registrado como participante da transação.
- Interface *coordinator* (ou coordenador): registra os objetos recursos como participantes da transação e fornece informações sobre a transação.
- Interface *resource* (ou recurso): implementa as funcionalidades do protocolo *commit* em duas fases, as quais são executadas através da interação com o terminador. Para ser considerado participante, todo objeto recurso precisa ser registrado pelo coordenador da transação.
- Interface *recoverycoordinator* (ou coordenador de recuperação): gerencia o processo de recuperação quando necessário. Existe um coordenador de recuperação para cada recurso participante.

A partir da especificação das interfaces foram construídos modelos que descrevem a estrutura da aplicação, o comportamento dinâmico envolvido e as funcionalidades que o ST oferece. Os modelos são apresentados na Seção 3.3.

3.3. Modelos do ST

A modelagem é um processo que permite descrever de forma abstrata o sistema a ser desenvolvido. Através dos modelos obtém-se um melhor conhecimento do problema, incrementa-se o controle sobre o mesmo e garante-se grande parte da documentação da aplicação. Favorece também a detecção de erros, antes que se inicie a implementação com uma linguagem de programação específica. O ST foi descrito usando o método OMT para a descrição estática do sistema e SDL para a descrição dinâmica e funcional do sistema. Estes modelos são descritos nas seções 3.3.1 e 3.3.2, respectivamente.

3.3.1. Modelo de classes

O ST mostrado no modelo de classes, está baseado na especificação do ST da OMG [10]. Neste documento são apresentadas as interfaces básicas de um ST através de especificações IDL. Este modelo representa a estrutura da aplicação. Foi construído utilizando o diagrama de classes do método OMT, e auxiliado pela ferramenta LOV/OMT [16].

O ponto de partida da modelagem é a identificação de classes. Cada classe possui características próprias e únicas e é responsável por realizar alguma atividade específica.

As classes identificadas são mostradas no dicionário de dados da Figura 3-1:

Classe factory (ou fábrica). Inicia uma nova transação a ser gerenciada. Para isto *factory* deve criar um objeto do tipo *control*.

Classe control (ou controle). Permite o acesso às duas classes que fornecem as funcionalidades básicas do ST: registrar recursos e gerenciar a finalização da transação. Estas classes são denominadas *coordinator* e *terminator*.
Quando uma transação é iniciada pelo cliente, é criado um objeto *control* único que permite o acesso ao *coordinator* e ao *terminator*.

Classe terminator (ou terminador). Gerencia o término da transação, considerando que ela pode terminar com sucesso ou não. Para isto, o terminador usa o protocolo chamado *commit* em duas fases para garantir atomicidade da transação, fundamental em ambientes distribuídos [10]. Para executar este protocolo é preciso interagir com cada objeto *resource* registrado como participante da transação.

Classe coordinator (ou coordenador). Registra os objetos *resource* como participantes da transação e fornece informações sobre a transação.

Classe resource (ou recurso). Participa da transação iniciada pelo cliente. Implementa as funcionalidades do protocolo *commit* em duas fases, as quais só serão executadas através da interação com o terminador. Todo objeto recurso, para ser considerado participante, precisa ser registrado pelo coordenador da transação. A transação pode ter um ou mais recursos participantes.

Classe recoverycoordinator (ou coordenador de recuperação). Gerencia o processo de recuperação quando for necessário; existindo um para cada recurso participante.

Figura 3-1: Dicionário de dados das classes do ST.

A identificação das classes a partir do documento da OMG foi um trabalho relativamente simples. O estudo deste documento mostrou que cada interface representa uma classe; mas, entender exatamente o que cada classe faz, quais as suas responsabilidades e como elas se relacionam não foi uma tarefa simples, pois o documento apresenta uma descrição geral (e em alguns casos confusa) de cada interface, demandando uma cuidadosa interpretação.

Visando melhorar o modelo, foram acrescentadas duas classes. A classe *transaction* mantém as informações relacionadas com a transação iniciada e a classe *factoryrec* (fábrica de recursos) mantém o controle dos recursos criados. O dicionário de dados que as descreve é apresentado na Figura 3-2.

Classe transaction (ou transação). Mantém as informações sobre a transação. O acesso aos atributos desta classe é realizado através do coordenador da transação.
Classe factoryrec (ou fábrica de recursos). Cria objetos da classe *resource* para cada transação e solicita ao coordenador registrá-los como participantes da transação.

Figura 3-2: Dicionário de dados das classes adicionais ao ST.

O conjunto final de classes com a notação OMT é mostrado na Figura 3-3 a seguir:

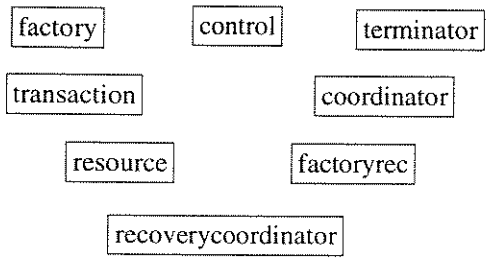


Figura 3-3: Conjunto de classes com a notação OMT.

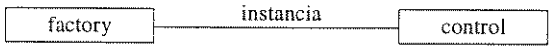
As classes não são entidades isoladas, elas interagem visando colaborar para fornecer um serviço mais amplo. Por isso, é importante identificar que tipo de relação existe entre as classes. A identificação desses relacionamentos é o assunto a seguir.

3.3.1.1.Relacionamentos entre as classes

A relação existente entre as classes é representada por meio de **associações**. As associações definem quais as classes podem ser afetadas por uma determinada classe e vice-versa. Para o ST as associações identificadas são descritas a seguir. Ao final, a Figura 3-4 dá uma visão global de todos os relacionamentos.

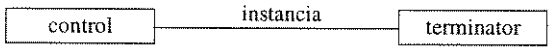
Relacionamento *factory - control*:

Uma fábrica (classe *factory*) instancia um e somente um objeto controle (classe *control*). Sempre que for iniciada uma transação, será necessário que o cliente peça para a fábrica criar um objeto controle. Isto significa que existe um relacionamento entre essas duas classes: *factory* instancia *control*.



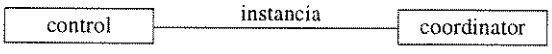
Relacionamento *control - terminator*:

Um objeto controle (classe *control*) permite o cliente ter acesso ao terminador (classe *terminator*) sendo responsável pela sua criação. A associação que mostra este relacionamento é a seguinte: *control* instancia *terminator*.



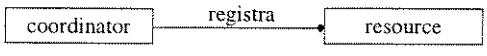
Relacionamento *control - coordinator*:

Um objeto controle (classe *control*) é responsável por instanciar o coordenador (classe *coordinator*), único para a transação, permitindo, dessa forma, que a fábrica de recursos tenha acesso àquele coordenador da transação. Portanto, foi estabelecida a seguinte associação: *control* instancia *coordinator*.



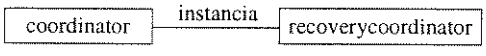
Relacionamento *coordinator - resource*:

Coordinator registra zero ou mais recursos (classe *resource*). Esta associação aparece pela necessidade de identificar os recursos participantes da transação.



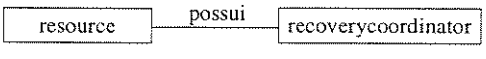
Relacionamento *coordinator - recoverycoordinator*:

Quando um recurso é registrado, o coordenador cria um coordenador de recuperação (classe *recoverycoordinator*). A associação: *coordinator* instancia *recoverycoordinator*, expressa esta característica.



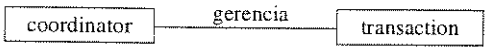
Relacionamento *resource - recoverycoordinator*:

Existe um objeto *recoverycoordinator* para cada recurso registrado. A associação: *resource* possui *recoverycoordinator*, expressa esta situação.



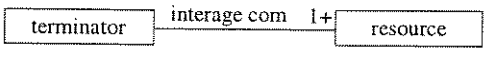
Relacionamento *coordinator - transaction*:

O coordenador é responsável por gerenciar todas as informações vinculadas à transação. Por isso, existe a associação: *coordinator* gerencia *transaction*.



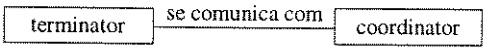
Relacionamento *terminator - resource*:

Um objeto da classe *terminator* interage com um ou mais *resources*. O objeto terminador gerencia o término da transação, através do protocolo *commit* em duas fases. Para alcançar este objetivo, ele precisa se comunicar com os recursos participantes da transação.



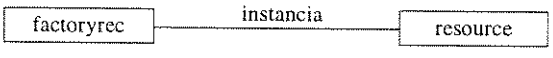
Relacionamento *terminator - coordinator*:

Fazendo uma análise do comportamento entre o objeto terminador e o objeto coordenador pode-se observar que ambos precisam se comunicar para compartilhar informações relacionadas ao protocolo e à transação. Por este motivo, a associação *terminator* se comunica com *coordinator* foi estabelecida.



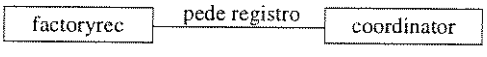
Relacionamento *factoryrec - resource*:

Uma *factoryrec* instancia *resource*. Esta associação mostra que a fábrica de recursos (classe *factoryrec*) é responsável pela criação dos objetos da classe *resource*.



Relacionamento *factoryrec - coordinator*:

Uma *factoryrec* pede para o coordenador (classe *coordinator*) registrar um recurso. Esta associação ilustra a responsabilidade da fábrica de recursos de solicitar ao *coordinator* o registro dos recursos por ele criados.



A Figura 3-4 mostra o modelo de classes do ST, com todos os relacionamentos descritos anteriormente. Em seguida, são mostrados os atributos e os métodos vinculados às classes do ST.

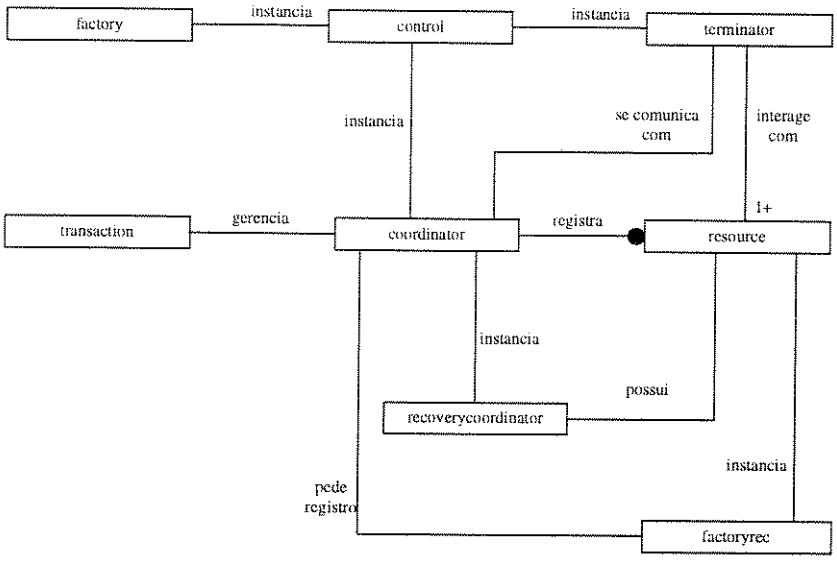


Figura 3-4: Classes e Associações.

3.3.1.2.Atributos e métodos das classes

As classes possuem atributos (ou dados) e operações (ou métodos) que definem, respectivamente, as suas características e o seu comportamento. Os métodos definem os serviços fornecidos pela classe. Os atributos definem as propriedades da classe.

Baseado no documento de especificação da OMG, foram identificados os métodos e os atributos de cada classe. Embora tenha-se no documento a definição da maioria dos métodos que cada classe possui, a grande dificuldade foi compreender o significado dos mesmos tanto para a classe quanto no contexto global. A Figura 3-5 mostra o modelo de classes incluindo os métodos e atributos definidos nas interfaces.

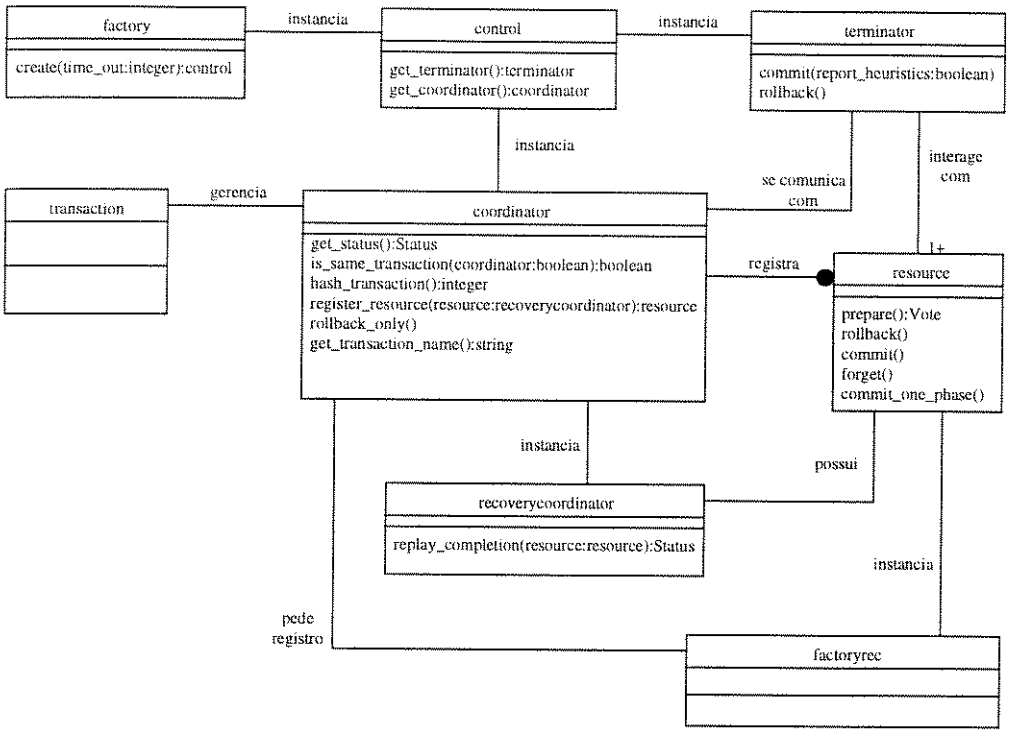


Figura 3-5: Modelo incluindo atributos e métodos definidos no documento da OMG.

Após analisar a funcionalidade de cada método, foi verificada a necessidade de incluir outros métodos e atributos para auxílio da implementação dos métodos predefinidos no documento. O modelo resultante é mostrado na Figura 3-6. A descrição dos métodos e atributos encontram-se no Anexo 2, obtido com auxílio da ferramenta LOV/OMT.

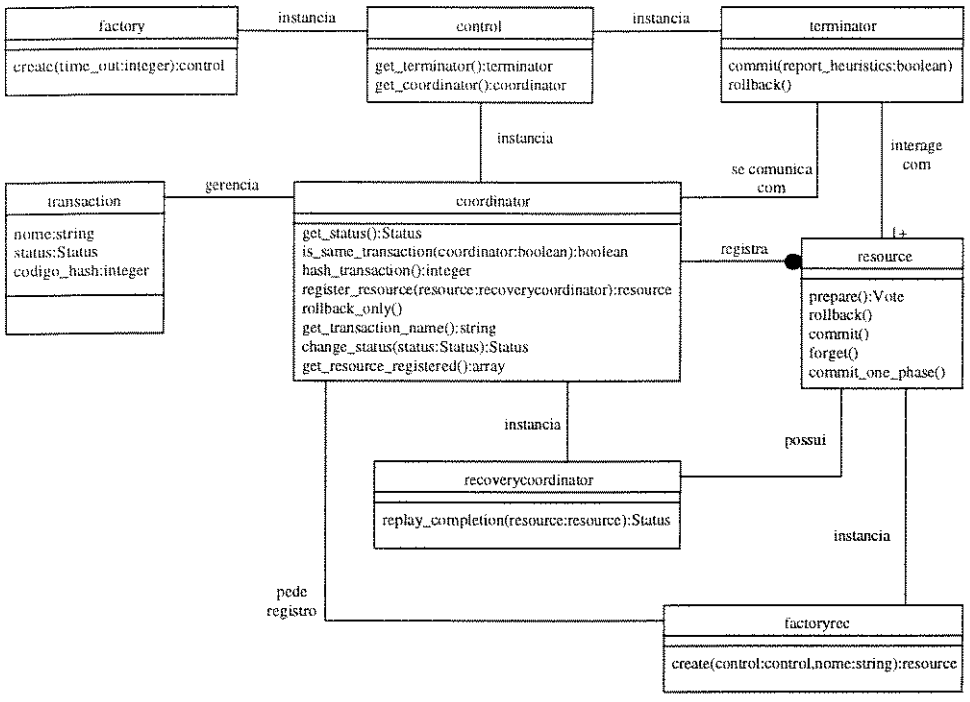


Figura 3-6: Modelo com novos atributos e novos métodos.

Como pode ser observado, na Figura 3-6 foram acrescentados dois métodos na classe *coordinator*. O método change_status() é necessário para mudar o estado da transação (o coordenador é o responsável por esta atividade). O método get_resource_registered() é necessário para obtenção da informação sobre os recursos registrados como participantes da transação. O método *create*, da classe *factoryrec*, foi acrescentado para controlar a criação de objetos, da classe *resource*, que participam da transação. Também foram acrescentados os atributos da classe *transaction*, identificados a partir do conhecimento do problema (*transaction* não possui uma definição de interface em IDL).

3.3.1.3.Estratégia de implementação das associações

Associações podem ser implementadas de diversas formas, utilizando ponteiros, classes ou estruturas de dados. Dependendo das necessidades, estas associações podem ser unárias (em um sentido só) ou binárias (em ambos sentidos). Para o modelo do ST todas as associações são unárias. As estratégias de implementação adotadas foram: como ponteiros para a classe associada; como uma outra classe; e como uma estrutura de dados.

A associação um-para-um: "*terminator* se comunica com *coordinator*" foi implementada através da definição de sinais no modelo construído em SDL apresentado na seção 3.3.2 e a partir desse modelo é gerado o código automaticamente.

As associações um-para-um restantes foram implementadas no ambiente CORBA utilizando ponteiros que indicam o objeto com quem a classe deve se relacionar. Para implementar as associações um-para-muitos: "*coordinator* registra *resource*" e "*terminator* interage com *resource*" foram definidos no modelo em SDL *arrays* ou tabelas. A partir dessas definições é gerado o código de forma automática.

A associação entre as classes *coordinator* e *transaction* foi implementada como uma estrutura de dados dentro da classe *coordinator* . Esta definição foi feita no modelo em SDL, isto porque a classe *transaction* guarda unicamente informações, não possuindo outras operações nem outras informações que necessitam ser encapsuladas. Os seus atributos podem ser vistos por qualquer outra classe. A Figura 3-7 ilustra as estratégias adotadas. Os detalhes da codificação são apresentados na seção 3.5.

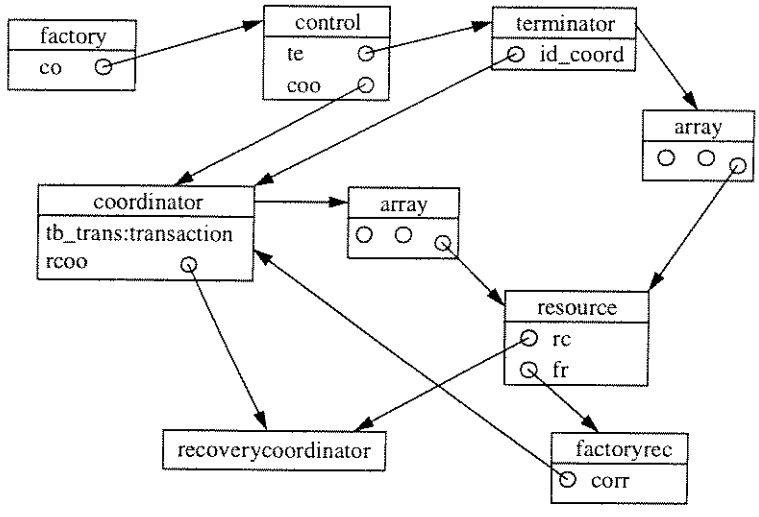


Figura 3-7: Estratégia de implementação das associações.

A modelagem estática termina após a construção do diagrama de classes e do dicionário de dados, que descreve cada elemento do sistema.

Tendo a estrutura do sistema é necessário analisar também a complexidade do comportamento de cada classe. Aquelas que apresentarem uma complexidade maior precisam ser modeladas de forma mais elaborada. A complexidade depende das interações que os objetos da classe realizam com os outros objetos do sistema e/ou com o ambiente externo. A modelagem dinâmica e funcional do ST é apresentada a seguir. São apresentados os critérios seguidos para sua construção e as técnicas e ferramentas usadas para este propósito.

3.3.2. Modelo dinâmico e funcional

O modelo dinâmico deve mostrar como os objetos reagem a estímulos ou eventos externos. Um primeiro passo na modelagem dinâmica foi identificar as seqüências de interações entre os objetos e com o ambiente. Para isto, foi utilizado o Diagrama de

Seqüência de Mensagens (MSC = Message Sequence Chart)[5] [8], o qual mostra como os objetos envolvidos no ST interagem internamente e com o ambiente externo. Esta notação permite estabelecer cenários de comportamento do sistema.

Inicialmente foi construído um cenário global, ilustrado na Figura 3-8, que inclui as principais classes envolvidas no ST. O MSC tem sido também muito utilizado neste trabalho na geração de cenários para os testes no processo de simulação, descrevendo as seqüências de execução esperadas. Os cenários construídos fazem parte da documentação do sistema.

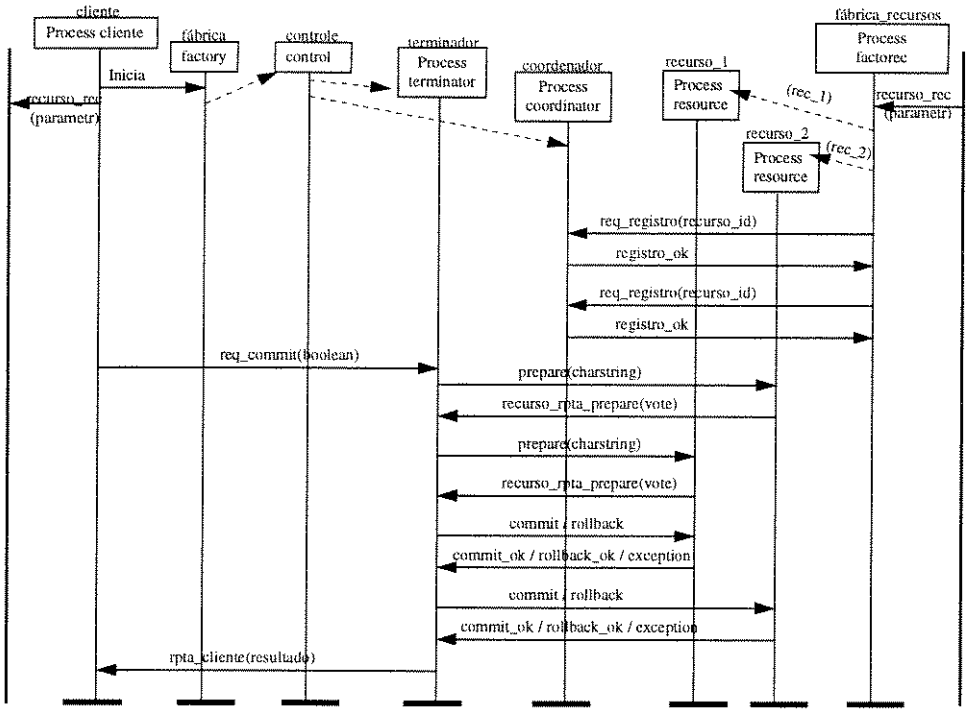


Figura 3-8: Cenário do ST utilizando a notação MSC.

A Figura 3-8 mostra a interação entre as classes que compõem o ST na forma de troca de mensagens. As linhas verticais nos dois extremos da figura representam o ambiente externo. Na figura, foi incorporado o cliente do ST, que faz parte do ambiente externo. Isto foi feito para uma melhor visualização da interação. O início da seqüência de mensagens é o sinal "inicia" enviado pelo cliente. Este sinal define o início da transação. A seta pontilhada representa a criação de uma instância do processo. Iniciada a transação o cliente pode interagir com diversos recursos, os quais devem ser instanciados e registrados como participantes da transação. O controle destas atividades pertence à fábrica de recursos, quem recebe a requisição do cliente através do sinal recurso_rec. Na figura, o sinal recurso_rec inicia no extremo esquerdo e continua no extremo direito, significando que o sinal é enviado pelo cliente, mas é recebido do ambiente pela fábrica de recursos. Em seguida, a fábrica de

[5]O MSC, especificado pela ITU-T, pode ser aplicado na especificação do comportamento de sistemas de tempo real e, particularmente, em sistemas de telecomunicações.

recursos instancia e solicita o registro dos recursos. Quando o cliente decide fechar a transação envia o sinal req_commit, que produz o início do protocolo *commit* em duas fases gerenciado pelo terminador.

A partir da análise deste cenário pode-se observar que, principalmente, as classes *terminator* e *coordinator* possuem um comportamento complexo na execução das suas funções. Isto se deve à execução do protocolo *commit* em duas fases, visível mais claramente no cenário da Figura 3-9. Esta figura ilustra de modo geral as interações de ambas as classes.

Dado que existe um grande número de interações e trocas de sinais possíveis, é necessário construir um modelo dinâmico para elas, mostrando os seus estados internos e as operações realizadas nas transições dos seus estados.

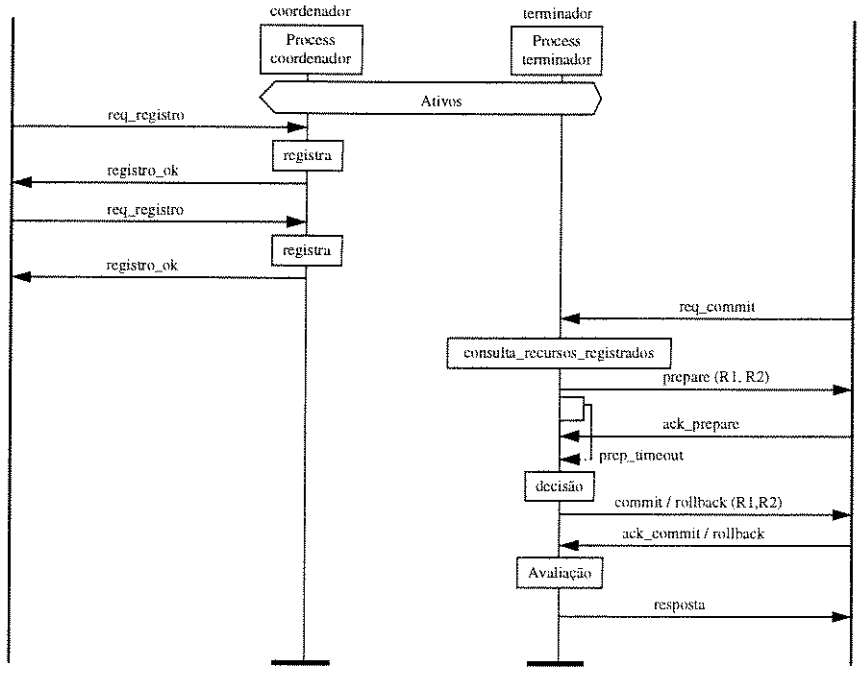


Figura 3-9: Protocolo commit em duas fases utilizando a notação MSC.

A Figura 3-9 ilustra as entradas e saídas que o terminador e o coordenador possuem. Coordenador e terminador devem estar ativos no início da sua execução. Os retângulos representam um conjunto de ações realizadas no processo. prep_timeout representa um *time out*. O protocolo inicia após ter um conjunto de recursos registrados e um pedido de finalização da transação expresso através da mensagem req_commit. Após conhecer os recursos registrados, o terminador envia sinais de *prepare* para cada recurso e espera uma resposta por um tempo. Ao receber a resposta de todos os recursos e analisá-las, o terminador envia para os recursos uma mensagem de *commit* ou *rollback*. O terminador espera novamente uma resposta. Após uma avaliação das respostas o terminador envia o resultado (se for necessário) para o cliente.

Dado que SDL permite especificar de forma detalhada a parte funcional dos processos através de tarefas internas executadas em cada transição de estado, foi decidido utilizá-lo também na modelagem funcional. O objetivo é explicitar ao máximo como o sistema pode ser implementado.

A construção dos modelos em SDL iniciou com a análise e transformação das classes OMT em termos de processos e Tipos Abstratos de Dados (ADTs) de SDL.

Dentre as classes identificadas para o ST, *terminator*, *coordinator* e *transaction* são as que apresentam um comportamento dinâmico mais complexo. Dado que *terminator* e *coordinator* são classes ativas, ou seja, que solicitam serviços de outras classes, elas são transformadas em processos SDL (PROCESS terminador e PROCESS coordenador, respectivamente). Por outro lado, *transaction* é uma classe passiva, pois apenas atende as solicitações de serviços de outras classes (é gerenciada pela classe *coordinator*, e não possui em si mesma um comportamento dinâmico). Esta classe é transformada em um ADT em SDL (NEWTYPE tb_transaction), sendo gerenciada pelo processo coordenador.

É importante ressaltar que não existe uma regra para a transformação das classes OMT para os processos SDL. Os critérios de transformação estão baseados em trabalhos envolvendo OMT e SDL, tais como as referências [14] e [15].

A partir desta análise, foi construída a estrutura hierárquica do sistema (Figura 3-10) composta pelos processos definidos anteriormente. Após isto, foi feita a definição dos sinais e a construção do diagrama de interconexão (Figura 3-11). O diagrama mostra a relação entre os processos terminador e coordenador, e os sinais trocados entre os processos e com o ambiente externo.

Entenda-se por ambiente externo tudo o que está fora dos limites do sistema (ou parte do sistema) especificado em SDL. No ST, o ambiente externo é constituído pelas classes *factory*, *control*, *resource*, *factoryrec* e *recoverycoordinator*, que não fazem parte da especificação em SDL.

Vale ressaltar que o ambiente externo se torna importante no processo de simulação, onde a simulação do seu comportamento visa testar a reação dos processos aos sinais externos e a suas atividades internas.

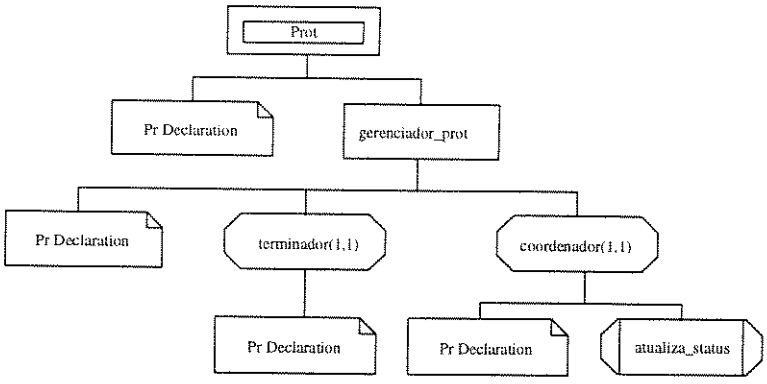


Figura 3-10: Estrutura hierárquica do modelo em SDL.

O protocolo implementado a partir da especificação em SDL é apresentado na hierarquia da Figura 3-10. Nesta hierarquia, o sistema Prot é formado por um único bloco e as declarações globais ao sistema. O bloco gerenciador_prot é formado por 2 processos: terminador e coordenador. As suas definições ficam visíveis unicamente dentro do bloco. Cada processo possui as suas próprias definições e pode possuir procedimentos (atualiza_status) dentro da sua especificação.

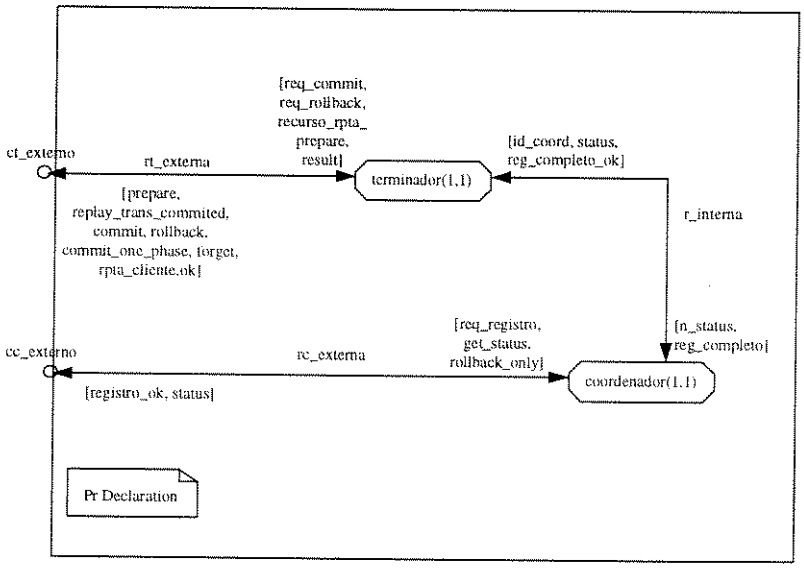


Figura 3-11: Diagrama de interconexão do modelo em SDL a nível de bloco.

O diagrama da Figura 3-11 é formado pelos processos definidos na hierarquia, os sinais trocados, as rotas de sinal dentro do bloco (rt_externa, r_interna, rc_externa) e os nomes dos canais que conectam o bloco com o exterior (ct_externo, cc_externo).

A partir deste ponto iniciou-se a construção das EFSMs que descrevem o comportamento interno dos processos terminador e coordenador. Cada EFSM mostra

estados seguidos de eventos que disparam transições (estes eventos podem ser sinais ou condições).

As ações representadas nas transições constituem a parte funcional da especificação. Nelas são representadas condições, cálculos, envio de sinais, procedimentos, controles de tempo, mudanças de estado, entre outros. Toda especificação em SDL é sempre acompanhada das definições de ADT, variáveis, constantes, etc., necessários para a sua execução.

Antes de apresentar as EFSMs destes processos serão descritas as construções mais usadas, algumas definições e situações especiais que foram modeladas. O objetivo é facilitar a compreensão das EFSMs. Para isto, será utilizada a representação textual de SDL (SDL_PR).

3.3.2.1.Construções mais usadas na modelagem

As EFSMs, resultado da modelagem feita em SDL, contêm diversas construções para descrever os aspectos dinâmico e funcional dos processos coordenador e terminador. Basicamente, foram utilizadas as seguintes construções:

- TASK: Usada para realizar acessos e atualizações das tabelas, variáveis e contadores, e mostrar comentários.
- DECISION: Usada para tomar decisões em função de um valor verdadeiro ou falso. Cada decisão determina um caminho de execução diferente.
- OUTPUT: Usada para envio de sinais a um determinado receptor seja este um processo ou um ambiente externo.
- INPUT: Usada para a recepção de sinais de outro processo ou do ambiente externo.
- JOIN: Usada para transferir o fluxo de controle de uma parte da especificação para outra.
- PROCEDURE: Usado para agrupar em um procedimento seqüências da especificação que se repetem em outras partes da mesma.

3.3.2.2.Definições utilizadas na modelagem

Para construir os modelos em SDL, é necessário definir ADTs, sinais, constantes e variáveis. Dependendo do nível em que serão utilizadas, as definições podem ser feitas a nível de sistema, bloco ou processo.

A seguir, são dadas algumas definições utilizadas no modelo. Os ADTs foram definidos em função dos tipos predefinidos que são oferecidos por SDL.

No nível mais alto da hierarquia foram definidos tipos de dados que são usados nos níveis inferiores. Entre eles podem ser encontrados os seguintes:

➢ Definição de tipos numéricos (SYNTYPE). São números que pertencem a um intervalo. No modelo pode ser encontrada a seguinte definição:

```
SYNTYPE index1 integer
  CONSTANTS 1:n;
ENDSYNTYPE index1;
```

Esta é a declaração de um tipo index1, que é um conjunto de inteiros variando de 1 até n.

➢ Definição de constantes (SYNONYM). É uma constante de qualquer tipo predefinido. No modelo, é usada para a definição de uma constante de tipo PID, que representa a identificação da tarefa externa (entidade do ambiente externo) que interage com a aplicação em SDL, fornecendo os sinais de entrada. Também é usada para a definição de uma constante: n. As definições são as seguintes:

```
SYNONYM ext_tarefa_1 PID = NULL;
SYNONYM n natural = 100;
```

➢ Definição de outros tipos de dados (NEWTYPE). São definidos em função dos tipos predefinidos. Algumas definições são mostradas a seguir:

⇒ NEWTYPE vote: mostra a definição de valores *string* constantes e ordenados. É similar a tipos *enum* em outras linguagens.

```
NEWTYPE vote
      LITERALS VoteCommit, VoteRollback, VoteReadOnly;
OPERATORS ORDERING;
ENDNEWTYPE vote;
```

Este tipo é usado para representar as possíveis respostas dos recursos a um pedido de *prepare*.

⇒ NEWTYPE vote_type: representa uma estrutura de valores do tipo *vote*.

```
NEWTYPE vote_type
      STRUCT
            var_voto vote;
ENDNEWTYPE vote_type;
```

⇒ NEWTYPE tb_vote: representa uma tabela que contém os valores das respostas ao pedido de *prepare*, fornecidos por cada recurso participante da transação. Este tipo de definição é considerado em SDL uma construção do tipo ARRAY (tipo predefinido) com duas componentes, uma do tipo index1 e a outra do tipo vote_type, definidos anteriormente.

```
NEWTYPE tb_vote
      ARRAY (index1,vote_type);
ENDNEWTYPE tb_vote;
```

⇒ NEWTYPE tb_transaction: define uma estrutura de dados. Estes dados pertencem à transação corrente.

```
NEWTYPE tb_transaction
    STRUCT
        var_nome_trans   charstring;
        var_status   status;
        var_hash_cod   integer;
ENDNEWTYPE tb_transaction;
```

➢ Definição de sinais. No nível mais alto da hierarquia são definidos também os sinais que são trocados com o ambiente externo. Eles podem ser observados graficamente através do diagrama de interconexão apresentado anteriormente na Figura 3-11. Algumas definições de sinais utilizadas no ST são as seguintes:

⇒ SIGNAL req_registro (charstring);

Este é um sinal de requisição de registro de um recurso. O parâmetro é um tipo predefinido de SDL, que representa uma *string*, que identifica o recurso a ser registrado. Este sinal é recebido pelo processo coordenador do ambiente externo.

⇒ SIGNAL registro_ok;

É um sinal de confirmação do registro do recurso. Este sinal é enviado para o ambiente externo pelo processo coordenador.

⇒ SIGNAL req_commit (boolean);

Este é um sinal de requisição para fechar a transação executando *commit.* O parâmetro pode possuir um valor *true* ou *false* que indica a requisição ou não de um informe das exceções. Este sinal é recebido pelo processo terminador do ambiente externo.

⇒ SIGNAL *prepare* (charstring);

Sinal de requisição para executar a função *prepare*, feita para cada recurso registrado. O parâmetro é do tipo *string* e identifica o recurso que recebe o sinal. Este sinal é enviado para o ambiente externo pelo processo terminador.

No nível de bloco, o segundo nível da hierarquia, foram definidos os sinais trocados entre processos que pertencem àquele bloco. Neste trabalho, os processos terminador e coordenador foram definidos no mesmo bloco. Uma forma deles se comunicarem é através de troca de sinais. Estes sinais são transparentes fora dos limites do bloco. Um destes sinais é definido a seguir:

⇒ SIGNAL n_status(status);

É um sinal trocado entre o processo coordenador e o terminador contendo como parâmetro um valor do tipo *status*. Este sinal é usado pelo

processo terminador para requisitar a mudança do valor do estado da transação.

No nível de processo, para cada um são definidos os sinais de entrada (SIGNALSET) e as variáveis usadas nas transições de estado. Todas as variáveis (definidas com DCL ou DCL REVEALED) devem ser declaradas antes de serem usadas.

A seguir são apresentadas algumas das situações especiais modeladas nas EFSMs.

3.3.2.3.Situações especiais

3.3.2.3.1.Acesso e atualização de tipos ARRAY e STRUCT em SDL

Sempre que for necessário o acesso a um valor de uma construção do tipo ARRAY ou STRUCT deve-se executar uma tarefa interna (TASK) como é mostrado a seguir:

rec_regist é uma variável que pertence ao processo terminador. Seu tipo é tb_rec_reg, definido como ARRAY. Esta variável contém as identificações dos recursos registrados. Para ter acesso a algum valor armazenado nesta variável será necessário realizar uma tarefa interna contendo o seguinte:

TASK rec_id := rec_regist(i)(var_charstr);

Onde: rec_id é uma variável do tipo predefinido *charstring* que recebe um valor do *array*; rec_regist é o nome do *array*; i determina a posição no *array*; e var_charstr é uma variável do tipo charstring que contém o valor do *array* na posição i.

Para atribuir valores a um *array*, deve-se, como no caso anterior, executar uma tarefa interna (TASK) como mostrado a seguir:

TASK tb_recursos(i)(var_charstr) := id_recurso;

A variável tb_recursos recebe na posição i o valor de outra variável id_recurso, onde: tb_recursos define um nome para o *array*; i define a posição dentro do *array* a ser preenchida; var_charstr é uma variável do tipo *charstring* que conterá o novo valor na posição i; e id_recurso é uma variável do tipo *charstring* que contém o valor a ser armazenado no *array*.

Quando se trata de uma estrutura STRUCT deve-se usar o símbolo "!" para identificar o campo da estrutura requerido. Para colocar um valor na estrutura realiza-se uma tarefa interna contendo o seguinte:

TASK tb_trans!var_status := novo_status;

Onde: tb_trans é o nome da estrutura; var_status é o campo da estrutura; e novo_status é o valor a ser colocado no campo da estrutura.

Para obter um valor da estrutura, a tarefa interna a ser executada é a seguinte:

TASK status_trans := tb_trans!var_status;

Onde: status_trans, do tipo status, recebe o valor do campo var_status da estrutura tb_trans.

Outros exemplos podem ser encontrados nas EFSMs do processo terminador e coordenador na sua representação gráfica (SDL-GR) no Apêndice 1.

3.3.2.3.2.Comunicação entre os processos

A comunicação entre os processos terminador e coordenador é realizada de duas formas: usando a construção OUTPUT (a forma mais comum de comunicação) e através de compartilhamento de variáveis com o mecanismo REVEALED-VIEWED (veja Anexo 1). Foi utilizado este último mecanismo, dado que ambos os processos rodam na mesma máquina e foram definidos no mesmo bloco, diminuindo a quantidade de interação por troca de sinais e aumentando a concorrência entre os processos. A seguir é detalhado como este mecanismo de comunicação é utilizado.

O processo coordenador é dono das informações relacionadas com a transação e com os recursos registrados. Quando o processo terminador recebe um pedido para fechar a transação ele precisa saber quais recursos estão registrados como participantes da transação. Para obter essa informação, o terminador deve pedi-la ao coordenador.

Esta informação é obtida com o mecanismo REVEALED-VIEWED, descrita na Tabela 3-II.

Processo coordenador	Processo terminador
O coordenador declara uma variável do tipo tb_rec_reg, um *array* que contém as informações dos recursos registrados. Isto é feito da seguinte forma: DCL REVEALED tb_recursos tb_rec_reg;	O terminador deve definir uma variável do mesmo tipo que o *array* do coordenador usando a seguinte expressão: VIEWED tb_recursos tb_rec_reg; Adicionalmente é declarada uma variável que guarda a informação obtida: DCL rec_regist tb_rec_reg; Quando o processo precisa ter acesso à informação, ele deve realizar a tarefa interna: TASK rec_regist:=VIEW(tb_recursos,coord_id);

Tabela 3-II: Mecanismo REVEALED - VIEWED.

O processo terminador precisa da identificação do processo coordenador para ter acesso à informação. Neste caso, a identificação do coordenador está contida na variável coord_id, cujo valor é recebido no momento da inicialização da aplicação.

3.3.2.3.3.Controle de tempo

Um mecanismo usado na modelagem é o controle de tempo usando *time out*. O processo terminador deve enviar para o ambiente externo um sinal *prepare*, que será recebido por cada recurso. Para cada sinal enviado, o processo terminador deve esperar por um tempo determinado a resposta contendo um valor do tipo *vote*. Enquanto a resposta não chegar, o processo terminador não pode enviar o sinal para o próximo recurso. Um *time out* acontece quando o tempo expirou e a resposta esperada ainda não chegou. Este controle é descrito a seguir.

No processo terminador é definido um relógio ou TIMER:

```
TIMER prep_timeout;
```

Durante a execução da operação de envio de sinais de *prepare*, o terminador executa as seguintes tarefas:

```
OUTPUT prepare(rec_id) TO ext_tarefa_1;
SET (now+5, prep_timeout);
WAIT espera_confirm;
```

Onde a primeira sentença representa o envio de um sinal ao ambiente externo, a segunda inicializa a contagem do tempo e a terceira define um estado em que o terminador espera a resposta.

No estado espera_confirm o processo terminador espera uma resposta do recurso ou *time out*. Se a resposta do recurso chegar antes da expiração do tempo, o relógio é zerado usando a expressão:

```
RESET (prep_timeout);
```

Caso contrário, é executada a transição que segue ao sinal de *time out* :

```
INPUT prep_timeout;
```

3.3.2.3.4.Ativação de uma transição por uma condição verdadeira

Este mecanismo pode ser observado no processo terminador. Ao finalizar o envio dos sinais de *prepare* o terminador executa uma tarefa interna mudando o valor de uma variável de condição, inicializada como *false*, para *true*, ativando a transição do estado decisão.

No modelo, isto é representado como:

```
TASK decisão := TRUE;
NEXSTATE decisão;
.....
STATE decisão;
PROVIDED decisão = true;
```

O estado decisão é ativado quando a variável decisão assume o valor *true* (PROVIDED decisão = true).

3.3.2.4.Características das EFSMs dos processos terminador e coordenador

No modelo SDL do ST existem 2 processos: terminador e coordenador. Eles são ativados na inicialização do sistema e cada um pode iniciar o seu processamento de forma independente (concorrentemente). Não existe uma seqüência preestabelecida de atividades entre eles. A execução das atividades depende dos eventos ou sinais enviados a eles.

Dentro de cada processo, pode-se observar também estados a espera de vários sinais de entrada possíveis. Estes sinais são concorrentes e não possuem uma ordem de ocorrência preestabelecida nem uma quantidade fixa de ocorrências. Os sinais são enfileirados na ordem de chegada.

As transições de estado não produzem necessariamente mudanças de estado. Isto depende dos valores dos dados internos ao processo.

As EFSMs para cada processo foram construídas com o auxílio do editor da ferramenta GEODE [19]. Uma visão global das suas características é dada a seguir. Os modelos na sua representação gráfica (SDL_GR) são mostrados no Apêndice 1.

3.3.2.4.1.EFSM do processo terminador

A EFSM do processo terminador, mostrado no Apêndice 1, tem como tarefa básica o gerenciamento do protocolo *commit* em duas fases, interagindo com o processo coordenador e com os recursos registrados, os quais fazem parte do ambiente externo.

O processo terminador é composto por diversos estados (Figura 3-12) os quais serão alcançados, dependendo da ocorrência de determinados eventos.

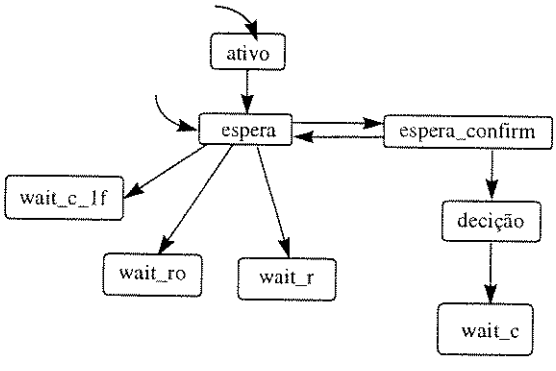


Figura 3-12: Estados do processo terminador.

O detalhe dos estados da Figura 3-12 são mostrados a seguir.

Estado ativo:

O sinal esperado neste estado é: id_coord(parâmetro). Este sinal possui a identificação do processo coordenador e produz a mudança para o estado espera. Não executa nenhuma atividade na transição.

Estado espera:

Os sinais esperados neste estado são:

1) req_commit(parâmetro). Este sinal é enviado pelo cliente (faz parte do ambiente externo) e solicita o fechamento da transação com *commit*. O parâmetro indica a requisição ou não do informe de exceções.

   Atividade na transição:

   - Enviar o sinal n_status ao coordenador para pedir a mudança do estado da transação.

   Próximo estado:

   - espera: não há mudança de estado.

2) status(parâmetro). Sinal enviado pelo coordenador. Contém o novo valor do estado da transação.

   Atividades possíveis na transição:

   - Iniciar a execução do protocolo *commit* em duas fases.
   - Pedir para o coordenador a informação dos recursos registrados.
   - Enviar o sinal *prepare* para os recursos registrados.
   - Enviar o sinal req_rollback para si mesmo.
   - Enviar o sinal commit_one_phase para um recurso.

   Próximos estados possíveis:

   - wait_c_1f: quando existe um único recurso registrado.
   - espera_confirm: quando existem vários recursos registrados e é preciso esperar a resposta de todos ao sinal de *prepare*.
   - espera: quando o estado da transação é StatusMarkedRollback. Não se produz uma mudança de estado.

3) req_rollback. Sinal enviado pelo cliente ou gerado pelo próprio terminador. Visa solicitar o fechamento da transação com *rollback*.

   Atividades possíveis na transição:

   - Verificar se o coordenador completou o registro dos recursos.

- Pedir para o coordenador a informação dos recursos registrados.
- Enviar o sinal *rollback* para os recursos registrados.

Próximos estados possíveis:

- espera: quando se precisa da confirmação do coordenador do registro dos recursos. Não se produz uma mudança de estado.
- wait_ro: quando a requisição de *rollback* foi feita pelo cliente.
- wait_r: quando a requisição de *rollback* foi feita pelo próprio terminador.

4) reg_completo_ok. Sinal enviado pelo coordenador para confirmar que o registro dos recursos foi completado.

Atividade na transição:

- Enviar o sinal *rollback* para os recursos registrados.

Próximos estados possíveis:

- wait_ro: quando a requisição de *rollback* foi feita pelo cliente.
- wait_r: quando a requisição de *rollback* foi feita pelo próprio terminador.

Estado wait_c_1f:

O sinal esperado neste estado é:

1) result (parâmetro). Este sinal possui a resposta do recurso ao sinal commit_one_phase que foi enviado para o único recurso registrado. O processo terminador pode finalizar a sua execução neste estado.

Atividades possíveis na transição:

- Enviar o sinal n_status ao coordenador para pedir mudança do estado da transação.
- Enviar o sinal rpta_cliente ao cliente.
- Enviar o sinal *forget* ao recurso.

Estado espera_confirm:

Os sinais esperados neste estado são:

1) recurso_rpta_prepare(parâmetro). Sinal que possui a resposta dos recursos ao sinal *prepare*.

Atividades possíveis na transição:

- Enviar o sinal req_rollback para si mesmo.
- Ativar o estado decisão.

Próximos estados possíveis:

- espera: quando executa o pedido de *rollback* para si mesmo.
- decisão: quando é necessário uma avaliação antes de fechar a transação com *commit*.

2) prep_timeout. Sinal recebido quando acontece um *time out*. Este sinal acontece quando algum recurso não responde ao sinal de *prepare*.

Atividade na transição:

- Enviar o sinal req_rollback para si mesmo.

Próximo estado:

- espera: para poder iniciar o envio dos sinais de *rollback* para os recursos registrados.

Estado decisão:

Este estado não é ativado por sinal, ocorre quando a variável decisão for verdadeira.

Atividade na transição:

- Avaliar respostas dos recursos antes de fechar a transação com *commit*.

Próximo estado:

- wait_c: quando a transação será fechada com *commit*.

Estado wait_c:

O sinal esperado neste estado é:

1) result (parâmetro). Este sinal é enviado pelo recurso que recebeu um sinal *commit*. O processo terminador pode finalizar a sua execução neste estado.

Atividades possíveis na transição:

- Enviar o sinal n_status ao coordenador para pedir a mudança do estado da transação.
- Enviar o sinal rpta_cliente ao cliente.
- Enviar o sinal *forget* aos recursos.

Próximo estado:

- wait_c: o processo não muda de estado até receber todas as respostas ao sinal *commit*.

Estado wait_r:

O sinal esperado neste estado é:

1) result (parâmetro). Este sinal é enviado pelos recursos que receberam o sinal *rollback*. O processo terminador pode finalizar a sua execução neste estado.

Atividades possíveis na transição:

- Enviar o sinal n_status ao coordenador para pedir a mudança do estado da transação.
- Enviar o sinal rpta_cliente ao cliente.
- Enviar o sinal *forget* ao recurso.

Próximo estado:

- wait_r: o processo não muda de estado até receber todas as respostas ao sinal *rollback*.

Estado wait_ro:

O sinal esperado neste estado é:

1) result (parâmetro). Este sinal é enviado pelos recursos que receberam um sinal de *rollback*. O processo terminador pode finalizar a sua execução neste estado.

Atividades possíveis na transição:

- Enviar o sinal n_status ao coordenador para pedir mudança do estado da transação.
- Enviar o sinal rpta_cliente ao cliente.
- Enviar o sinal *forget* ao recurso.

Próximo estado:

- wait_ro: o processo não muda de estado até receber todas as respostas ao sinal *rollback*.

Os dois últimos estados são aparentemente iguais. Na realidade, ambos possuem o mesmo objetivo, mas as suas execuções são diferentes. Isto pode ser notado na execução das decisões dentro da transição de estado.

3.3.2.4.2.EFSM do processo coordenador

Este processo se concentra em registrar os recursos e devolver o estado da transação corrente. Sempre será possível perguntar pelo estado da transação, enquanto que o registro de recurso somente pode ser executado quando este processo está no estado ativo. No Apêndice 1 é apresentada a EFSM do processo coordenador na sua representação gráfica (SDL_GR).

O coordenador sai do estado ativo quando chegar a informação de que a transação só poderá ser fechada executando *rollback* (através do sinal rollback_only) ou quando um sinal do terminador pedindo a confirmação da finalização do registro de recursos é recebido através do sinal reg_completo. A Figura 3-13 mostra os estados do processo coordenador.

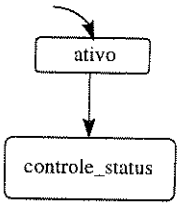


Figura 3-13: Estados do processo coordenador.

O detalhe dos estados do coordenador são apresentados a seguir.

Estado ativo:

Os sinais esperados neste estado são:

1) req_registro(parâmetro). Sinal enviado pela fábrica de recursos. Possui como parâmetro a identificação de um recurso.

   Atividade na transição:

   - Registrar o recurso como participante da transação, guardando a sua identificação.

   Próximos estados possíveis:

   - ativo: não muda de estado porque espera mais requisições de registro ou outros sinais.
   - controle_status: quando finalizou o processo de registro, ou seja, não é mais permitido o registro de recursos.

2) get_status. Sinal recebido do ambiente requisitando informações sobre a transação.

   Atividade na transição:

   - Devolver o estado da transação.

Próximo estado:

- ativo: não existe mudança de estado porque este evento é só uma consulta.

3) rollback_only. Sinal recebido do ambiente externo. Produz a mudança no estado da transação.

Atividade na transição:

- Mudar o estado da transação para aquele que só permite a posterior execução de *rollback* (StatusMarkedRollback).

Próximo estado:

- controle_status: esta mudança de estado significa que não será mais permitido o registro de recursos.

4) n_status(parâmetro). Sinal enviado pelo terminador. Possui como parâmetro o novo estado da transação.

Atividades possíveis na transição:

- Mudar o estado da transação para um valor vinculado ao protocolo *commit* em duas fases (StatusPrepared, StatusCommit, StatusRollback).
- Enviar o novo estado da transação.

Próximo estado:

- controle_status: significa que não será mais permitido o registro de recursos.

5) reg_completo. Sinal requisitando a confirmação do término do registro de recursos participantes.

Atividade na transição:

- Retorno de confirmação da finalização do registro dos recursos com o envio do sinal reg_completo_ok.

Próximo estado:

- controle_status: significa que não será mais permitido o registro de recursos.

Estado controle_status:

Os sinais esperados neste estado são os seguintes:

1) get_status. Sinal recebido do ambiente requisitando informações sobre o estado da transação.

Atividade na transição:

- A atividade neste estado é a mesma que no estado ativo, ou seja, devolver o estado da transação.

Próximo estado:

- controle_status: o processo permanece no mesmo estado.

2) rollback_only. Sinal que produz a mudança de estado da transação para o valor StatusMarkedRollback.

Atividade na transição:

- É a mesma que no estado ativo, ou seja, mudar o estado da transação para um valor que só permita executar *rollback* (StatusMarkedRollback).

Próximo estado:

- controle_status: o processo permanece no mesmo estado.

3) n_status(parâmetro). Sinal enviado pelo terminador requisitando a mudança do estado da transação.

Atividade na transição:

- Mudar o estado da transação para um valor vinculado ao protocolo *commit* em duas fases (StatusPrepared, StatusCommit, StatusRollback).

Próximo estado:

- controle_status: o processo permanece no mesmo estado.

O processo de construção das EFSMs não foi uma tarefa simples pois precisa-se levar em consideração tanto a especificação da OMG quanto a sintaxe e notação de SDL.

Durante a construção do modelo a sintaxe foi verificada, visando preparar o modelo para a etapa de simulação e verificação.

As atividades realizadas durante o processo de simulação são descritas a seguir.

3.4. Simulação e verificação do modelo SDL

Visando testar o modelo, foi realizado um processo de simulação, onde foi verificado o seu comportamento e validada a sua execução, segundo os requisitos especificados pela OMG.

A simulação é considerada fundamental no desenvolvimento de SDAs e em geral de sistemas que realizam muitas interações. Isto permite observar como o sistema está se comportando, favorecendo a melhoria do modelo inicial, possibilitar o teste e detecção de casos especiais de comportamento e preparar o sistema para a codificação.

O ponto de partida da simulação foi a elaboração de cenários do comportamento esperado do sistema. Construir cenários permite testar a funcionalidade do sistema. Foram elaborados alguns cenários básicos usando o MSC, suportado pela ferramenta GEODE [19].

A medida que foram realizadas as primeiras simulações encontrou-se principalmente erros de modelagem e situações de *deadlock*, além de cenários não previstos, podendo assim aumentar a abrangência dos testes sobre o modelo.

As atividades embutidas no processo de simulação são apresentadas a seguir. Inicialmente, é realizada uma descrição do ambiente de simulação. Após, são apresentados 2 casos básicos de simulação. Em seguida, são mostrados os resultados obtidos em função dos cenários envolvidos e, finalmente, as vantagens e problemas encontrados durante este processo.

3.4.1. Ambiente de simulação

A simulação baseada no modelo construído em SDL foi realizada no simulador da ferramenta GEODE [20].

Cada simulação gera automaticamente um ou vários cenários contendo a seqüência de execução testada, mostrando também os valores das variáveis envolvidas e as falhas observadas. Os cenários gerados podem ser iguais aos especificados pelos cenários na notação MSC. Se forem diferentes é necessário analisar se o cenário gerado está correto ou não.

Durante a simulação podem ser incorporados os cenários construídos com o MSC, que atuam como observadores do comportamento do modelo. O simulador compara o comportamento esperado, descrito nos observadores, com o comportamento do modelo sendo simulado.

É importante notar que neste trabalho os MSCs construídos não foram usados como observadores dentro da simulação, mas como sugerido na literatura [8] e [17], foram utilizados para representar comportamentos corretos esperados, sendo ponto de partida para outros novos cenários de comportamento.

3.4.2. Processo de simulação

As EFSMs especificadas em SDL constituem um modelo que contém as funcionalidades envolvidas no processo terminador e coordenador (correspondem respectivamente às classes *terminator* e *coordinator* do modelo de classes OMT). Basicamente, estes processos devem controlar o registro de recursos participantes na transação e a execução esperada do protocolo *commit* em duas fases.

Para realizar a simulação, este modelo precisa dos componentes externos com que deve interagir; por isso, foi necessário acrescentar 3 processos ao modelo inicial. Um processo cliente, que faz o papel de quem inicia e pede para fechar uma transação. Um segundo processo, denominado recurso, que simula um participante da transação. Após ser criado, o recurso é registrado pelo coordenador. Cada instância do processo recurso deve estar preparada para reagir aos pedidos do terminador, responsável por gerenciar o protocolo *commit* em duas fases. Finalmente, um terceiro processo, chamado fábrica de recursos, que cria as instâncias do processo recurso.

A estrutura hierárquica, com a notação SDL, do modelo de simulação é mostrada na Figura 3-14 e o diagrama de interconexão na Figura 3-15.

Na Figura 3-14, pode-se observar o sistema (ST) composto de um bloco (gerenciador_prot) e as declarações globais do sistema. O bloco gerenciador_prot é formado pelos 5 processos anteriormente definidos e as declarações a nível de bloco. Cada processo possui também as suas próprias declarações. As declarações incluem definições de tipos, variáveis, constantes, sinais, relógios, etc.

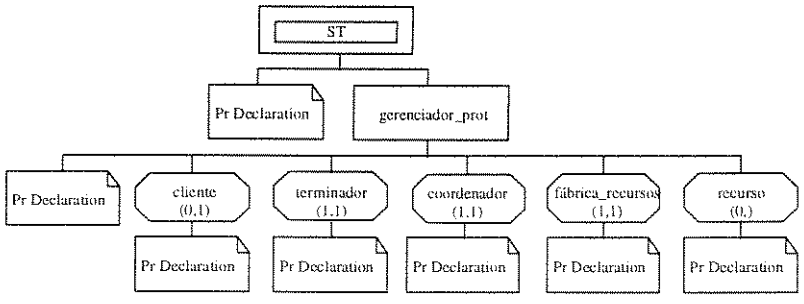


Figura 3-14: Estrutura hierárquica do modelo de simulação.

A Figura 3-15 mostra, com a notação em SDL, como os processos se comunicam e quais sinais chegam a cada processo pelas diferentes rotas. O processo fábrica de recursos não interage com os outros processos, ele somente cria instâncias do processo recurso.

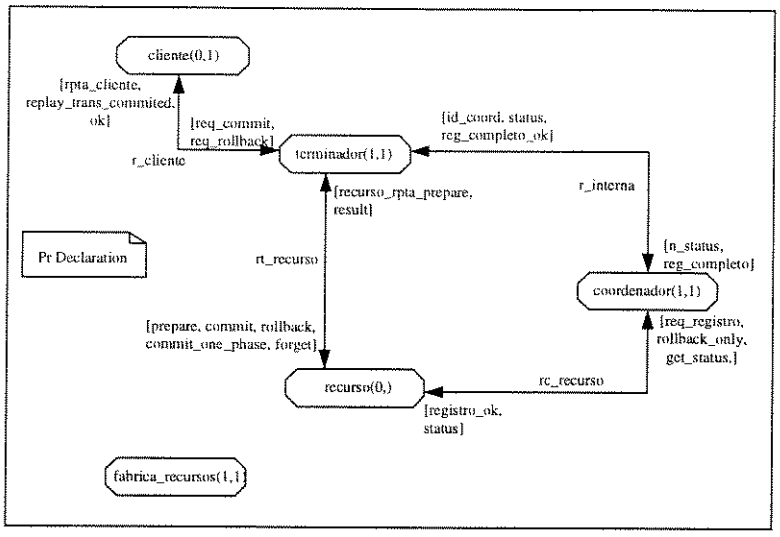


Figura 3-15: Diagrama de interconexão do modelo de simulação.

Durante a simulação foi testado o seguinte:

- O registro dos recursos, que é de responsabilidade do processo coordenador.
- A execução do protocolo *commit* em duas fases segundo os cenários estabelecidos, cujo gerente é o processo terminador.
- A atualização do estado da transação, a partir das decisões dos processos coordenador ou terminador.
- A reação dos recursos aos sinais do terminador segundo os cenários estabelecidos.

Foram controlados os seguintes aspectos:

- Existência de um sinal de requisição de *commit* do cliente sempre que existir pelo menos um recurso registrado.
- Requisições de registro de recursos que chegaram antes do pedido de *commit* do cliente devem ser atendidas primeiro. Requisições que chegaram depois do pedido de *commit* serão descartadas.

3.4.3. Casos de simulação

A simulação foi dividida em duas partes. A primeira considera a simulação e verificação do modelo, ignorando situações de exceção (especificadas no documento da OMG). Para isto foram criados sete cenários básicos de comportamento usando os MSCs. Um dos cenários é mostrado na Figura 3-16. O objetivo desta parte é corrigir o modelo testando-se a execução do protocolo *commit* em duas fases; mais especificamente, testando-se a execução do registro dos recursos e o fechamento da transação com *commit* ou *rollback*.

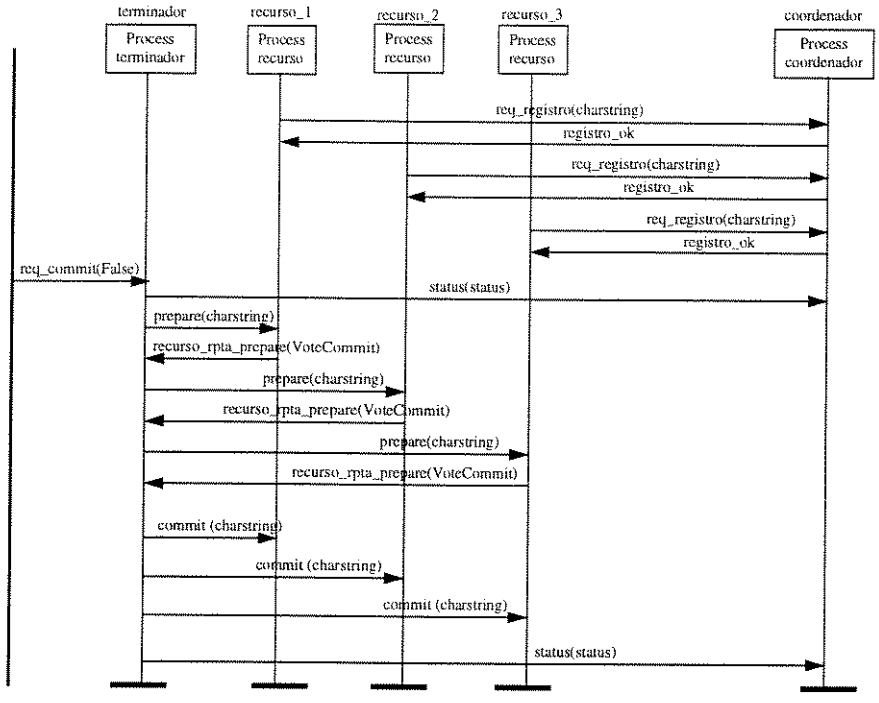


Figura 3-16: Cenário básico de simulação sem considerar exceções.

O cenário da Figura 3-16 mostra o registro de recursos no coordenador por iniciativa dos próprios recursos. Assume-se que é o recurso que pede para se registrar com o coordenador. Na implementação, esta tarefa é realizada pela fábrica de recursos, que, após criar um recurso, solicita o seu registro ao coordenador.

Também é mostrado na figura o pedido de *commit* feito pelo cliente com um parâmetro que indica se ele quer ou não um informe de exceções. Neste caso, o cliente não estará esperando informe de exceções.

O terminador inicia o protocolo com o envio do sinal *prepare* para todos os recursos registrados. Após a análise das respostas dos recursos, o terminador decide fechar a transação com *commit* enviando um sinal de *commit* para todos os recursos. Neste caso, o terminador não espera confirmação deste sinal. Pode-se observar também o pedido de mudanças de estado da transação.

A segunda parte da simulação considera a presença de exceções durante a execução do protocolo *commit* em duas fases. Para esta segunda parte também foram construídos onze cenários com MSCs. Um dos cenários é mostrado na Figura 3-17.

Esta abordagem considera a inclusão de um tipo de confirmação da execução das operações nos recursos (*commit*, *rollback*, commit_one_phase) através de sinais enviados para o processo terminador. Esta inclusão responde a uma necessidade dada pela própria especificação da OMG. Segundo a especificação, o cliente precisa saber se ocorreu alguma situação de exceção em algum recurso. A inclusão também foi feita por uma questão de modelagem. Quando um sinal *commit* ou *rollback* é enviado aos recursos, isto é, ao ambiente externo, o processo terminador não tem como saber se a operação no recurso foi executada ou não, precisando de uma resposta.

Quando ocorre uma exceção nos recursos, a resposta para o terminador será diferente de commit_ok ou rollback_ok. As exceções geram um informe de exceções ao cliente.

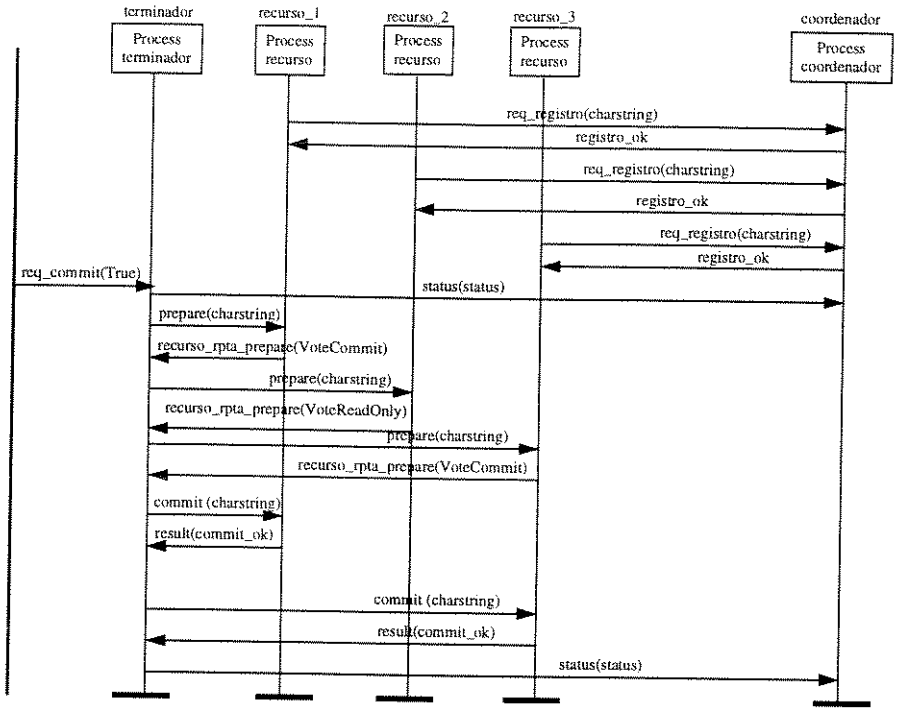


Figura 3-17: Cenário básico de simulação considerando exceções.

A Tabela 3-III contém os parâmetros considerados no primeiro caso de simulação que não considera o informe de exceções para o cliente.

Primeiro caso de simulação	
Número de recursos:	3 (testar para um recurso e para mais de um).
Existe informe de exceções para o cliente? (Parâmetro report_heuristic) :	FALSE.
Testes:	Registro de recursos. Pedido de *commit* e de *rollback* do cliente.
	Execução do protocolo: operações de envio dos sinais *prepare, commit, rollback.*
	Variações nas respostas dos recursos ao sinal *prepare.*

Tabela 3-III: Parâmetros considerados no primeiro caso de simulação.

A Tabela 3-IV contém os parâmetros considerados no segundo caso de simulação que considera o informe de exceções para o cliente.

Segundo caso de simulação	
Número de recursos:	3 (testar para um recurso e para mais de um).
Existe informe de exceções para o cliente? (Parâmetro report_heuristic) :	TRUE.
Testes:	Registro de recursos. pedido de *commit* e de *rollback* do cliente.
	Execução do protocolo: operações de envio dos sinais *prepare*, *commit*, *rollback*
	Variações nas respostas dos recursos ao sinal *prepare*.
	Controle do término de execução das operações dos recursos após o envio do sinal *commit* ou *rollback*, podendo gerar situações de exceção.

Tabela 3-IV: Parâmetros considerados no segundo caso de simulação.

3.4.4. Tipos de simulação utilizados

Os dois casos de simulação apresentados nas tabelas III e IV foram testados utilizando três tipos de simulação:

1) Simulação interativa, onde o usuário escolhe quais transições de estado disparar entre os diversos processos. Esta simulação foi usada para corrigir o modelo e verificar se o seu funcionamento corresponde ao comportamento expressado nos cenários básicos.

2) Simulação randômica, onde o simulador se encarrega de disparar as transições aleatoriamente. Esta simulação foi usada para detectar cenários não previstos inicialmente, resultando em modificações no modelo.

3) Simulação exaustiva, onde todas as transições disparáveis do modelo são executadas. Foi usada para obter informações globais da execução do modelo, gerar cenários automáticos com problemas de *deadlock* e corrigir o modelo. Esta forma de simulação é recomendável para avaliar modelos complexos quando não se pode ter um controle através da simulação interativa e randômica, quando a possibilidade de ocorrência de *deadlock* é grande ou quando é preciso de informações globais sobre o número de estados e transições simulados, duração da simulação, porcentagem de transições executadas e estados alcançados na simulação, transições nunca executadas, etc.

Em todos os tipos de simulação é possível obter informações sobre a cobertura dos estados alcançados e das transições disparadas durante a simulação. O detalhe destas informações é apresentado na seção 3.4.5.

3.4.5. Resultados da simulação

- A simulação do modelo permitiu testar todos os caminhos de execução existentes no modelo e observar o comportamento do sistema em diferentes cenários de execução.
- Foram gerados automaticamente arquivos com extensão .log contendo os cenários simulados.
- A simulação interativa e randômica reduz a presença de erros no modelo.
- Todos os tipos de simulação devolvem informações da cobertura de estados alcançados e de transições executadas.

A Tabela 3-V apresenta informações da porcentagem de estados alcançados e transições executadas em cada tipo de simulação. Para obter estes resultados foi considerado o primeiro cenário para o primeiro caso de simulação, mostrado na Figura 3-16.

Processos	Simulação interativa		Simulação randômica		Simulação exaustiva	
ST	%E	%T	%E	%T	%E	%T
cliente	100.00	40.00	100.00	40.00	100.00	40.00
terminador	62.50	46.67	62.50	46.67	75.00	53.33
coodenador	100.00	40.00	100.00	40.00	100.00	50.00
fabrica_recursos	100.00	66.67	100.00	66.67	100.00	66.67
recurso	100.00	50.00	100.00	50.00	100.00	62.50

Tabela 3-V: Cobertura de estados e transições na simulação. E significa estado e T significa transição.

Na Tabela 3-V, os resultados da simulação interativa e randômica se referem a uma única seqüência de execução do modelo, podendo-se obter resultados semelhantes testando outros cenários. A simulação exaustiva mostra os resultados em função de todas as possíveis seqüências de execução para o modelo.

Os resultados da Tabela 3-V podem ser melhor entendidos com o detalhe das tabelas 3-VI e 3-VII.

Processos ST	S. interativa	S. randômica	S. exaustiva
terminador :			
Estados alcançados:	62.50% 5	62.50% 5	75.00% 6
Estados não alcançados:	37.50% 3	37.50% 3	25.00% 2
Transições executadas:	46.67% 7	46.67% 7	53.33% 8
Transições não executadas:	53.33% 8	53.33% 8	46.66% 7
coordenador			
Estados alcançados:	100.00% 2	100.00% 2	100.00% 2
Estados não alcançados:	0.00% 0	0.00% 0	0.00% 0
Transições executadas:	40.00% 4	40.00% 4	50.00% 5
Transições não executadas:	60.00% 6	60.00% 6	50.00% 5

Tabela 3-VI: Detalhe das informações obtidas durante a simulação.

Na Tabela 3-VI pode-se ver que 62.50% dos estados alcançados do processo terminador se refere a 5 estados. Sendo que este processo possui 8 estados, 3 deles não foram alcançados. Uma interpretação semelhante pode ser feita para todos os tipos de simulação. O resultados para o processo cooordenador mostram que todos os estados foram alcançados com os três tipos de simulação.

Para o caso de cobertura de transições pode-se observar que o 46.67% de transições foram executadas, enquanto que 8 delas não foram executadas. A interpretação dos resultados do processo coordenador é a mesma que no caso anterior.

O fato de obter resultados menores de 100% é explicável devido à presença de estados alternativos (por exemplo estados que emanam de uma condição de teste) e transições alternativas nos processos.

É importante notar que os resultados da simulação exaustiva mudam em relação aos outros tipos de simulação devido a que se baseia em mais de uma seqüência de execução e portanto testa todos os caminhos possíveis de execução do modelo.

Na Tabela 3-VII encontram-se informações detalhadas da simulação do processo terminador. São mostrados o número de vezes que um estado do processo é alcançado e uma transição é executada.

Processo terminador do ST	S. interativa	S. randômica	S. exaustiva
terminador :			
Estados alcançados	62.50%	62.50%	75.00%
ativo:	1	1	288
espera:	2	2	396
wait_c:	3	3	54
espera_confirm:	3	3	192
decisão:	1	1	24
wait_r:	0	0	0
wait_ro:	0	0	0
wait_c_1f:	0	0	66
Transições executadas	46.67%	46.67%	53.33%
start:	1	1	288
from_ativo_input_id_coord:	1	1	270
from_espera_input_req_commit:	1	1	126
from_espera_input_status:	1	1	186
from_espera_input_req_rollback:	0	0	0
from_espera_input_reg_completo_ok:	0	0	0
from_wait_r_input_result:	0	0	0
from_wait_ro_input_result:	0	0	0
from_wait_c_1f_input_result:	0	0	33
from_espera_confirm_input_recurso_rpta_prepare:	3	3	96
from_espera_confirm_input_prep_timeout:	0	0	0
from_decisao_provided:	1	1	24
from_wait_c_input_result:	3	3	24
discard:	0	0	0
timeout_prep_timeout:	0	0	0

Tabela 3-VII: Detalhes de simulação do processo terminador.

Na tabela Tabela 3-VII pode-se observar claramente quais estados do processo nunca foram alcançados. Dependendo da aplicação este resultado pode ser favorável ou não. Quando se sabe que são estados alternativos então o resultado é o esperado para aquele cenário; caso contrário, o resultado mostra que há algum problema no modelo que deve ser revisado. A mesma situação acontece com os resultados das transições executadas.

Note-se que a simulação exaustiva mostra o número de vezes que os estados foram alcançados e as transições executadas, durante o teste de todos os caminhos possíveis de execução. Observe-se que ainda existem estados e transições que não foram testados. Isto se deve aos parâmetros do cenário que está-se simulando. No cenário da Figura 3-16 pode-se notar que as respostas dos recursos são a favor da

execução de *commit*, portanto, estados alcançados em situações de *rollback* não devem ser executados para este caso.

A simulação exaustiva provê outro tipo de informações que são explicitados a seguir:

⇒ Duração da simulação: 6 segundos. O teste de todos os estados e transições durou 6 segundos.

⇒ Número de *deadlocks*: 27. É o número de cenários detectados pelo simulador contendo *deadlocks*. Na verdade, os 27 *deadlocks* detectados não correspondem ao conceito de *deadlock* como encontrado na literatura. Todos os cenários retornados estão corretos, mas devido a uma indefinição no documento de especificação da OMG, o coordenador continua no estado ativo mesmo após o encerramento das suas atividades. Esse comportamento é considerado pelo simulador como *deadlock*.

O correto seria que o coordenador terminasse a sua execução e deixasse de existir, pois cada vez que uma transação é iniciada, é criado um novo terminador e um novo coordenador.

⇒ Taxa de cobertura de transições: 53.66 % (19 transições não cobertas). Significa que do total de transições existentes no modelo (41 transições), 53.66% (equivalente a 22 transições) foram executadas durante a simulação. 46.34% das transições do modelo nunca foram testadas. Esta porcentagem representa 19 transições.

⇒ Taxa de cobertura de estados: 85.71% (2 estados não cobertos). Isto significa que do total de estados no modelo (14 estados) 85.71% dos estados (equivalente a 12 estados) foram executados pelo menos uma vez durante a simulação. 2 estados nunca foram executados.

Estes dois últimos resultados variam de acordo com as condições dadas no início da simulação. A medida que os cenários vão mudando, outras transições vão sendo ativadas mudando os resultados.

3.4.6. Vantagens da simulação

- Permite forçar o ambiente externo a se comportar da forma desejada com o objetivo de simular diferentes situações. Isto significa simular um comportamento externo sobre o qual não se tem controle.
- É útil para acrescentar controles ao modelo.
- A partir dos diversos cenários simulados é possível completar e corrigir o modelo. Por exemplo, localizar transições incompletas ou com erros de interpretação.
- Permite rastrear no transcorrer do tempo os valores de variáveis, os sinais nas filas de entrada de cada processo e parâmetros definidos no modelo.

- Detectar problemas de *deadlock*. Os cenários das situações que os produzem são gerados, facilitando o a correção do comportamento do sistema.
- Permite visualizar em forma gráfica os estados atuais dos processos e as seqüências de execução simuladas, facilitando o acompanhamento das situações de comportamento correto e incorreto.
- Auxiliar no teste dos caminhos de execução do modelo. No modelo do ST todos os caminhos possíveis foram testados.
- Preparar o modelo para a geração do código com o mínimo de erro.
- Permite simular partes do modelo, acrescentando em cada parte algumas características adicionais.
- Ajudar na descoberta de cenários indesejáveis na execução do sistema.
- Ajudar a questionar o comportamento do modelo, isto é, fornecer informações necessárias para responder a perguntas como: é isso o que o sistema deve fazer?.
- Facilitar a análise de modelos relativamente grandes, onde pode-se usar a simulação exaustiva para obter cenários de comportamento automaticamente.

3.4.7. Restrições encontradas durante a simulação

- O ciclo de simulação (preparação e análise de cenários, avaliação dos resultados, etc.) é um processo demorado.
- Nem todas as construções do modelo podem ser exercitadas na simulação [20], por exemplo, a simulação de atrasos na transmissão de sinais não é possível pois o particionamento de um canal não é permitido pelo simulador.
- O relógio do simulador não funciona em tempo real, ou seja, o tempo consumido pelas tarefas dentro de uma transição não pode ser testado.
- A simulação foi feita em uma única máquina envolvendo apenas um processador, pois a ferramenta GEODE não permite a simulação de forma distribuída. Por exemplo, a comunicação entre máquinas diferentes não foi testada.
- A simulação utilizando procedimentos não permite visualizar o que acontece dentro dos procedimentos.
- A simulação interativa leva um tempo considerável e existe o problema da incerteza de que todos os caminhos de execução foram testados, sobretudo quando o modelo é complexo.
- Na simulação randômica existe a incerteza de que todos os tipos de comportamento do modelo foram explorados, mesmo depois de uma longa simulação. Este tipo de simulação se torna mais apropriado para modelos com maior número de estados, onde a simulação interativa se torna difícil de controlar e quando não se dispõe de muita memória.

- Modelos abertos (que interagem com o ambiente) não podem ser simulados exaustivamente. É preciso criar outros processos que simulem o comportamento desejado.

Terminada esta etapa de simulação, foi iniciada a codificação do sistema.

3.5. Codificação

Esta seção descreve as tarefas realizadas para a geração de código executável do ST. São apresentadas inicialmente as decisões tomadas para a codificação, tais como a plataforma utilizada, as linguagens de programação utilizadas, etc. Em seguida, é apresentada a arquitetura da implementação do sistema e, finalmente, as atividades realizadas nesta etapa.

3.5.1. Decisões iniciais na geração de código

- A implementação realizada preocupa-se em fornecer as funcionalidades básicas do protocolo *commit* em duas fases, portanto, não foi feita uma implementação total do ST. Mecanismos de recuperação, os quais não são claramente explicitados no documento de especificação, não foram contemplados.
- A aplicação executa em ambiente Orbix (implementação do padrão CORBA). Por este motivo, deve-se considerar que o ST deve manter as suas interfaces definidas em IDL.
- As classes envolvidas no protocolo são as seguintes: *terminator*, *coordinator* e *transaction*. Elas foram implementadas usando o gerador de aplicações da ferramenta GEODE, a partir da especificação em SDL. As classes que formam o ambiente da aplicação (*factory*, *control*, *resource*, *recoverycoordinator*) foram parcialmente implementadas usando Orbix e C++.

3.5.2. Arquitetura da Implementação

O sistema está composto por objetos Orbix, que possuem a sua interface descrita em IDL e por uma aplicação em SDL. A aplicação em SDL fornece as funcionalidades das classes *terminator* e *coordinator*.

Os objetos Orbix foram implementados em diferentes servidores: o objeto *factory* faz parte de um servidor; *control*, *terminator* e *coordinator* formam o segundo servidor; e *resource* e *factoryrec* formam um terceiro servidor.

Um objeto cliente (*client*) foi definido para interagir com o conjunto de classes que formam o sistema.

As classes *coordinator* e *terminator* cooperam na execução do protocolo. As funcionalidades destas classes foram implementadas a partir da especificação em SDL, ficando unicamente nelas a responsabilidade de receber requisições dos outros objetos Orbix e estabelecer a comunicação com a aplicação SDL.

A geração de código a partir da especificação em SDL foi feita utilizando a ferramenta GEODE. Considerando que esta ferramenta não possui uma interface para Orbix, o código gerado deverá se comunicar com um programa externo [21]. Este programa externo, chamado tarefa externa, age como uma ponte entre o sistema SDL e os objetos *coordinator* e *terminator,* que rodam em ambiente Orbix e cuja funcionalidade é específica para comunicação com a aplicação em SDL. Para estabelecer esta comunicação foi utilizado o mecanismo IPC (Inter Processes Comunication) do UNIX.

A Figura 3-18 ilustra a arquitetura da implementação do ST.

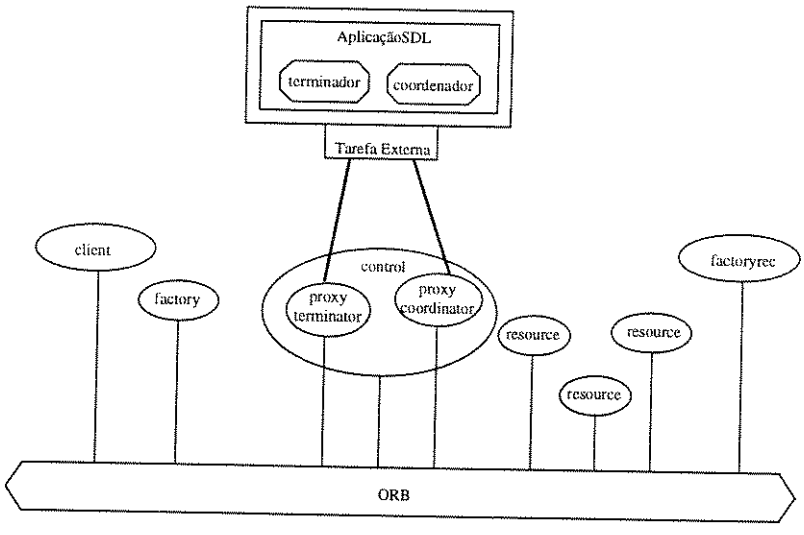


Figura 3-18: Arquitetura de implementação do ST.

A Figura 3-18 representa os objetos Orbix através de uma elipse. Estes objetos devem interagir entre si através do ORB, representado na parte inferior da figura, que é responsável pela comunicação entre os objetos. Na figura foram introduzidas as denominações de *proxy terminator* e *proxy coordinator,* que correspondem aos objetos *terminator* e *coordinator,* respectivamente. Um *proxy* é um elemento que recebe requisições de outros objetos e as envia para o responsável em fornecer o serviço requisitado. Dada a semelhança do comportamento dos objetos *terminator* e *coordinator* com este conceito foi adotada essa denominação. Isto foi feito com o objetivo de facilitar o entendimento e a visualização, evitando confusões entre os processos terminador e coordenador com os objetos *terminator* e *coordinator.*

Ainda na Figura 3-18, o retângulo com borda dupla representa a aplicação SDL com os seus processos coordenador e terminador. As linhas mais espessas representam a comunicação entre os processos SDL e os objetos Orbix.

A interação entre os vários elementos da figura é ilustrada nas figuras 3-19 e 3-20.

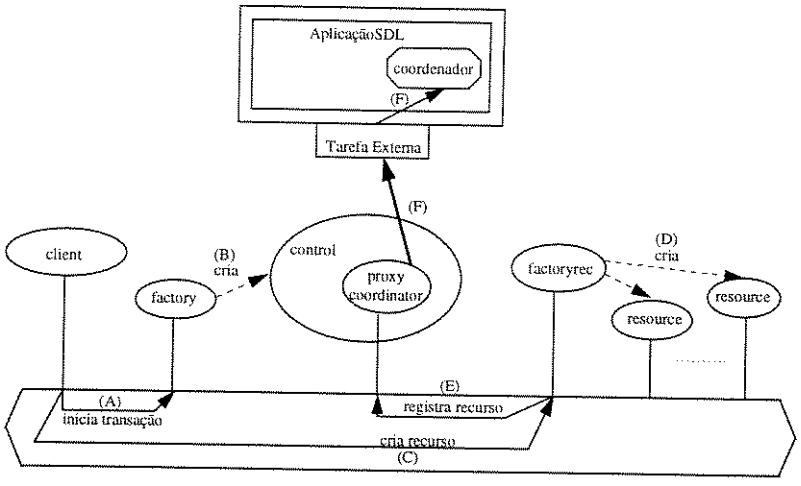


Figura 3-19: Interações para iniciar transação e registrar recursos participantes.

Na Figura 3-19 um cliente pede para *factory* iniciar uma transação (A). Para isto, *factory* cria um objeto *control* (B) e devolve esse objeto ao cliente. Por outro lado, o cliente solicita ao objeto *factoryrec* que crie objetos *resources* (C), passando para ele a identificação do objeto *control*. A *factoryrec*, após criar os recursos (D), deve pedir o registro deles para o *proxy coordinator*. A identificação do *proxy coordinator* é fornecida pelo objeto *control*. Com esta informação, o objeto *factoryrec* pede o registro dos recursos(E). O *proxy coordinator*, ao receber um pedido, envia uma mensagem para a aplicação SDL (F) pedindo para executar este serviço.

Quando o cliente decidir fechar a transação ele precisa solicitar este serviço ao *proxy terminator*, como mostrado na Figura 3-20.

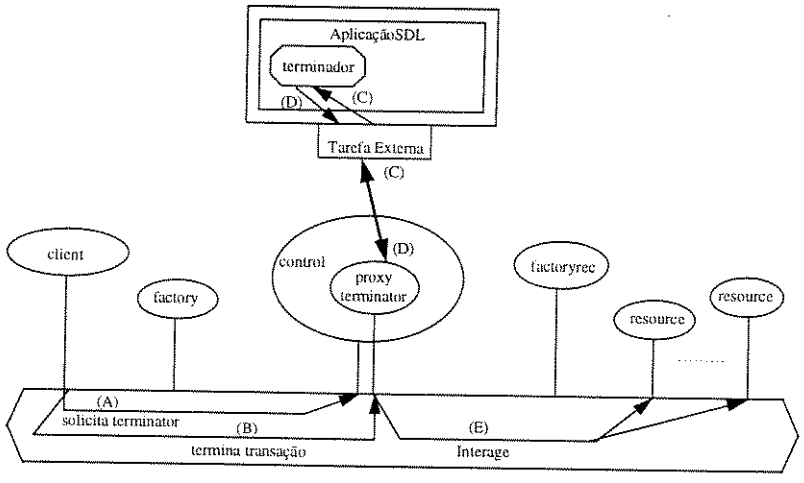


Figura 3-20: Interações para fechar a transação.

A Figura 3-20 mostra que para obter a identificação do *proxy terminator* o cliente pede para o objeto *control* fornecer esta informação (A). Com a identificação do *proxy terminator*, o cliente pode solicitar o término da transação. Este é o ponto de início do protocolo. Quando o *proxy terminator* recebe o pedido de término da transação solicitado pelo cliente (B), ele envia uma mensagem para a aplicação SDL

(C) pedindo executar este serviço. Na aplicação SDL o processo terminador interage com o processo coordenador para requisitar informações dos recursos registrados.

Quando o terminador (processo SDL) precisar interagir com os objetos *resource* (processos Orbix), ele deverá enviar os sinais correspondentes ao *proxy terminator* (D) e este, através do ORB, deverá interagir com os objetos *resource* (E).

3.5.3. Geração de código

3.5.3.1.Implementação do código em Orbix

As classes consideradas no ambiente da aplicação SDL foram implementadas utilizando Orbix. Estas classes são: *factory*, *control*, *resource*, *factoryrec*, *terminator* e *coordinator*. Do conjunto de classes, somente *coordinator* e *terminator* interagem com os processos coordenador e terminador, construídos em SDL.

As classes *terminator* e *coordinator* recebem as requisições de clientes Orbix ou das outras classes e estabelecem a comunicação com a aplicação em SDL.

As classes do ST foram agrupadas em servidores Orbix, os quais foram distribuídos em nós diferentes de uma rede. Para cada servidor foi construído um programa denominado programa servidor, que tem como função inicializar os objetos das classes relacionadas a ele.

Foi implementado também um programa cliente que interage com o conjunto de servidores através do ORB. O programa é responsável por iniciar uma transação, fazendo um pedido para a classe *factory*, e por terminar a transação, fazendo um pedido para a classe *terminator*. O cliente também pede à classe *factoryrec* para criar os objetos da classe *resource* que serão os participantes da transação.

Para executar a aplicação deve-se chamar o executável "ServTrans", o qual ativa o cliente e inicia o processamento do ST.

Observando o modelo de classes do ST da Figura 3-6, pode-se notar que as operações get_register_resource e changes_status não fazem parte da interface IDL. Estas operações foram criadas pela necessidade de comunicação entre o processo terminador e o processo coordenador em SDL. As operações são transparentes para a aplicação em Orbix.

3.5.3.2.Implementação da especificação em SDL

A geração automática de código da especificação em SDL, utilizando o gerador de aplicações da ferramenta GEODE[21], produz código executável na linguagem C. Basicamente a especificação em SDL contém as funcionalidades necessárias para gerenciar o protocolo *commit* em duas fases.

3.5.3.3.Comunicação entre as aplicações SDL e Orbix

Para realizar a comunicação entre as implementações SDL e Orbix, foi criada uma tarefa externa, que utiliza as bibliotecas da linguagem C e as bibliotecas específicas para as aplicações em SDL. O objetivo da tarefa externa é servir de interface entre a aplicação em Orbix e a aplicação em SDL, recebendo e retornando os sinais trocados entre eles.

O mecanismo usado pela tarefa externa para a troca de sinais com a aplicação em Orbix é o IPC (Inter Processes Communication) nativo do UNIX. Dentre os mecanismos disponíveis no IPC (*mailbox*, memória compartilhada e semáforos) o mecanismo de *mailbox* foi escolhido pela sua simplicidade.

4. Avaliação

Este capítulo apresenta inicialmente uma avaliação da modelagem estática, dinâmica e funcional no contexto de SDAs, descrevendo as necessidades de projeto e implementação de SDAs e o que os métodos utilizados oferecem para este objetivo. Em seguida, é apresentada uma avaliação das ferramentas utilizadas e, finalmente, algumas considerações gerais do processo de desenvolvimento.

4.1. Representação estática do sistema

O modelo estático de um SDA não possui características especiais que o diferencie de outros sistemas. Portanto, nenhum recurso adicional é necessário. As representações comumente usadas na modelagem estática são suficientes, faz-se necessário apenas a escolha de um bom modelo, de modo que este aspecto do sistema seja bem capturado e de fácil entendimento.

Algumas características da modelagem estática abordada por OMT e SDL são apresentadas a seguir.

4.1.1. O Método OMT

No caso de estudo deste trabalho (ST), o Modelo de Classes do método OMT se mostrou adequado para modelagem estática do ST. Uma de suas vantagens é a facilidade de ser empregado.

OMT é um método que se preocupa principalmente com a fase de análise, fase em que a definição do sistema é o principal objetivo. Um tempo bastante considerável do processo de desenvolvimento é dedicado a esta fase. Durante a modelagem estática são identificadas as partes que compõem o sistema em termos de classes, atributos, operações e relacionamentos. O modelo fornece uma visão em alto nível do sistema, facilitando o entendimento dos requisitos sem se preocupar com os detalhes de implementação.

A notação do Modelo de Classes do método OMT é muito rica para modelagem estática de sistemas, capturando este aspecto de forma simples e clara. Fornece uma abstração em alto nível do que é o sistema, o que deve fazer e como está dividido,

obtendo também uma representação em alto nível das fronteiras de cada classe e do sistema global.

Uma vantagem da modelagem estática OMT é permitir que o Modelo de Classes seja completado a medida que se obtém maior experiência e conhecimento do problema.

Por último, o fato da especificação feita pela OMG ser baseada no conceito de objetos, reforça a vantagem de se utilizar um modelo orientado a objetos para capturar, pelo menos, o aspecto estático do ST.

4.1.2. A técnica SDL

A notação em SDL não é muito rica para a representação estática de sistemas. Ela é mais recomendada para a parte de projeto de sistemas pois a sua notação é mais apropriada para representar como o sistema deve funcionar.

A notação da parte estática em SDL é uma hierarquia que divide o sistema em blocos e processos. Esta notação não permite identificar quais as entidades são necessárias no sistema, como elas se relacionam e o que devem fazer. Esta notação é mais útil quando o desenvolvedor já conhece as partes constituintes do sistema, e, portanto, deve-se concentrar os esforços em estruturar o sistema para a futura implementação.

A modelagem estática em SDL também foi útil, pois a sua estrutura hierárquica ajudou a decidir como estruturar o sistema para a implementação. Por exemplo, foi decidido que cada bloco representa uma máquina em uma rede. Na Figura 4-1, cada bloco representa um servidor. Os servidores poderão ou não ser distribuídos em diferentes máquinas.

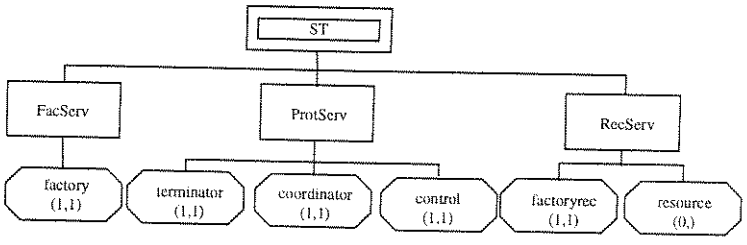


Figura 4-1: Representação hierárquica do ST em SDL.

4.2. Representação dinâmica do sistema

O modelo dinâmico é usado para descrever o comportamento de cada entidade do sistema. São criados cenários de comportamento, contendo situações para as quais o sistema deve estar preparado, ou seja, situações que ilustram como o sistema muda no transcorrer do tempo.

Um SDA tem características dinâmicas que o diferenciam dos sistemas convencionais. Ele possui restrições temporais, diversas formas de comunicação e alto nível de interação entre as suas partes. No caso do ST, a modelagem do tempo e

da comunicação é mais crítica, pois suas partes estão distribuídas e precisam se comunicar.

Baseado nestas características, a modelagem dinâmica em SDL se tornou mais apropriada devido aos recursos de sua notação.

Algumas características da modelagem dinâmica em OMT e SDL são apresentadas a seguir.

4.2.1. Método OMT

O Modelo Dinâmico em OMT representa as mudanças nos objetos no decorrer do tempo. O modelo descreve a seqüência de operações que ocorrem em resposta a um estímulo externo, sem considerar o que as operações fazem ou como elas são implementadas.

Apesar da notação do Modelo Dinâmico em OMT ser complexa, ela se preocupa mais em mostrar quais são os estados de um objeto e quais os eventos a fazem mudar para um outro estado. Não se preocupa em mostrar quem gerou os eventos.

Em geral, a complexidade da notação dificulta a construção do modelo dinâmico de sistemas como SDAs, sendo difícil identificar todas as situações possíveis de interação no sistema. Os modelos para estes sistemas tornam-se complexos e difíceis de serem implementados.

Nem sempre fica claro o que acontece em cada estado, podendo existir interpretações ambíguas para o mesmo modelo. Por exemplo, não é possível discriminar o tipo de evento que está envolvido, se é um sinal, se é uma mudança no valor de uma variável, etc.

Considerando que é difícil representar todos os comportamentos possíveis seguindo todas as combinações de eventos, é difícil responder a perguntas como "O que acontece se..." ou detectar *deadlocks* nas interações entre os objetos sem uma ferramenta que automatize estas atividades.

Não há uma representação clara para situações de tempo. Fica difícil representar, por exemplo, quanto tempo um objeto leva para realizar as tarefas embutidas no seus estados, representar que um objeto fica esperando um certo evento durante um tempo determinado, situações de *time out* ou como e com que freqüência os objetos interagem entre si.

O modelo não se preocupa em mostrar quem produz os eventos, qual a seqüência de acontecimentos que afeta o objeto após a ocorrência de um evento ou como o ambiente influencia no sistema. A implementação de SDA a partir deste modelo é bastante difícil.

Em resumo, as observações apresentadas formam um conjunto de razões que inviabilizam a modelagem dinâmica e implementação de SDAs usando o Modelo Dinâmico de OMT.

Com a finalidade de esclarecer como é feita a abordagem dinâmica no método OMT, foi construído um Modelo Dinâmico OMT para as classes *terminator* e *coordinator*.

O Modelo Dinâmico do processo terminador é apresentado na Figura 4-2 e um detalhe de um dos seus estados é ilustrado na Figura 4-3.

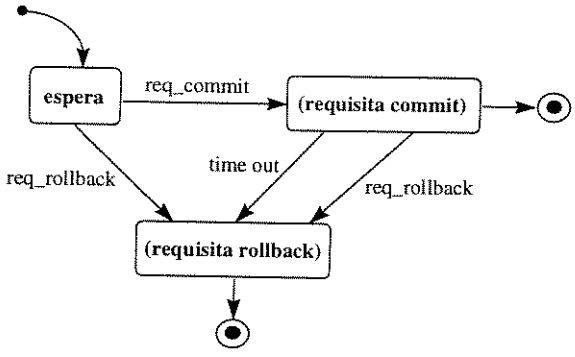


Figura 4-2: Modelo Dinâmico para a classe terminator.

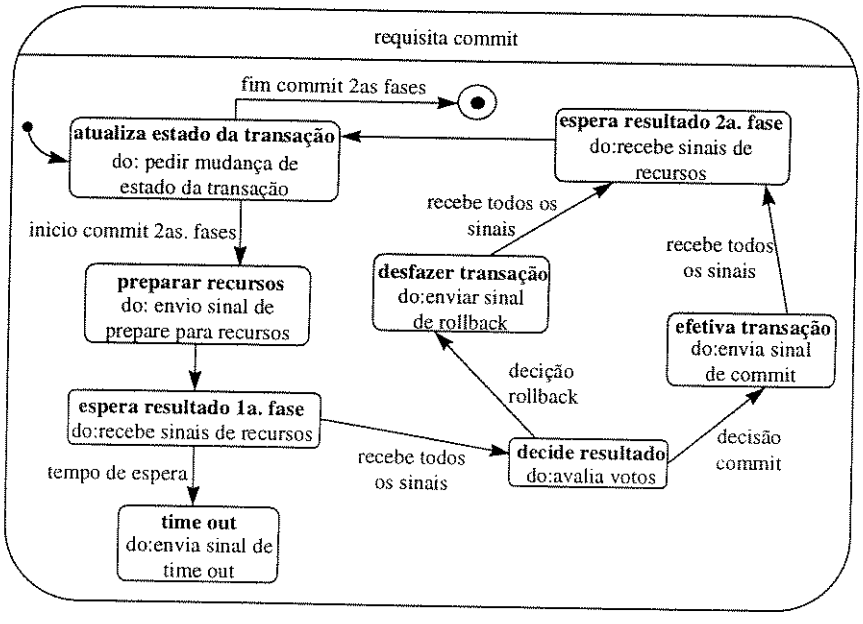


Figura 4-3:Detalhe do estado requisita commit da classe terminator.

A Figura 4-4 ilustra o Modelo Dinâmico da classe *coordinator*.

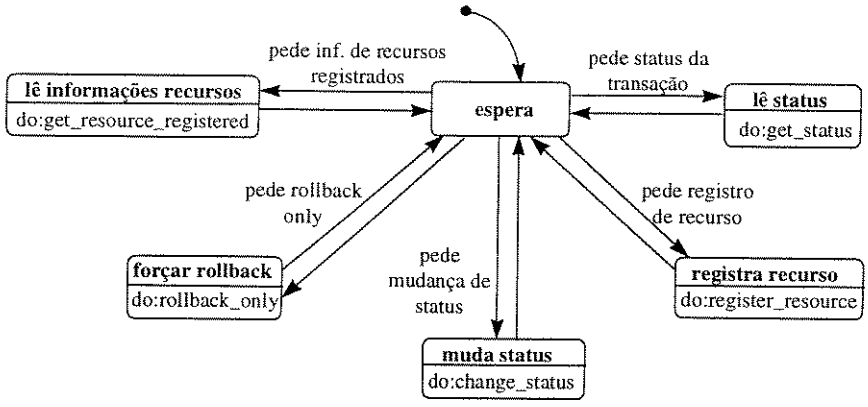


Figura 4-4: Modelo Dinâmico para a classe coordinator.

Os modelos obtidos representam abstratamente o comportamento das classes *coordinator* e *terminator*. Como pode ser observado, em ambos os casos os modelos dão uma visão geral de como cada classe se comporta quando ocorre um evento, mas não fica claro com que classes interagem. Um *time out* é considerado como qualquer outro evento, sendo difícil saber como e quando este evento ocorreu. Além disso, a implementação do modelo não é claramente explicitada por Rumbaugh et al. em [1].

Ao se tentar descer no nível de abstração o modelo se torna muito mais complexo. Por tanto se o objetivo é fornecer um alto nível de abstração este objetivo é garantido.

Algumas perguntas são difíceis de serem respondidas com o Modelo Dinâmico OMT, por exemplo, quanto tempo demorou o envio de um sinal entre os objetos? Como modelar *broadcasting*, ou comunicação via *mailbox* ou memória compartilhada? Como modelar troca de mensagens entre os objetos?. Estas perguntas não são claramente respondidas.

Por outro lado, a existência de níveis hierárquicos entre estados ajuda a visualizar o comportamento sob diferentes níveis de complexidade/abstração.

De um modo geral, a notação do Modelo Dinâmico OMT, como proposta em [1], se mostra bastante satisfatória, entretanto, para modelar SDAs, nota-se que muitas questões essenciais ficam pendentes neste modelo tal como o fator tempo.

4.2.2. Técnica SDL

SDL é mais apropriada para modelar sistemas concorrentes, onde os fatores de tempo e de comunicação são críticos. Possui formas particulares de representar estes aspectos, ficando mais clara a representação das interações entre os processos.

O modelo dinâmico em SDL tem um nível de abstração mais próximo da implementação, podendo ser mais difícil de ser entendido por quem não está familiarizado com o sistema e com a notação.

O modelo dinâmico em SDL aborda as mudanças nos processos no tempo. Descreve a seqüência de operações que ocorrem em resposta a um estímulo externo, considerando o que elas fazem e proporcionando uma descrição de alto nível de como elas podem ser implementadas.

O formalismo de sua sintaxe permite a simulação e geração de código a partir da especificação. Através das simulações é possível detectar a presença de *deadlocks* e fazer perguntas como "O que acontece se...".

A construção do modelo é simples, porém passa a ficar crítico quando o número de estados e eventos aumentam.

O modelo baseado em SDL dá uma sugestão de como pode ser realizada a implementação. Isto não impede que possam ser usadas outras formas de implementação.

4.3. Representação funcional do sistema

O modelo funcional tem por objetivo descrever a transformação de dados dentro do sistema.

A representação funcional de SDAs não difere dos sistemas convencionais, a diferença ocorre na fase de codificação, onde é necessário uma construção mais apropriada.

As modelagens dinâmica e funcional geralmente estão fortemente relacionadas, sendo que esta última é o detalhamento daquilo que uma classe ou um processo deve fazer em determinado estado.

Algumas características da modelagem funcional com OMT e SDL são mostradas a seguir.

4.3.1. Método OMT

Em OMT o Modelo Funcional é representado através de Diagramas de Fluxo de Dados (DFDs). Os DFDs mostram a transformação dos dados, desde a entrada no sistema até a geração da saída. Este modelo é utilizado para explicitar o que acontece durante as mudanças de estado, definidas no Modelo Dinâmico, ou para detalhar a funcionalidade dos métodos das classes do Modelo de Classes. Este modelo não se preocupa em como ou quando as transformações são executadas, qual é a seqüência de execução dos processos.

Neste trabalho, OMT não foi utilizado na modelagem funcional. Considerou-se mais natural construir o modelo funcional usando SDL.

4.3.2. Técnica SDL

Em SDL, a representação funcional é simples e está embutida na própria especificação das EFSMs.

Diferentemente de OMT, SDL modela as transformações dentro das próprias transições de estado do sistema, preocupando-se em detalhar quando as transformações são executadas e qual é a seqüência de execução dos processos.

Para o caso de SDAs, a representação em SDL é adequada pois em sua notação há construções comuns a este tipo de sistemas, tais como sinais, troca de mensagens, tempo, *time out* e *deadlock*.

Uma vantagem é a existência de um gerador de aplicações dentro da ferramenta GEODE que implementa automaticamente as EFSMs contendo as funcionalidades do sistema. Esta facilidade não foi essencial, mas a tradução das construções do modelo em SDL para uma linguagem de programação demandaria um esforço considerável.

4.4. Avaliação das ferramentas usadas

4.4.1. LOV/OMT

O uso desta ferramenta foi importante na edição do modelo e na elaboração do dicionário de dados descrevendo cada elemento constituinte do modelo (classes, métodos, atributos, associações). Isto foi fundamental dado que o modelo sofre várias mudanças à medida que se tem mais conhecimento do sistema a ser desenvolvido facilitando a manutenção, a alteração, a verificação de consistência e a documentação dos modelos.

Os resultados gerados pela ferramenta constituem uma parte importante da documentação do sistema.

Como aspecto negativo, a ferramenta possui recursos limitados para definições de tamanhos das figuras e tipos de letras. Possui limitações na impressão dos modelos, por exemplo, não é possível alterar o tamanho das figuras do modelo. Os formatos das figuras não podem ser alterados por um editor de texto. Isto é necessário quando se precisa incluir a figura dentro de um documento.

4.4.2. GEODE

O uso desta ferramenta foi importante pelo suporte da notação SDL gráfica (SDL_GR) e textual (SDL_PR), bem como para simulação e geração de código.

Foi muito útil na edição e manutenção do modelo, que muda à medida que se tem maior conhecimento do problema.

Ajudou na correção da sintaxe, preparando o modelo para a simulação e geração de código.

Ajudou na verificação do comportamento do sistema através dos recursos de simulação, permitindo a correção de erros principalmente de comportamento, de *deadlock* e recepção de sinais não especificadas.

O simulador ajudou na validação do comportamento do modelo em relação ao comportamento especificado nos cenários (MSCs).

O gerador de aplicações permitiu obter 100% de código, a partir da especificação em SDL.

Uma das vantagens em utilizar uma ferramenta como GEODE é a melhoria da qualidade do projeto do sistema, já que se obtém um modelo mais consistente e com menos erros.

A ferramenta favoreceu a documentação de diversos aspectos do sistema, tais como a estrutura hierárquica, as EFSMs, os cenários e a descrição de cada elemento declarado na especificação.

A ferramenta apresenta alguns problemas, entre eles os recursos limitados para impressão da especificação, detectados também na ferramenta LOV/OMT. Existem alguns problemas durante a edição que fazem o sistema sair sem guardar as alterações. Isto ocorreu enquanto se realizou trabalhos com muitas janelas abertas.

Em caso de erros os manuais não possuem informações completas.

A ferramenta não tem interface para IDL, o que impede a geração de código para sistemas compatíveis com a especificação CORBA. Isto obriga o usuário a construir uma tarefa externa que consiga conectar o código gerado ao *broker*.

4.5. Considerações gerais

4.5.1. Tempo de desenvolvimento

Inicialmente, foram realizadas atividades que são independentes do uso de ferramentas, tais como o estudo do Serviço de Transações, a interpretação dos conceitos envolvidos, a identificação das classes, as suas funcionalidades, o funcionamento dos seus métodos, a identificação dos atributos e os relacionamentos entre elas. Em relação a SDL, foram feitas atividades como a identificação de processos, a definição dos cenários básicos a serem testados, a construção da tarefa externa que serve de interface entre os objetos Orbix e SDL. Em relação ao Orbix, foi realizada a definição do código das classes a ser implementado.

As ferramentas ajudaram na diminuição do tempo de desenvolvimento, auxiliando a manutenção de consistência entre os diferentes níveis de especificação. Em OMT isto é caracterizado pela presença de diferentes visões do sistema, as quais são atualizadas automaticamente a cada modificação do modelo. Em relação a SDL, a ferramenta GEODE deu suporte às duas formas de representação, gerando automaticamente a especificação textual a partir da especificação gráfica. A ferramenta ajudou também na verificação da sintaxe do modelo. Isto foi importante porque prepara o modelo para as fases de simulação e geração de código. Em relação ao Orbix, ele gerencia a comunicação entre objetos locais e remotos, providenciando

a transparência na comunicação entre os objetos e diminuindo o tempo de implementação de mecanismos de comunicação entre objetos remotos.

4.5.2. Documentação

As ferramentas de modelagem ajudaram na edição dos modelos e na geração da documentação a partir das definições no dicionário de dados e nas definições em SDL.

A facilidade de atualização e modificação dos modelos ajuda a manter a documentação sempre atualizada.

A documentação fornecida pela ferramenta LOV/OMT é composta de diferentes visões do modelo de classes e pelo dicionário de dados.

A documentação fornecida pela ferramenta GEODE é composta de modelos nas representações textual e gráfica, pelos cenários e pelos diferentes resultados da simulação.

Orbix não dá suporte à documentação.

4.5.3. Combinação de métodos

A deficiência de um método é compensada pelo outro. O uso de OMT com SDL é uma alternativa para criar modelos de sistemas com características como: concorrência, restrições temporais e comunicação.

A combinação destes métodos neste trabalho buscou aproveitar o melhor de cada um deles para o contexto de SDAs. É importante comentar que pela não disponibilidade de uma ferramenta, a transformação das classes OMT para SDL foi realizada manualmente.

4.5.4. Comentários

Os comentários apresentados a seguir descrevem de forma sucinta como OMT e SDL se comportaram no desenvolvimento de SDAs e algumas das suas limitações.

- OMT se comportou como o esperado para a modelagem estática. SDL não é recomendável para este objetivo.
- OMT é complexo e incompleto na sua notação para a parte dinâmica, sendo de difícil implementação.
- SDL, embora possua muitos detalhes na notação, possui construção simples e mais próxima de conceitos de tempo real, comunicação local e remota, troca de sinais, etc. Provê também uma alternativa para a implementação.
- Nem todas as características dos SDAs podem ser modeladas usando-se a combinação de OMT e SDL. Por exemplo, não há como modelar fluxos contínuos de dados dos sistemas multimídia, características de qualidade de serviço, transparência, distribuição, etc.

Não foi objetivo deste trabalho mapear todas as construções do modelo de classes OMT para construções SDL, mas, se o modelo de classes tivesse construções mais complexas como: herança, agregação, associações modeladas como classes, etc., o mapeamento seria mais complexo, precisando de regras bem definidas. Uma outra dificuldade neste ponto seria o fato da ferramenta GEODE não suportar o padrão SDL orientado a objetos [7], portanto, podem existir construções de OMT não suportadas pela ferramenta. Uma alternativa para resolver este problema poderia ser a obtenção de uma outra ferramenta que suporte a versão orientada a objetos de SDL (não disponível no momento), podendo-se ampliar a complexidade do modelo com o uso de herança e agregação, reutilização, polimorfismo, etc.

Considerando a existência de SDL orientado a objetos e de ferramentas que suportam tanto OMT quanto SDL, seria mais fácil o trabalho de mapeamento de OMT para SDL. Para trabalhos futuros é importante verificar se estas novas ferramentas suportam todas as construções de um Modelo de Classes OMT.

A nova versão da ferramenta GEODE suporta OMT e SDL orientados a objetos. Segundo uma revisão feita das suas características é proposta inicialmente a utilização do Modelo de Classes OMT, seguido da definição de cenários com MSC e a construção das EFSMs. Como pode ser notado, esta nova versão propõe um tipo de desenvolvimento similar ao adotado neste trabalho. Sendo que não tem sido verificados os critérios de mapeamento do modelo de classes para SDL, este aspecto pode ser assunto para ser pesquisado em trabalhos futuros.

Para o contexto de SDAs, pode ser importante revisar a nova versão da ferramenta. Segundo a sua descrição, ela permitiria a geração de código de forma distribuída utilizando o protocolo TCP/IP na comunicação.

5. Conclusão

5.1. Introdução

Como considerações finais ao presente trabalho, este capítulo apresenta inicialmente os resultados obtidos. Em seguida, são explicitadas as dificuldades enfrentadas durante o desenvolvimento, bem como algumas recomendações. Finalmente, é apresentada uma análise crítica do trabalho e sugestões que possam dar continuidade ao trabalho realizado.

5.2. Resultado

A necessidade de formalizar e documentar o processo de desenvolvimento de SDAs levou à utilização de métodos, técnicas e ferramentas resultando em um processo mais organizado. Este processo permitiu representar o sistema, esclarecer dúvidas sobre o sistema, documentá-lo melhor e facilitar a sua implementação, obtendo com isto um sistema consistente com a sua especificação.

O processo partiu da especificação do sistema chegando até a codificação, portanto todas as fases do processo de desenvolvimento puderam ser consideradas no estudo.

A seguir são explicitados resultados específicos obtidos com os métodos e as ferramentas usados.

Combinação de métodos

No trabalho foram aplicados o Modelo de Classes do método OMT e a técnica de especificação formal SDL. Os resultados obtidos são apresentados a seguir.

OMT

1. O Modelo de Classes OMT, pela sua abordagem orientada a objetos, induz a modelagem de sistemas mais modulares, pois cada classe ou conjunto de classes pode ser interpretada como um módulo com fins específicos. As conseqüências da modularidade são bastante conhecidas na área de Engenharia de Software: facilita

a localização de erros do sistema e diminui o impacto da propagação dos efeitos de uma alteração, pois apenas as partes dentro do domínio de um módulo (classe) são afetadas. Os benefícios são tanto para o processo de desenvolvimento quanto para a manutenção do sistema.

2. A notação do Modelo de Classes é bastante completa e expressiva. É considerada completa pois possui construções que atendem às necessidades de modelagem do aspecto estático dos SDAs. É considerada expressiva pois a sua representação é de fácil entendimento.
3. O Modelo de Classes e o dicionário de dados gerados na modelagem constituem uma parte importante da documentação do sistema. A conseqüência de uma boa documentação reflete positivamente no trabalho de manutenção do sistema.

SDL

1. A notação de SDL é bastante completa pois é capaz de capturar características comuns aos SDAs, tais como tempo, concorrência e comunicação. Sendo esta representação uma forma de abstração, constitui um recurso valioso para abordar a complexidade dos aspectos dinâmico e funcional, inerente aos SDAs.
2. A especificação em SDL nas suas duas representações (SDL-GR, SDL-PR) e os cenários da notação MSC constituem um documento importante dos aspectos dinâmico e funcional do sistema. Este documento juntamente com o Modelo de Classes OMT e o dicionário de dados fornecem informações muito úteis à manutenção do sistema.

Uso das ferramentas CASE

LOV/OMT:

Esta ferramenta favoreceu a edição e alteração do Modelo de Classes e do dicionário de dados, descrevendo cada elemento do modelo. Foi útil também na impressão do modelo e do dicionário de dados, os quais fazem parte da documentação do sistema. Estas facilidades diminuem significativamente o esforço de manipulação dos diagramas do Modelo de Classes.

GEODE:

- Esta ferramenta facilitou o trabalho de edição e modificação do modelo construído em SDL facilitando a manipulação dos diagramas e das definições durante o processo de desenvolvimento. Foi útil também na edição e alteração dos cenários construídos com a notação MSC pois facilita a sua manipulação.
- Durante a edição do modelo SDL, erros de sintaxe são detectados pela ferramenta, contribuindo para a sua consistência.
- A facilidade de impressão foi importante pois possibilita a inclusão dos cenários e do modelo SDL como parte da documentação do sistema.

- O mapeamento dos modelos para código não é uma tarefa simples e normalmente demanda um tempo razoável. Este mapeamento foi beneficiado com a geração automática de código do modelo SDL. Esta facilidade da ferramenta GEODE contribui a diminuir o tempo envolvido na codificação do sistema.
- A ferramenta foi de grande valor para o trabalho de simulação. A simulação permitiu verificar a interação do sistema com o seu ambiente externo, verificar se o comportamento interno do sistema estava de acordo com o esperado e validar este comportamento com o que foi especificado. Com isto, obtém-se um nível maior de correção do sistema.
- Algumas limitações presentes na versão disponível, representaram restrições ao trabalho. Por exemplo, a impossibilidade de geração de código de forma distribuída impediu à aplicação SDL rodar em vários nós da rede, limitando-a a rodar em uma única máquina. O relógio do simulador não é atualizado em tempo real, impossibilitando o teste de tempos transcorridos em uma transição de estado, ou atrasos na comunicação. A ferramenta não gera código para Orbix, tendo que ser implementada manualmente a comunicação entre ambas.

Orbix:

- Orbix mostrou-se favorável na implementação de objetos distribuídos pelo modo transparente com que trata a comunicação e pela facilidade de descrever as interfaces dos objetos.

5.3. Problemas enfrentados

- Foi difícil coletar referências bibliográficas sobre SDL, pois não existem muitas publicações que tratem sobre esse assunto. Isto impôs limitações ao estudo desta técnica.
- O documento de especificação da OMG é considerado de difícil interpretação não sendo muito claro para descrever os elementos envolvidos no ST.
- A diversidade de tópicos abordados tais como especificação formal, modelagem orientada a objetos, Sistemas Distribuídos Abertos e arquitetura CORBA demandou um grande esforço no entendimento dos vários conceitos necessários ao desenvolvimento do trabalho.
- Durante o uso das ferramentas alguns fatores dificultaram o andamento dos trabalhos, tais como erros de configuração, *bugs*, instalação incompleta, etc..

5.4. Recomendações

- Da forma como foi modelado o ST é difícil manter o controle do fluxo de execução. Para resolver este problema, recomenda-se estruturar a hierarquia do modelo SDL utilizando procedimentos. Com os procedimentos pode-se agrupar

fluxos com alguma lógica específica. Uma vez definidos estes grupos, os procedimentos podem ser reutilizados em diferentes pontos de uma EFSM através de uma chamada de procedimento. Isto favorece o controle das seqüências de execução e a visualização do modelo pois, ao ocultar detalhes sobre algumas seqüências de execução o modelo fica mais claro e organizado.

- Durante o processo de simulação é recomendável a utilização dos cenários da notação MSC como observadores do modelo, obtendo com isto a verificação automática do modelo em relação ao cenário especificado. Isto é especialmente importante em SDAs que possuem comportamentos críticos e que precisam ser testados e validados de forma mais rigorosa.
- As referências bibliográficas [3, 4, 7, 8, 16, 17] referentes a SDL são um ponto de partida para quem estiver interessado neste assunto. Também é interessante consultar os endereços URL na Internet: http://www.nr.no/presentations/SDL e http://www.tdr.dk/public/SDL/SDL.html .

5.5. Análise crítica

- Como observações ao presente trabalho, devem ser mencionados que é possível que alguns elementos do documento de especificação proposto pela OMG[10] tenham sido interpretados de forma diferente. Possivelmente algumas destas diferenças ocorreram por causa da existência de pontos que não estão claros no documento de especificação.
- As EFSMs dos processos coordenador e terminador não foram organizadas de forma estruturada, como conseqüência são encontradas seqüências de tarefas redundantes e desvios no fluxo de controle que afetam o domínio sobre a seqüência de execução. Isto pode ser melhorado com a utilização de procedimentos (ver seção 5.4).
- No trabalho foi proposta uma estratégia de desenvolvimento para SDAs. Esta estratégia é válida para a maioria dos casos. Apesar disso, existem algumas restrições para casos de Sistemas Distribuídos Multimídia, onde tanto os métodos quanto as ferramentas não capturam adequadamente características como: mídias contínuas (áudio, vídeo e animação), relacionamento entre as mídias (por exemplo sincronização de voz e imagem de um vídeo) e requisitos de qualidade de serviço (QoS) como: "Se um vídeo for exibido com um baixo número de quadros por segundo, pode-se perder a sensação de movimento contínuo que causa uma perda da qualidade do sistema".

5.6. Atividades futuras

- Integração de OMT e SDL com CORBA. As ferramentas CASE comerciais certamente providenciarão esta funcionalidade.
- Incorporar restrições de qualidade de serviço no processo de desenvolvimento de Sistemas Distribuídos Abertos.
- Explorar temas como tolerância a falhas, teste e qualidade em ambientes distribuídos.
- Explorar e incorporar os modelos de qualidade como o CMM [23] (Capability Maturity Model) no processo de desenvolvimento de SDAs.

Bibliografia

[1] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F., LORENSEN W. **Object-Oriented Modeling and Design**. New York: Prentice Hall International, 1991. 500 p.

[2] PRESSMAN R.S. **Software Engineering. A practitioner's approach**. Third Edition. Singapore: Mc Graw Hill International, 1992. 793 p.

[3] SARACCO, R., SMITH J.R.W., REED, R. **Telecomunications System Engineering using SDL**. The Netherlands: North-Holland, 1989. 631 p.

[4] BELINA, F., HOGREFE, D., SARMA, A. **SDL with Applications from Protocol Specification**. Great Britain: Prentice Hall International, 1991.

274 p.

[5] ORFALI, R., HARKEY, D. **Client/Server survival guide with OS/2**. Van Nostrand Reinhold, 1994. Cap. 33: Object Request Brokers. p. 689-722. Cap. 34: Distributed Object Services. p. 723-744.

[6] DATE, C.J. **An Introduction to Database Systems**. The Systems Programming Series. Volume II. Ed. Addison. Wesley Publishing Company, Inc., July 1985. 383 p.

[7] ITU-T. **Specification and Description Language**. Recommendation Z.100. Switzerland, 1993.

[8] ITU-T. **Gráfico de Secuencias de Mensajes. Critérios para la utilización y aplicabilidad de técnicas de descripción formal**. Recomendation Z.120. Switzerland, 1994.

[9] OMG. **The Common Object Request Broker: Architecture and Specification**. Revision 2.0, July 1995.

[10] OMG. **CORBAservices: Common Object Services**. March 1995.

[11] TUTUMI, R. **Plataforma para o Desenvolvimento de Ferramentas Baseadas em Diagramas**. Campinas: IMECC, UNICAMP, 1996. Tese (Mestrado) - Instituto de Matemática Estatística e Ciências da Computação, Universidade Estadual de Campinas, 1996. p. 27-45.

[12] ARAÚJO, D. E. **Serviços de Gerenciamento ODP Utilizando a Arquitetura CORBA**. Campinas: FEEC, UNICAMP, 1996. Tese (Mestrado) - Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, 1996. p. 21-28.

[13] MARK, B. OMG's CORBA. Dr. Dobb's Special Report, Winter 1994-95.

p. 8-12.

[14] GUO F., MACKENZIE, T.W. Traslation of OMT to SDL-92. In: Seventh SDL Forum 25-29 September, 1995, Osb, Norway. http://www.nr.no/presentations/SDL/SDL.abs.html.

[15] WITASZEK, D., HOLZ, E., WASOWSKI, M., LAU, S., FISCHER J. A Development Method for SDL-92 Specifications Based on OMT. In: Seventh SDL Forum, 25-29 September, 1995, Osb, Norway. http://www.nr.no/presentations/SDL/SDL.abs.html.

[16] VERILOG. **LOV/OMT Editor - Reference Manual**. version 1.1, 1994.

[17] VERILOG. **GEODE, An Introduction to SDL and MSC**. October 1993.

[18] VERILOG. **GEODE Technical Presentation**. July 1994.

[19] VERILOG. **GEODE Editor - Reference Manual**. Version 2.2, October 1993.

[20] VERILOG. **GEODE Simulator - Reference Manual**. Version 2.1, October 1993.

[21] VERILOG. **GEODE Application Generators - Reference Manual**. Version 2.2, January 1995.

[22] IONA Technologies Ltd. **Orbix distributed object technology, Programmer's Guide**. Release 1.3.1, February 1995.

[23] PAULK, M., et. al. **Capability Maturity Model**. Version 1.1. Software Engineering Institute, Carnegie Mellon University. IEEE, July 1993.

Anexo 1 : Aplicações das construções SDL para Comunicação

Este anexo visa detalhar o modelamento da comunicação entre processos SDL.

SDL define os seguintes tipos de comunicação:

- Informação revelada
- Envio de informação a um receptor
- Broadcasting de informação
- Acesso de informação a partir de uma requisição
- Repositório comum de informação

1. Informação revelada

Nesta comunicação um processo deixa disponível algumas informações que podem ser acessadas por outros processos a critério destes. Uma analogia disso é o *display* de vídeo: qualquer um olhando ao *display* pode ler a informação apresentada.

Figura A- 1: Informação revelada.

Notação:

Processo dono da informação (Calendário)	Processo usuário da informação
DCL REVEALED Tipo_de_dia char;	VIEWED Tipo_dia char; DCL Tipo_dia char; ..... TASK Tipo_dia:=VIEW(Tipo_de_dia, Calendário);

O proprietário deve declarar a variável compartilhada como "revelada", permitindo assim, o seu acesso. Os usuários precisam declarar a variável que manterá o valor da variável compartilhada após o acesso (o acesso se dá executando a operação VIEW).

2. Envio de informação a um receptor

Este tipo de comunicação consiste em enviar uma informação a um receptor particular, cuja identidade é conhecida pelo emissor. O envio é completamente dependente da decisão do emissor.

Uma analogia deste tipo de comunicação é uma ligação telefônica: Se alguém quiser se comunicar por telefone tem que conhecer o número (endereço) e ligará quando decidir.

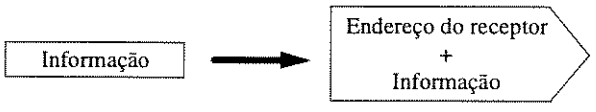


Figura A- 2: Envio de informação a um receptor.

Notação:

OUTPUT mensagem TO processo_receptor VIA canal_1

O envio de um sinal em SDL é apropriado para modelar este tipo de comunicação. "TO" permite associar o endereço do processo destino. "VIA" pode ser usado para restringir os receptores possíveis.

3. Broadcasting de informação

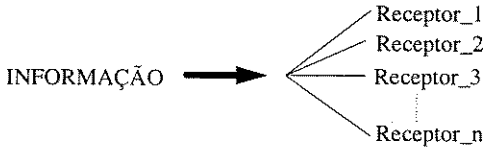


Figura A- 3: Broadcasting de informação.

Neste tipo de comunicação a informação é enviada a todos os receptores em potencial. A comunicação ocorre por decisão do emissor e não lhe é possível limitar o envio para um determinado grupo de receptores.

Notação :

Proc_dono da informação	Receptores
DCL EXPORTED Informação Info; DCL EXPORTED Broadcast Boolean :=False; ..... EXPORT Informação; TASK Broadcast:= True; EXPORT Broadcast;	DCL var_local Info; IMPORTED Informação Info; IMPORTED Broadcast Boolean; ..... STATE Espera; PROVIDED IMPORT (Broadcast, Pro_dono); TASK var_local:=IMPORT(Informação,Pro_dono); .....

A notação mostra que o processo emissor (Proc_dono) deve executar um *broadcasting* da "Informação". Para isto, primeiro é declarada a variável "Informação" e o *flag* "Broadcast" como exportáveis. Durante a transição, o processo exporta a informação, muda o valor de Broadcast e exporta este *flag*.

Os processos receptores têm declarados as mesmas variáveis como importáveis, sendo ativados quando a condição expressada com a palavra PROVIDED se tornar verdadeira ou seja, quando o *flag* Broadcast for TRUE. Após isto, uma variável local toma o valor da variável "Informação", através da importação.

4. Acesso de informação a partir de uma requisição

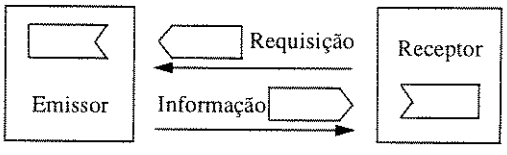


Figura A- 4: Acesso de informação a partir de uma requisição.

Neste caso, a comunicação baseia-se em uma requisição feita pelo usuário da informação (receptor), ao detentor da informação. O proprietário da informação responde à requisição enviando a informação solicitada.

Notação:

Processo Solicitante: (SENDER)	Processo dono da informação
OUTPUT Requisição TO Processo_dono;	INPUT Requisição; OUTPUT Resposta(inform) TO SENDER;

SENDER é uma variáveis do tipo predefinido PID que contém a identificação do processo que envia um sinal.

SDL define 4 variáveis predefinidas do tipo PID:

SENDER: identifica o processo que enviou um determinado sinal.

PARENT: identifica o processo que criou um outro.

OFFSPRING: identifica o último processo criado.

SELF: identifica o próprio processo.

5. Repositório comum de informação

Esta forma de comunicação permite o envio de informações para um de vários receptores em potencial, armazenando as mesmas em um repositório comum. Fica a critério do emissor decidir quando colocar a informação no repositório. Uma vez que a informação está no repositório o emissor perde o controle da mesma. O repositório funciona como uma caixa de correio. Cada receptor decide quando acessar o repositório para recuperar a informação.

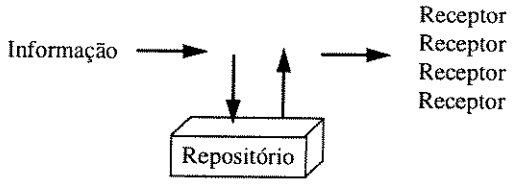


Figura A- 5: Repositório comum de informação.

Para modelar este tipo de comunicação pode-se combinar dois mecanismos vistos anteriormente: envio de sinais e informação compartilhada. Existe também um processo que gerencia o repositório.

Os processos enviam as informações ao processo gerenciador e este faz a informação visível aos usuários através do mecanismo de informação compartilhada.

Notação:

```
Processo_envia:
    DCL Sinal_1 boolean;
    .....
    OUTPUT Sinal_1 TO Processo_gerenciador;
Processo_gerenciador:
    DCL EXPORTED REVEALED Sinal_X boolean:=False;
    .....
    INPUT Sinal_1(Sinal_X);
    EXPORT Sinal_X;
    TASK Sinal_X :=TRUE;
    .....
```

Processos usuários:

.....
TASK var_x := IMPORT (Sinal_X, Processo_gerenciador);
.....
TASK var_y := VIEW (Sinal_X, Processo_gerenciador);

Esquemas de comunicação

- Envio sem espera
- Espera por sincronização

1. Envio sem espera (Send no Wait)

Neste esquema de comunicação o emissor envia a informação e continua o processamento independentemente da recepção da informação pelo receptor (Figura A- 6).

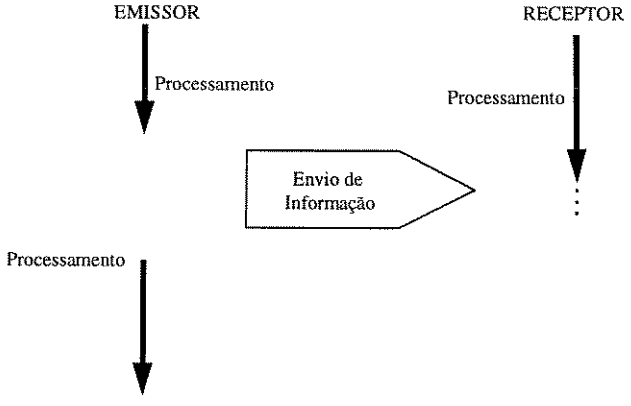


Figura A- 6: Envio sem espera.

É importante considerar que o envio de mensagens com este esquema envolve um tempo. As vezes é crucial que a recepção da mensagem não exceda um tempo limite.

Uma outra consideração importante é que as vezes é necessária uma confirmação da recepção da mensagem ou uma notificação de que ocorreu um erro. No primeiro caso, espera-se o receptor enviar a confirmação. No segundo caso, estipula-se um *time out* e considera-se que a comunicação falhou após este limite de tempo.

Este esquema é aplicável aos seguintes tipos de comunicação:

- Envio de informação a um receptor
- Broadcasting de informação
- Repositório comum de informação

Notação:

Processo Emissor

.....
OUTPUT Sinal_4 (informação) TO Instância_do_processo_n;
TASK .....;

No exemplo acima, o processo emissor envia um sinal e continua o seu processamento independentemente da recepção do sinal pelo receptor.

Processo Emissor	Processo Receptor
OUTPUT Sinal_3(NOW);	INPUT Sinal_3 (Tempo_de_envio); DECISION (NOW - Tempo_de_envio)<5/*msec*/; (FALSE) /*atraso inaceitável */;

Neste exemplo, especifica-se que o tempo entre o envio e a recepção do sinal deve ser menor que um limite de 5 milisegundos.

2. espera por sincronização (Wait for Synchronisation)

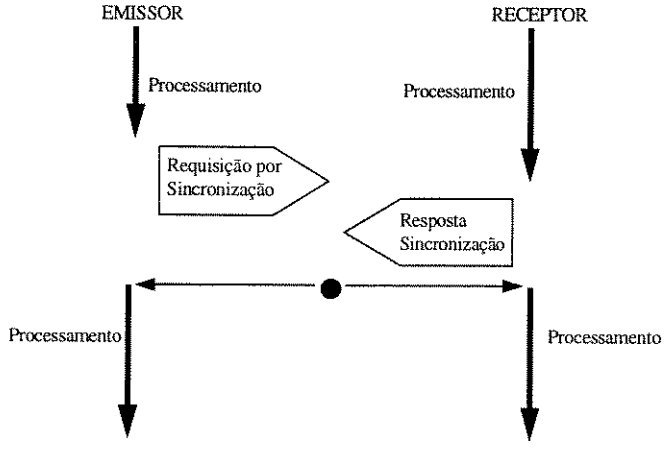


Figura A- 7: Espera por sincronização.

Neste mecanismo, o emissor requisita a atenção do receptor (requisição por sincronização) e suspende o seu processamento até receber uma resposta do receptor. Após a troca de informação, cada processo continua o seu processamento de forma independente.

Para evitar a presença de *deadlock*, é necessário introduzir limites de tempo através de *time outs*.

Este esquema tem a vantagem de que a troca de informação após a sincronização não toma tempo (ou é mínimo) e se tem a certeza da recepção da mensagem.

Este esquema é aplicável aos seguintes tipos de comunicação:

- Informação revelada
- Envio de informação a um receptor
- Mostrar informação a partir de uma requisição
- Repositório comum de informação.

Notação:

SENDER	RECEIVER
OUTPUT Requer_atenção; NEXTSTATE Espera_receptor_pronto; STATE Espera_receptor_pronto; SAVE *; INPUT Atenção_dada; OUTPUT Mensagem;	STATE ativado; INPUT Requer_atenção; OUTPUT Atenção_dada; NEXTSTATE Espera_mensagem; STATE Espera_mensagem; SAVE *; INPUT Mensagem;

Neste exemplo é usado o envio de um sinal para conseguir a sincronização. É importante considerar que o tempo gasto na transferência do sinal é nulo.

Notar nesta representação que se algum erro ocorrer as duas instâncias dos processos podem parar o processamento indefinidamente (*deadlock*). Para evitar isto, pode-se "setar" um relógio, conseguindo que o processo seja liberado com um sinal de *time out*, como no exemplo seguinte:

SENDER	RECEIVER
DCL Falha_sincroniz TIMER; ..... SET(NOW+100/*msec*/, Falha_sincroniz); NEXTSTATE Espera_receptor_pronto; STATE Espera_receptor_pronto; SAVE *; INPUT Atenção_dada; OUTPUT Mensagem; ..... INPUT Falha_sincroniz;	DCL Falha_sincroniz TIMER; ..... SET(NOW+5/*msec*/,Falha_sincroniz); NEXTSTATE Espera_mensagem; STATE Espera_mensagem; SAVE *; INPUT Mensagem; ..... INPUT Falha_sincroniz;

Anexo 2 : Dicionário de dados de operações e atributos do ST

O texto a seguir foi gerado pela ferramenta LOV/OMT a partir da modelagem estática do ST.

CLASS MODULE Serviço de Transações;

CLASSES

control, factory, factoryrec, recoverycoordinator,

resource, terminator, transaction;

NOTES

Definição do Serviço de Transações (ST)

O módulo Serviço de Transações apresenta um modelo com as classes que fazem parte deste serviço.

O ST deve controlar o escopo e a duração de uma transação e coordenar a sua finalização. Deve permitir também que múltiplos recursos participem em uma transação única e atômica.

O modelo apresentado é específico para o tipo de transações "Flat", definidas como aquelas transações que não possuem transações filhas.

CLASS control;

NOTES

Descrição da classe control

Fornece o acesso aos objetos *terminator* e *coordinator* que coordenam o protocolo *commit* em duas fases (two-phase commit). *Control* está associada implicitamente a uma transação específica. Existirá um objeto *control* para cada transação iniciada.

OPERATIONS

get_terminator() : terminator;

NOTES

Descrição: retorna um objeto da classe *terminator*. Esta operação é usada pelos programas clientes do ST. Após obter o objeto *terminator*, o cliente pode solicitar o fechamento da transação chamando as operações *commit* ou *rollback* do *terminator*.

get_coordinator() : coordinator;

NOTES

Descrição: retorna um objeto da classe *coordinator*. Esta operação é usada pela classe *factoryrec*. Após obter o objeto *coordinator*, o objeto *factoryrec* pode solicitar o registro dos recursos criados por ele, chamando o método register_resource.

Também é usada pelas classes que precisam de informação sobre o estado da transação. Após obter o objeto *coordinator*, podem solicitar informações sobre o estado da transação chamando o método get_status.

CLASS coordinator;

NOTES

Descrição da classe coordinator

Permite o registro de recursos como participantes da transação e o acesso às informações relacionadas com a transação. Existe um coordenador para cada transação.

OPERATIONS

get_status() : Status;

NOTES

Descrição: retorna o estado atual da transação corrente. É usada pelo objeto *terminator*, pelo próprio cliente, ou pelos recursos participantes.

is_same_transaction(coordinator : coordinator) : boolean;

NOTES

Descrição: retorna um valor *boolean*. É usado pelos recursos participantes para identificar a transação onde estiver participando. O parâmetro de entrada é o objeto *coordinator*. Esta operação não é utilizada no contexto do protocolo *commit* em duas fases.

hash_transacation() : integer;

NOTES

Descrição: retorna um valor inteiro correspondente ao código *hash* da transação. Esta operação não é utilizada no contexto do protocolo *commit* em duas fases.

register_resource(resource : resource) : recoverycoordinator;

NOTES

Descrição: registra objetos *resources* como participantes da transação. É usada pela *factoryrec* após a criação dos objetos *resource*. Possui como parâmetro de entrada o recurso a ser registrado. O retorno é um objeto *recoverycoordinator*.

rollback_only();

NOTES

Descrição: muda o estado da transação para StatusMarkedRollback. Isto significa que a transação só poderá ser desfeita. Esta operação é usada pelos recursos participantes da transação para forçar a execução de *rollback*.

get_transaction_name() : string;

NOTES

Descrição: retorna o nome da transação corrente. É utilizada pelos recursos participantes. Esta operação não é utilizada no contexto do protocolo *commit* em duas fases.

change_status(status : Status) : Status;

NOTES

Descrição: muda o estado da transação durante a execução do protocolo *commit* em duas fases. É chamada pelo objeto *terminator* durante a execução do protocolo. Se o *terminator* enviar o pedido de *prepare* para os recursos, o estado da transação deverá mudar para StatusPrepared. Se o *terminator* decidiu fechar a transação com *commit* o estado da transação será StatusCommitted, caso contrário se a decisão é fechar com *rollback* o estado da transação será StatusRolledBack. O retorno desta operação é o novo estado da transação.

get_resource_registered() : array;

NOTES

Descrição: devolve uma tabela contendo a identificação dos recursos registrados na transação. É usada pelo objeto *terminator* para iniciar o protocolo *commit* em duas fases.

CLASS factory;

NOTES

Descrição da classe factory

Inicia uma nova transação criando um objeto da classe *control*. É utilizada quando clientes iniciam transações.

OPERATIONS

create(time_out : integer) : control;

NOTES

Descrição: cria e retorna um objeto da classe *control*. É usada por clientes do ST para iniciar uma nova transação. Tem como parâmetro de entrada o tempo de duração da transação que é definida pelo cliente. O gerenciamento do parâmetro de tempo não é implementado neste trabalho.

CLASS factoryrec;

NOTES

Descrição da classe factoryrec

Criar objetos da classe *resource* e solicita o seu registro ao coordenador da transação.

OPERATIONS

create(control : control, nome : string) : resource;

NOTES

Descrição: cria objetos *resource* a serem registrados como participantes da transação. É utilizada pelo cliente que iniciou a transação. Possui como parâmetros de entrada o objeto *control* criado para a transação e um nome que identifica o recurso.

CLASS recoverycoordinator;

NOTES

Descrição da classe recoverycoordinator

Gerencia o processo de recuperação em certas situações. Existirá um único objeto desta classe para cada recurso registrado.

OPERATIONS

replay_completion(resource : resource) : Status;

NOTES

Descrição: retorna o estado corrente da transação. É usada pelos recursos participantes após a ocorrência de alguma falha. Pode ser invocado pelo recurso

associado após ter executado *prepare*. Esta operação não é utilizada no contexto do protocolo *commit* em duas fases.

CLASS resource;

NOTES

Descrição da classe resource

Fornece as funcionalidades do protocolo *commit* em duas fases.

OPERATIONS

prepare() : Vote;

NOTES

Descrição: é chamada pelo objeto *terminator*. Devolve um valor do tipo Vote contendo a resposta ao pedido de *prepare*. Este "voto" indica se o recurso concorda ou não com a execução de *commit* preparando-se para fechar a transação. O efeito desta operação no recurso é registrar todas as informações necessárias para desfazer as mudanças realizadas, caso o *terminator* decida fechar a transação com *rollback*.

rollback();

NOTES

Descrição: é chamada pelo objeto *terminator* para solicitar a execução de *rollback*. O efeito é desfazer todas as alterações feitas sobre dados durante a transação.

commit();

NOTES

Descrição: é chamada pelo objeto *terminator* para solicitar a execução de *commit*. O efeito desta operação é fazer permanentes todas as alterações feitas sobre dados durante a transação.

forget();

NOTES

Descrição: faz o recurso "esquecer" todo conhecimento sobre a transação. Esta operação é chamada pelo objeto *terminator* após a ocorrência de uma situação de exceção.

commit_one_phase();

NOTES

Descrição: é chamada pelo *terminator* quando existe um único participante na transação. O efeito é o mesmo que a operação *commit*.

CLASS terminator;

NOTES

Descrição da classe terminator

Suporta operações para o encerramento da transação. É usada pelo cliente que originou a transação. O objeto *terminator* pode fechar a transação com *commit* ou com *rollback*. Existe um *terminator* para cada transação.

OPERATIONS

commit(report_heuristics : boolean);

NOTES

Descrição: usada pelo cliente do ST que iniciou a transação.

Na recepção do pedido de *commit* o *terminator* deve verificar se o estado da transação não é StatusMarkedRollback e se a transação não teve tentativa de *rollback* (estado diferente de StatusRolledback).

Em seguida, o objeto *terminator* enviará uma requisição de *prepare* para cada recurso participante da transação. Para obter informações dos recursos participantes o *terminator* deve interagir com o objeto *coordinator* chamando a operação get_register_resource.

Após a requisição de *prepare*, os recursos respondem com um "voto". Se os votos foram a favor de *commit* (VoteCommit), o *terminator* enviará uma requisição de *commit* aos recursos. Se algum recurso responder com *rollback* (VoteRollback) o *terminator* enviará uma requisição de *rollback* àqueles recursos que responderam com VoteCommit. Se algum recurso responder com VoteReadOnly não será necessário o envio de requisição para ele.

Esta operação tem como parâmetro de entrada um valor *boolean,* que define a requisição ou não de um informe de exceções.

Após a decisão de *commit* e tendo o valor do parâmetro de entrada igual a *false*, a transação é considerada fechada com sucesso. Se ocorreu uma exceção e o parâmetro de entrada for *true* o Serviço de Transações deve informar o ocorrido ao cliente. Após isto, é enviada uma requisição de *forget* aos recursos onde ocorreu a exceção.

rollback();

NOTES

Descrição: usada pelo cliente do ST que iniciou a transação. Serve para fechar a transação desfazendo as alterações feitas durante a transação.

Na recepção deste pedido o *terminator* enviará um pedido de *rollback* para todos os recursos participantes da transação. Após a execução desta operação, a transação é considerada fechada sem sucesso, devendo-se iniciar o processo de recuperação nos recursos envolvidos.

CLASS transaction;

NOTES

Descrição da classe transaction

Armazena todas as informações sobre a transação corrente. Não possui operações, tem-se acesso as suas informações através do objeto *coordinator.*

ATTRIBUTES

nome : string;

status : Status;

código_hash : integer;

NOTES

Descrição dos atributos:

nome: mostra o nome atribuído à transação corrente. O acesso a este atributo é feito através da operação get_transaction_name do *coordinator.*

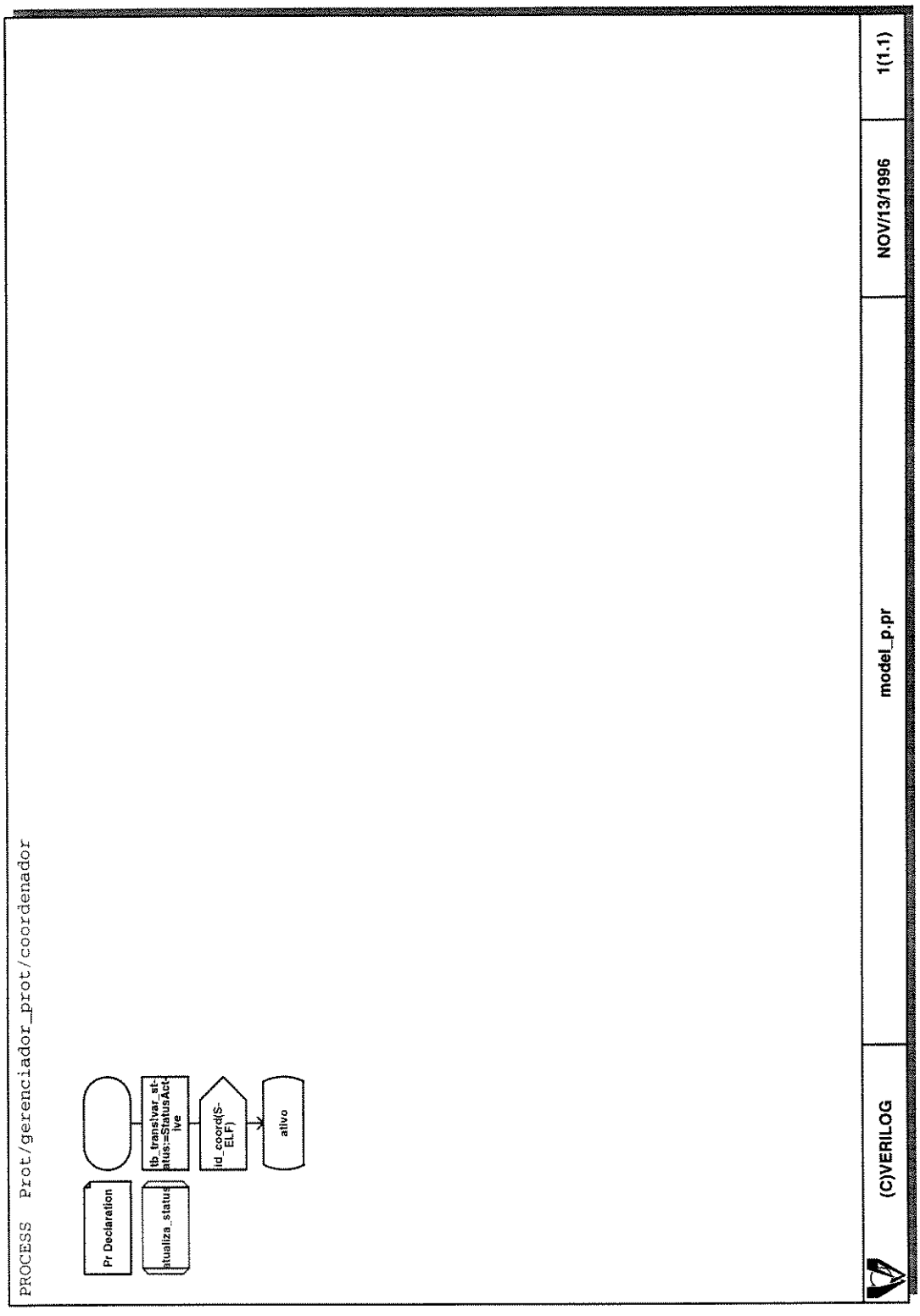
status: mostra o estado atual da transação corrente. O acesso a este atributo é feito através das operações get_status e change_status, ambas do *coordinator.*

código_hash: mostra o código *hash* associado com a transação corrente.

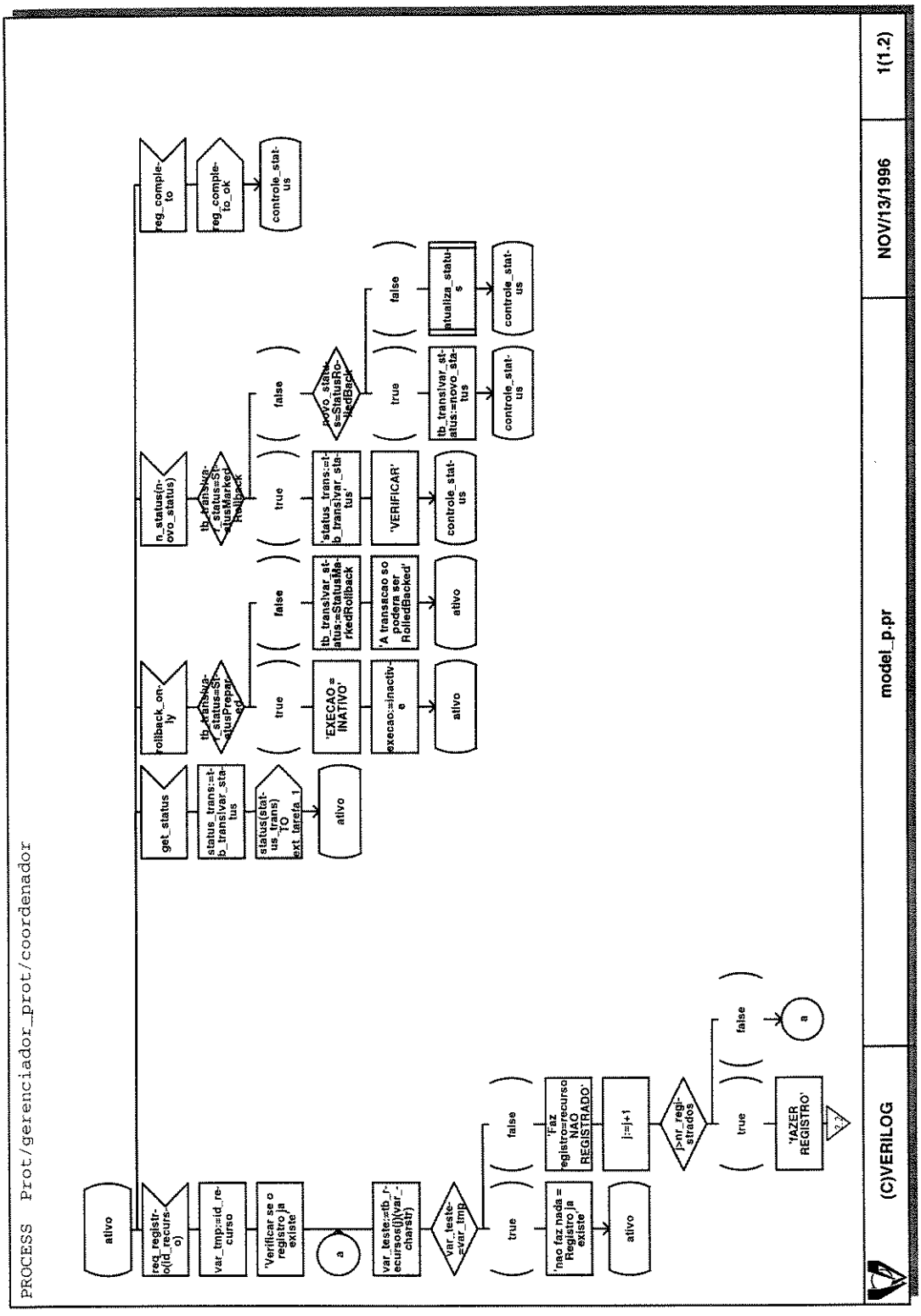
Apêndice 1: EFSMs dos processos coordenador e terminador

Neste apêndice são apresentadas as EFSMs dos processos coordenador e terminador na sua notação gráfica (SDL-GR). Estes modelos foram gerados pela ferramenta GEODE.

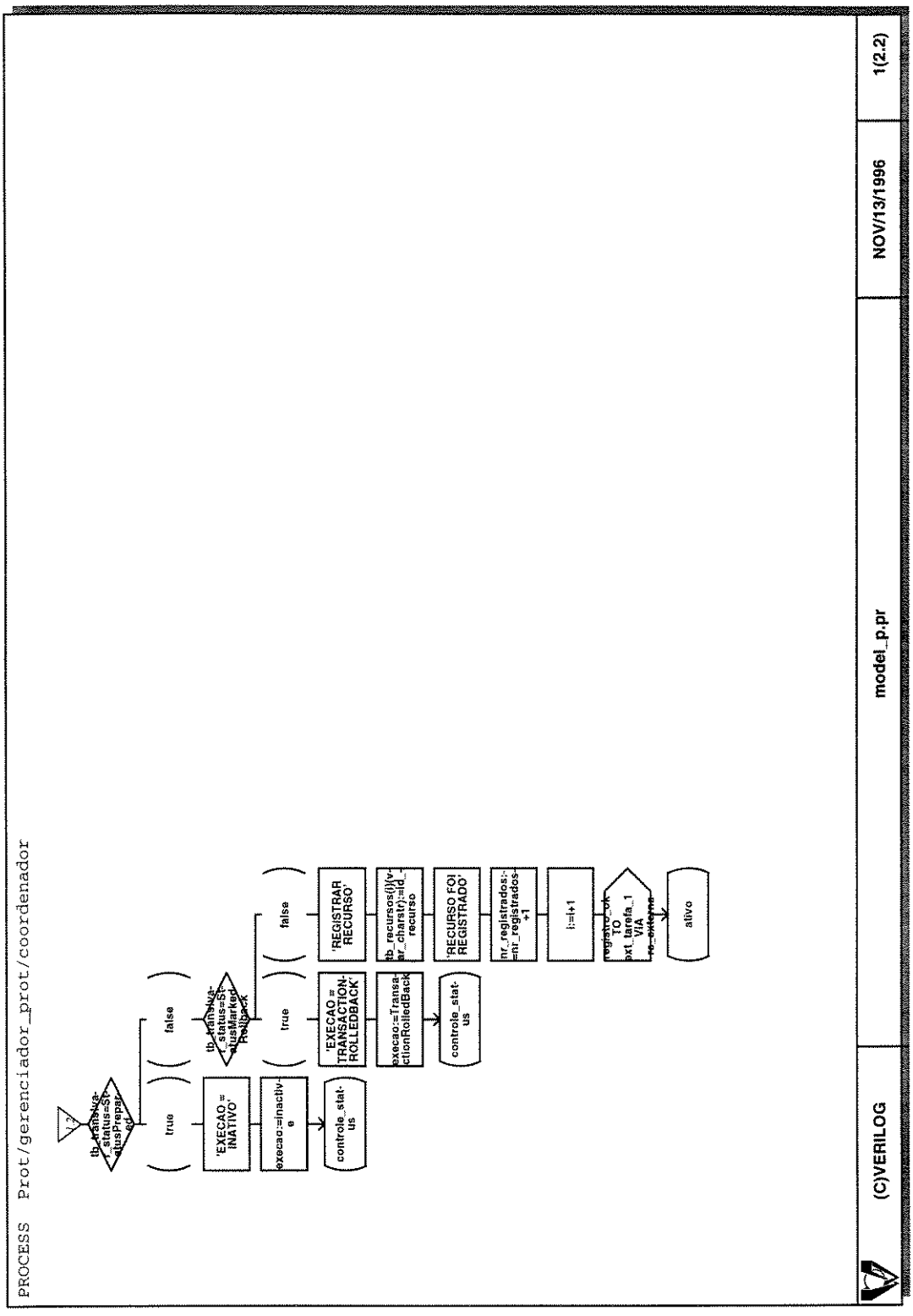
EFSM do processo coordenador



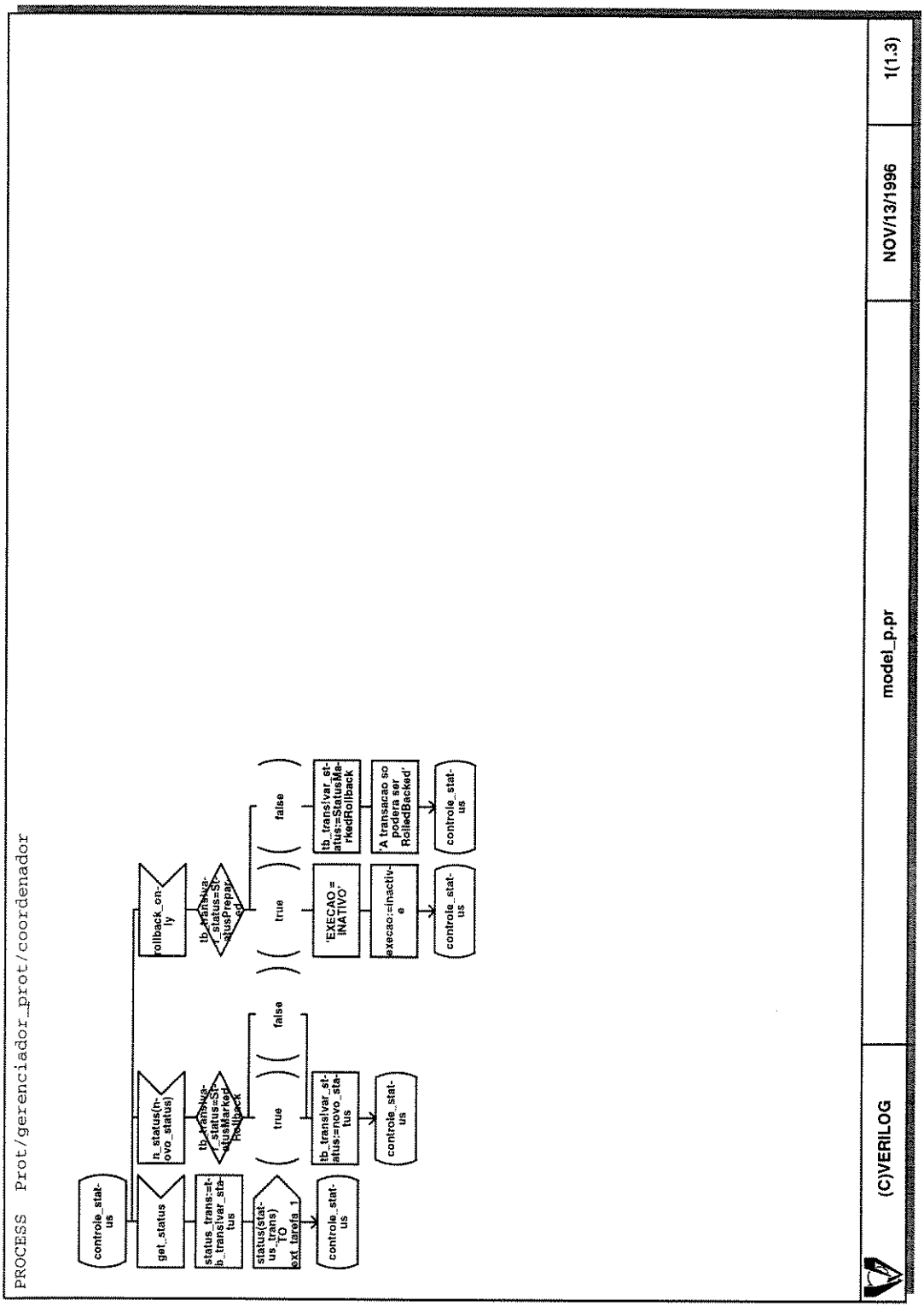
EFSM do processo coordenador



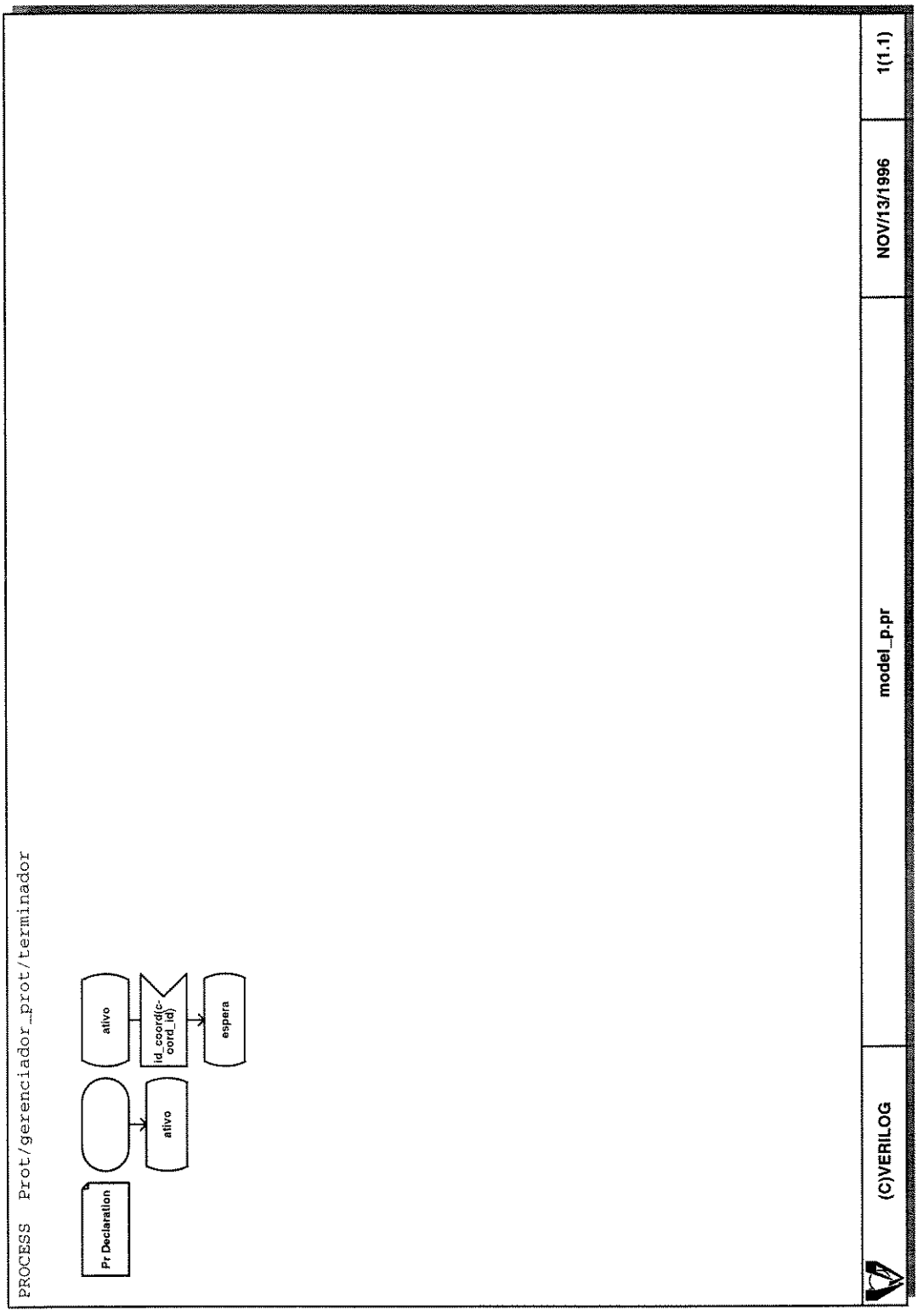
EFSM do processo coordenador



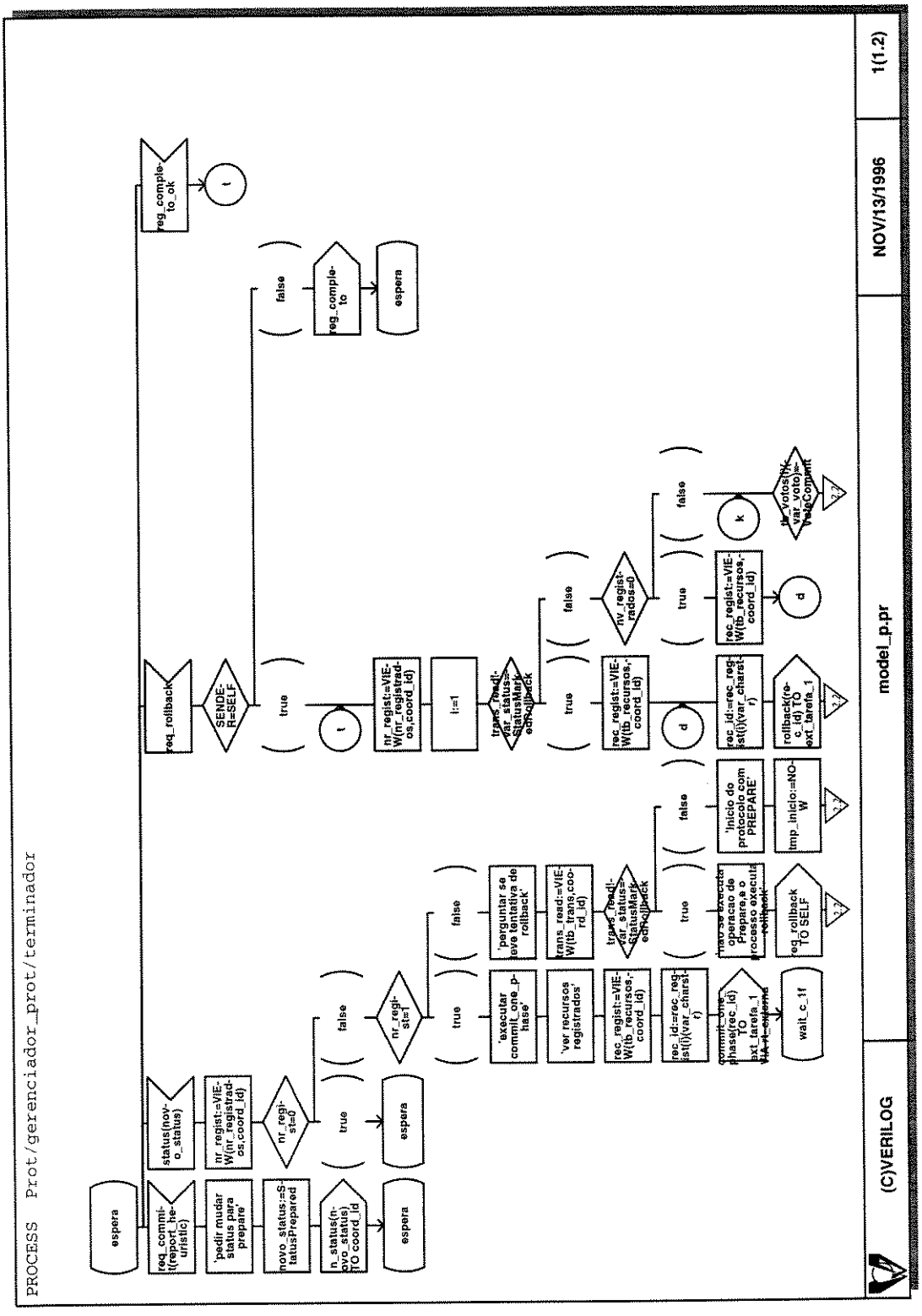
EFSM do processo coordenador



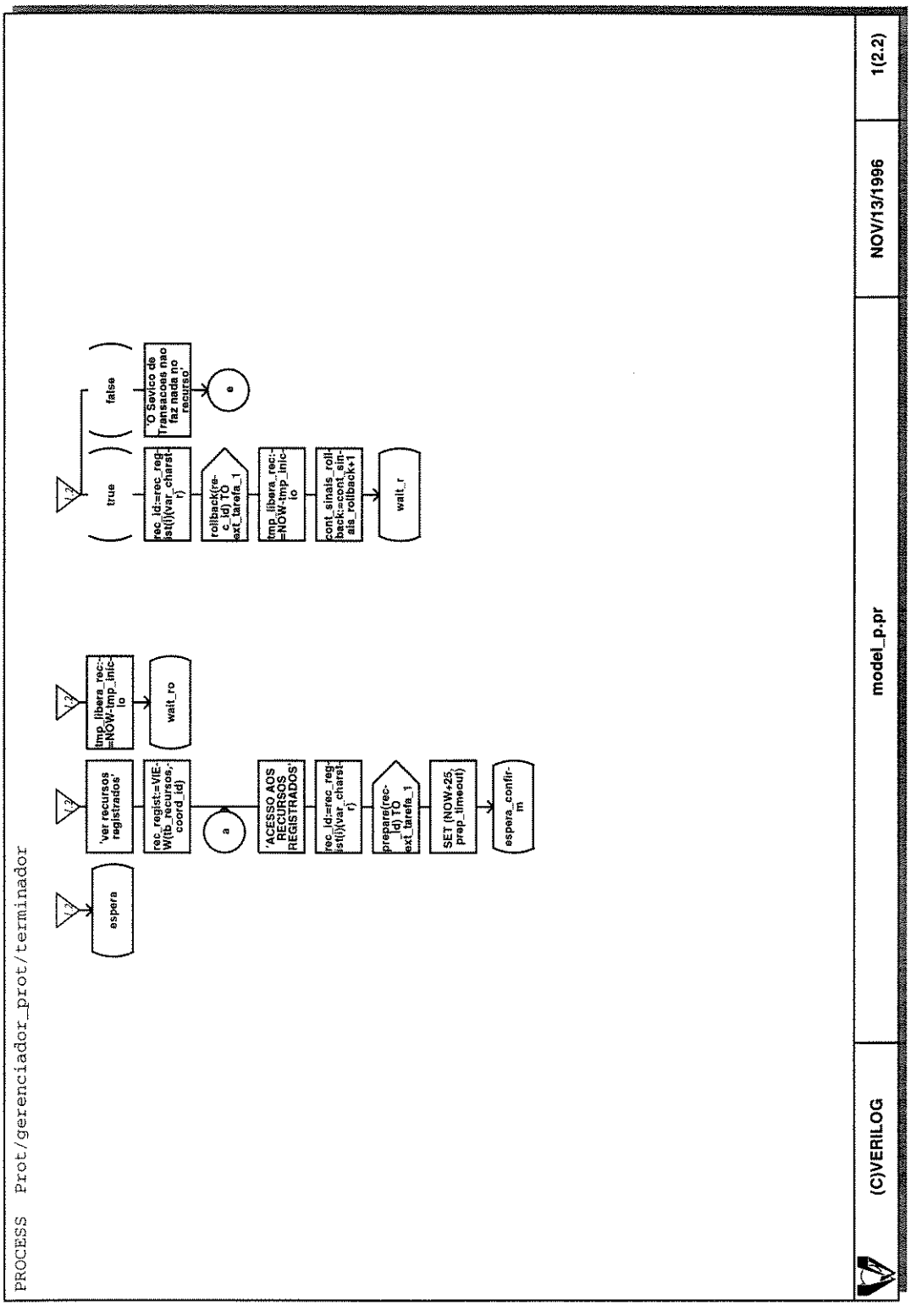
EFSM do processo terminador



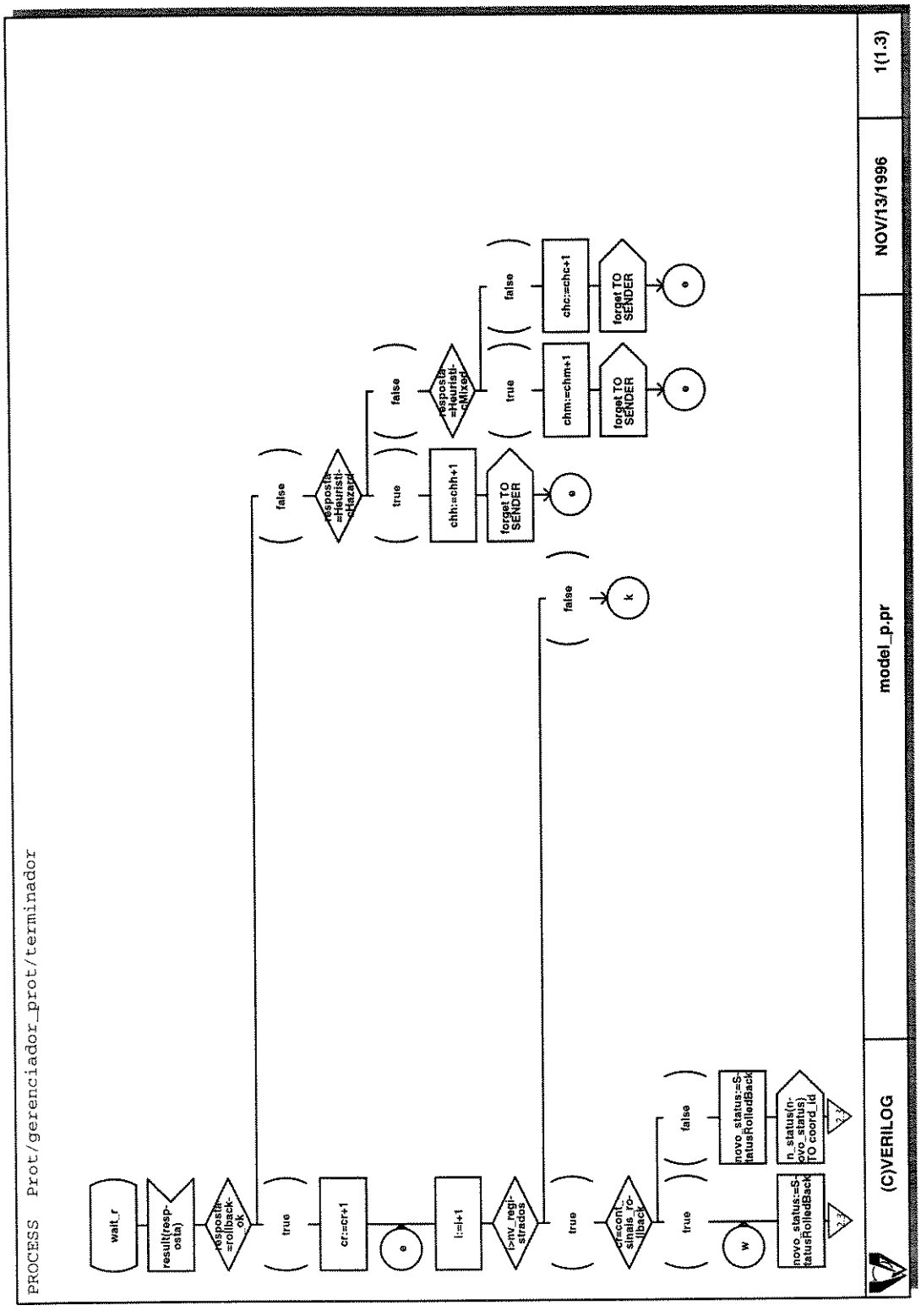
EFSM do processo terminador



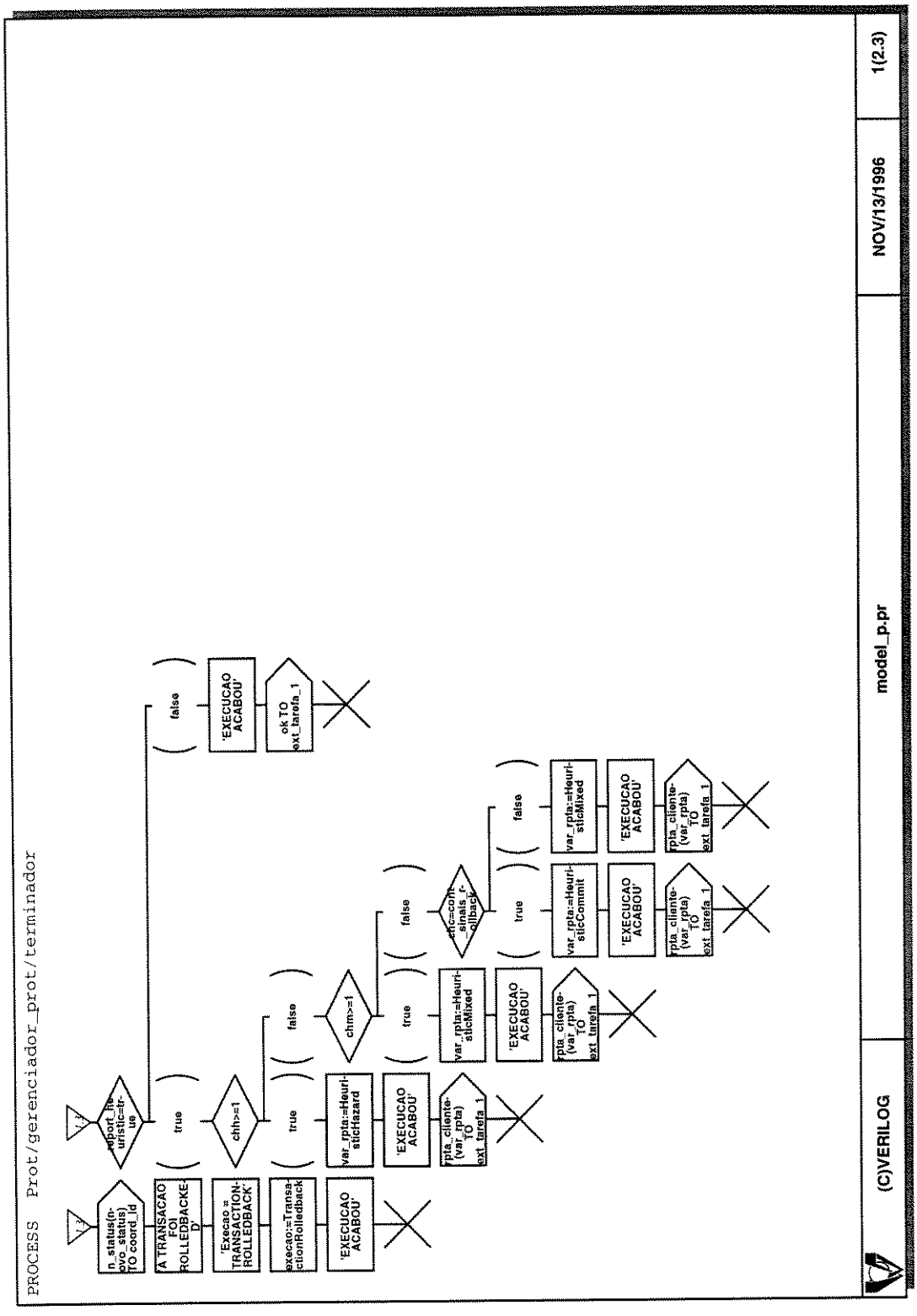
EFSM do processo terminador



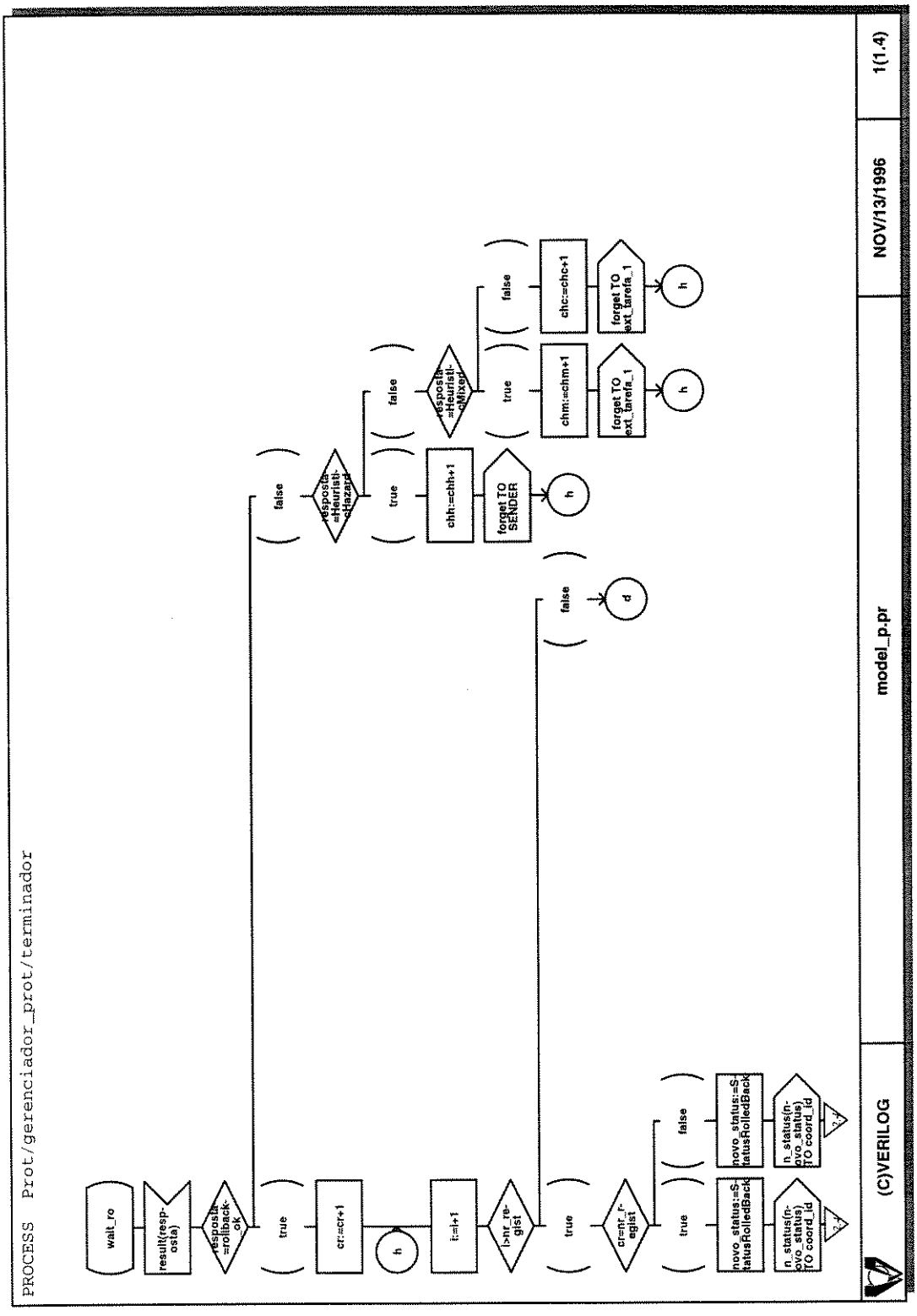
EFSM do processo terminador



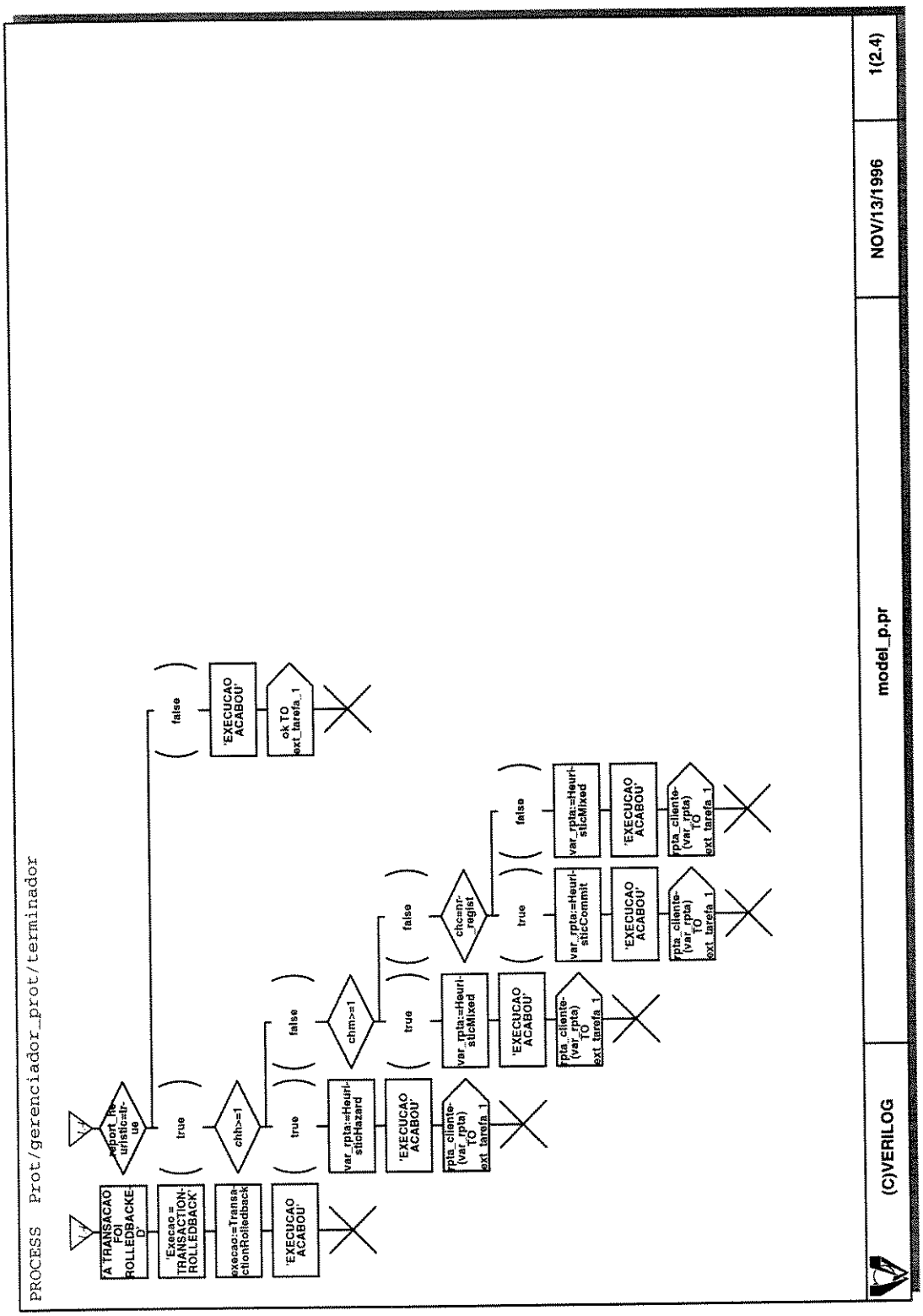
EFSM do processo terminador



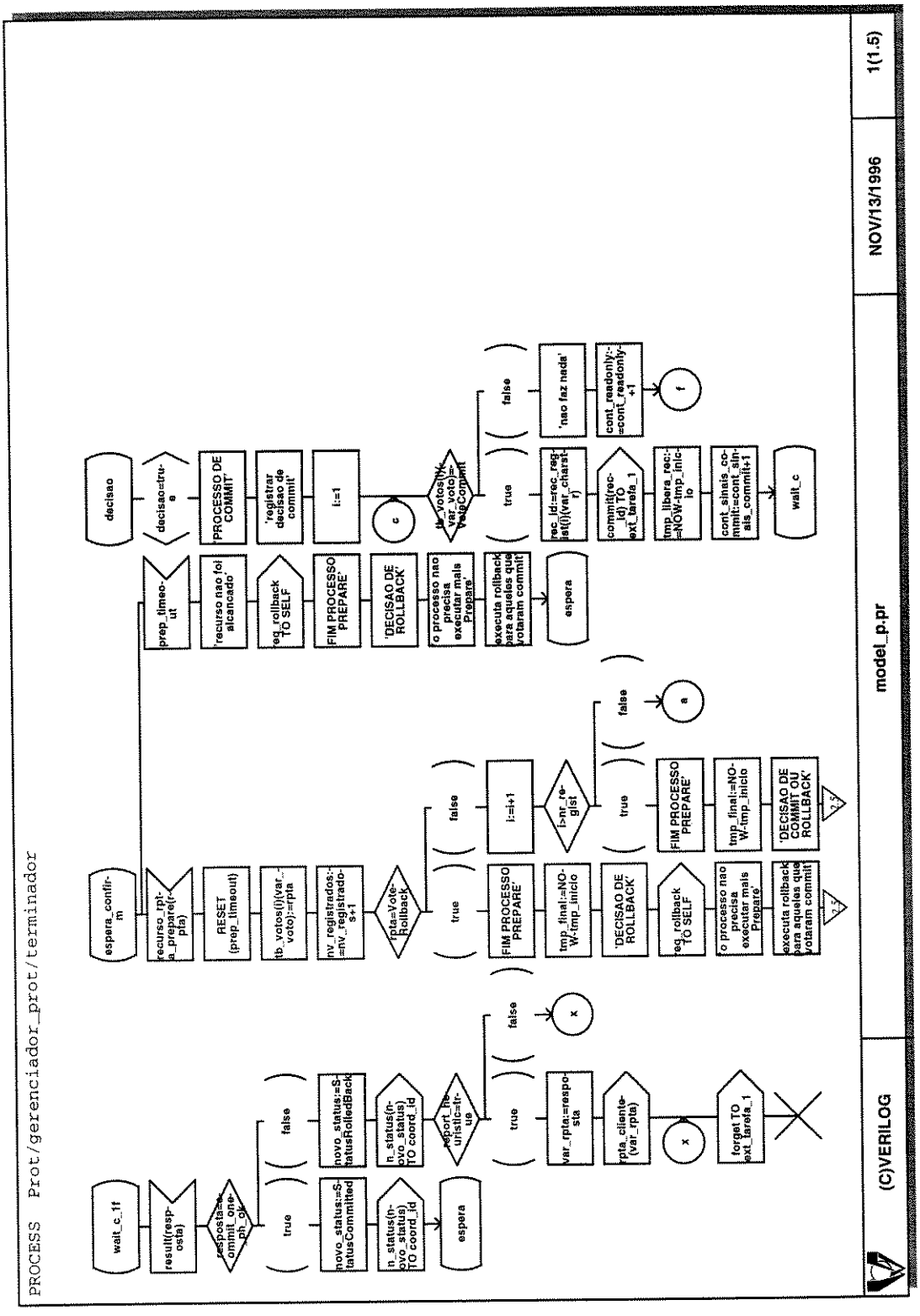
EFSM do processo terminador



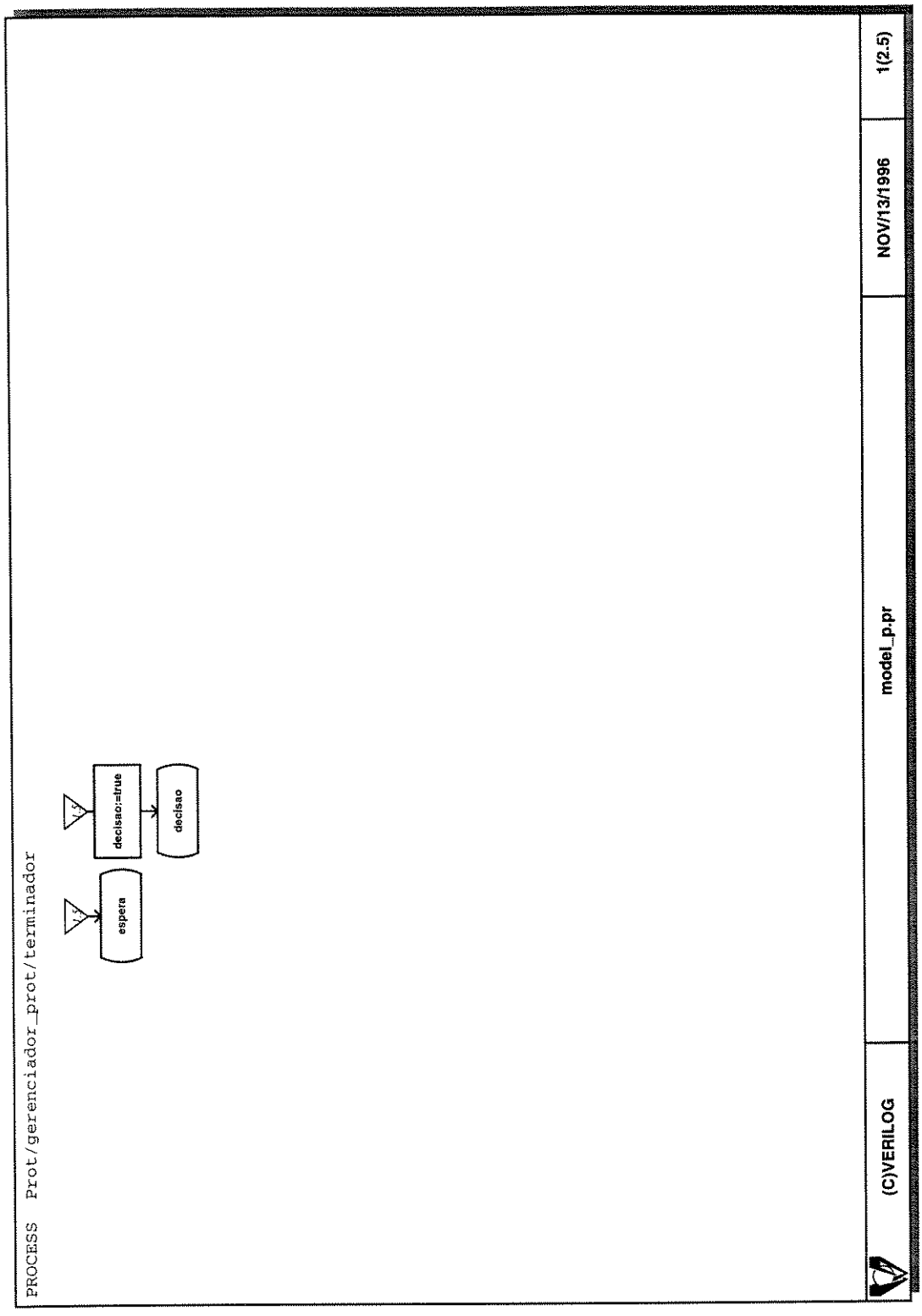
EFSM do processo terminador



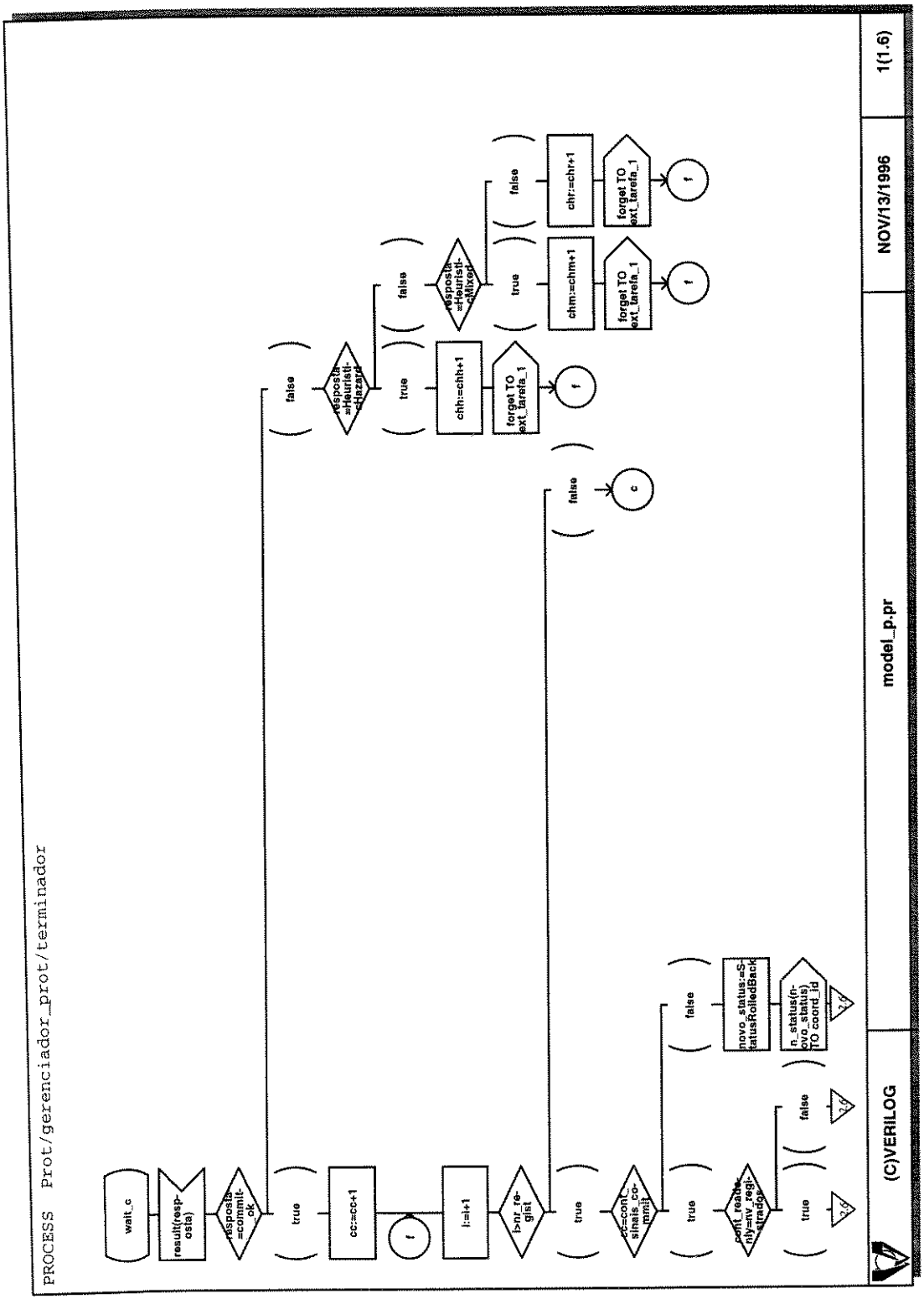
EFSM do processo terminador



EFSM do processo terminador



EFSM do processo terminador



EFSM do processo terminador