Detecção de propriedades estáveis em sistemas distribuídos

Gisele da Silva Craveiro

Dissertação de Mestrado

Detecção de propriedades estáveis em sistemas distribuídos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Gisele da Silva Craveiro e aprovada pela Banca Examinadora.

Campinas, 22 de maio de 1998.

Prof. Dr. Ricardo de Oliveira Anido
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Detecção de propriedades estáveis em sistemas distribuídos

Gisele da Silva Craveiro[1]

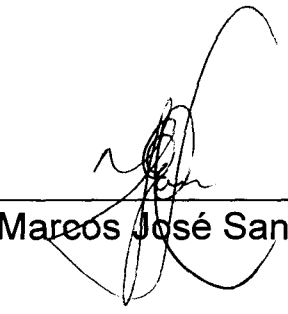
Abril de 1998

Banca Examinadora:

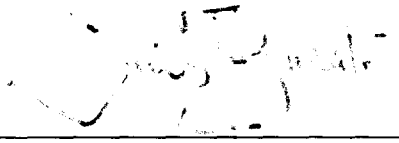
- Prof. Dr. Ricardo de Oliveira Anido (Orientador)
- Prof. Dr. Marcos José Santana
  Departamento de Ciências de Computação e Estatística - SCE
  Instituto de Ciências Matemáticas de São Carlos - ICMC- USP
- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação - UNICAMP
- Prof. Dra. Maria Beatriz F. de Toledo (Suplente)
  Instituto de Computação - UNICAMP

[1]**Apoio financeiro CNPq-FAPESP processo 96/10031-3**

Tese de Mestrado defendida e aprovada em 22 de maio de 1998 pela Banca Examinadora composta pelos Professores Doutores



Prof. Dr. Marcos José Santana



Prof. Dr. Luiz Eduardo Buzato

Prof. Dr. Ricardo de Oliveira Anido

© Gisele da Silva Craveiro, 1998.
Todos os direitos reservados.

Aos meus pais José e Célia
e à minha avó Dalila.

Aos milhões de brasileiros anônimos,
principalmente os analfabetos
e semi-alfabetizados.

Quem construiu a Tebas das sete portas ?
Nos livros constam os nomes dos reis.
Os reis arrastaram os blocos de pedra?

E a Babilônia tantas vezes destruída
Quem a ergueu outras tantas?

Em que casas da Lima radiante de ouro
Moravam os construtores?

Para onde foram os pedreiros
Na noite em que ficou pronta a Muralha da China?

A grande Roma está cheia de arcos de triunfo.
Quem os levantou?
Sobre quem triunfaram os césares?

A decantada Bizâncio só tinha palácios.
Para seus habitantes?

Mesmo na legendária Atlântida,
Na noite que o mar a engoliu,
Os que se afogavam gritavam pelos seus escravos.

O jovem Alexandre conquistou a India.
Ele sozinho?

César bateu os gauleses.
Não tinha pelo menos um cozinheiro consigo?

Filipe de Espanha chorou quando sua armada naufragou.
Ninguém mais chorou?

Frederico II venceu a Guerra dos Sete Anos.
Quem venceu além dele?

Uma vitória em cada página.
Quem cozinhava os banquetes da vitória?
Um grande homem a cada dez anos.
Quem pagava suas despesas?

Tantos relatos.
Tantas perguntas.

Bertold Brecht

A História, como sabemos, é feita não por homens isolados, mas por sociedades.

Agradecimentos

Agradeço a Deus,

pela vida,
por minha harmoniosa família,
pela oportunidade de estudar e me desenvolver,
pelas irmãs que pude escolher,
pelos meus amigos,
pelo ensinamento encontrado em colegas, professores e funcionários da UNICAMP,
pelas pessoas marcantes, tanto aquelas que me despertaram a admiração e o respeito, como aquelas que me ensinaram a não seguir o seu exemplo,
pela dura ausência e distância das pessoas queridas (temos que aprender a perder)
por sofrer com a morte e assim aprender o nada-e-tudo que a vida é,
por aumentar minha fé ("pedir o que já se tem"),
por estar me ensinando através das pessoas a desafiar e respeitar os limites (tantos os meus como os dos outros),
por me mostrar o Amor (quando a linguagem universal da Matemática não é capaz de expressar em números quanto é uma simples conta de um mais um),
por me mostrar o Amor (quando se descobre que se é uma colcha de retalhos feitos dos outros),
por me mostrar o Amor (quando você se sente retalho em outras colchas),
por me mostrar a Dor (crescer dói),
por me mostrar a Dor (remédio que arde, mas cura).

E, principalmente, agradeço por tudo que pedi e não recebi, pois o nada que ganhei era tudo que precisava.

Meus agradecimentos se estendem a todas as pessoas citadas em minha oração. Nomes próprios não foram escritos, mas com certeza sempre serão lidos.

Agradeço às pessoas que nem imaginam que cheguei aqui, mas graças a elas os órgãos de fomento à pesquisa viabilizaram esse trabalho.

O meu muito obrigado.

Resumo

Elaborar uma solução para um problema distribuído que atenda aos critérios de segurança e progresso está muito longe de ser uma tarefa trivial. Nem mesmo algoritmos publicados escaparam de contestação posterior, mostrando que são incorretos.

Com o objetivo de encontrar uma modelagem mais adequada, alguns pesquisadores tomam a direção das derivações. Verificando comportamentos similares e características comuns, pretende-se desenvolver um método que facilite o projeto de algoritmos distribuídos para resolver problemas que apresentem algumas similaridades. Uma característica muito importante comum a certos problemas é a propriedade da estabilidade, isto é, se o problema ocorre, ele persistirá até que seja detectado e tratado. Assim acontece com *deadlocks* e quando uma computação distribuída alcança o término de sua atividade.

Esta dissertação concentra-se nos seguintes pontos. Do estudo das áreas de *deadlock* e terminação distribuídos resulta a apresentação da modelagem de cada problema, uma visão em largura das abordagens propostas para a detecção do *deadlock* distribuído e para a detecção da terminação distribuída e a discussão mais detalhada de alguns algoritmos propostos para esses dois problemas. Às contribuições citadas soma-se a proposição de um método que derive soluções para o problema da terminação distribuída a partir de algoritmos para a detecção do *deadlock*, preservando as propriedades de segurança e progresso da solução base.

Abstract

Devising a solution to a distributed problem, satisfying the requirements of progress and safety is clearly not an easy task. Several published algorithms have been shown incorrect, despite having been "proven" correct at time of publication.

An interesting approach to devise a new solution to a problem is to use a known solution to a different problem that has similar requirements and characteristics. In this dissertation we present a methodology to derive solutions to the problem of detecting the termination of a distributed computation based on solutions to the deadlock problem. A common characteristic of these problems is that both are *persistent*: once attained the condition of deadlock or termination, the condition will hold thereafter, until some external action is taken.

Besides presenting the derivation methodology, this dissertation presents a broad study of the more interesting solutions to the deadlock and termination problems found in the literature.

Conteúdo

Agradecimentos viii

Resumo x

Abstract xi

1 Introdução 2
- 1.1 Organização do Trabalho . . . 3
- 1.2 Modelo do Sistema . . . 3
  - 1.2.1 Propriedades do Sistema . . . 4
  - 1.2.2 Representação em diagrama temporal . . . 5
- 1.3 Ordenação causal e Estado Global . . . 5
- 1.4 Representação em grafos . . . 6
- 1.5 Complexidades . . . 7

2 A Teoria da Terminação 8
- 2.1 O comportamento do sistema . . . 8
- 2.2 O problema . . . 9
- 2.3 Algumas classes de algoritmos . . . 10
  - 2.3.1 O algoritmo bifásico e variantes . . . 11
  - 2.3.2 Utilização de *timestamps* . . . 12
  - 2.3.3 Contadores vetoriais . . . 12
  - 2.3.4 Algoritmos de contagem de canal . . . 13
  - 2.3.5 Algoritmos de recuperação de crédito . . . 14
- 2.4 Comentários . . . 15

3 Algoritmos de Detecção de Terminação 17
- 3.1 Algoritmos bifásicos . . . 17
  - 3.1.1 O algoritmo de Dijkstra e Scholten 80 . . . 17
  - 3.1.2 O algoritmo de Dijkstra, Feijen e Van Gasteren 83 . . . 21

3.1.3 O algoritmo de Mattern 90 25
3.2 Algoritmos monofásicos 28
3.2.1 Uso de timestamp - O algoritmo de Rana 83 28
3.2.2 Contagem de canal- o algoritmo de Shavit e Francez 86 31
3.2.3 Recuperação de crédito - o algoritmo de Mattern 89 34

4 A Teoria do *Deadlock* 37
4.1 Conceitos básicos 39
4.1.1 O modelo do sistema 39
4.1.2 Os modelos da teoria dos grafos 40
4.1.3 A hierarquia dos modelos de requisição 40
4.2 Definições de *Deadlock* 45
4.3 Classificação das abordagens de detecção de *deadlock* 47
4.3.1 Algoritmos *path-pushing* 47
4.3.2 Algoritmos caça-aresta (*edge-chasing*) 48
4.3.3 Computações difusas 48
4.3.4 Detecção do estado global 48
4.4 Considerações gerais 49
4.5 Comentários 49

5 Algoritmos de Detecção de *Deadlock* 51
5.1 Algoritmos para o modelo de requisição AND 51
5.1.1 Algoritmo de Menasce e Muntz 79 51
5.1.2 Algoritmo de Obermarck 82 56
5.1.3 Algoritmo de Chandy, Misra e Haas 83 60
5.2 Algoritmo para o modelo OR 64
5.2.1 Algoritmo de Chandy, Misra e Haas 83 64
5.3 Algoritmos para o modelo $C_{n,k}$ 69
5.3.1 Algoritmo de Bracha e Toueg 87 69
5.3.2 Algoritmo de Kshemkalyani 94 77
5.3.3 Algoritmo de Brzezinski *et al.* 95 86
5.4 Comentários 94

6 Derivações 96
6.1 A Terminação Distribuída 97
6.2 O *Deadlock* Distribuído 97
6.3 Detecção da Terminação utilizando Detecção de *Deadlock* 99
6.4 Exemplo de Derivação 103

7 Conclusões e Trabalhos Futuros **106**
7.1 Algoritmos de Detecção de Terminação Distribuída 106
7.2 Algoritmos de Detecção de Deadlock Distribuído 107
7.3 Derivação de algoritmos 107
7.4 Sugestões para futuros trabalhos 110

Bibliografia **111**

Lista de Tabelas

7.1 Tabela comparativa de alguns algoritmos distribuídos de detecção da terminação 108
7.2 Tabela comparativa de alguns algoritmos distribuídos de detecção de *deadlock* 109

Lista de Figuras

1.1 Exemplo de um *deadlock* no trânsito. 2
1.2 Diagrama temporal 5
1.3 Diagrama de precedência 6

2.1 Diagramas de tempo 9
2.2 Onda de controle 10
2.3 Contra-exemplo do método dos contadores 10
2.4 Exemplo do método de utilização de *timestamps* 12

3.1 Estado incial do sistema 20
3.2 Exemplo de uma execução do Algoritmo de Dijkstra e Scholten 21
3.3 Exemplo de uma execução do Algoritmo de Dijkstra, Feijen e Van Gasteren 24
3.4 Exemplo de uma execução do algoritmo de Mattern 90 27
3.5 Exemplo de uma execução do algoritmo de Rana 83 29
3.6 Exemplo de uma execução do algoritmo de Shavit e Francez 86 33
3.7 Exemplo de uma execução do algoritmo de Mattern 89 36

4.1 Exemplos de ciclo e de *knot* 41
4.2 Exemplo de uma requisição *AND-OR* 43
4.3 Mapeamento em processos 43
4.4 Decomposição de uma requisição 44
4.5 Hierarquia dos modelos de requisição 50

5.1 Exemplo de execução do algoritmo de Menasce e Muntz 79 54
5.2 Exemplo de *deadlock* não detectado 56
5.3 Dois grafos *wait-for* locais 56
5.4 Os grafos *wait-for* com a adição do nó externo 57
5.5 O grafo aumentado do nó S_2 60
5.6 Exemplo de requisição de um recurso que não está no local do processo . . 61
5.7 Relações de dependência entre os processos 63
5.8 Dependências representadas no modelo do sistema assumido 64

5.9 Exemplo de execução do algoritmo de Chandy et al. para o modelo AND . 65
5.10 Exemplo de uma execução do algoritmo para o modelo de comunicação . . 67
5.11 Continuação do exemplo de uma execução do algoritmo para o modelo de comunicação 68
5.12 Exemplo de uma detecção de *deadlock* 75
5.13 Exemplo de uma detecção de *deadlock* realizada através do algoritmo de Bracha e Toueg 76
5.14 As relações de dependência entre os processos 82
5.15 Exemplo de uma detecção de *deadlock* 83
5.16 Exemplo de uma detecção de *deadlock* 84
5.17 Continuação do exemplo de execução do algoritmo 85
5.18 Primeira passagem do token pelo sistema 90
5.19 Segunda passagem do token pelo sistema 91

6.1 *Deadlock* detectado, mas terminação ainda não alcançada. 102

Lista de Algoritmos

3.1 Descrição do Algoritmo de Dijkstra e Scholten 80 19
3.2 Descrição do Algoritmo de Dijkstra, Feijen e Van Gasteren 83 23
3.3 Descrição do Algoritmo de Mattern 90 26
3.4 Descrição do Algoritmo de Rana 83 30
3.5 Descrição do Algoritmo de Shavit e Francez 86 32
3.6 Descrição do Algoritmo de Mattern 89 - Recuperação de crédito 35
5.1 Descrição do Algoritmo de Menasce e Muntz 53
5.2 Descrição do Algoritmo de Obermarck 82 59
5.3 Descrição do Algoritmo de Chandy, Misra e Haas 83 para o modelo de recurso 62
5.4 Descrição do Algoritmo de Chandy, Misra e Haas 83 para o modelo de comunicação 66
5.5 Descrição do Algoritmo de Bracha e Toueg 87 (primeira versão) 71
5.6 Algoritmo do Clousure 72
5.7 Descrição do Algoritmo de Bracha e Toueg 87 (segunda versão) 74
5.8 Mensagens computacionais da aplicação 78
5.9 O tratamento da mensagem FLOOD 80
5.10 O tratamento das mensagens ECHO e SHORT 81
5.11 Descrição do Algoritmo de Brzezinski et al. 95 89
6.1 Ações da computação básica 97
6.2 Requisições de objetos 98
6.3 Requisições de objetos 99
6.4 As ações básicas aumentadas 102
6.5 Algoritmo de *deadlock* sobreposto ao algoritmo 6.4 104
6.6 Algoritmo de detecção de terminação 105

Capítulo 1

Introdução

Esta dissertação focaliza dois problemas fundamentais da computação distribuída: o problema do *deadlock* e da terminação. O problema do *deadlock* ocorre nos mais diferentes contextos e analogias podem ser feitas em relação à vida cotidiana onde uma parte faz o papel de processo e a outra são os recursos envolvidos. Como exemplo, podem-se citar negociações (de paz, reajuste de salários, etc.) que chegam a um impasse. As conversas podem estar bloqueadas indefinidamente enquanto ambas as partes não cedem nas reinvidicações. Outro exemplo comum é o impasse no trânsito como ilustrado na Figura 1.1. Existe uma espera circular na qual os carros representam os processos e as vias do cruzamento são os recursos sendo controlados e/ou requisitados.

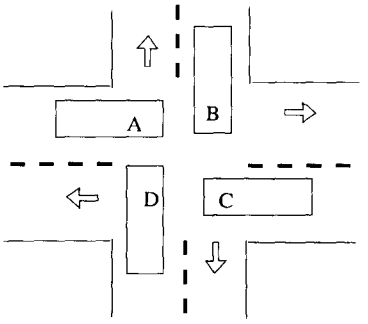


Figura 1.1: Exemplo de um *deadlock* no trânsito.

O problema da detecção da terminação de uma computação não existe em sistemas centralizados, pois ela pode ser facilmente identificada em um único ponto no fluxo de controle. Infelizmente, no caso distribuído, a terminação não é tão trivial de ser detectada, pois mesmo que um processo alcance uma configuração terminal, ele não tem conhecimento sobre os estados dos demais processos e dos canais de comunicação.

Esses dois problemas citados apresentam uma característica comum muito interessan-

te: a propriedade de **estabilidade** ou **persistência**, ou seja, uma vez que um sistema está em *deadlock* ou tenha alcançado a terminação, essas situações persistirão até que uma interferência externa as detectem e as tratem.

Na verdade, a semelhança que existe entre os dois é muito maior, ao ponto de um poder ser modelado no outro, conforme será visto no Capítulo 6. Considere a seguinte analogia: associando mensagens da aplicação aos recursos e a configuração terminal de cada processo à requisição de mensagens (recursos) para os demais processos do sistema, pode-se observar que a terminação do sistema implica na formação de um *deadlock* onde cada processo espera pelo restante. Com a mesma facilidade é feito o caminho inverso da modelagem.

A tantos pontos em comum, junta-se mais um: ambos os problemas têm sido exaustivamente estudados e a literatura compreende algoritmos do final da década de 70 até os dias de hoje. Analisando os trabalhos propostos, verifica-se que o projeto de um algoritmo distribuído é uma tarefa complexa e não raro conduz a soluções incorretas que chegaram, inclusive, a ter a sua correção "provada".

Esta dissertação concentra-se nos seguintes pontos: na apresentação de uma visão em largura das abordagens propostas para a detecção do *deadlock* distribuído e para a detecção da terminação distribuída e na proposição de um método que derive soluções para o problema da terminação distribuída a partir de algoritmos para a detecção do *deadlock*, preservando as propriedades de segurança e progresso da solução original.

1.1 Organização do Trabalho

O trabalho está organizado da seguinte forma. O restante da introdução apresenta definições e conceitos que serão utilizados por toda a dissertação. O próximo Capítulo descreve os conceitos da teoria da terminação, no Capítulo 3 os conceitos apresentados anteriormente são ilustrados em alguns algoritmos. O Capítulo 4 discute o *deadlock*, suas caracterizações e modelos que o classificam. No Capítulo 5 são apresentados algoritmos de detecção do *deadlock* distribuído. O Capítulo 6 apresenta o mecanismo de derivação de um algoritmo de detecção de terminação a partir de um algoritmo de detecção de *deadlock*. Finalmente no Capítulo 7 são apresentadas as conclusões e sugestões de futuros trabalhos.

1.2 Modelo do Sistema

Os sistemas distribuídos diferem dos centralizados em três aspectos essenciais: ausência de conhecimento do estado global, ausência de um relógio global e não determinismo. A

combinação dessas três características faz com que o projeto de algoritmos distribuídos seja uma tarefa intricada.

O estado do sistema muda em passos discretos chamados **eventos**. Além dos eventos comuns (eventos internos) um processo realiza os eventos de **envio** e **recebimento** para interagir com os demais processos. No restante deste trabalho, o termo evento é utilizado para essas duas primitivas de comunicação e, quando necessário, os demais eventos serão tratados especificamente como eventos internos.

Os processos comunicam-se por troca de mensagens que podem ser transmitidas de forma **síncrona** ou **assíncrona**. Uma troca de mensagem é dita síncrona quando os eventos de envio e recebimento estão coordenados, ou seja, um processo somente envia uma mensagem quando o destinatário está pronto para recebê-la. A ênfase deste trabalho é dada em algoritmos para sistemas assíncronos.

A comunicação entre os processos é determinada também pelos canais de comunicação, que podem ser: **confiáveis**, quando eles não corrompem, não duplicam e nem consomem uma mensagem; e **FIFO**[1], quando preservarem a ordem de envio das mensagens. Os algoritmos apresentados neste trabalho, supõem canais de comunicação confiáveis, entrega das mensagens em um tempo finito, mas não necessariamente FIFO.

1.2.1 Propriedades do Sistema

Dado um determinado algoritmo distribuído, é necessário demonstrar que esse algoritmo é uma solução correta para o problema que ele se propõe resolver. O problema especifica quais propriedades um algoritmo deve ter e a solução deve mostrar que possui tais propriedades.

As propriedades, de uma forma geral, podem ser classificadas em um dos dois requisitos: de **segurança** (*safety*) ou de **progresso** (*liveness*). O requisito de segurança impõem que uma certa propriedade deve valer para cada execução em *toda* configuração alcançada naquela execução. Um requisito de progresso impõem que uma certa propriedade deve valer para cada execução em *alguma* configuração alcançada naquela execução [Tel94].

A propriedade de segurança de um algoritmo é informalmente colocada como "Asserção P é sempre verdadeira". A propriedade de progresso é formulada como "Asserção P será verdadeira em um tempo finito".

Além desses requisitos existem outros. Alguns exemplos bastante conhecidos são a propriedade de tolerância à falhas e a propriedade de assimetria de processos.

[1] Em inglês *First In First Out*.

1.2.2 Representação em diagrama temporal

Uma execução de um sistema distribuído pode ser representada em um diagrama espaço-temporal como apresentado na Figura 1.2. No diagrama, uma linha horizontal representa o eixo temporal da execução em cada processo, os pontos são os eventos (de envio, recebimento e interno) e as setas indicam a transmissão de uma mensagem, orientada do evento de envio para o respectivo recebimento.

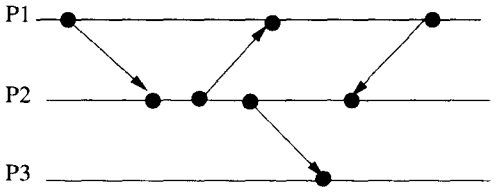


Figura 1.2: Diagrama temporal

1.3 Ordenação causal e Estado Global

Esta seção apresenta resumidamente os resultados de [CL85] relevantes a esta dissertação.

Denota-se o conjunto de eventos por E. O **estado local** de um processo p consiste na história de todos os eventos que ocorreram em p. Seguindo Lamport [Lam78], é definido ordem parcial $\preceq \subseteq E \times E$ como segue:

Definição 1 *Seja $e_1, e_2 \in E$. Então $e_1 \preceq e_2$ (e_1 precede e_2) se*

1. *ou e_1 e e_2 estão no mesmo processo p, e e_1 ocorreu antes em p que e_2;*
2. *ou e_1 é um evento de envio e e_2 é o recebimento correspondente;*
3. *ou $(\exists e' \in E : e_1 \preceq e' \wedge e' \preceq e_2)$.*

O item 1 da Definição 1 diz que os eventos dentro do mesmo processo são totalmente ordenados. No segundo item é descrito o fato que uma mensagem somente é recebida depois de ser enviada. E, por último, tem-se a transitividade da relação $\preceq$.

A história de um sistema e sua relação de precedência pode ser representada por um diagrama espaço-temporal como representado na Figura 1.3.

Definição 2 *Um* **corte** *c de E é uma partição de E em dois conjuntos P_c e F_c, denotando passado e futuro respectivamente.*

Definição 3 *Um corte é* **consistente** *se F_c é fechado em $\preceq$.*

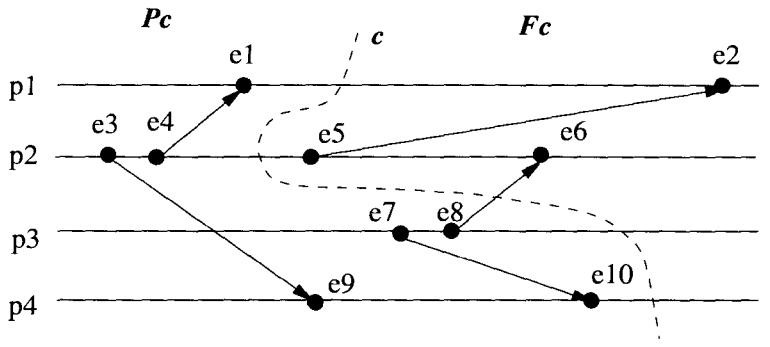


Figura 1.3: Diagrama de precedência

Um corte consistente define um estado consistente. De forma intuitiva pode se dizer que um corte consistente é aquele que não contém em F_c um evento de envio cujo recebimento correspondente está em P_c.

Olhando novamente o exemplo da Figura 1.3, vê-se que $P_c = \{e_1, e_3, e_4, e_7, e_8, e_9, e_{10}\}$ e $F_c = \{e_2, e_5, e_6\}$. Além disso, sabe-se que c é um corte consistente já que F_c é fechado em $\preceq$. Agora é necessário estender a relação $\preceq$ para estados consistentes.

Definição 4 *Seja S_1, S_2 estados consistentes. Então $S_1 \preceq S_2$, se $P_{S_1} \subseteq P_{S_2}$.*

Definição 5 *Seja S um estado consistente e $\mathbf{e} \in E$, tal que $P_S \bigcup \{\mathbf{e}\}$ defina um estado consistente S'. Então $S \vdash S'$ (S' é* **alcançável** *a partir de S).*

Definição 6 *Uma sequência de eventos $\sigma = (e_1, e_2, \cdots, e_n)$ é uma* **escala** *se $S \vdash^{e_1} S_1 \vdash^{e_2} \cdots \vdash^{e_{n-1}} S_{n-1} \vdash^{e_n} S'$.*

Essa fórmula é expressa resumidamente como $S \vdash^{\sigma} S'$

Segue o resultado de [CL85]:

Lema 7 $S \preceq S' \Rightarrow \exists escala\ \sigma :: S \vdash^{\sigma} S'$

Esse é o resultado fundamental utilizado em alguns algoritmos distribuídos.

1.4 Representação em grafos

A estrutura de um sistema distribuído é frequentemente modelada em um grafo $G = (V, E)$, onde os vértices representam os processos e as arestas são os canais de comunicação entre pares de processos. De acordo com o tráfego que permite, um canal pode ser **bidirecional** ou **unidirecional**, sendo que neste último caso, é representado por uma aresta orientada.

Os grafos modelam tanto a estrutura física da rede, como a topologia lógica sobre a qual os processos organizam-se durante a execução do algoritmo distribuído. Como exemplo, tem-se a passagem de ficha por um anel, a redução dos nós de uma árvore, o controle centralizado de uma estrela, etc.

Uma topologia pode ser **estática** se permanece fixa durante a computação ou **dinâmica** quando as mudanças ocorrem durante a computação.

1.5 Complexidades

Para comparar diferentes algoritmos para o mesmo problema verifica-se a quantidade de recursos consumidos. Para cada tipo de recurso tem-se uma medida diferente:

Complexidade de mensagem: número de mensagens trocadas na execução do algoritmo.

Complexidade de *bit*: é levado em consideração nessa medida o tamanho das mensagens transmitidas.

Complexidade de espaço: quantidade de memória necessária para a execução do processo.

Complexidade de tempo: a complexidade de tempo de um algoritmo assíncrono é o tempo máximo da execução calculado sob a hipótese que a computação local de um processo tem custo zero e a transmissão de cada mensagem é de uma unidade de tempo.

Capítulo 2

A Teoria da Terminação

Em sistemas distribuídos, onde os processos comunicam-se somente por mensagens, a detecção de terminação da computação distribuída não é trivial, já que nenhum processo tem conhecimento do estado global e um relógio global, em princípio, não existe.

A terminação é uma propriedade do estado global de uma computação distribuída. Essa propriedade é estável (ou persistente), isto é, uma vez que tal estado é alcançado ele persistirá.

Neste capítulo serão apresentados alguns conceitos e discutida uma classificação para os algoritmos de detecção da terminação.

2.1 O comportamento do sistema

Além das características do modelo suposto no Capítulo 1, como comunicação exclusiva por troca de mensagens, entrega das mensagens em um tempo arbitrário e finito, comunicação assíncrona e canais não necessariamente FIFO, é necessário que agora introduzir o comportamento do sistema para esse problema:

1. Em qualquer tempo, um processo encontra-se passivo ou ativo.
2. Somente processos ativos enviam mensagens.
3. Em qualquer momento um processo pode mudar de ativo para passivo.
4. Ao receber uma mensagem, um processo muda de passivo para ativo.

Supõe-se que os processos tornar-se-ão passivos em um tempo finito. Por causa disso, a duração do intervalo de tempo em que o processo permanece ativo é irrelevante ao problema. Como o tempo de entrega das mensagens é arbitrário, todas as mensagens que são enviadas dentro deste intervalo também poderiam ter sido enviadas no início do

período de atividade, só que umas demorariam mais do que as outras para chegarem. Veja as representações na Figura 2.1.

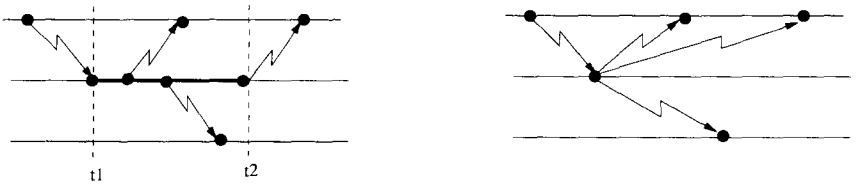


Figura 2.1: Diagramas de tempo

2.2 O problema

Francez [Fra80] deu proeminência ao problema da terminação distribuída e, desde então, várias soluções foram sugeridas baseadas em um modelo de comunicação síncrono. Além da discussão dos algoritmos que trabalham no modelo síncrono, esta dissertação estuda o problema em um contexto mais geral da comunicação assíncrona.

No modelo suposto, a **condição da terminação** é assim definida: *nenhum processo está ativo e nenhuma mensagem está em trânsito.*

Para que um ou mais processos sejam capazes de detectar se a condição de terminação é valida, é preciso sobrepor à computação básica uma computação de controle. Quando um ou mais processos detectam a condição de terminação diz-se que o sistema alcançou o estado final ou que está **quiescente**; como alguns pesquisadores preferem denominar.

Uma estratégia bastante simples para descobrir se alguma mensagem ainda não foi recebida é analisada em [Mat87]. Ela consiste de um mecanismo que faça com que cada processo conte as mensagens da aplicação enviadas e recebidas armazenando, respectivamente nos contadores E_i e R_i.

O valor desses contadores é informado por demanda ao processo iniciador da detecção de terminação. Obtendo esses valores de cada processo, o iniciador pode acumular os contadores. A Figura 2.2 mostra os instantes em que os processos receberam as mensagens de controle da detecção e informaram os valores dos seus contadores ao iniciador, procedimento que na figura é simbolizado pelos pontos mais claros. Estes pontos são ligados por uma linha que representa uma **onda de controle** que induz um corte de tempo.

Infelizmente, esse algoritmo simples não é correto, porque mensagens "vindas do futuro" corrompem o valor dos contadores (Figura 2.3, os eventos e_3 e e_2). Isso mostra que o corte é inconsistente, isto é, leva a um resultado incorreto já que indica que uma mensagem foi enviada e uma foi recebida, apesar da computação não estar terminada.

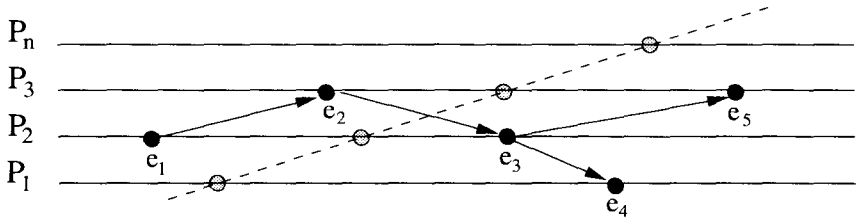


Figura 2.2: Onda de controle

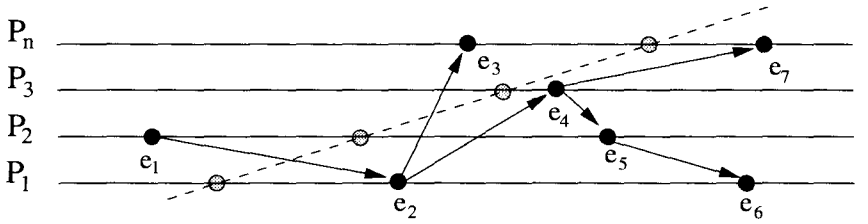


Figura 2.3: Contra-exemplo do método dos contadores

2.3 Algumas classes de algoritmos

Um algoritmo de detecção de terminação deve atender aos seguintes critérios [Tel94]:

Não interferência A detecção não deve influenciar a computação básica.

Segurança Se a terminação é detectada, então ela existe.

Progresso Se a condição de terminação é válida, então, em algum tempo finito ela será detectada.

No primeiro trabalho que trata da terminação [Fra80], Francez propõe uma solução que não atende ao primeiro critério. A interferência se dá, por exemplo, quando a solução propõe o congelamento da aplicação, desabilitando todos os eventos para que se possa verificar a terminação do sistema. Os critérios mencionados excluem esta solução e requerem que os algoritmos de detecção sejam *"on-the-fly"*, isto é, trabalhem enquanto a computação básica prossegue.

Os algoritmos de detecção de terminação podem ser classificados de acordo com os métodos utilizados para corrigir a situação de cortes [Mat87]:

- Detectando cortes inconsistentes, isto é, detectando a chegada de mensagens vindas do futuro e reiniciando o algoritmo mais tarde.
- Passando outras informações nas mensagens, não deixando a detecção a cargo somente de contadores.

- Impondo um controle mais rígido sobre as mensagens da aplicação, seja através de um congelamento do sistema, ou enviando um reconhecimento para cada mensagem recebida ou, ainda, seguindo uma ordenação FIFO. Apesar de existirem abordagens que detectam corretamente a terminação do sistema utilizando esse controle, elas não obedecem ao critério de não-interferência ou restringem seriamente o modelo geral da computação distribuída adotado.

Os algoritmos de detecção de terminação podem adotar as mais variadas estratégias para lidar como o problema, mas de uma forma geral eles podem ser classificados de acordo com o número de fases de execução. Partindo do estado quiescente (processos passivos, canais livres) é contado o número de *rounds* ou fases necessárias à detecção. O que verificou-se nesse estudo foi uma divisão bem nítida de algoritmos bifásicos e monofásicos.

2.3.1 O algoritmo bifásico e variantes

Esses algoritmos baseiam-se no mecanismo de detecção através de duas ondas de controle: uma para verificar a atividade do sistema e uma segunda para ratificar essa informação.

Os valores dos contadores que mantêm o número de mensagens recebidas e das enviadas podem ser corrompidos se existe atividade à direita da primeira onda, como visto na Seção 2.2. Para que o mesmo não ocorra, são utilizadas variáveis de controle (*flags*) para detectar tal atividade, as quais são inicializadas na primeira onda e têm seu valor alterado pelos processos quando eles recebem mensagens. Mattern [Mat87] denomina esse esquema de "algoritmo cético".

A segunda onda basicamente verificará se as variáveis tiveram modificação que, sendo confirmada, será suposta a corrupção dos contadores de mensagens. A desvantagem do esquema é que geralmente pelo menos duas ondas serão necessárias para se detectar a terminação.

O uso de diferentes topologias de controle deu origem a algumas variações desse princípio. Existem algoritmos com a topologia lógica de uma estrela, onde o iniciador pergunta para todos os outros processos individualmente. Outras implementações são baseadas em um anel e outras configurações de controle obedecem à uma árvore geradora.

Os algoritmos que utilizam a idéia da árvore geradora podem ser subdivididos em dois grupos: aqueles que usam árvores fixas e os que as constroem dinamicamente. As árvores estáticas são usadas em algoritmos baseados em comunicação síncrona, como [Top84, Fra80, FR82]. As árvores geradoras são também construídas dinamicamente [LR86] ou implicitamente através do algoritmo de eco [Cha82]. Esse algoritmo é dividido em duas fases e utiliza o seguinte mecanismo: uma fase de expansão e outra de retração.

O mecanismo de eco é também conhecido como **computação difusa**. Para o caso da terminação, esse esquema foi empregado de forma muito interessante e alguns trabalhos relevantes onde há a sua utilização são [DS80, Cha82] (originários do princípio) e [MC82, Mat87].

A filosofia do "ver para crer" aparece nitidamente em algoritmos que percorrem o sistema organizado logicamente em anel, como também nas duas fases do mecanismo de eco. As fases de expansão e de retração desse mecanismo podem ser utilizadas perfeitamente como as duas ondas necessárias ao algoritmo "cético", vide Seção 3.1.3. O mecanismo de eco permite uma implementação do algoritmo cético sem que apresente um gargalo tão sério quanto o da topologia lógica em estrela.

Esse mecanismo de computação difusa originou tanto algoritmos de detecção de terminação como soluções para o problema da detecção do *deadlock* distribuído. O mecanismo produziu soluções desde a sua criação até os dias de hoje [KS97].

2.3.2 Utilização de *timestamps*

Nesse tipo de algoritmo, cada processo é equipado com um relógio lógico local representado por um contador iniciado em zero. Uma onda de controle é iniciada no tempo i, acumula os valores dos contadores e "sincroniza" os relógios locais, atualizando-os em $i+1$. Dessa forma a onda de controle separa o passado do futuro.

Um processo tem conhecimento que recebeu uma mensagem do futuro, isto é, uma mensagem que cruzou a onda da esquerda para direita e adulterou os contadores devido ao *timestamp* da mensagem ser maior que o relógio local (Figura 2.4). Depois que tal mensagem é recebida, a onda de controle atual é anulada quando chega ao processo P_3.

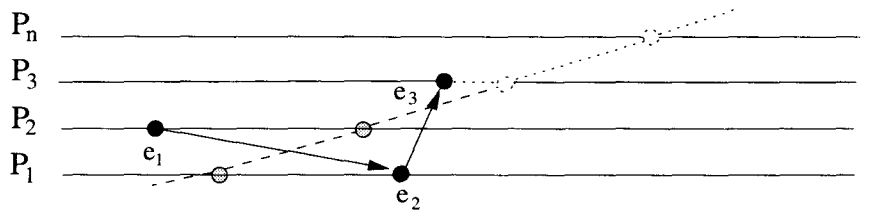


Figura 2.4: Exemplo do método de utilização de *timestamps*

2.3.3 Contadores vetoriais

Mattern [Mat87] propôs um algoritmo que mantém um contador para cada destino. Para o processo p o contador da mensagem é um vetor $mc_p[P]$, P o número de processos. Quando p envia para q, p atualiza o contador para $mc_p[q] := mc_p[q] + 1$ e quando p recebe uma mensagem, ele faz $mc_p[q] := mc_p[q] - 1$.

A ficha também transporta um vetor de contadores e quando ela está sendo mantida por p, mc_p é adicionado ao vetor da ficha e reinicializado com zero em todos os elementos. A terminação é detectada quando todos os contadores do vetor na ficha são iguais a zero.

Uma vantagem da estratégia é que a terminação é detectada mais rápido, mais precisamente em um *round* depois que a terminação ocorre. Outra vantagem é que quanto mais o sistema demorar a alcançar a condição de terminação, mais vezes a ficha é suspensa. Isso reduz o número de mensagens de controle trocadas pelo algoritmo.

A maior desvantagem dos algoritmos baseados em contadores vetoriais é que a ficha carrega uma grande quantidade de informação que deve ser transmitida em cada mensagem de controle, porém essa desvantagem não é tão séria se o número de processos for pequeno.

Outra desvantagem desse princípio é o conhecimento sobre a identidade dos outros nós. Inicialmente cada processo tem que saber pelo menos a identidade dos seus vizinhos (para incrementar o contador corretamente), e as outras identidades serão conhecidas durante a computação.

2.3.4 Algoritmos de contagem de canal

Na abordagem baseada em contadores vetoriais cada mensagem é contada duas vezes: uma pelo emissor e outra pelo receptor.

O método de contagem de canais pode ser visto como um refinamento do método de contadores vetoriais. O receptor registra o emissor e mantém o número de mensagens recebidas de cada canal, usando um contador apropriado. O princípio utilizado é de uma árvore construída dinamicamente [Kum85].

Cada processo P_j tem $2n$ contadores, n para mensagens que chegam e n para as que saem. O contador $in_C_j[i]$ é incrementado quando é recebida uma mensagem do processo P_i e $out_C_j[k]$ é incrementado ao enviar uma mensagem para o processo P_k. Cada processo informa ao iniciador o valor dos contadores sempre que isso é solicitado. O inciador chega à conclusão que o sistema terminou quando $in_C_j[i] = out_C_i[j]$ para todo i, j. Note que se existissem mensagens no canal, implicaria que a condição exigida seria válida para alguns i, j e não para todo i, j.

O método de contagem do canal não apresenta muitas vantagens sobre o método de contadores vetoriais no caso das topologias de anel e de estrela. A quantidade de informação de controle é maior, já que existem dois contadores por aresta em um grafo de comunicação, ou no caso completo, n^2 contadores.

Esse método foi combinado com o algoritmo de eco [Cha82] no trabalho de Mattern [Mat90] (Seção 3.1.3).

2.3.5 Algoritmos de recuperação de crédito

Nessa abordagem é atribuído um valor de crédito para cada processo e para cada mensagem.

Ao iniciar a computação básica, o nó iniciador distribui um crédito de valor total 1, dividindo-o em partes iguais pelas mensagens transmitidas aos processos. Ao receber uma mensagem, o processo incorpora o crédito recebido ao valor que mantém. Um processo ativo possui a variável crédito com valor maior que zero e uma mensagem de ativação carrega um crédito também maior que zero. É garantido pelo algoritmo que, em qualquer momento da execução, a soma de todos os créditos do sistema, contidos nos processos e nas mensagens, é igual a um.

Após o início da computação básica, o iniciador começa a recuperar os valores enviados da seguinte forma: um processo ao tornar-se passivo, devolve ao iniciador a soma dos créditos que mantém. O processo iniciador é responsável pela totalização desses valores. Quando há a recuperação de todos os créditos (a soma é igual a um), o iniciador anuncia a terminação do sistema.

Essa estratégia utiliza o mecanismo de *piggybacking*, ou seja, a informação de controle é incluída na mensagem básica. Na prática, um problema surge quando os créditos são armazenados em um número fixo de *bits*.

Quando surge uma situação de divisão do menor crédito possível, Tel [Tel94] sugere uma intervenção do iniciador que retira uma certa quantidade q (a variável crédito pode se tornar negativa), envia para o processo que está lidando com o crédito muito pequeno, para que, assim, a divisão possa ser feita e a transmissão transcorra normalmente.

Ao invés de adotar essa solução que viola o princípio da não-interferência, já que irá bloquear a computação básica até que o aumento do crédito seja resolvido, nós sugerimos a utilização de dois campos na mensagem para armazenar o crédito: um de numerador e outro de denominador. Dessa forma, um pequeno espaço de armazenamento pode ser utilizado para representar um crédito suficientemente pequeno. Essa solução foi obtida na implementação do algoritmo de Kshemkalyani e Singhal [KS94] que utiliza a recuperação de crédito. Além de preservar a não-interferência, não é necessária a manipulação de números em ponto flutuante e não existem erros de arredondamento.

O princípio da abordagem de recuperação de crédito foi inspirado em um algoritmo de detecção de lixo distribuído [Bev87]. Na verdade, o algoritmo de Mattern é uma derivação da solução proposta por Bevan, aplicando as regras de derivação de um algoritmo de terminação distribuída a partir de esquemas para coleta de lixo distribuído propostas em [TTvL87].

Um outro exemplo de algoritmo que utiliza o princípio da recuperação de crédito é apresentado em [Hua89] e utilizado no algoritmo de detecção de *deadlock* distribuído da Seção 5.3.2.

2.4 Comentários

Reunindo os pontos apresentados de cada classe de algoritmos, pode-se sugerir a seguinte caracterização dos algoritmos para detecção da terminação distribuída:

- Os algoritmos "céticos":
  1. Cada processo deve ser visitado pelo menos duas vezes.
  2. Várias topologias são possíveis, por exemplo, anel, estrela ou árvore. Os anéis são simétricos, porém são sequenciais. As estrelas e árvores permitem uma execução paralela. Nos casos das topologias de estrela e de árvore estática, um processo específico é responsável pelo início da detecção.
  3. Qualquer processo pode iniciar o algoritmo, possibilitando que várias ondas de detecção estejam ativas concorrentemente. Como consequência, a quantidade de informação de controle é aumentada. Se por um lado o uso de *flags* diminui essa quantidade de informação, por outro isso resulta em um algoritmo mais complexo para gerenciar as ondas concorrentes.
  4. Não existe um limite superior para o número de mensagens de controle. É preciso escolher um tempo arbitrário para que o algoritmo recomece depois de uma tentativa fracassada.
- Os algoritmos baseados em *timestamp*:
  1. É necessário que se visite o nó somente uma vez.
  2. Os comentários feitos anteriormente sobre topologias e limite do número de mensagens de controle também aplicam-se aqui.
  3. Cada ação básica deve ser aumentada com um *timestamp*.
- O método de contadores vetoriais:
  1. Também necessita de uma única visita a cada nó para detectar a terminação.
  2. Mensagens básicas não precisam ser trocadas, como é feito na estratégia baseada em *timestamp*.
  3. Cada processo deve ter o conhecimento do tamanho da rede, ou pelo menos de quantos vizinhos possui.
  4. Em algumas implementações, o teste da terminação permanece ativo até que a computação básica acabe, ou seja, nunca é retornada uma resposta negativa.
- O algoritmo de contagem de canal:

1. Os itens do método anterior aplicam-se nesse caso.

- A abordagem de recuperação de crédito:

1. Detecta a terminação em uma fase.
2. Direta ou indiretamente deve ser formada uma estrela centrada no nó iniciador.
3. Cada mensagem é aumentada com o campo do crédito.
4. Nunca é retornada uma resposta negativa. Enquanto a totalização não chega a 1 no iniciador, ele espera pelo retorno dos créditos.

No próximo Capítulo esses conceitos e abordagens serão utilizados em alguns exemplos de trabalhos propostos na área de detecção da terminação distribuída.

Capítulo 3

Algoritmos de Detecção de Terminação

Neste capítulo são apresentados alguns algoritmos de detecção da terminação distribuída que estão divididos de acordo com o número de fases necessárias à detecção.

Como descrito no capítulo anterior, considera-se que o sistema alcançou a terminação e a partir desse ponto são contados os *rounds* ou fases do algoritmo de detecção até o momento do anúncio da terminação do sistema.

3.1 Algoritmos bifásicos

Os algoritmos que apresentam no mínimo duas fases para detectar a terminação do sistema são denominados de algoritmos "céticos". Esse termo é bastante adequado para descrever de maneira geral o mecanismo utilizado: uma primeira fase percorre o sistema, tendo obtido um veredito favorável à terminação, o sistema é percorrido uma segunda vez para certificar que não houve alteração do estado entre o final do primeiro percurso e o veredito.

3.1.1 O algoritmo de Dijkstra e Scholten 80

Esse algoritmo utiliza a idéia de computação difusa. Os canais de comunicação entre os processos permitem que um grafo orientado seja definido, onde os nós são os processos. Um nó especial é chamado **ambiente** e não possui pai (nó sem arestas de entrada).

Uma computação difusa é iniciada quando o ambiente envia por conta própria uma mensagem para os seus sucessores. Supõe-se que isso é feito somente uma vez. Depois da recepção da primeira mensagem, um nó interno é livre para enviar mensagens para os seus sucessores. A essa característica dá-se o nome de **computação difusa**.

Dijkstra e Scholten adaptaram esse esquema para o problema da terminação distribuída da seguinte forma: a terminação é verificada de tal forma que o processo ambiente seja informado quando a computação termina e não quando há uma pseudo-terminação.

O Algoritmo

Todos os processos estão passivos inicialmente, exceto o ambiente que envia mensagens para os seus descendentes no grafo G. A idéia básica é associar a cada mensagem enviada pelo processo P_i para o processo P_j uma mensagem especial que faz uma espécie de reconhecimento do envio. Essa mensagem é chamada de **signal**.

O envio de um *signal* está sujeito a uma regra que expressa a propriedade de terminação de maneira que, quando o processo ambiente recebe os reconhecimentos de todas as mensagens que enviou, conclui-se que a computação terminou.

Para que se consiga isso é utilizado o conceito de **déficite**. A cada aresta, isto é, a cada canal de comunicação unidirecional entre pares de processos, é associado um número que representa a diferença entre mensagens transmitidas e os *signals* retornados.

São utilizados dois contadores para formular uma regra que controlará o envio de mensagens e de sinais de reconhecimento de maneira que seja garantida a detecção da terminação. A cada processo são associadas duas variáveis, C e D, que representam, respectivamente, o somatório dos déficites das arestas de entrada e o somatório dos déficites das arestas de saída.

Esses contadores são manipulados como descrito no pseudocódigo do Algoritmo 3.1.

Exemplo

Nessa seção será ilustrada a estratégia apresentada no Algoritmo 3.1.

A Figura 3.1 representa o sistema com os nós e a relação entre eles que vai ser formada a partir da transmissão de mensagens básicas. No grafo, têm-se arestas representando as transmissões. A árvore geradora orientada está enraizada no nó ambiente e será construída durante o envio das mensagens. O nó que enviou a primeira mensagem recebida por um nó i é considerado o pai de i.

O comportamento do sistema é representado passo a passo na Figura 3.2.a - .i, onde é utilizada a seguinte representação: as setas cheias representam as mensagens da aplicação (básicas) e as setas pontilhadas são mensagens da detecção (controle).

O processo ambiente, o nó A, inicia enviando mensagens para 1, 2 e 3 (Figura 3.2.a). Na sequência, há uma sucessão de retransmissões de mensagens e envio de respostas, representada nas letras b-i). No último quadro da Figura 3.2, o nó 3 envia a resposta que estava faltando para que a configuração inicial do nó ambiente fosse alcançada e, dessa forma, detectando a terminação do sistema.

```
(S1)when received (mensagem, emissor)
        do
                if (C = 0) then
                        pai := emissor;
                else
                        predecessores := predecessores + emissor;
                end if;
                C := C + 1;
        end do;

(S2) when received (signal, emissor)
        do
                D := D - 1;
        end do;

(S3) before P_i send (mensagem, i) to j
        do
                if (C ≠ 0) then
                        D := D + 1;
                        send (mensagem, i) to j;
                end if;
        end do;

(S4) before P_i send (signal)
        do
                if (C > 1 or (C = 1 and D = 0)) then
                        if (C = 1)
                                then send (signal,i) to pai;
                                else
                                        pred := um elemento de (predecessores);
                                        predecessores := predecessores - {pred};
                                        send (signal,i) to pred;
                        end if;
                        C := C - 1;
                end if;
        end do;
```

Algoritmo 3.1: Descrição do Algoritmo de Dijkstra e Scholten 80

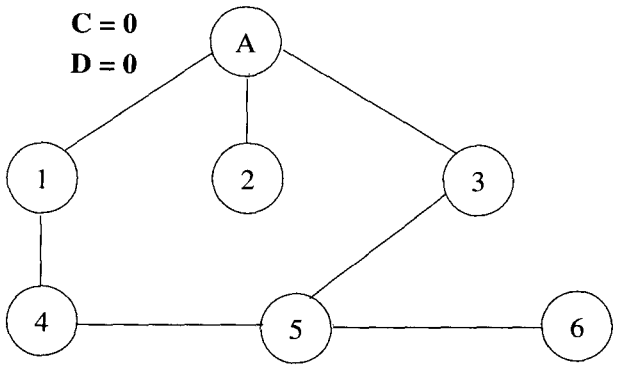


Figura 3.1: Estado incial do sistema

Desempenho

Esse algoritmo apresenta um equilíbrio entre o número de mensagens da aplicação e o número de mensagens de controle: para cada mensagem básica enviada de p para q, é enviada uma mensagem de controle de q para p. Portanto, se a computação básica troca M mensagens, a detecção gastará extamente M mensagens.

Correção

Quando o sistema alcança o estado terminado, nenhum processo pode enviar mensagens básicas ou *signals*, logo pode-se afirmar o seguinte sobre os contadores de mensagens presentes em cada processo, exceto o ambiente: $C > 1$ ou $(C = 1$ e $D = 0)$. Para o processo ambiente, a raiz da árvore, os valores são: $C = 0$ e $D \geq 0$.

Então quando a computação está terminada, $\forall P_i : C_i \leq D_i$. Como não há nenhum tipo de mensagem em trânsito, o somatório dos C_i's é igual ao somatório dos D_i's. Portanto, $\forall P_i : C_i = D_i$ e o processo ambiente tem $C = D = 0$. A configuração inicial da raiz é alcançada.

Para provar que o algoritmo não detecta pseudo-terminações, é útil observar algumas colocações encontradas em [Tel94]. O algoritmo de detecção da terminação mantém uma árvore $T = (V_T, E_T)$ que obedece às seguintes propriedades:

1. Ou T é vazia, ou T é uma árvore orientada com raiz no nó ambiente.

2. O conjunto V_T inclui todos os processos ativos e todas as mensagens em trânsito.

Quando o ambiente detecta a terminação $(C = D = 0)$, tem-se que ambiente $\notin V_T$. Pela primeira propriedade, T é vazia e, pela segunda propriedade, a condição de terminação é válida.

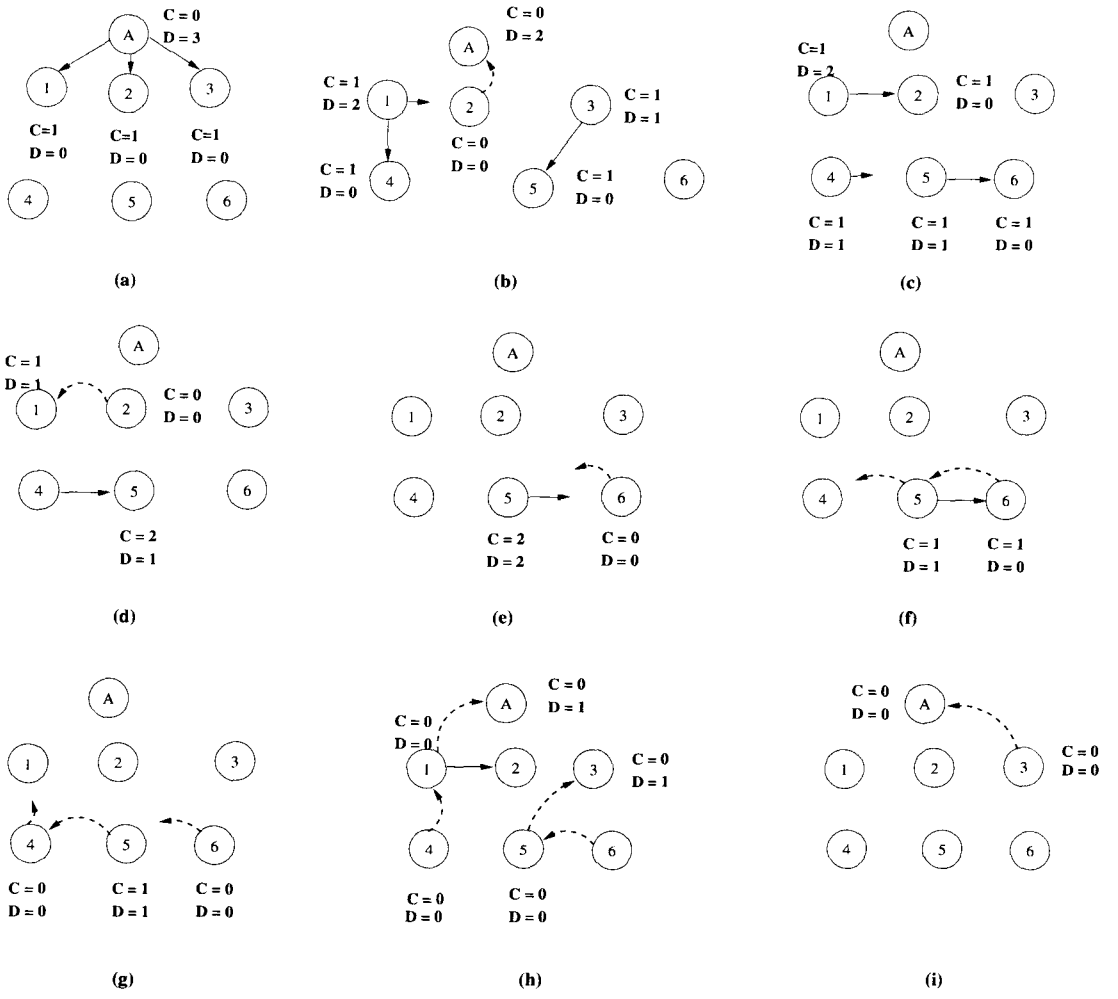


Figura 3.2: Exemplo de uma execução do Algoritmo de Dijkstra e Scholten

3.1.2 O algoritmo de Dijkstra, Feijen e Van Gasteren 83

No modelo síncrono de troca de mensagens, o esquema de contadores não é necessário, já que uma transmissão síncrona não pode cruzar as duas ondas, mas as variáveis para detectar o corte inconsistente ainda são necessárias. Alguns algoritmos com base nesse princípio são encontrados em [DFG83, FR82, Top84]. Nessa seção é visto com maiores detalhes o Algoritmo proposto por Dijkstra *et al.*

Os processos estão dispostos em anel na ordem $P_0, P_1, \cdots, P_{n-1}, P_0$ e a comunicação é feita unidirecionalmente de P_i para P_{i-1}. O procedimento de detecção da terminação será iniciado por um único processo, o qual é chamado de P_0.

O Algoritmo

A idéia é simples e consiste em uma ficha circulando pelo anel. A primeira transmissão da ficha é feita por P_0 na ocasião em que ele se torna passivo. Essa ficha é recebida por P_{n-1} que a retransmitirá somente quando estiver passivo. Esse procedimento se sucede por todo o anel.

O problema surge quando um processo passivo P_i recebe a ficha, a repassa para P_{i-1} e depois P_i é reativado por uma mensagem vinda de um processo que ainda não tinha sido visitado pela ficha. Para lidar com isso e, assim evitar pseudo-terminações, é dada uma cor à ficha.

Inicialmente a ficha é branca e a sua cor é mudada de acordo com regras que serão descritas a seguir. Se uma ficha percorre todo o anel e chega ao ponto inicial com a cor branca, então isso mostra que o sistema chegou à terminação.

Seja a seguinte situação: suponha que a ficha chega branca ao processo P_i, ou seja, os processos que antecedem P_i no anel estão passivos. Além disso, existe um processo P_j ($j < i$, logo não visitado ainda) que envia uma mensagem a outro processo P_k, com $k > j$. Como P_j desconhece a posição da ficha no anel, ele deve registrar a possibilidade de reativação que a mensagem enviada causará.

Exemplo

Nesse exemplo é ilustrada a necessidade de percorer o anel por duas vezes depois que o sistema cessa sua atividade.

Na Figura3.3.a. P_1 registra o envio de uma mensagem básica para P_2 colorindo-se de preto. A mensagem ativa P_2 e depois de tratada o processo torna-se passivo, a partir desse instante todos os processos estão passivos e os canais estão livres de mensagens.

O processo inciador P_0 envia a ficha que está branca para verificar o estado do sistema (Figura 3.3.b). Um processo preto ao receber a visita da ficha, a retransmite transferindo a própria cor para ela e tornando-se branco. Pode-se dizer que a ficha na primeira passagem funciona como uma "esponja" que branqueia os canais e os processos (Figura3.3 letras (c)-(g)).

O segundo percurso verifica que nenhum processo foi ativado, portanto a ficha, que foi enviada branca, retorna da mesma cor ao iniciador que anuncia, então, a terminação do sistema (Figura 3.3.h).

Desempenho

Para cada computação básica que gaste M mensagens, o algoritmo de Dijkstra, Feijen e Van Gasteren gasta, no pior caso, $M(N-1)+N$ mensagens de controle, para N igual ao número de processos. Nessa situação, o último processo a ser visitado envia M mensagens

```
(S1)when received(mensagem)
        do
                estado := ativo;
        end do;

(S2) when sent (mensagem) to j
        do
                if ( my_id < j) then
                        cor_proc := preto;
                end if;
                send (mensagem) to j;
        end do;

(S3) when received (token, cor) from i+1
        do
                tok_pres := true;
                if( my_id = 0 ) then
                        if (cor_proc = branco and cor_tok = branco)
                                then
                                        terminação detectada;
                                else
                                        send (token, branco) to n-1;
                        end if;
                else
                        if (cor_proc = branco)
                                        then send (token, cor) to i-1;
                                        else send (token, preto) to i-1;
                        end if;
                end if;
        end do;

(S4) when sent (token, cor) to i-1
        do
                if (estado = passivo and tok_pres) then
                        if (cor_proc = preto) then
                                cor_tok := preto;
                        end if;
                        tok_pres := false;
                        send (token, cor_tok) to i-1;
                        cor_proc := branco;
                end if;
        end do;

(S5) when waiting (mensagem) or end
        do
                 estado = passivo;
        end do;
```

Algoritmo 3.2: Descrição do Algoritmo de Dijkstra, Feijen e Van Gasteren 83

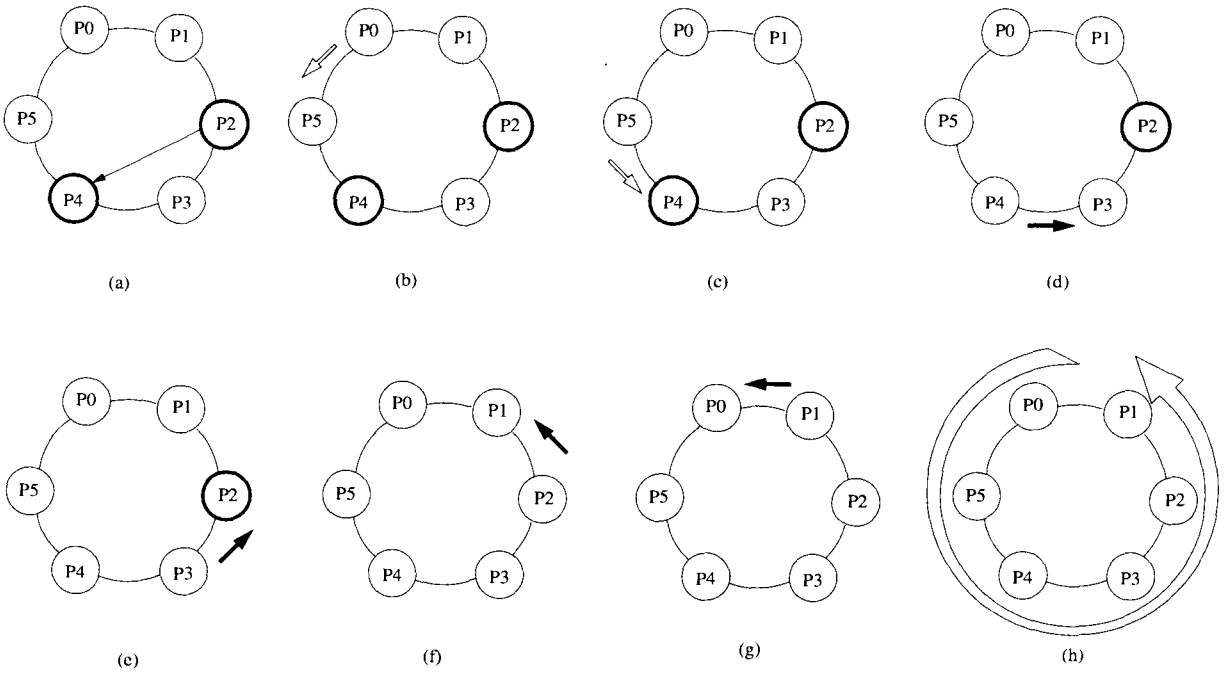


Figura 3.3: Exemplo de uma execução do Algoritmo de Dijkstra. Feijen e Van Gasteren

básicas ao processo que inicia a detecção imediatamente depois que cada ficha parte do iniciador. Esse comportamento acaba invalidando cada uma das M passagens que gastam $N-1$ mensagens de controle cada. A última passagem que gasta N mensagens é a que detecta a terminação.

Correção

Esse algoritmo apresenta as seguintes propriedades [DFG83]:

P_0 : $\forall i, N > i > t$, processo i está passivo. (t é o número do processo o qual mantém a ficha.)

P_1 : $\exists j. t \geq j \geq 0$, o processo j está preto.

P_2: a ficha está preta.

O algoritmo foi projetado de forma que o predicado $P \equiv (P_0 \vee P_1 \vee P_2)$ seja uma invariante do algoritmo. O sistema está terminado quando o processo p_0 é branco e recebe uma ficha branca. Isso implica em $\neg P_2$, pois a ficha é branca. $t = 0$ implica em $\neg P_1$ e o estado de p_0 implica que foi alcançada a condição de terminação, pois todos os processos estão passivos e nenhuma mensagem de ativação foi enviada.

Para mostrar a propriedade de progresso, assume-se que a computação terminou e, a partir de então, os processos recebem e retransmitem a ficha. Ao retornar ao iniciador, a ficha está branca como todos os processos do sistema o que indica que a atividade cessou.

3.1.3 O algoritmo de Mattern 90

Para se detectar se mensagens ainda estão em trânsito, várias contribuições foram dadas, cada uma com diferentes propriedades. O algoritmo desse trabalho baseia-se no princípio de contagem de mensagens.

É um sistema distribuído estático, com processos identificados de maneira distinta e ordenada, além da identificação ser diferente de 0. Os processos são conectados bidirecionalmente formando um grafo de comunicação conexo.

O algoritmo

O princípio desse método é simples e pode ser descrito basicamente em duas ondas: a primeira, onda de *query*, que parte do iniciador em direção às folhas e desliga uma *flag* de estado de cada nó (tornando-os passivos). A outra onda é a onda de *echo* que acumula contadores locais de mensagens e a *flag* de estado. Se o iniciador detecta que o número total de mensagens enviadas é igual ao das recebidas e que a *flag* de comunicação não foi ligada, o que acontece devido à recepção de uma mensagem básica, então ele sinaliza a terminação.

Uma variável inteira S controla o número de mensagens enviadas e recebidas e outra variável binária *comunica* é uma *flag* que indica se o nó está passivo ou ativo.

O tratamento de uma mensagem básica dá-se da seguinte forma:

when received (msg)	**when send** (msg)
$S := S - 1$;	$S := S + 1$;
comunica := **true**;	

Outras variáveis são necessárias ao algoritmo e sua descrição segue agora. N é um contador do número de mensagens de controle recebidas. *pred* contém a identidade do nó predecessor (aquele que envia a primeira mensagem de controle). Para o nó iniciador a identidade de *pred* é 0. A variável *accu* acumula os valores S recebidos como resposta em mensagens eco.

Um processo passivo inicia o algoritmo de eco como descrito no pseudocódigo do Algoritmo 3.3.

```
(S1)when received QUERY(t)
        if ( t > clock )
        then
                clock := t;
                if (pred = 0 and N ≠ |NEIGHBOURS|)
                        then failed;
                end if;
                pred := p;
                if (|NEIGHBOURS| = 1 )
                        then send ECHO(t,S,false) to p;
                        else N := 1;
                end if;
                START:
                        accu := 0; comunica := false;
                send QUERY(clock) to NEIGHBOURS - {pred};

        else
                if (t = clock)
                        then CHECK_COMPLETED();
        end if;


(S2)when received ECHO(t,soma,FAILED)
        do
                if ( t := clock )then
                        accu := accu + soma;
                        comunica := comunica or failed;
                        CHECK_COMPLETED();
                end if;
        end do;


CHECK_COMPLETED()
begin

        N := N + 1;
        if ( N = |NEIGHBOURS| )then
                if ( pred <> 0 )
                        then send ECHO(t,accu+S,comunica) to pred;
                        else if ( (accu + S) = 0 and not(comunica)
                                then terminated;
                                else failed;
end;
```

Algoritmo 3.3: Descrição do Algoritmo de Mattern 90

Exemplo

Esse algoritmo apresenta a idéia da abordagem cética implementada através do algoritmo de eco. A primeira onda é disparada pelo processo iniciador P_1, que se expande até alcançar os nós folhas (Figura 3.4.a e .b). A partir de então a retração verifica o estado dos processos e o valor da *flag comunica*(Figura 3.4.c - .e). Esses resultados serão recebidos pelo iniciador que obterá o resultado da detecção(Figura 3.4.f).

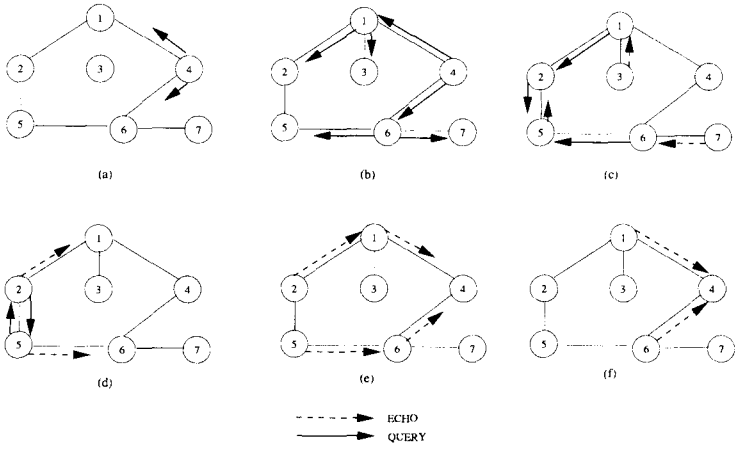


Figura 3.4: Exemplo de uma execução do algoritmo de Mattern 90

Desempenho

Qualquer detecção iniciada após a terminação do sistema, terá duas ondas para verificar o estado do sistema: uma onda de expansão (*query*) percorrerá a árvore no sentido do iniciador para as folhas e a segunda onda de retração (*echo*) no sentido inverso.

No pior caso, o qual o algoritmo trabalha sobre um grafo completo de N vértices, a fase de expansão terá uma mensagem por aresta que liga o iniciador aos seus vizinhos, mais duas mensagens por aresta que une os vértices não iniciadores aos seus vizinhos (exceto o vizinho iniciador). Tem-se o total de $(N-1)+(N-1)(N-2)$ nessa primeira onda. Para fase de retração, serão gastas $N-1$ mensagens vindas dos nós não iniciadores em direção ao iniciador. No total tem-se $N(N-1)$ mensagens, resultando numa complexidade $O(N^2)$.

Correção

O algoritmo utiliza o princípio de *echo algorithm* (o iniciador recebe os valores acumulados de todos os nós em tempo finito) e a correção desse esquema pode ser encontrada em [Cha82].

Para provar a propriedade de progresso, assume-se que nenhum processo permanece indefinidamente no estado ativo. Se o sistema já estava no estado terminado no início da detecção, ao final desse *round* ela será detectada, pois se todas as mensagens básicas que foram enviadas foram recebidas, o contador de mensagens acumuladas é igual a 0 e a *flag* *comunica* não é atribuída verdadeira durante a execução do algoritmo de detecção. Logo, se a aplicação termina em algum tempo finito, a terminação será detectada.

Para provar o critério de segurança, é usado o seguinte esquema de cores: inicialmente todos os nós e arestas estão brancos. O iniciador torna-se vermelho ao começar a detecção. As mensagens de *query* são vermelhas e as de *echo* são azuis. Uma mensagem colore com a sua própria cor a aresta pela qual é transmitida. Uma mensagem vermelha torna o nó vermelho e um nó vermelho muda para azul quando recebe mensagens em todas as arestas incidentes.

Pela construção do algoritmo, se uma mensagem básica é recebida por um processo vermelho, a variável *comunica* torna-se verdadeira. O mesmo acontece com a *flag* do iniciador toda vez que um processo vermelho envia uma mensagem básica.

Se, ao final da detecção, o sistema não terminou e a *flag* não foi setada, então existe uma mensagem em trânsito. Como um processo azul não se ativa espontaneamente, ele deve ter recebido uma mensagem de um processo não azul. Assumindo que a *flag* não foi setada, logo o processo emissor não pode ser vermelho; por exclusão ele é branco.

Se a mensagem enviada por um processo branco para um processo azul e *comunica* é falso, pelo algoritmo o contador é maior que zero.

Portanto, se ao final da detecção, o sistema não terminou, então ou *comunica* é verdadeira ou o contador é maior que zero, logo o algoritmo não detecta pseudo-terminações.

3.2 Algoritmos monofásicos

Esta seção apresenta alguns exemplos das seguintes abordagens: detecção baseada em *timestamp*, contagem de canal, recuperação de crédito.

3.2.1 Uso de timestamp – O algoritmo de Rana 83

Esta subseção discute uma solução ao problema da detecção da terminação baseada no uso de *timestamps*. É assumido que os processos utilizam o esquema de relógio lógico de Lamport [Lam78]. O princípio da detecção foi proposto por Rana [Ran83].

O algoritmo baseia-se na verificação de que se, em um certo instante t, todos os processos cessaram a atividade nesse tempo, então a terminação foi alcançada. Isso é feito por uma onda, a qual pede que cada processo confirme que estava passivo a partir de

um determinado tempo em diante. Um processo que estava em atividade nada responde e assim extingue a onda de detecção.

O algoritmo de Rana é simétrico, ao contrário das abordagens sugeridas em [DS80] e [DFG83]. Ele utiliza comunicação síncrona, nenhuma restrição sobre a topologia de canais de comunicação e os processos estão dispostos em um anel, onde cada um ocorre uma e somente vez.

Algoritmo

Cada processo P_i que encontra-se passivo inicia a detecção enviando uma mensagem para o próximo no ciclo de processos. Essa mensagem carrega duas informações: o tempo no qual o processo tornou-se passivo e o número de processos já visitados por essa mensagem. Quando a mensagem contém esse número igual ao número total de processos, a terminação é anunciada.

O princípio desse algoritmo está na extinção seletiva de mensagens. Quando P_i está passivo e recebe uma mensagem de detecção, ele compara o tempo da mensagem recebida com o tempo que está na mensagem que ele enviou. Se o tempo contido na mensagem recebida for mais recente, P_i a repassa para o sucessor com o contador incrementado de um. No caso do tempo da mensagem não ser o mais recente ou se P_i não estiver ativo, a mensagem recebida é simplesmente descartada. Uma descrição apresentada na forma de pseudocódigo é encontrada no Algoritmo 3.4.

Exemplo

Na ilustração que é mostrada na Figura 3.5.a), todos os processos estão em estado passivo e duas detecções estão em curso. É verificado que a mensagem de menor *timestamp* é descartada (Figura 3.5.b) pelo processo que havia conhecimento de uma detecção mais recente. A mensagem que permanece em curso completa uma volta no anel detectando a terminação (Figura3.5.c e Figura3.5.d).

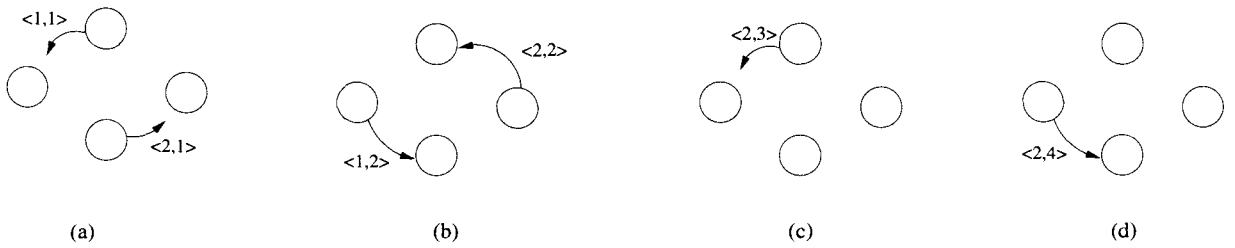


Figura 3.5: Exemplo de uma execução do algoritmo de Rana 83

```
Inicialmente
        clock := 0;
        passivo := false;
        terminou := false;


(S1)when received (mensagem)
        do
                passivo := false;
        end do;


(S2)when passivo changes to true
        do
                tempo := clock;
                 send (tempo, 1) to sucessor;
        end do;

(S3)when received (t_msg, n_visit)
        do
                if (n_visit = N and passivo) then
                begin
                        terminou := true;
                        send(terminado) to sucessor;
                end if;

                if(n_visit ≠ N and passivo) then
                begin
                        if (t_msg > tempo) then
                                        send (t_msg, n_visit + 1) to sucessor;
                                end if;
                end if;
        end do;

(S4)when received (terminado)
        do
                if (not terminou) then
                        terminou := true;
                        send (terminado) to sucessor;
                end if;
        end do;
```

Algoritmo 3.4: Descrição do Algoritmo de Rana 83

Correção

Assumindo que todos os processos vão alcançar a terminação supõem-se um processo P_u como o último que atribuiu verdadeiro para *passivo* no tempo t. Como os *timestamps* das mensagens dos outros processos é menor que t, sabe-se que a mensagem de P_u não vai ser descartada e, assim, P_u receberá a sua mensagem de volta e anunciará a terminação do sistema.

Para provar que o algoritmo não detecta pseudo-terminações, deve-se mostrar que não há recebimento de mensagem de detecção portando o contador igual ao número de processos, mas sem haver terminação do sistema. A prova da propriedade de segurança é feita por absurdo em [Ran83], e será omitida nesse texto.

Variação

Esse algoritmo pode ser modificado para que seja aplicável ao caso de comunicação assíncrona. Para isso são adotadas as seguintes abordagens [Ran83]:

- Modificando a condição de terminação global: todos os processos estão passivos e não há mensagem pendente para ser transmitida.
- Modificando localmente o seguinte: é enviado um reconhecimento para cada mensagem básica recebida. O processo é considerado passivo quando sua atividade cessou e quando recebeu todas as mensagens de reconhecimento que esperava.

Em [Tel94] é encontrado um estudo mais detalhado dessa variação sugerida.

3.2.2 Contagem de canal– o algoritmo de Shavit e Francez 86

Para aplicar-se o algoritmo considerado nesta seção, é necessária a utilização de canais de comunicação bidirecionais. Da mesma forma visto no Algoritmo de Dijkstra e Scholten, para cada mensagem da aplicação trocada entre dois processos, é retornado um *signal* em sentido contrário ao envio.

O algoritmo

Esse algoritmo é uma generalização do algoritmo visto em [DS80]. Ao invés dos nós estarem dispostos em uma árvore geradora, essa estratégia trata as computações básicas decentralizadas sendo a topologia, nesse caso, uma floresta com cada árvore enraizada no iniciador da computação.

Como no Algoritmo de Dijkstra e Scholten a terminação é detectada quando o grafo está vazio. No caso da floresta, cada iniciador verifica se a própria árvore está vazia, o que

não implica que toda floresta assim estará. A verificação de todas as árvores é feita em uma onda. Para facilitar essa varredura, a floresta é mantida de forma que uma árvore ao tornar-se vazia permanecerá assim. Se um processo é ativado depois que a árvore a qual pertence está vazia, ele será inserido em uma árvore de outro iniciador.

Para detectar a terminação, uma onda percorre os processos que pertencem às árvores que estão com a computação concluída. Ao final da varredura é anunciado o resultado.

```
(S1)only possible when estado = ativo do
        send (msg) to q;
        sc := sc + 1;
end do

when P received (msg) do
        estado := ativo;
        if(pai_p = NULL) then
                pai_p := q;
        else
                send (signal) to q;
        end if
end do


(S3)possible only when estado = ativo do
        estado := passivo;
        if(sc = 0) then
                if(pai_p = p)
                        then vazio := true;
                        else send (signal) to pai_p;
                end if
                pai_p := NULL;
        end if
end do


(S4)when P received (signal) do
        sc := sc - 1;
        if(sc = 0 and estado = passivo) then
                if(pai_p = p)
                        then vazio := true;
                        else send (signal) to pai_p;
                end if
                pai_p := NULL;
        end if
end do
```

Algoritmo 3.5: Descrição do Algoritmo de Shavit e Francez 86

A onda que percorre a floresta não está representada no pseudocódigo 3.5, mas é um procedimento que os processos executam concorrentemente. Um processo p somente participa dessa onda, se a variável *vazio* é verdadeira. Obtendo a participação de todos, a terminação é anunciada.

Exemplo

No exemplo a seguir é apresentada a detecção da terminação de um sistema que tem em curso duas computações, uma iniciada pelo processo a e outra pelo processo d (Figura 3.6.a e Figura 3.6.b).

Ao final de uma das computações, a árvore enraizada em a está vazia (Figura 3.6.c). O processo c, antes passivo, recebe uma mensagem de ativação do processo h, e assim passa a fazer parte da árvore enraizada em d (Figura 3.6.d).

A terminação é detectada quando a raiz d recebe o último *signal* pelo qual estava esperando para atribuir verdadeiro a *vazio* e, assim, anunciar a terminação do sistema (Figura 3.6.e).

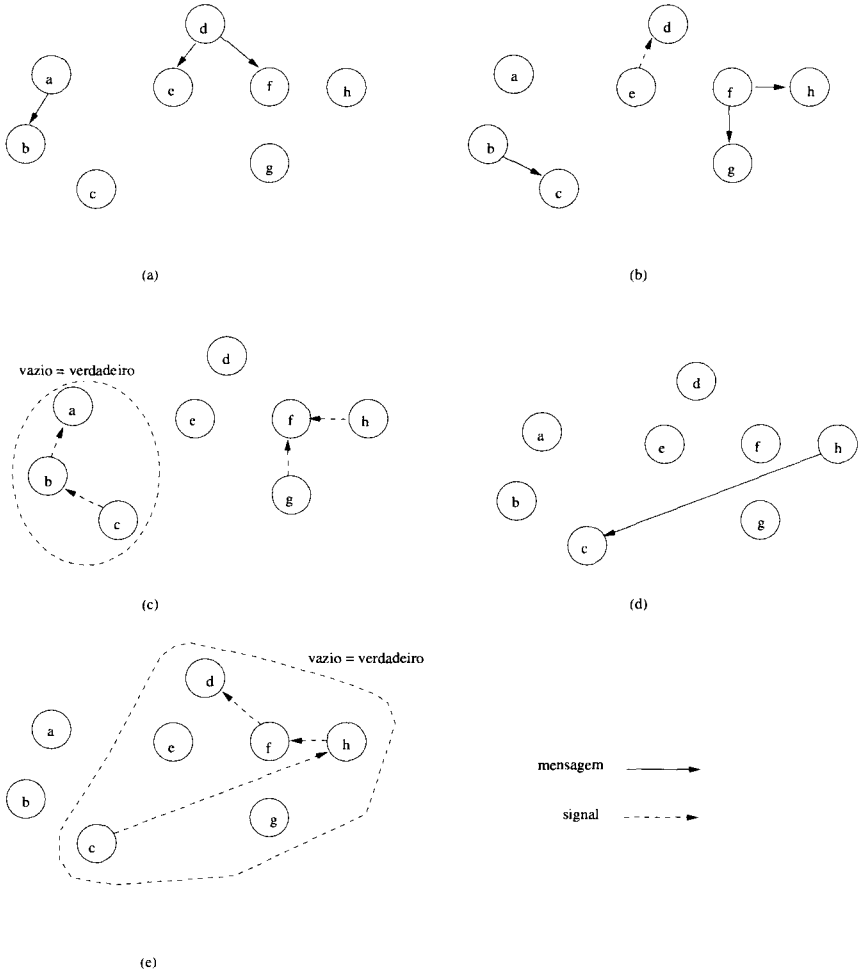


Figura 3.6: Exemplo de uma execução do algoritmo de Shavit e Francez 86

Desempenho

Pode ser provado, como visto no Algoritmo de Dijkstra e Scholten 80, que o número de *signals* não excede o número de mensagens da aplicação. Além desses *signals*, são enviadas mensagens de controle de somente uma onda, então o número total de mensagens de controle trocadas é de no máximo $M + W$.

Correção

Assuma que o sistema atingiu o estado de terminação e que dentro de um número finito de passos o algoritmo alcançará a configuração terminal. Como visto na prova de correção do Algoritmo de Dijkstra e Scholten (Seção 3.1.1), o grafo está vazio. Como consequência, todos os eventos da onda foram habilitados em todos os processos e a configuração terminal implica que todos os eventos foram executados, incluindo a última decisão que anuncia a terminação.

Para provar que não há detecção de pseudo-terminações, observe que a detecção é anunciada quando uma decisão ocorre no algoritmo de onda. Isto implica que cada processo p enviou uma mensagem de detecção ou tomou uma decisão e a árvore que contém p está vazia e não há nada que mudará isso. Se para cada p isso acontece, então conclui-se que o grafo está vazio e logo o sistema está terminado.

3.2.3 Recuperação de crédito – o algoritmo de Mattern 89

O Algoritmo

Mattern propôs um algoritmo que detecta a terminação em uma unidade de tempo após sua ocorrência. O algoritmo detecta a terminação de uma computação e assume uma topologia lógica em estrela, pois requer que todo processo seja capaz de comunicar-se com o iniciador diretamente.

A cada mensagem e a cada processo é associado um **crédito**, uma variável que assume valores entre 0 e 1.

Um processo passivo envia o seu crédito para o iniciador. O iniciador faz o papel do banqueiro, acumulando os créditos enviados na variável *retornado*. Quando todos os créditos foram coletados, implica que não há mais nenhum crédito no sistema, significando que não há processo ativo e nenhuma mensagem em trânsito, portanto a terminação foi alcançada.

O algoritmo assegura as seguintes invariantes:

1. Em qualquer momento, a soma dos todos créditos mantidos por processos e mensagens é igual a 1.

2. Quando um processo está ativo, o seu crédito é maior que 0.

3. Uma mensagem de ativação que está em trânsito possui crédito maior que 0.

Inicialmente

```
        if(p = p0) then
                credito := 1;
                retornado :=0;
        else
                credito := 0;
        end if

(S1)before send (msg,cred)
        do
                credito := credito/2;
                send (msg, credito);
         end do

(S2)when received (msg, cred)
        do
                estado := ativo;
                credito := credito + cred;
        end do

(S3)when estado changes to passivo
        do
                send (ret, credito) to p0;
                credito := 0;
         end do

(S4)when p0 received (ret, cred)
        do
                retornado := retornado + cred;
                if (retornado = 1) then terminacao detectada;
                end if
        end do
```

Algoritmo 3.6: Descrição do Algoritmo de Mattern 89 - Recuperação de crédito

Exemplo

Na Figura 3.7 é ilustrado o comportamento desse algoritmo. O nó que inicia a computação básica distribui o seu crédito pelas mensagens que envia (letra (a) da Figura 3.7). Os valores são divididos enquanto há a transmissão de mensagens (setas cheias)(Figura 3.7.b-.f)). Quando um processo torna-se passivo, o crédito é retornado ao iniciador (setas pontilhadas). Obtendo a totalização de créditos igual a um é anunciada a terminação do sistema (letra (g) da Figura 3.7).

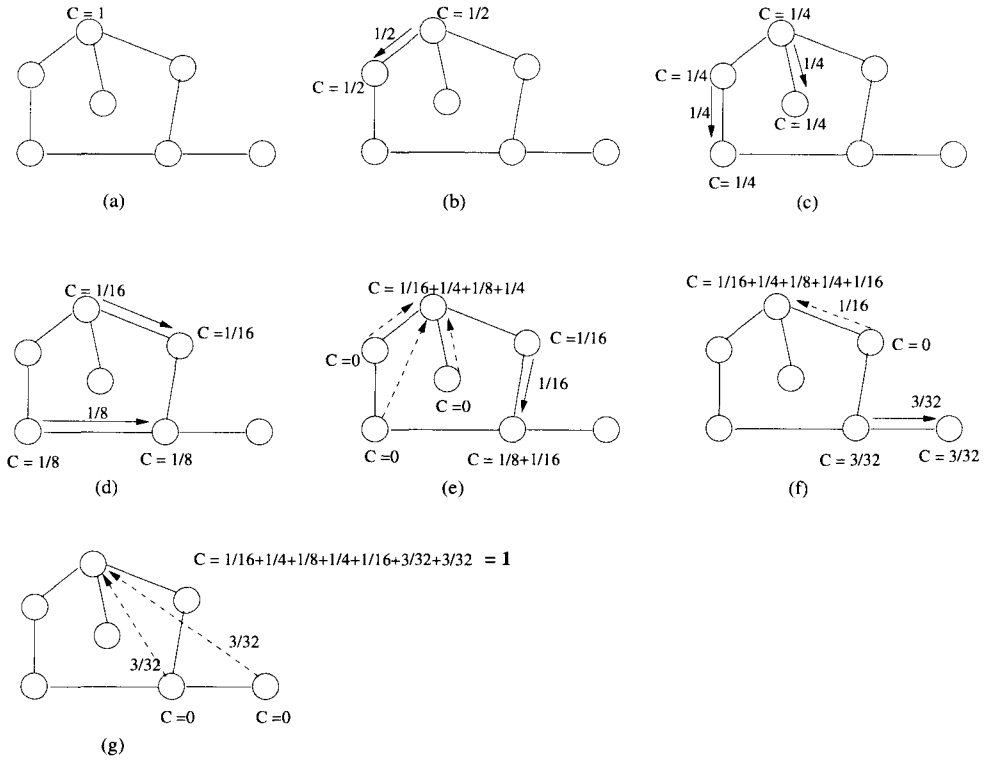


Figura 3.7: Exemplo de uma execução do algoritmo de Mattern 89

Desempenho

Cada processo ao tornar-se passivo envia uma mensagem contendo o seu crédito ao iniciador. Contando a partir do instante que o sistema alcançou a terminação, serão gastas exatamente N mensagens de controle para transferência de crédito.

Correção

A propriedade de segurança pode ser facilmente verificada. Se o iniciador obteve todo o somatório dos créditos igual a 1 então, pela primeira invariante, nenhum outro processo e nenhuma mensagem possuem crédito maior que zero. Por causa das outras invariantes, conclui-se que todos os processos estão passivos e não há mensagem em trânsito.

Por construção do algoritmo, verifica-se que quando o sistema alcança a terminação não há mensagem em trânsito (os canais têm crédito igual a zero) e os créditos nos processos não serão alterados. Como todos os créditos são transferidos para o iniciador em um tempo finito, o somatório chegará a um e, portanto a propriedade de progresso está garantida.

Capítulo 4

A Teoria do *Deadlock*

Pelo fato dos sistemas distribuídos apresentarem um alto grau de compartilhamento, se a sequência de alocação de recursos para os processos não for controlada, um *deadlock* pode ocorrer.

Informalmente, um *deadlock* refere-se à situação na qual existe um grupo de processos esperando por atendimento, de maneira que nenhum processo nesse grupo possa enviar qualquer mensagem (liberação de recurso) até que receba a mensagem pedida (recurso) de outro processo no grupo. Quando essa situação ocorre, todos esses processos esperam permanentemente e não há mais progresso.

Um *deadlock* só ocorre quando as seguintes condições são válidas [CEA71]:

Exclusão mútua: Os recursos envolvidos não são compartilháveis.

Retém e aguarda: Os processos mantêm a posse do recurso enquanto esperam pelo atendimento dos novos pedidos.

Nenhuma preempção: Não há preempção de recurso que possa ser feita sem que o processo que o mantém seja abortado.

Espera circular: Em uma cadeia circular de processos, cada um mantém recursos que estão atualmente sendo requisitados pelo próximo na cadeia.

As abordagens que lidam com o problema do *deadlock* podem ser divididas em três categorias principais apresentadas em vários trabalhos, por exemplo [IM80, SG94, Tan95]:

1. Estratégias que previnem a ocorrência de *deadlock*: somente são iniciados os processos que não podem produzir um *deadlock* durante todo o seu tempo de execução. Para isso, pelo menos uma das quatro condições descritas anteriormente deve ser negada.

2. Estratégias que evitam a ocorrência de *deadlock*: é utilizado um algoritmo que se antecipa à transformação de um potencial *deadlock* em uma ocorrência real. Nesse caso, é recusado o atendimento de um pedido que não conduza o sistema a um estado seguro (estado onde existe pelo menos uma sequência de alocação que leve todos os processos até o final da execução). É uma noção distinta da noção de prevenção a qual assegura que não existirão *deadlocks* em potencial.

3. Estratégias que detectam um *deadlock*: detectam e recuperam o sistema de um estado de *deadlock*.

Dependendo da aplicação, uma estratégia é mais adequada que outra. Por exemplo, as abordagens de prevenir e evitar *deadlocks* são conservadoras, sendo preferíveis no caso da ocorrência de um *deadlock* ser frequente ou altamente indesejável.

Em contraste, a detecção é uma estratégia otimista, que fornece um recurso disponível a um pedido, esperando que isso não leve a um *deadlock*. Singhal [Sin89] confronta as duas primeiras estratégias com a detecção de *deadlock* e obtém os seguintes resultados:

1. A estratégia de prevenção é ineficiente porque diminui a concorrência do sistema, restringindo a execução das tarefas para evitar pelo menos uma das condições de *deadlock*. Nas abordagens que evitam *deadlock* é caro e ineficiente verificar se cada estado é seguro.

2. A informação necessária às estratégias para prevenir ou evitar *deadlocks* pode não estar à disposição. Em alguns sistemas, por exemplo, a prevenção é impossível pelo fato de não existir informação a respeito de futuras requisições.

Como o *deadlock* possui a propriedade de ser estável (persistirá até que seja detectado e quebrado), os processos não podem prosseguir, pois vão esperar que um outro processo libere algum recurso. Assim, se os *deadlocks* forem infrequentes, a estratégia otimista permite o melhor aproveitamento dos recursos do sistema e torna possível uma execução do algoritmo de detecção paralela às outras atividades normais do sistema, de maneira a não causar um grande impacto no desempenho.

Além disso, a estratégia de detecção pode ser integrada a outros problemas relacionados como, por exemplo, detecções de terminação e de processos órfãos que devem ser identificados em tempo de execução. Se já existem funções no sistema que necessitam de detecção de estado global (por exemplo, depuração distribuída ou monitorização de um sistema distribuído [BM93]), isto facilita uma extensão do algoritmo utilizado para que ele possa realizar também a detecção de *deadlock*.

Nesse capítulo são apresentados os princípios teóricos que servirão de base para o entendimento dos algoritmos de detecção de *deadlock* a serem apresentados no capítulo seguinte.

4.1 Conceitos básicos

Para investigar o problema de *deadlock*, será necessário introduzir o modelo do sistema, alguns conceitos e definições da teoria dos grafos e a descrição dos modelos de requisição. Será apresentada também uma classificação proposta por Knapp [Kna88] para as diversas abordagens que lidam com detecção.

4.1.1 O modelo do sistema

No modelo considerado neste trabalho supõe-se que todo processo pode se comunicar com qualquer outro processo, direta ou indiretamente. A maioria dos algoritmos supoem, também, que as mensagens enviadas são recebidas em um tempo finito e na ordem que foram enviadas.

Os processos em sistemas de computação podem interagir explicitamente trocando mensagens, ou implicitamente, competindo por objetos como recursos de E/S. Em qualquer desses dois tipos de interação pode ocorrer o bloqueio de processos. Será usado o termo **recurso** para referir-se a qualquer objeto cuja requisição possa bloquear um processo.

Existem dois tipos de recursos: os **recursos reutilizáveis**, que modelam a competição por objetos e os **recursos consumíveis**, que modelam a troca de mensagens. A diferença fundamental entre um recurso reutilizável e um consumível é que as instâncias de um reutilizável não são destruídas nem criadas, mas apenas transferidas; em contraste, um recurso consumível é produzido (criado) e consumido. Shih [SS91], apresenta a seguinte descrição para diferenciar os dois tipos de recursos: recursos reutilizáveis são usados para modelar a competição entre os objetos, tais como dados compartilhados e *buffers* de memória. Para modelar as interações explícitas entre os processos (sincronização ou troca de mensagens) é utilizada a idéia dos recursos consumíveis.

Em operação normal, um processo pode utilizar um recurso somente na seguinte sequência de operações:

Requisita recurso Se ele não pode ser atendido imediatamente, deve esperar até que possa obter o recurso. Um processo está bloqueado desde o momento que faz o pedido até o momento que é atendido. Dessa maneira, não pode requisitar ou liberar qualquer outro recurso, mas nada o impede de enviar ou receber mensagens para executar outras tarefas (detecção de *deadlock*, por exemplo).

Utiliza recurso Faz uso do recurso por um tempo finito.

Libera recurso

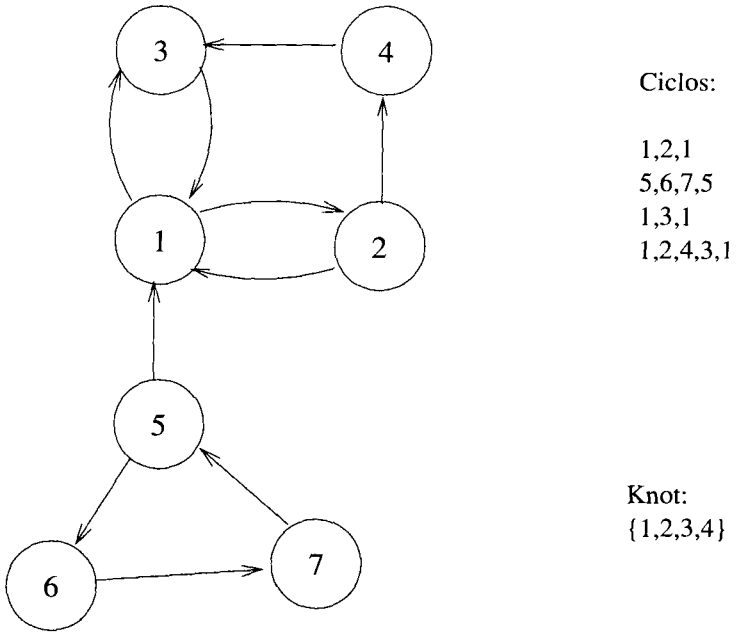


Figura 4.1: Exemplos de ciclo e de *knot*

sumidas. Um processo ativo torna-se passivo ao requisitar algumas mensagens (recursos). Tendo as recebido, torna-se ativo novamente.

A cada processo passivo está associado um conjunto de processos dos quais ele está esperando receber mensagens, o **conjunto dependente** (DS)[1]. Um processo passivo somente é ativado se a condição de ativação definida sobre o conjunto dependente é satisfeita.

A formulação dessas condições depende dos modelos que as aplicações utilizam para expressarem as requisições. Esses modelos caracterizam-se pelas restrições que impõem à forma que as requisições podem assumir (uma descrição mais detalhada pode ser encontrada em [Kna88, BHRS95]).

O Modelo de Recurso Único

Nesse modelo um processo só pode ter uma requisição pendente por vez. A presença de um ciclo no grafo de estado corresponde a um *deadlock*.

O Modelo AND

Um processo pode requisitar um conjunto de recursos. Esse processo estará bloqueado até que **todas** as suas requisições sejam atendidas, ou seja, a condição de ativação é

[1]Em inglês *dependent set*.

que o processo passivo receba uma mensagem de cada processo que está no seu conjunto dependente.

O modelo AND também é conhecido como **modelo de recurso**. O problema da detecção se reduz a encontrar um ciclo de processos bloqueados, onde cada processo depende do outro nesse ciclo.

O Modelo OR

Esse modelo é denominado também como **modelo de comunicação**. Um processo pode solicitar um conjunto de recursos e esse processo estará bloqueado até que lhe seja atribuído **qualquer um** dos recursos que solicitou. Basta apenas uma mensagem vinda de qualquer processo no conjunto dependente para que o processo seja ativado.

O fato de existir um ciclo no grafo é uma condição insuficiente para a existência do *deadlock* nesse caso. Um conjunto S de processos está em *deadlock* se:

1. todos os processos em S estão bloqueados;

2. o conjunto de dependentes de todo o processo em S é um subconjunto de S, e

3. não existem mensagens de atendimento em trânsito entre os processos em S.

Ou seja, a condição necessária e suficiente para que um processo bloqueado p esteja em *deadlock* é:
(i)ele fazer parte de um *knot* ou
(ii)p alcançar somente processos pertencentes a um *knot*.

O Modelo AND-OR

Esta é uma generalização do modelo anterior onde são permitidas requisições da forma *a and (b or c)*. O que normalmente é feito a partir de uma requisição desse tipo é mapear a expressão que representa a requisição para uma árvore de processos. O mapeamento é uma forma de representação da requisição *AND-OR* em uma forma regular como a forma normal disjuntiva.

Um exemplo é apresentado na Figura 4.2, onde a transação t_1 espera pelo atendimento de $(t_2$ *and* $t_3)$ *or* t_4 *or* t_5. Na figura, uma requisição *AND* é representada por arestas conectadas por uma linha e uma requisição *OR* por uma aresta simples.

Para o mapeamento, é utilizada uma diferenciação que denota os processos *AND* por p' e os processos *OR* por p.

Seja P o conjunto de N processos $\{P_1, P_2, \ldots, P_N\}$. O conjunto dependente para esse modelo é definido como $DS = DS_1 \bigcup DS_2 \bigcup \cdots \bigcup DS_q$, onde $DS_i \subseteq P$ para todo i.

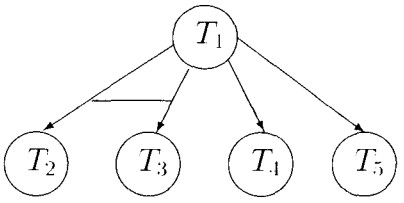


Figura 4.2: Exemplo de uma requisição *AND-OR*

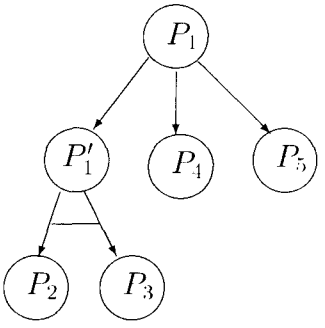


Figura 4.3: Mapeamento em processos

A condição de ativação é satisfeita somente depois do recebimento de uma mensagem vinda *de cada* processo em DS_1, *ou* uma mensagem vinda *de cada* processo em DS_2, *ou* .... *ou* uma mensagem vinda *de cada* processo em DS_q.

O Modelo $C_{n,k}$

Esse modelo permite a especificação de uma requisição para se obter quaisquer k recursos disponíveis a partir de um repositório de tamanho n.

O modelo $C_{n,k}$ também é encontrado na literatura como modelo de requisição k *out of* n. O sistema que segue esse tipo de modelo permite que um processo solicite n recursos. Esse processo permanecerá bloqueado até que sejam conseguidos quaisquer k dos n recursos.

Esse tipo de combinação de elementos pode ser mapeada para uma árvore de processos usando um esquema similar ao modelo anterior. O mapeamento é ilustrado através de um exemplo extraído de [BT83]. Considere um processo p que requisita serviços dos processos $p_1, ..., p_6$ e necessita que as seguintes condições sejam satisfeitas para que possa voltar à execução:

$$p_1 \text{ or } (\text{ 2- out-of- } (p_1, p_3, p_5) \text{ and 2 out-of } (p_2, p_4, p_6))$$

Com a introdução de processos intermediários, essa fórmula é decomposta em requisições que contêm um único conector (*AND, OR, out-of*).

A notação k/n ao lado do vértice na Figura 4.4 indica que o nó está bloqueado e precisa de k dos n recursos pendentes para que possa desbloquear.

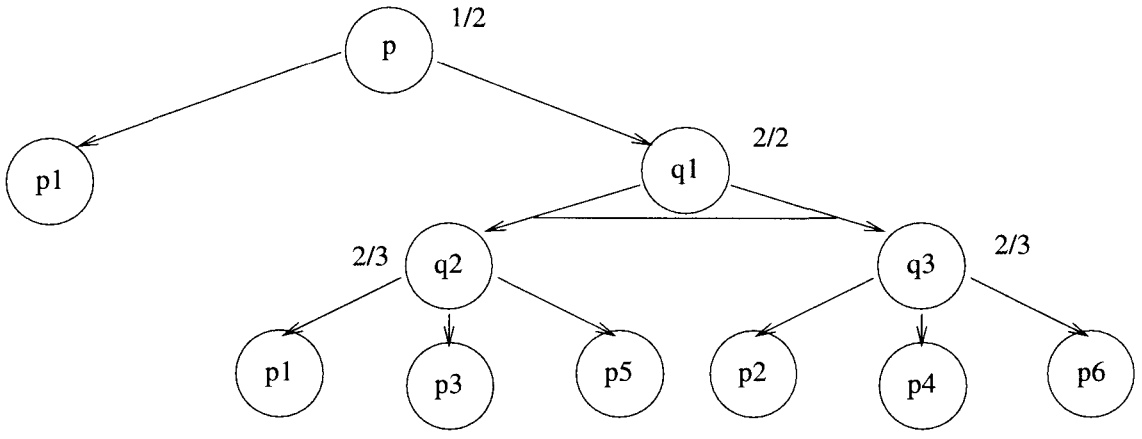


Figura 4.4: Decomposição de uma requisição

Toda requisição no modelo $C_{n,k}$ pode ser expressa no modelo AND-OR e vice-versa. Tome-se como exemplo os casos especiais:

a and b and c and d $\cong C_{4,4}$ requisição no modelo AND $\cong C_{n,n}$

a or b or c or d $\cong C_{4,1}$ requisição no modelo OR $\cong C_{n,1}$

Note-se que os operadores AND e OR podem expressar uma requisição P *out of* Q, mas o tamanho da fórmula AND-OR correspondente é $P * C_{P,Q}$.

O poder dos modelos AND-OR e $C_{n,k}$ é idêntico, entretanto, o modelo $C_{n,k}$ expressa uma requisição de forma mais compacta. Isso é muito mais conveniente e mais interessante por questões de implementação. Uma representação reduzida requer menos armazenamento e menos tempo de verificação se uma condição está satisfeita.

Para descrever a condição de ativação, esse modelo é dividido em dois submodelos: o básico e o disjunto. Na verdade, é um único modelo, mas casos particulares são visualizados com mais facilidade no modelo básico e a generalização é expressa pelo modelo disjunto.

1. Modelo Básico

   A cada processo passivo existem associados um conjunto dependente DS e um inteiro k tal que $1 \leq k \leq |DS|$. O processo é ativado ao receber k mensagens vindas de processos distintos em DS.

   Este modelo inclui o modelo AND (quando $k = |DS|$) e o modelo OR (quando $k = 1$)

2. Modelo Disjunto

   A cada processo passivo estão associados o conjunto dependente $DS = DS_1 \bigcup DS_2 \bigcup \cdots \bigcup DS_q$ e inteiros $k_1, k_2, \ldots, k_q$, onde $DS_i \subseteq P$ e $1 \leq k_i \leq |DS_i|$ para todo i.

   Um processo se torna ativo depois que chegam k_1 mensagens vindas de processos distintos em DS_1, ou k_2 mensagens vindas de processos distintos em DS_2, ou ..., ou k_q mensagens vindas de processos distintos em DS_q.

O Modelo Irrestrito

A única coisa que pode-se supor nesse modelo é a propriedade da estabilidade do *deadlock*. Portanto, todos os algoritmos que lidam com esse modelo geral podem ser utilizados para detectar outras propriedades estáveis também. Contudo, esses algoritmos possuem mais valor teórico do que prático, já que não se pode contar com informações da estrutura da computação subjacente, sendo necessário lidar com uma grande sobrecarga que pode ser evitada quando são utilizados algoritmos para modelos mais simplificados. Muitos algoritmos relacionados ao modelo irrestrito tem o seu estudo apresentado em [FL82, CL85, Hel89, BM93, BR89, HMR93, MS94, SS94].

4.2 Definições de *Deadlock*

Além da caracterização de um *deadlock* por topologia encontrada em grafo de dependência, pode-se defini-lo de uma maneira mais formal por predicados que utilizam a idéia de condição de ativação vista anteriormente para cada modelo de requisição. Para maiores detalhes, é sugerido o trabalho [BHRS95].

Seja *deadlock*(B) um predicado significando que B, um conjunto não vazio de processos, está em *deadlock*. Além desse predicado, alguns outros serão introduzidos para auxiliar nas definições:

- *passivo*: verdadeiro se e somente se P_i está passivo.
- $chegou_i(j)$: verdadeiro se e somente se uma mensagem chegou de P_j e ainda não foi consumida por P_i.
- $vazio_i(j)$:verdadeiro se e somente se todas as mensagens enviadas de P_j para P_i chegaram.

Observe que a condição $vazio_i(j) \bigwedge \neg chegou_i(j)$ significa que todas as mensagens de P_j para P_i chegaram e foram consumidas. Uma outra observação necessária é que, por conveniência, a expressão $(\forall P_i :: P_i \in B :: \ldots)$ substitui $(\forall i :: (1 \leq i \leq N) \bigwedge (P_i \in B) :: \ldots)$.

Seja DS_i o conjunto dependente de P_i.

Definição 8 (*Deadlock* **no modelo AND**) $deadlock(B) \equiv$
$(B \subseteq P) \bigwedge (B \neq \emptyset) \bigwedge (\forall P_i :: P_i \in B :: (passivo_i \bigwedge (\exists P_j :: P_j \in DS_i \bigcap B :: (vazio_i(j) \bigwedge \neg chegou_i(j))))$

Definição 9 (*Deadlock* **no modelo OR**) $deadlock(B) \equiv$
$(B \subseteq P) \bigwedge (B \neq \emptyset) \bigwedge (\forall P_i :: P_i \in B :: (passivo_i \bigwedge (DS_i \subseteq B) \bigwedge (\forall P_j :: P_j \in DS_i :: (vazio_i(j) \bigwedge \neg chegou_i(j)))))$

Para $i, 1 \leq i \leq N$ e $r, 1 \leq r \leq q_i$, $DS_{i,r}$ denota os subconjuntos do conjunto dependente de P_i.

Definição 10 (*Deadlock* **no modelo AND-OR**) $deadlock(B) \equiv$
$(B \subseteq P) \bigwedge (B \neq \emptyset) \bigwedge (\forall P_i :: P_i \in B :: (passivo_i \bigwedge (\forall r :: 1 \leq r \leq q_i :: (\exists P_j :: P_j \in DS_{i,r} \bigcap B :: (vazio_i(j) \bigwedge \neg chegou_i(j))))))$

Seja $n_i = |DS_i|$, então tem-se que $1 \leq k_i \leq n_i$.

Definição 11 (*Deadlock* **no modelo** *P-out of-Q* **básico**) $deadlock(B) \equiv$
$(B \subseteq P) \bigwedge (B \neq \emptyset) \bigwedge (\forall P_i :: P_i \in B :: (passivo_i \bigwedge (\exists D_i :: D_i \subseteq DS_i \bigcap B :: ((|DS_i \setminus D_i| < k_i) \bigwedge (\forall P_j :: P_j \in D_i :: (vazio_i(j) \bigwedge \neg chegou_i(j)))))))$

Isto significa que cada processo P_i em B está associado a um conjunto de processos D_i tal que as chegadas de novas mensagens de processos em D_i não são possíveis já que $D_i \subseteq B$ e $(\forall P_j :: P_j \in D_i :: (vazio_i(j) \bigwedge \neg chegou_i(j))$. Dessa forma, P_i pode receber no máximo $|DS_i \setminus D_i|$ das mensagens esperadas, que não são suficientes para ativar P_i porque $|DS_i \setminus D_i| < k_i$.

Para $i, 1 \leq i \leq N$ e para $r, 1 \leq r \leq q_i$, $k_{i,r}$ denota o número de mensagens que P_i deve receber dos processos que pertencem aos subconjuntos dependentes $DS_{i,r} (1 \leq k_{i,r} \leq |DS_{i,r}|$.

Definição 12 (*Deadlock* **no modelo** *P-out of-Q* **disjunto**) $deadlock(B) \equiv$
$(B \subseteq P) \bigwedge (B \neq \emptyset) \bigwedge (\forall P_i :: P_i \in B :: (passivo_i \bigwedge$
$(\forall r :: 1 \leq r \leq q_i ::$
$(\exists D_{i,r} :: D_{i,r} \subseteq DS_{i,r} \bigcap B :: ((|DS_{i,r} \setminus D_{i,r}| < k_{i,r})$
$\bigwedge (\forall P_j :: P_j \in D_{i,r} ::$
$(vazio_i(j) \bigwedge \neg chegou_i(j))))))$

É apresentada uma definição de *deadlock* independente do modelo de requisição subjacente. Para isso, é definido $CHEGOU_i$ como o conjunto de todos os processos P_j tal que $chegou_i(j) = verdadeiro$. Além disso, NV_i denota o conjunto de todos os processos P_j tal que $vazio_i(j) = falso$.

Para que se abstraia da condição de ativação, o predicado *fulfilled*(A) é introduzido, onde A é subconjunto de P. O predicado *fulfilled*(A) é verdadeiro se as mensagens que chegaram de todos os processo que pertencem a A são suficientes para ativar o processo P. O predicado *fulfilled* associado a P_i é chamado de $fulfilled_i$.

Um conjunto de processos B está em *deadlock* segundo a definição dada abaixo:

Definição 13 (Uma definição abstrata de *Deadlocks*) $deadlock(B) \equiv (B \subseteq P) \bigwedge (B \neq \emptyset) \bigwedge (\forall P_i :: P_i \in B :: (passivo_i \bigwedge \neg fulfilled_i(CHEGOU_i \bigcup NV_i \bigcup (P \setminus B))))$

Em outras palavras, $P_i \in B$ não poderia ser ativado mesmo depois que todas as mensagens tenham chegado de todos os processos em $CHEGOU_i \bigcup NV_i$ e de todos os processos que não estão em *deadlock*, pois elas não seriam suficientes para satisfazer a condição de ativação de qualquer processo em B.

4.3 Classificação das abordagens de detecção de *deadlock*

A hierarquia de modelos de requisição apresentada também pode ser utilizada para classificar os algoritmos de detecção de *deadlock* de acordo com a complexidade de requisição de recursos que eles permitem. Relacionada a essa complexidade, está a topologia que caracteriza um *deadlock* em cada um dos modelos.

Dependendo da topologia que deseja-se encontrar no grafo de estado, algumas abordagens são mais adequadas que outras. De acordo com isso, os algoritmos publicados têm seus fundamentos resumidos nos quatro métodos de resolução descritos a seguir que formam outra classificação.

4.3.1 Algoritmos *path-pushing*

A idéia básica dessa classe de algoritmos é construir um WFG simplificado em cada nó do sistema. Para isso, mensagens são enviadas para transmitir os caminhos do grafo até que algum nó tenha informação do estado global do sistema e, assim, possa anunciar se existe ou não *deadlock*.

4.3.2 Algoritmos caça-aresta (*edge-chasing*)

A característica fundamental dessa classe é a utilização de uma mensagem especial (sonda), diferente das mensagens de pedido ou atendimento. A propagação das sondas indica ou não a presença de um ciclo no grafo.

4.3.3 Computações difusas

Nessa categoria, uma computação é sobreposta à aplicação. Um WFG é construído implicitamente através do envio de mensagens *query* ou mensagens *reply* (essas mensagens são distintas das mensagens de pedido ou atendimento para o recurso). Esse princípio foi introduzido por Dijkstra e Scholten [DS80] para detectar a terminação de uma computação básica centralizada.

Uma computação difusa[2] também é conhecida como **algoritmo de eco** [Cha82]. Esse algoritmo é dividido em duas fases: uma fase de expansão, na qual a primeira mensagem é recebida e propagada para os nós vizinhos ao receptor, e uma fase de retração, quando as mensagens de eco são recebidas dos vizinhos.

Essa idéia constitui um importante princípio que deu origem a vários algoritmos de detecção de *deadlock* e de detecção de terminação (Seções 5.2.1, 5.3.1, 5.3.2, 5.3.3, 3.1.1 e 3.1.3).

4.3.4 Detecção do estado global

Um *snapshot* (estado global) do sistema é construído sem suspender temporariamente a aplicação e, sobre esse estado registrado, simula-se o atendimento dos pedidos para verificar a existência ou não de um *deadlock*.

Relacionando com o problema do *deadlock*, o estado do sistema é o WFG e as escalas são seqüências de transformações do WFG. Baseado nos resultados de [CL85] (vide Seção 1.3), diz-se que uma transação está em *deadlock* se ela está em *deadlock* no WFG_t, o WFG no tempo t.

Chandy e Lamport mostraram como obter um estado global consistente de um sistema distribuído propagando-se marcadores pelos canais do sistema. Um estado global consistente obtido dessa maneira é também chamado de *snapshot* do sistema. No próximo capítulo será vista a aplicação dessa teoria em uma das abordagens propostas para detecção de *deadlock*.

[2]Em inglês *diffusing computation*.

4.4 Considerações gerais

Para que um algoritmo de detecção de *deadlock* seja correto, ele deve satisfazer duas condições. A primeira é a propriedade básica de **progresso** que toda solução deve ter, ou seja, todo *deadlock* será detectado em tempo finito. A segunda é a propriedade de **segurança** que assegura que se um *deadlock* é detectado ele realmente existe.

Os algoritmos distribuídos para detecção de *deadlock* baseiam-se na cooperação dos nós do sistema para determinar se há uma determinada topologia no grafo de estado. Um grafo que represente as verdadeiras relações de dependência é imprescindível para se detectar *deadlocks*.

A manutenção dos grafos de estado enquanto o sistema opera é uma tarefa complicada que se não for bem conduzida, pode levar à detecção de *deadlocks* falsos.

As mensagens de atualização do grafo podem ser entregues fora da ordem que foram enviadas. Por exemplo, se uma mensagem de remoção de uma aresta chega antes da mensagem de criação dessa mesma aresta, ela não pode ser simplesmente ignorada. Se isso ocorrer, poderão ser geradas transações **aparentemente bloqueadas**[3], ou seja, transações que, apesar de estarem ativas, aparecem bloqueadas em alguns grafos de dependência.

Como resultado da inabilidade de manutenção de uma visão consistente do status das transações, alguns *deadlocks* que não existem são detectados. Além dessa indesejável característica, grafos que não refletem o estado do sistema causam a transmissão de algumas mensagens desnecessárias pelo algoritmo de detecção que são desnecessárias e podem gerar outros *deadlocks* falsos.

4.5 Comentários

A complexidade da detecção de *deadlock* depende de como os processos podem fazer requisições. O modelo de requisição mais simples possível é aquele em que um processo pode pedir, no máximo, um recurso por vez. Além desse, tem-se o modelo AND (também conhecido como **modelo de recurso**), o modelo OR (também chamado de **modelo de comunicação**) e o modelo AND-OR, que é uma generalização dos dois modelos anteriores. Porém, o modelo mais genérico é o modelo AND-OR e o equivalente modelo $C_{n,k}$ ou *k out-of n*. A Figura 4.5 ilustra a hierarquia dos modelos de requisição em relação ao seu poder de expressão.

Os modelos de requisição apresentados classificam os algoritmos de detecção de *deadlock* distribuído de acordo com a complexidade das requisições que eles permitem. Além

[3]Em inglês *ostensibly blocked*.

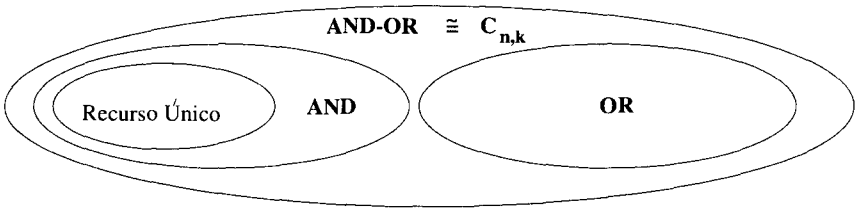


Figura 4.5: Hierarquia dos modelos de requisição

dessa classificação, foi apresentada uma outra que baseia-se no princípio fundamental utilizado: computações difusas, esquema de *path-pushing*, de *edge-chasing* ou detecção do estado global.

Ambas classificações poderão ser melhor ilustradas no próximo capítulo, que apresentará na forma de algoritmos algumas idéias discutidas até aqui.

Capítulo 5

Algoritmos de Detecção de *Deadlock*

Neste capítulo serão vistas algumas abordagens que mostram o desenvolvimento do estudo e da resolução do problema. Os algoritmos serão classificados de acordo com o modelo de requisição subjacente, partindo de modelos menos complexos, como o modelo AND, e chegando ao modelo generalizado, ou seja, o modelo $C_{n,k}$.

5.1 Algoritmos para o modelo de requisição AND

Nessa classe de algoritmos serão apresentados três exemplos que cobrem duas abordagens: dois algoritmos que representam a filosofia *path-pushing* e um que representa a caça-aresta.

Como será constatado no esquema *path-pushing*, todo nó constrói um grafo de dependência que representa uma parte do grafo total da qual ele tem conhecimento. Essas visões dependem do comportamento dinâmico do sistema e são construídas a partir do recebimento de mensagens que descrevem as dependências na forma de um caminho orientado ($P_i \rightarrow P_j$, por exemplo, indica a dependência e sua orientação). A idéia é que, se um *deadlock* existe, um ciclo aparecerá em pelo menos um dos grafos parciais.

A segunda abordagem utiliza uma idéia que baseia-se na transmissão de uma mensagem especial (sonda) que percorre o sistema detectando ciclos, que é a caracterização de um *deadlock* no modelo AND de requisição.

5.1.1 Algoritmo de Menasce e Muntz 79

Será apresentado nesta seção um dos primeiros algoritmos a tratar a detecção de *deadlock* no caso distribuído. Sua descrição completa pode ser encontrada no artigo de Menasce e Muntz [MM79].

A descrição concentra-se em uma das duas abordagens apresentadas no artigo sobre detecção de *deadlock*. A outra estratégia não é distribuída, mas, sim, hierárquica. A

atenção será focalizada no algoritmo distribuído por causa da repercussão nos trabalhos posteriores na área de detecção de *deadlock* distribuído.

O Algoritmo e Estruturas de Dados

Trata-se da primeira estratégia a usar um grafo *wait-for* condensado, isto é, o algoritmo de detecção de *deadlock* de Menasce e Muntz propaga somente os dois pontos finais de um caminho orientado, ao invés de todo o caminho. É utilizado um grafo de dependência (TWF), como aquele visto anteriormente no Capítulo 4.

Uma transação é considerada **não bloqueada**, se o vértice associado a ela é um sorvedouro, ou seja, não tem aresta de saída. Um **conjunto bloqueante** de T (denotado por $conjunto_bloqueante(T)$) é definido como conjunto de todas as transações não bloqueadas que podem ser alcançadas a partir do nó representando a transação T, seguindo todos os caminhos dirigidos no grafo TWF. Este é o conjunto de transações responsáveis pelo bloqueio de T.

Para cada transação T_1 no $conjunto_bloqueante(T)$, o par (T, T_1) é chamado **par bloqueante**, o sítio (nó) de origem da transação T é denotado por $S_{orig}(T)$ e o grafo no sítio K é denotado por $TWF(K)$.

Quando uma transação T é bloqueada, para cada transação T_i no conjunto T, o algoritmo envia o par bloqueante (T, T_i) para os sítios de origem de T e de T_i. Em outras palavras, a informação sobre o grafo TWF condensado é enviada pelos caminhos do grafo TWF global. O algoritmo é descrito resumidamente a seguir, podendo ser acompanhado pelo pseudocódigo descrito no Algoritmo 5.1. Além das variáveis citadas anteriormente, faz-se necessária uma estrutura de dados para registrar o grafo de dependência (TWF). Essa estrutura é atualizada com a função $adicione_aresta(T, T_i)$, quando é registrada a dependência de T para T_i, e percorrida através de $existe_ciclo(TWF)$, para se detectar a presença da topologia que caracteriza um *deadlock* nesse modelo de requisição.

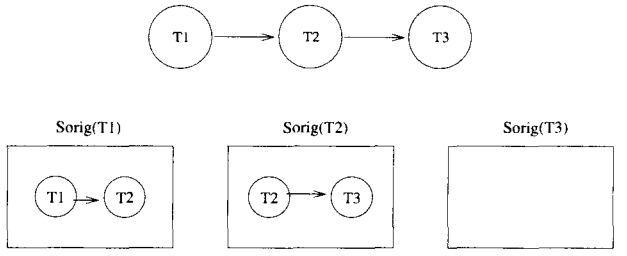
Funções ou procedimentos que montam, atualizam ou percorrem um grafo de dependência são os mesmos utilizados no caso centralizado, contudo não encontram-se descritos nesse trabalho. O leitor interessado deve consultar [Gra78].

Quando uma transação T requer um recurso **r** no sítio S_k e **r** está sendo utilizado pelas transações $T_1, T_2, \ldots, T_n$, uma aresta é adicionada a partir do nó denotado T, para cada uma das transações $T_1 \ldots T_n$. Se essa adição ocasiona um ciclo em $TWF(K)$, então existe um *deadlock*. Para cada transação T' no conjunto bloqueante(T), um par bloqueante (T, T') é enviado a $S_{orig}(T)$, se $S_{orig}(T) \neq S_k$ e para $S_{orig}(T')$, se $S_{orig}(T') \neq S_k$.

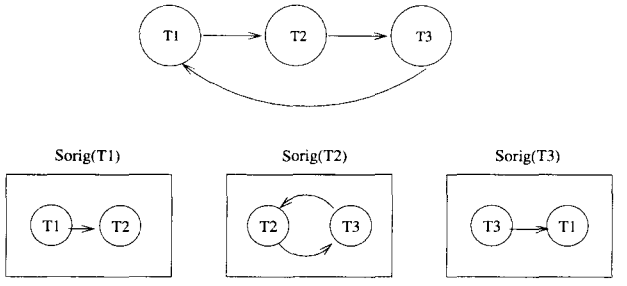
Ao ser recebido um par bloqueante (T, T') no sítio S_k, uma aresta é adicionada de T para T' no $TWF(K)$. Se isso resulta em um ciclo, então um *deadlock* é detectado. Se T' está bloqueado e $S_{orig}(T) \neq S_k$, então para cada transação T'' no conjunto bloqueante(T), é enviado o par bloqueante (T, T'') para $S_{orig}(T'')$ se $S_{orig}(T'') \neq S_k$.

```
(S1)when T precisa de um recurso r no sitio K
        do send req(r,K);
        end do;

(S2) when receive req(r,K) from T
        do
                if ( r está sendo utilizado ) then
                        for ∀Ti que utiliza r do
                                adicione_aresta(T,Ti); //de T para Ti
                        end for;
                        if (existe_ciclo(TWF)) then
                                acuse deadlock;
                        end if;
                        for ∀T' ∈ conjunto_bloqueante(T) do
                                if (S_orig(T) ≠ S_k) then
                                        send par_bloq(T,T') to S_orig(T);
                                end if;
                                if (S_orig(T') ≠ S_k) then
                                        send par_bloq(T,T') to S_orig(T');
                                end if;
                        end for;
                end if;
        end do;

(S3) when receive par_bloq(T,T') de S_k
        do
                adicione_aresta(T,T'); //de T para T'
                if (existe_ciclo(TWF)) then
                        acuse deadlock;
                end if;
                if (T' está bloqueado and S_orig(T) ≠ S_k) then
                        for ∀T'' ∈ conjunto_bloqueante(T) do
                                if (S_orig(T'') ≠ S_k) then
                                        send par_bloq(T,T'') to S_orig(T'');
                                end if;
        end do;
```

Algoritmo 5.1: Descrição do Algoritmo de Menasce e Muntz



(a) Antes de T3 esperar por T1



(b) Depois de T3 esperar por T1

Figura 5.1: Exemplo de execução do algoritmo de Menasce e Muntz 79

Exemplo

A Figura 5.1 ilustra a execução do algoritmo para detectar um *deadlock* envolvendo as transações T_1, T_2 e T_3. Inicialmente T_1 é bloqueada por T_2 e T_2 por T_3. Os sítios de origem de T_1 e de T_2 têm conhecimento do grafo TWF $T_1 \rightarrow T_2$ e $T_2 \rightarrow T_3$, respectivamente. Agora, quando T_3 faz uma requisição e é bloqueada por T_1, o par bloqueante (T_3, T_2) é enviado ao sítio de origem de T_2. Isto adiciona uma aresta de T_3 para T_2 no TWF no sítio de origem de T_2, resultando no ciclo $T_3 \rightarrow T_2 \rightarrow T_3$ e a detecção de um *deadlock* nesse sítio.

Esse exemplo pode ser generalizado para uma situação envolvendo **k** vértices. Inicialmente, as transações de T_1 até T_{k-1} estão bloqueadas e, em cada sítio de origem de cada uma delas, só existe conhecimento a respeito da transação consecutiva no ciclo. Ou seja, o TWF em cada local estaria na seguinte forma:

$S_{orig}(T_1)$: $T_1 \rightarrow T_2$
$S_{orig}(T_2)$: $T_2 \rightarrow T_3$
$\vdots$
$S_{orig}(T_{k-1})$: $T_{k-1} \rightarrow T_k$

Se T_k requisita um recurso que está sendo mantido por T_1, será enviado o par bloqueante (T_k, T_2) para $S_{orig}(T_2)$. Note que, numa visão local, T_2 é um vértice não bloqueado,

sem arestas de saída. Portanto, ele faz parte do conjunto bloqueante de T_k.

Ao chegar o par bloqueante no sítio de T_2, ele dará origem ao par bloqueante (T_k, T_3) que é enviado a $S_{orig}(T_3)$ pelas mesmas razões explicadas acima. Esse processo se repetirá sem alterações até que o par bloqueante (T_k, T_{k-1}) chegue a $S_{orig}(T_{k-1})$, onde causará a adição de uma aresta no grafo de dependência local. Essa aresta formará o ciclo $T_k \rightarrow T_{k-1} \rightarrow T_k$ e o *deadlock* será detectado.

Desempenho

Para um ciclo envolvendo n sítios do sistema, a mensagem de par bloqueante será transmitida $n-2$ vezes até que ela chegue no sítio que acrescentará uma aresta que formará o ciclo no local, detectando assim, o *deadlock*. A complexidade de mensagens é, portanto, $O(n^2)$.

Correção do Algoritmo

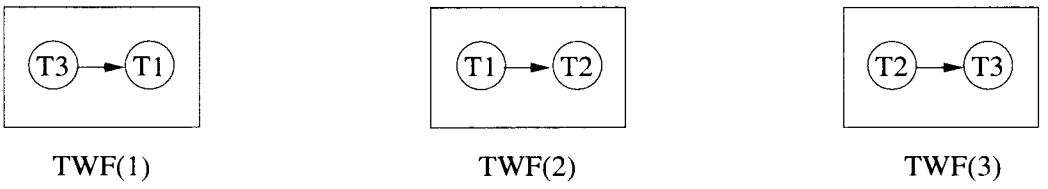
Gligor e Shattuck [GS80] mostraram que este algoritmo pode falhar na detecção de alguns *deadlocks* por duas razões: primeiro, no caso de uma requisição remota, a determinação do bloqueio ou não de uma transação é incorreta porque a determinação não pode ser feita até que a resposta chegue do sítio remoto. Ou seja, uma transação pode ser considerada desbloqueada, quando, na verdade, existe mensagem em trânsito que contraria isso.

Segundo, mesmo que essa resposta chegue para determinar corretamente se a transação está bloqueada ou não, o algoritmo não faz uso dela.

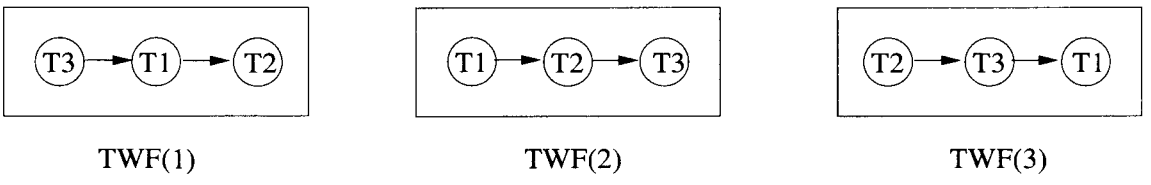
Considere a seguinte situação: assuma uma rede com 3 sítios. S_i o sítio i que contém o recurso R_i e a transação T_i. Além disso, $S_{orig}(T_i) = S_i$. Cada T_i pede R_i e é atendida. Depois disso, T_1 pede o recurso R_2 que está em S_2. Nesse momento é inserida a aresta $T_1 \rightarrow T_2$ em $TWF(2)$ e enviado o par bloqueante (T_1, T_2) para S_1. Enquanto isso, T_2 pede R_3 e T_3 faz requisição de R_1, que são situações similares àquela que foi descrita anteriormente. A Figura 5.2.a ilustra os grafos de dependência até esse ponto.

Apesar de já existir um *deadlock* global, ele não é detectado em nenhum sítio. Mesmo depois que as mensagens de pares bloqueantes chegam aos sítios destinos (dando origem aos grafos da Figura 5.2.b), o *deadlock* continua invisível em qualquer grafo local e a atividade de detecção é encerrada.

Uma modificação proposta para consertar a falha é definir precisamente o status de todas as transações (ativas, bloqueadas ou esperando pela chegada de uma requisição remota de recurso) e propagar pares bloqueantes apropriados quando existe a certeza que uma transação esperando está bloqueada.



(a) Envio das mensagens de pares bloqueantes



(b) Grafos depois do recebimento dos pares bloqueantes

Figura 5.2: Exemplo de *deadlock* não detectado

5.1.2 Algoritmo de Obermarck 82

A idéia apresentada por [Obe82] é na verdade uma melhoria do trabalho de [MM79].

O grafo *wait-for* mantido em cada nó é muito semelhante ao grafo utilizado em técnicas centralizadas, a diferença está no vértice adicional EX (externo). Uma aresta $P_i \rightarrow$EX existe se P_i está esperando por um recurso em outro nó que está sendo utilizado por qualquer processo. O raciocínio é análogo para o caso da aresta EX$\rightarrow P_i$.

A Figura 5.3 mostra a relação de dependência entre os processos em dois nós do sistema.

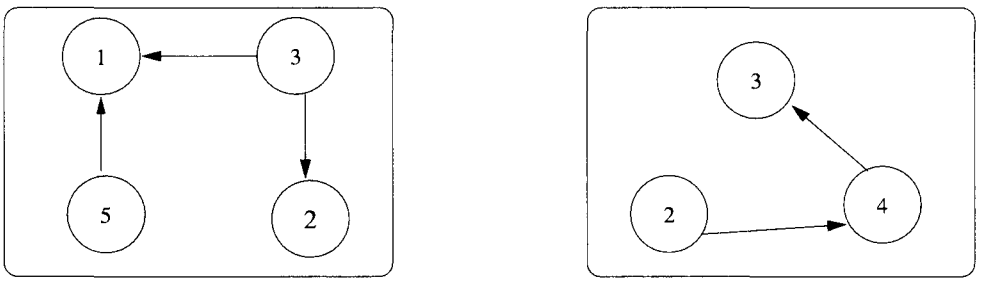
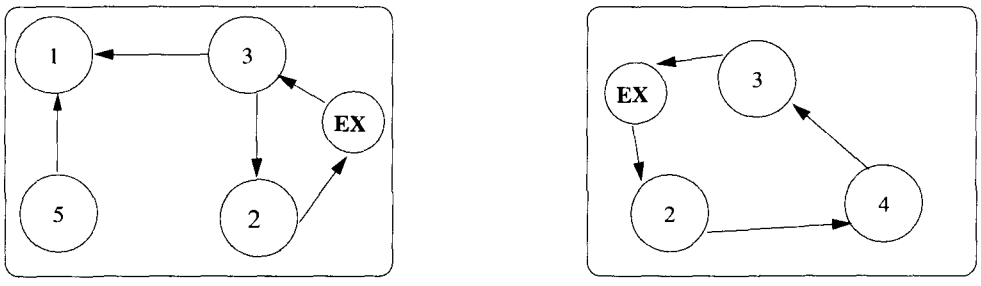


Figura 5.3: Dois grafos *wait-for* locais

A adição do nó externo resulta nos grafos da Figura 5.4.



Nó S1 Nó S2

Figura 5.4: Os grafos *wait-for* com a adição do nó externo

Se um grafo de estado contém um ciclo que não envolve EX, então o sistema está em *deadlock*. A existência de um ciclo contendo EX indica um *deadlock* em potencial. Para ter a certeza que um *deadlock* existe, a informação do provável ciclo é enviada para os processos potencialmente envolvidos.

O Algoritmo e Estruturas de Dados

Cada processo P_i mantém as seguintes estruturas:

- TWFG: estrutura que armazena o grafo de dependência que o processo tem conhecimento;
- In$[P_j]$, $\forall j$: com base no conhecimento de P_i, esse conjunto é formado pelas identificações das transações que estão sendo bloqueadas por P_j;
- Out$[P_j]$, $\forall j$: é análogo ao conjunto In, com a diferença que são guardadas as identificações das transações que bloqueiam P_j.
- n_ciclos: número de ciclos existentes no TWFG;
- v: identificação da vítima selecionada;
- Vítimas: conjunto que registra a identificação de vítimas escolhidas para seja possível descartar cadeias recebidas que incluam transações já abortadas.
- Ciclos: estrutura derivada do TWFG que armazena, para cada ciclo, as identificações das transações que o compõe.

O algoritmo utiliza algumas funções e procedimentos que serão descritos abaixo:

- `construa(TWFG,In,Out)`: construção do grafo de dependência a partir de informações existentes nas mensagens recebidas e nas estruturas de controle da concorrência, tais como tabelas de alocação de recursos ou filas de transações.
- `arestas(id_origem, id_destino, TWFG)`: atualização do grafo a partir das mensagens recebidas.
- `liste_ciclos(TWFG)`: listagem de todos os ciclos encontrados no grafo de dependência.
- `selecione(Ciclos(k))`: Seleciona uma vítima para quebrar o $ciclo_k$.
- `aborte_transação(id_vítima)`: liberação dos recursos e atualização das estruturas In, Out e TWFG das quais a vítima fazia parte.

Os algoritmos *path-pushing*, de um modo geral, são resumidos na repetição cíclica de recebimento de caminhos e construção do grafo até que um veredito sobre a existência ou não de um *deadlock* seja obtido. A abordagem de Obermarck não foge à regra e constitui um dos melhores exemplos, como pode ser observado no pseudocódigo do Algoritmo 5.2.

Exemplo

Para ilustrar o Algoritmo de Obermarck, considere o grafo na Figura 5.4. Suponha que S_1 descobriu o ciclo $P_3 \rightarrow P_2 \rightarrow EX$. Já que P_2 está esperando por um recurso que está no nó S_2, uma cadeia descrevendo o ciclo detectado é transmitida de S_1 para S_2. Quando S_2 recebe essa mensagem, ele atualiza seu grafo, obtendo o grafo de estado da Figura 5.4. Este grafo contém o ciclo: $P_3 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3$ que não contém o nó EX. Assim, o sistema está em um estado de *deadlock* e uma estratégia apropriada é chamada para que faça a recuperação do sistema.

A estratégia chegaria ao mesmo resultado se S_2 tivesse descoberto primeiro o ciclo. No pior caso, ambos os nós teriam descoberto o ciclo e seriam enviadas mensagens de S_1 para S_2 e de S_2 para S_1.

O último laço *for* de (S1) no Algoritmo 5.2 evita essa situação, procurando reduzir o tráfego de mensagens. Assim, quando um nó S_k descobre que seu grafo de estado local possui um ciclo que contém EX, a cadeia somente será enviada se o identificador do primeiro processo no ciclo for maior que o identificador do último processo do ciclo que é anterior a EX.

Desempenho

Considerando questões de desempenho, Obermarck [Obe82] mostra que se n nós estão envolvidos em um *deadlock*, então, no máximo $n(n-1)/2$ mensagens são enviadas, já que

```
(S1) when P_i inicia a detecção
       do
              construa(TWFG, In, Out);
              //Criação das arestas EX→ P_i (obedecendo o critério de existência mencionado)
              arestas(EX,Pi,TWFG);
              //Criação das arestas P_i → EX (obedecendo o critério de existência mencionado)
              arestas(Pi,EX,TWFG);
               n_ciclos:= liste_ciclos(TWFG,Ciclos);
              for i:= 1 to n_ciclos do
                     if ( EX ∉ Ciclos(i)) then
                            v:= selecione(Ciclos(i));
                            remova os ciclos que incluíam a vítima;
                            Vitimas := Vitimas + {v};
                            if (v tem um agente nesse nó) then
                                   for all n, onde n é nó que possui agente da vítima do
                                          send abort(v) to n;
                                   end for;
                            else
                                   for all n, onde n é nó de onde veio uma cadeia com v do
                                          send abort(v) to n;
                                   end for;
                            end if;
                     end if;
              end for;
              for all ciclo = EX → P_1 → P_2 → ... → P_j → EX do
                     if ( id(P_1)>id(P_j) ) then
                            k := Out(P_j); //nó de onde P_j está esperando por uma resposta
                            send cadeia ('EX,P_1,P_2,...,P_j') to k;
                     end if;
              end for;
       end do;


(S2)when received cadeia('Ex,···')
       do
              if (∀id ∈ Vitimas ⋂ cadeia) then
                     descarte mensagem; //transação já abortada
              else
                     faça as ações de (S1);
              end if;
       end do;


(S3)when received abort(v)
       do
              aborte_transação(v);
       end do;
```

Algoritmo 5.2: Descrição do Algoritmo de Obermarck 82

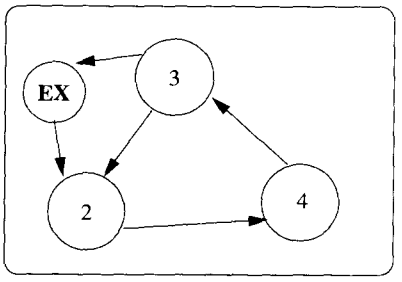


Figura 5.5: O grafo aumentado do nó S_2

as mensagens serão transmitidas para os n nós envolvidos no ciclo, desde o agente que está esperando pelo recurso até o agente responsável pela resposta, seguindo o caminho orientado que representa o ciclo global. Isso resulta em uma complexidade de $O(n^2)$ mensagens trocadas.

O tráfego de mensagens é reduzido pela metade pelo fato da informação de ciclos potenciais ser transmitida somente quando, no ciclo, o primeiro identificador do processo é lexicalmente maior que o último identificador.

Correção do algoritmo

Esse algoritmo, bem como o anterior, estão presentes neste trabalho por razões históricas, pois ambos não são corretos. O algoritmo de Obermarck detecta *deadlocks* falsos e falha na detecção dos existentes, porque os grafos *wait-for* construídos não representam um estado global consistente, como foi observado por Elmagarmid em [Elm86].

5.1.3 Algoritmo de Chandy, Misra e Haas 83

Será apresentado nesta seção um algoritmo simples para detectar *deadlocks* no modelo de recurso. Essa abordagem é devida a Chandy, Misra e Haas [CMH83] que desenvolveram esse trabalho utilizando o princípio caça-aresta (Seção 4.3.2). Na próxima seção haverá a apresentação de outro algoritmo que é encontrado nesse mesmo artigo mas, nesse caso, ele está voltado para o modelo de comunicação, o modelo OR.

O algoritmo para o modelo de recurso

Antes da descrição do algoritmo, será introduzido resumidamente o modelo de requisição sobre o qual os autores trabalharam. Um sistema distribuído, no caso do artigo foi considerado um banco de dados distribuído, é constituído por recursos, controladores e

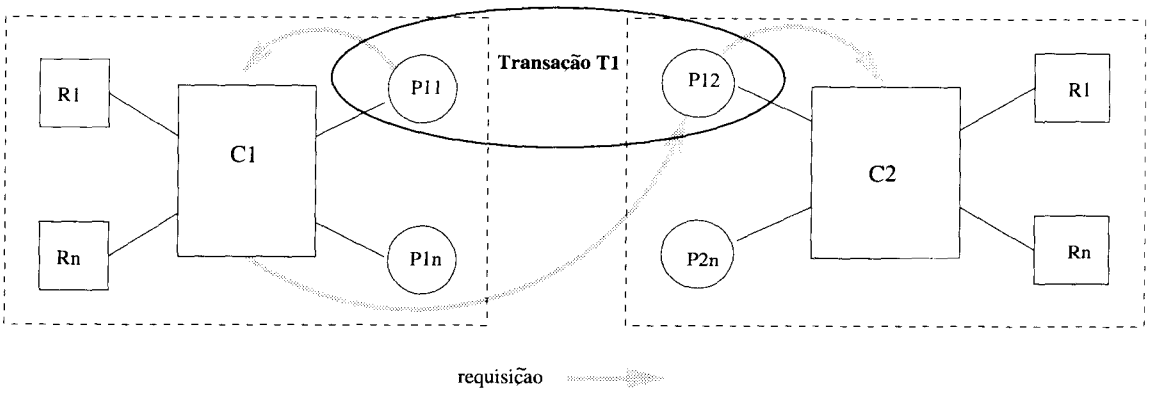


Figura 5.6: Exemplo de requisição de um recurso que não está no local do processo

processos. Um conjunto de **recursos** está associado a um **controlador** que gerencia requisições de um conjunto de **processos**. Um processo só pode requisitar recursos a partir do seu controlador: se esse recurso não se encontra no mesmo local, o pedido é feito de uma forma indireta, como ilustra a Figura 5.6.

Um processo P_i é **dependente** de um processo P_k ($P_i \rightarrow P_k$) se existe uma sequência de processos bloqueados: $P_i, P_{j_1}, P_{j_2}, \ldots, P_{j_m}, P_k$, tal que todos os processos, exceto o primeiro da sequência, mantêm um recurso que está sendo requisitado pelo anterior. P_i é **localmente dependente** se todos os processos da sequência estão associados ao mesmo controlador.

Para determinar se um agente ocioso está em *deadlock*, o seu controlador inicia uma **detecção baseada em sonda** [1]. Uma mensagem **sonda** percorre as arestas (representando as dependências entre os processos) para se detectar ou não a existência de um ciclo. Como foi visto anteriormente, um ciclo é condição necessária e suficiente para a caracterização de *deadlock* no modelo de recurso.

Uma sonda (i, j, k) consiste de uma mensagem formada pelo rótulo que identifica a qual *probe computation* pertence e pela aresta (P_j, P_k). A detecção é iniciada nas seguintes situações:

1. P_j está ocioso,
2. P_j está esperando por P_k ou
3. P_i é dependente de P_j.

As sondas recebidas podem ser aceitas ou não. Para que sejam consideradas relevantes, P_k deve estar ocioso e o controlador de P_k não tinha conhecimento que P_i era dependente

[1] Em inglês *probe computation*.

de P_k e agora pode deduzir essa informação. Para verificar se uma sonda é relevante, o teste da dependência é implementado com ajuda do vetor *dependent*, que está associado a cada controlador. Se $dependent_k(i) = verdadeiro$, então o controlador associado ao processo P_k sabe que $P_i \rightarrow P_k$. A formulação dessas considerações pode ser encontrada em termos de pseudocódigo no Algoritmo 5.3.

```
(S1)if (P_i é localmente dependente dele mesmo) then
        declare deadlock;
 else for ∀P_a, P_b such that
        (P_i é localmente dependente de P_a) and
        (P_a está esperando por P_b) and
        (P_a e P_b estão em diferentes controladores)do

        send probe (i, a, b); //envia a mensagem probe(sonda)do controlador de a para o controlador de b
correspondente à detecção iniciada por i
        end for;
 end if;

(S2)when received probe(i, j, k)
        do
                if (P_k está ocioso) and
                   (dependent(k)(i) = false ) and
                   (P_k ainda não respondeu (positivamente) às req.  de P_j) then
                        dependent(k)(i) = true ;
                        if (k = i) then
                                P_i está em deadlock;
                        else for ∀P_a, P_b such that
                                (P_k é localmente dependente de P_a) and
                                (P_a está esperando por P_b) and
                                (P_a e P_b estão em diferentes controladores) do

                                send probe (i, a, b); // envia a mensagem probe(sonda)do controlador de a para
o controlador de b correspondente à detecção iniciada por i
                                end for;
                        end if;
                end if;
        end do;

(S3)when P_k se ativa
        do
                for all i do
                        dependent_k(i):= false;
                end for;
        end do;
```

Algoritmo 5.3: Descrição do Algoritmo de Chandy, Misra e Haas 83 para o modelo de recurso

Exemplo

Será utilizado um exemplo para ilustrar o comportamento do algoritmo proposto por Chandy *et al* [CMH83]. Para tanto, considere as seguintes relações de dependência entre os vários processos que se encontram na Figura 5.7.

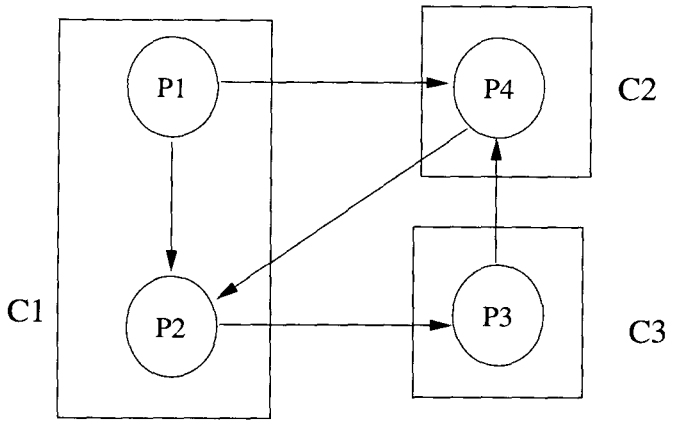


Figura 5.7: Relações de dependência entre os processos

Obedecendo à descrição da forma com que as requisições remotas são feitas, o estado do sistema pode ser representado como na Figura 5.8. As setas mais claras não estão fisicamente implementadas, elas apenas ajudam a associar uma representação (Figura 5.7) à outra.

Prosseguindo com o exemplo, a Figura 5.9 exibe representações simplificadas da Figura 5.8 que servem de base para a explicação do funcionamento do algoritmo.

Correção do Algoritmo

A prova de que um processo está realmente em *deadlock* quando isso é detectado segue do fato (provado por indução) que $dependent_k(i)$ é verdadeiro somente se P_i é dependente de P_k e P_k está passivo.

Por outro lado, pode-se provar que P_i será declarado estar em *deadlock* se existe um ciclo de processos $P_i, P_{j_1}, \ldots, P_{j_m}, P_i$ (onde cada processo na sequência é dependente do próximo) e é iniciada para P_i uma detecção baseada em sonda. A prova segue da hipótese de indução: $dependent_{j_k}(i)$ será atribuído verdadeiro e P_{j_k} enviará a sonda para o próximo na sequência para $1 \leq k \leq K$, para todo K. As provas detalhadas podem ser encontradas em [CM82].

Desempenho

O Algoritmo 5.3 possui mensagens de tamanho fixo. Pelo fato da sonda percorrer as arestas do grafo até que chegue ao ponto inicial e, assim, detectando um ciclo, toda detecção de *deadlock* envolve não mais que a mensagens, onde a é o número de pares de processos que se comunicam na rede. No pior caso, onde a rede de n processos é totalmente conexa, $a = n(n-1)$, resultando na complexidade de $O(n^2)$ mensagens.

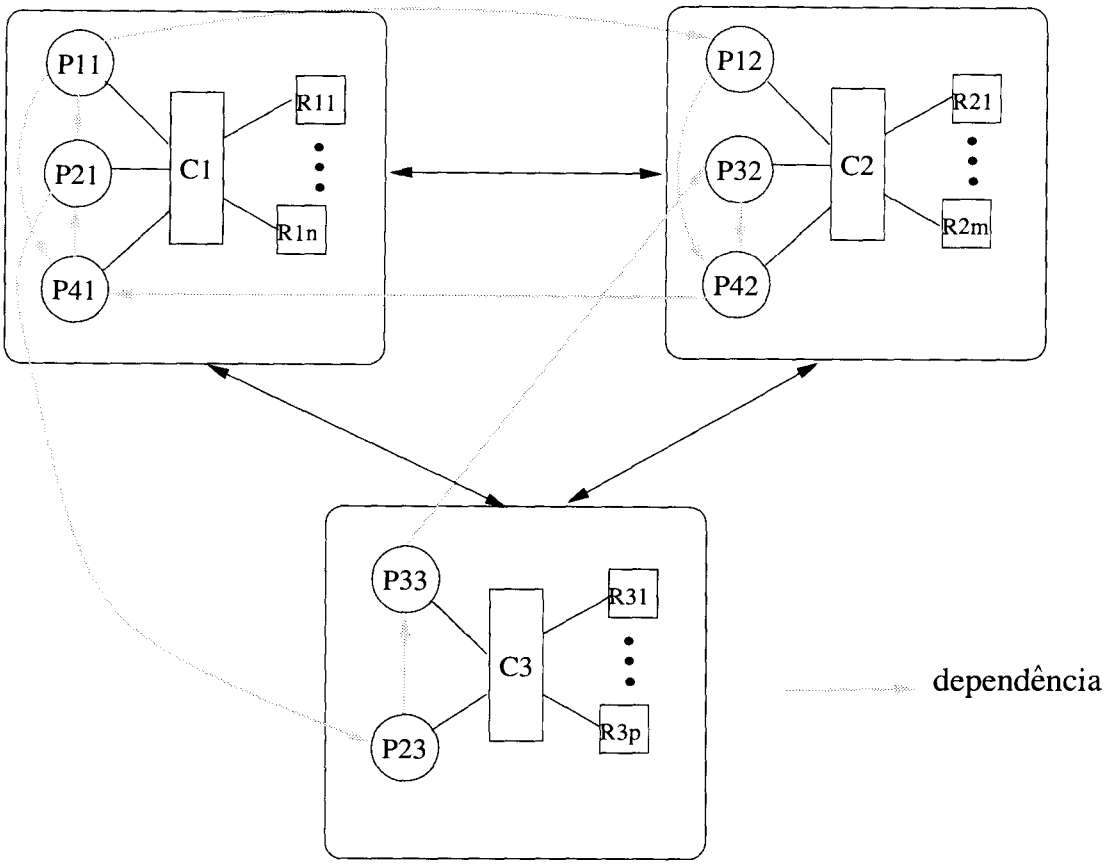


Figura 5.8: Dependências representadas no modelo do sistema assumido

5.2 Algoritmo para o modelo OR

5.2.1 Algoritmo de Chandy, Misra e Haas 83

O modelo de comunicação é mais geral, permitindo tratar do caso que ocorre quando um processo espera por uma combinação lógica de recursos. Para se caracterizar um *deadlock* deve-se verificar a existência de um *knot*. Esse algoritmo é mais uma aplicação da técnica de computações difusas (vide Seção 4.3.3).

O algoritmo do modelo de comunicação

A detecção de *deadlock* é feita através de uma **detecção baseada em consulta** [2], distinta do processo da aplicação. Para isso, utilizam-se as mensagens: $query(i, m, j, k)$ e $reply(i, m, j, k)$. Essas mensagens pertencem ao m-ésimo *round* de detecção iniciado pelo

[2]Em inglês *query computation*.

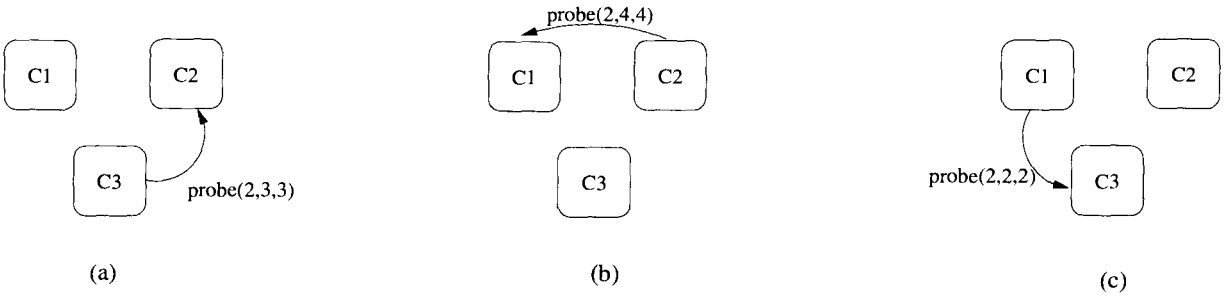


Figura 5.9: Exemplo de execução do algoritmo de Chandy et al. para o modelo AND

processo i e são enviadas de P_j para P_k. Os parâmetros, em ordem, podem ser descritos assim: iniciador(i), número de sequência (m), emissor(j) e receptor(k). Existirá, no máximo, uma mensagem na forma $query(i, m, j, k)$ e, no máximo, uma mensagem na forma $reply(i, m, j, k)$ para aquela mensagem $query$.

As *query computations* possuem as seguintes propriedades:

1. Se P_i está em *deadlock* quando inicia sua m-ésima computação, então receberá $reply(i, m, j, i)$ correspondente a toda mensagem $query(i, m, i, j)$ que é enviada.

2. Se P_i (iniciador) recebeu $reply(i, m, j, i)$ correspondente a toda consulta $query(i, m, j, i)$ que foi enviada, então P_i está em *deadlock*.

Variáveis do algoritmo:

- $latest(i)$ maior número de sequência m em qualquer $query(i, m, j, k)$ recebida por P_k. Inicialmente, $latest(i) = 0$ para qualquer i.

- $engager(i)$, $i \neq k$ é a identidade (j) do processo que ocasionou que $latest(i)$ fosse colocado para o atual valor m, ao enviar $query(i, m, j, k)$ a P_k.

- $num(i)$ número total de mensagens da forma $query(i, m, j, k)$ enviadas por P_k, menos as mensagens $reply(i, m, j, k)$ recebidas por P_k.

- $wait(i)$ se está verdadeiro indica que P_k está ocioso desde a última atualização de $latest(i)$.

- S_i conjunto dos processos dos quais P_i depende.

Exemplo

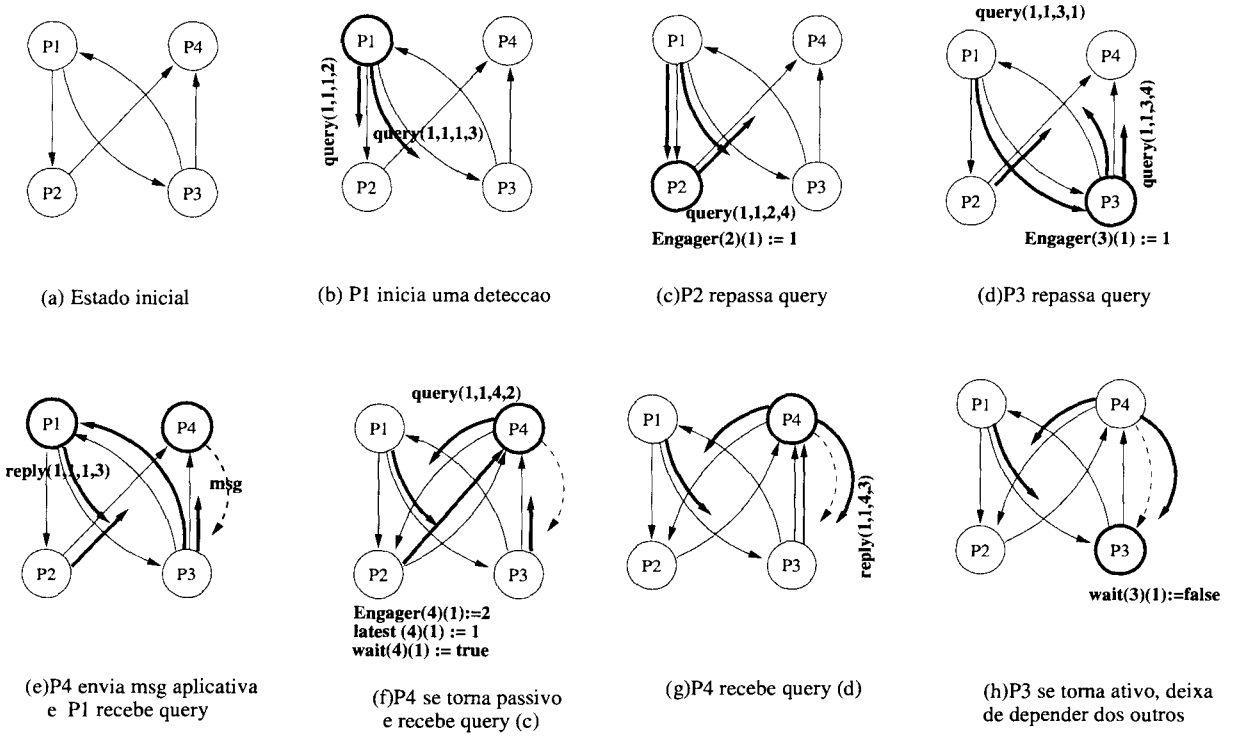
Nessa seção será ilustrada a estratégia apresentada no Algoritmo 5.4. Nas Figuras 5.10 e 5.11 é utilizada a seguinte representação: as setas de menor espessura representam as

```
(S1)when P_i inicia a computação da consulta and estado = passivo
        do
                latest(i):= latest(i) + 1;
                wait(i):= true ;
                send query(i,latest(i),i,j) to ∀P_j ∈ S_i;
                num(i):= |S_i|;
        end do;


(S2)when P_k em execução
        do
                for all i
                        wait(i):= false ;
                end for;
                while (estado = executando) do
                        descarte todas as consultas e respostas recebidas;
                end while;
        end do;


(S3)when P_k ocioso received query(i, m, j, k)
        do
                if (m > latest(i)) then
                        latest(i) := m;
                        engager(i) := j;
                        wait(i) := true ;
                        for all processes P_r ∈ S_k
                                send query(i,m,k,r);
                        end for;
                        num(i) :=|S_k|;
                else
                        if (wait(i)) and (m = latest(i)) then
                                send reply(i,m,k,j) to P_j;
                        end if;
                end if;
        end do;


(S4)when received reply(i, m, r, k)
        do
                if (m = latest(i)) and (wait(i)) then
                        num(i) := num(i) - 1;
                        if (num(i) = 0) then
                                if (i=k) then declare que P_k está em deadlock ;
                                else send reply(i,m,k,j) to P_j, onde j = engager(i);
                                end if;
                        end if;
                end if
        end do;
```

Algoritmo 5.4: Descrição do Algoritmo de Chandy, Misra e Haas 83 para o modelo de comunicação

dependências entre os processos, as de maior espessura são mensagens do algoritmo e as setas pontilhadas são mensagens da aplicação responsáveis pela ativação e desbloqueio do processo.

Inicialmente tem-se a seguinte situação: P_1 espera por P_2 ou P_3, P_2 espera por uma mensagem de P_4, P_3 está dependendo de P_1 ou P_4 e P_1,por sua vez está ativo, como pode ser observado na Figura 5.10.a.



(a) Estado inicial

(b) P1 inicia uma deteccao

(c)P2 repassa query

(d)P3 repassa query

(e)P4 envia msg aplicativa e P1 recebe query

(f)P4 se torna passivo e recebe query (c)

(g)P4 recebe query (d)

(h)P3 se torna ativo, deixa de depender dos outros

Figura 5.10: Exemplo de uma execução do algoritmo para o modelo de comunicação

P_1 inicia a *query computation*, enviando mensagens *query* para os processos em S_i. Na seqüência, há uma sucessão de envios e recebimentos de mensagens do algoritmo e da aplicação, representados nas letras (b)-(p) das Figuras 5.10 e 5.11. Os vértices destacados em cada quadro são aqueles cujo o comportamento é enfocado naquele determinado estágio da execução do algoritmo.

Correção do Algoritmo

No artigo [CMH83] o algoritmo é demonstrado correto através das provas dos seguintes teoremas:

Teorema 14 *Se o iniciador de uma computação difusa está em deadlock quando inicia a computação, então posteriormente ele se declarará em deadlock.*

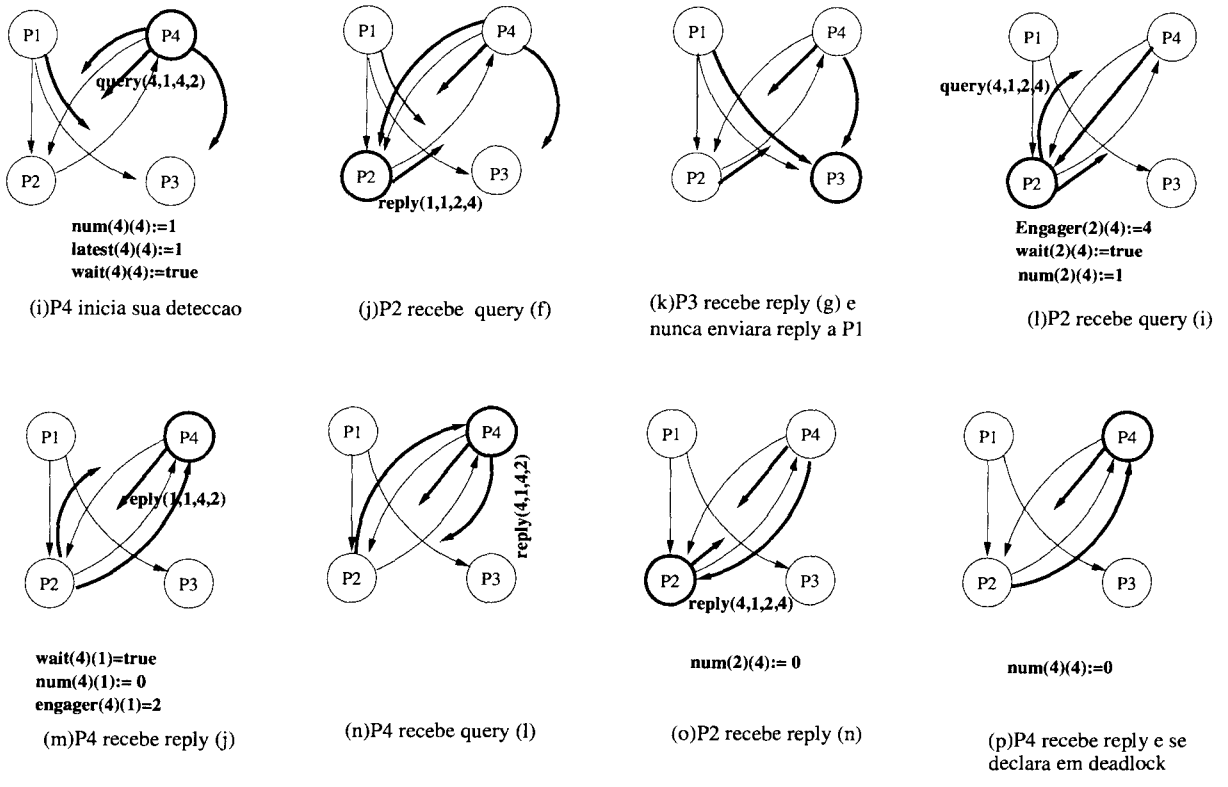


Figura 5.11: Continuação do exemplo de uma execução do algoritmo para o modelo de comunicação

PROVA. Considere um conjunto de processos P que está em *deadlock* no início da detecção. Todos os processo desse conjunto não vai alterar o próprio conjunto de dependentes e não mensagem de atendimento em trânsito entre os processos de P. Como notado em [CMH83] essa situação é equivalente ao problema da terminação distribuída considerada por Dijkstra e Scholten [DS80] e as provas desse trabalho aplicam-se também nesse algoritmo de detecção de *deadlock* que utiliza o princípio da computação difusa [DS80].□

Teorema 15 *Se o iniciador de uma computação difusa se declara em deadlock, então ele pertence a um conjunto em deadlock.*

PROVA. Seja P um conjunto de processos, incluindo o iniciador, que recebe as mensagens *query* durante a detecção. Todo processo antes de responder de volta, deve ter recebido *replies* de todas as *queries* que enviou. Deste fato, segue a prova por indução.

Se o iniciador se declara em *deadlock*, uma resposta deve ter sido enviada para q *queries* da detecção, para todo q. Logo, todo processo em P responde na fase de retração. Segue que, se o processo P_k em P torna-se ativo em um tempo t, depois de ter recebido uma mensagem *query*, então algum processo P_j em P torna-se ativo no tempo t' depois de ter

ter recebido uma *query*, $t' < t$. Usando isso indutivamente, resulta que nenhum processo em P pode voltar à atividade depois de ter recebido uma *query*, portanto P é um conjunto em *deadlock*. □

Teorema 16 *Pelo menos um processo em cada conjunto em deadlock irá detectar o deadlock já que todo processo inicia uma nova computação difusa sempre que se bloqueia.*

PROVA. Pelo teorema 14, o último processo a se bloquear em um conjunto em *deadlock* anunciará que está em *deadlock*.□

Desempenho

A análise do desempenho é similar àquela feita para o algoritmo do modelo AND apresentado pelos mesmos autores. As mensagens são de tamanho fixo e toda detecção de *deadlock* envolve não mais que *a queries* e *a replies*, onde a é o número de pares de processos que se comunicam na rede. No pior caso, $a = n(n-1)$, resultando na complexidade de mensagens $O(n^2)$.

5.3 Algoritmos para o modelo $C_{n,k}$

Para essa classe de algoritmos serão discutidas as seguintes idéias: duas se fundamentam na abordagem de computações difusas e utilizam também a teoria de detecção de estado global; a terceira além de se diferenciar pela filosofia *edge-chasing*, apresenta soluções para questões de tolerância a falhas.

5.3.1 Algoritmo de Bracha e Toueg 87

No modelo $C_{n,k}$, uma transação pode fazer uma requisição com uma combinação *and-or* arbitrária de requisições $C_{n,k}$. Essa combinação é mapeada para uma árvore de processos usando um esquema igual aquele discutido na Seção 4.1.3, com cada processo tendo uma única requisição $C_{n,k}$.

Um algoritmo para esse modelo é apresentado em [BT87] que consiste de uma aplicação da técnica de detecção de estado global descrita na Seção 4.3.4. Será discutido em algum detalhe a primeira das três versões do algoritmo dado por Bracha e Toueg, que assume comunicação síncrona entre processos e um WFG estático.

A segunda versão do algoritmo, que também será discutida, relaxa de forma limitada a restrição de sincronismo. O estado de uma aresta incidente num processo deve ser conhecido por aquele processo para que o algoritmo esteja correto. Dessa forma, o segundo tratamento é basicamente síncrono também — é apenas mais um esquema para simular

sincronismo enviando mensagens de status para frente e para trás, uma idéia denominada como algoritmo de eco ou computação difusa (vide Seção 4.3.3)

O terceiro algoritmo determina primeiro um *snapshot* global do WFG usando a técnica introduzida em [CL85]. Este estado global pode então ser usado para executar um dos dois primeiros algoritmos para detectar *deadlock*.

Um processo é bloqueado quando envia n requisições e volta à execução quando recebe k atendimentos. Nesse caso, ele envia mensagens de liberação (*relinquish*) para os $n - k$ processos, informando-os que a aresta criada no pedido não existe mais. Mensagens de liberação são necessárias porque cada processo tem que ter conhecimento do seu conjunto de arestas de entrada (IN) e do seu conjunto de arestas de saída (OUT).

Uma mudança de estado é formalmente modelada através de transformações no grafo. Se, ao aplicar uma transformação s no grafo G, é resultado o grafo G', essa situação é denotada por $G \vdash^s G'$. O *deadlock* é definido pelos autores do trabalho como segue:

Definição 17 *Um processo p está em deadlock em um WFG G se não existe um escalonamento σ tal que $G \vdash^\sigma G'$ e p está em execução.*

O primeiro algoritmo é uma chamada encadeada de computações difusas com uma pequena mudança feita quando um processo deixa seu estado neutro. Somente a primeira consulta que cada processo recebe é que o fará participar da detecção. Todas as consultas posteriores são respondidas imediatamente, mesmo se o processo retornou ao seu estado neutro. Uma consequência desse comportamento é que o número de mensagens trocadas durante a invocação do algoritmo é reduzido. Outra consequência, como observado em [Kna88], "é que muitas das propriedades das computações difusas são perdidas e as provas de correção tornam-se complicadas e quase incompreensíveis".

A primeira idéia encontra-se descrita no Algoritmo 5.5. Esse algoritmo, como se encontra demonstrado no trabalho original[BT87], pode ser descrito de outra forma, como um aninhamento de duas instâncias de um algoritmo similar a uma computação difusa, que Bracha e Toueg chamam de *Clousure* (Algoritmo 5.6) Em linhas gerais, isso funcionaria da seguinte maneira:

Invocação mais externa de Clousure: Encontre o conjunto S de todos os processos executando que são alcançáveis através da propagação de mensagens de consulta, começando por algum iniciador i.

1. Cada $p \in S$ simula o atendimento de todos os recursos que mantém e pelos quais os outro processos estão esperando. Uma instância de Clousure é invocada para cada p.

Inicialização

```
OUT := {u | (v,u) ∈ E };
IN := {u | (u,v) ∈ E };
notified := false;
free := false;
n_granted := 0;


(S1)when v received (NOTIFY) from u
        do
                if (not notified) then Notify();
                end if;
                send (DONE) to u;
        end do;

(S2)when v received (GRANT) from u
        do
                n_granted := n_granted + 1;
                if ( not free and (n_granted ≥ n)
                        then Grant();
                end if;
                send (ACK) to u;
        end do;

procedure Notify()
begin
        notified := true;
        for all w ∈ OUT do
                send (NOTIFY) to w;
        end for;
        if (n = 0) then
                Grant();
        end if;
        for all w ∈ OUT do
                await (DONE) from w;
        end for;
end

procedure Grant()
begin
        free := true;

        for all w ∈ IN do
                send (GRANT) to w;
        end for;
        for all w ∈ IN do
                await (ACK) from w;
        end for;
end
```

Algoritmo 5.5: Descrição do Algoritmo de Bracha e Toueg 87 (primeira versão)

2. Os atendimentos são propagados e o número a de atendimentos simulados são recebidos em cada processo alcançável p. Esse número recebido é comparado com o número r de recursos necessários para que p volte à execução novamente. Se $a<r$, então p nunca será atendido em todas as suas requisições, logo está em *deadlock*.

No pseudocódigo apresentado no Algoritmo 5.6, é utilizada a variável *in_C* que serve como um mecanismo que evita que o nó execute *Clousure* mais de uma vez.

```
Inicialização

OUT := {u|(v,u) ∈ E};
in_C := false;
IN := ∅;

procedure Clousure()
begin
        in_C := true;
        for all w ∈ OUT do
                send (NOTIFY) to w;
        end for;
        for all w ∈ OUT do
                await (DONE) from w;
        end for;
end

when v received (NOTIFY) from u
        do
                IN := IN ⋃ {u};
                if (not in_C and ( v ∈ S or P(v,IN))
                        then Clousure();
                end if;
                send (DONE) to u;
        end do;
```

Algoritmo 5.6: Algoritmo do Clousure

A segunda versão do algoritmo para detecção de *deadlock* utiliza um esquema de cores nas arestas para modelar os estados dos canais. Uma aresta (u, v) é:

cinza, se u envia *request* para v que ainda não foi recebido e u não enviou *relinquish* para v.

preta, se v recebeu *request* de u mas ainda não enviou *reply* para u e u não enviou *relinquish* para v.

branca, se v enviou *reply* para u que ainda não foi recebido e u não enviou *relinquish* para v

translúcida, se u enviou *relinquish* para v e ainda não foi recebido.

Seja v um nó ativo em um grafo de dependência G. Utilizando esse esquema de cores, as transformações que o grafo G sofre podem ser descritas da seguinte forma:

1. Adicionando k arestas cinzas para o nó v sem arestas de saída e atribuindo $n_v := r$ $(1 \leq r \leq k)$.
2. Mudando uma aresta de cinza para preta.
3. Mudando uma aresta (u, v) de preta para branca.
4. Removendo uma aresta (u, v) branca e diminuindo n_u de 1. Se $n_u = 0$, então todas as arestas de saída de u são feitas translúcidas.
5. Removendo uma aresta translúcida.

A terceira abordagem aplica-se a um sistema que está em constante mudança. Utilizam-se mensagens *freeze* que são propagadas pelo sistema. Quando um processo recebe essa mensagem, ele registra um estado local. Sobre esse estado fixo é executado o Algoritmo 5.5 ou o Algoritmo 5.7.

O embasamento teórico sobre construção de *snapshots* consistentes é obtido resumidamente na Seção 4.3.4. Para um maior detalhamento, consulte [CL85].

Exemplo

Para ilustrar a primeira versão do algoritmo que se encontra descrito no Algoritmo 5.7, observe as Figuras 5.12 e 5.13.

O algoritmo procede da seguinte forma: um nó iniciador começa a fase de notificação ao enviar mensagens *notify* para o seu conjunto de saída. Essas mensagens vão se propagando até que cheguem aos nós folhas.

A partir desse ponto, começa a fase dos atendimentos, onde mensagens *grant* simularão as respostas aos pedidos. Mensagens *ack* e *done* são necessárias à terminação do algoritmo de detecção.

No exemplo, na letra (h) da Figura 5.13, é recebida a última mensagem *done* pelo nó iniciador. Como a variável $free$ continua falsa, é declarado um *deadlock*.

Correção do Algoritmo

Observe que todas as instâncias internas de Clousure terminarão antes da externa. O algoritmo sempre termina e na terminação cada processo sabe se está em *deadlock* ou não. Esta é uma característica de uma computação difusa.

A prova dada pelos autores utiliza argumentos puramente operacionais e nenhuma invariante é dada. Além do mais, a segunda versão do algoritmo não pode estar correta,

Inicialização

```
OUT_black := {u | (v,u) ∈ E and is black};
IN_black := {u | (u,v) ∈ E and is black};
notified := false;
free := false;
n_granted := 0;


(S1)when v received (NOTIFY) from u
        do
                if (not notified) then Notify();
                end if;
                send (DONE) to u;
        end do;

(S2)when v received (GRANT) from u
        do
                n_granted := n_granted + 1;
                if ( not free and (n_granted ≥ n - n_greywhite)
                        then Grant();
                end if;
                send (ACK) to u;
        end do;

procedure Notify()
begin
        notified := true;
        for all w ∈ OUT_black do
                send (NOTIFY) to w;
        end for;
        if (n - n_greywhite ≤ 0) then
                Grant();
        end if;
        for all w ∈ OUT_black do
                await (ACK) from w;
        end for;
end

procedure Grant()
begin
        free := true;

        for all w ∈ IN_black do
                send (GRANT) to w;
        end for;
        for all w ∈ IN_black do
                await (ACK) from w;
        end for;
end
```

Algoritmo 5.7: Descrição do Algoritmo de Bracha e Toueg 87 (segunda versão)

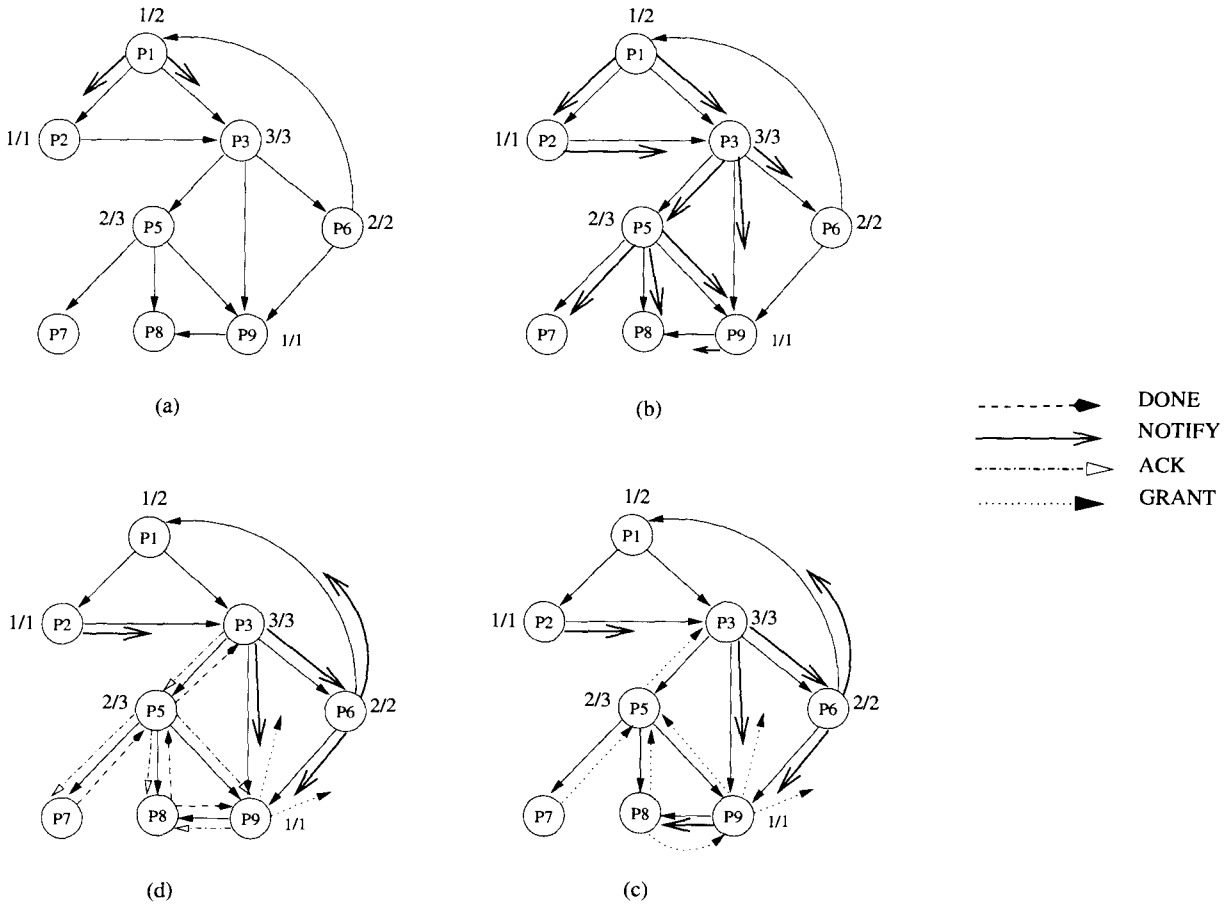


Figura 5.12: Exemplo de uma detecção de *deadlock*

como Knapp [Kna88] observa. Segundo Bracha e Toueg " se um iniciador i começa o algoritmo de detecção em um WFG G colorido, então o algoritmo termina. Além disso, quando i termina, a variável local $free_i \equiv$ true se, e somente se, i não está em *deadlock* em G ".

Deadlock é colocado como na definição 17, com G contendo arestas pretas somente. Um contra-exemplo da definição de Bracha e Toueg é um ciclo de arestas cinzas em G. Essas arestas tornar-se-ão pretas em um tempo finito. Logo, não existe escala σ tal que ($G \vdash^{\sigma} G'$ e quaisquer dos processos no ciclo estão ativos em G'). Logo, pela definição 17, todos os processo no ciclo estão em *deadlock*.

Como Bracha e Toueg consideram inexistentes as arestas cinzas, se o iniciador está entre os processos do ciclo, ele erroneamente sentenciará que não está em *deadlock*. Apesar desse *deadlock* ser detectado em algum tempo posterior, isso não salva o algoritmo de não atender à sua especificação.

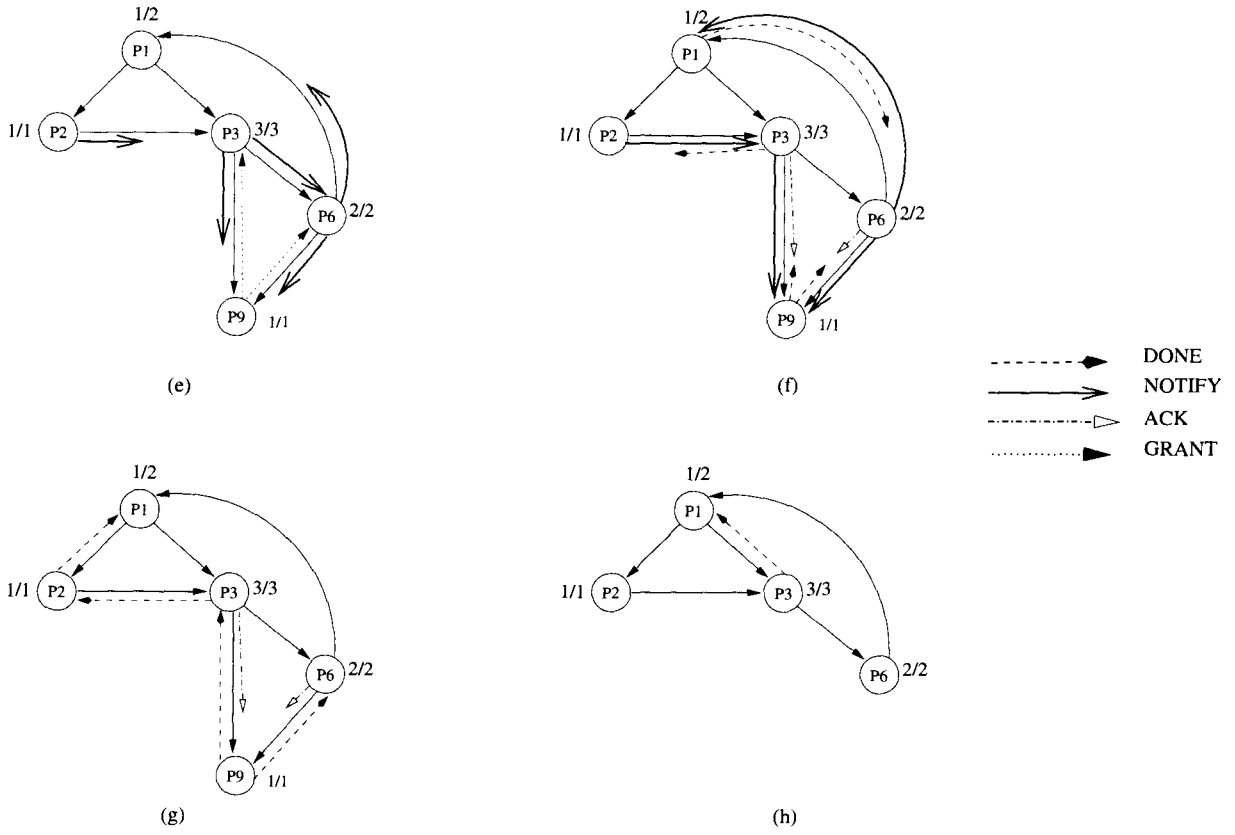


Figura 5.13: Exemplo de uma detecção de *deadlock* realizada através do algoritmo de Bracha e Toueg

Desempenho

A análise de desempenho é resumida da seguinte forma. No caso síncrono, no máximo, $4n(n-1)$ mensagens são necessárias, sendo n o número de nós. Na versão assíncrona, mais mensagens têm que ser trocadas para determinar o estado das arestas entre os processos. No pior caso uma sobrecarga de $O(n^2)$ mensagens é suficiente. Já que o algoritmo primeiro toma um *snapshot* e depois executa a detecção de *deadlock*, o tempo entre a ocorrência de um *deadlock* e a sua detecção pode ser significativo.

Comparado a outros esquemas de detecção, este algoritmo explora pouco a concorrência. Primeiro, porque não existe uma verdadeira concorrência entre a detecção e a aplicação subjacente. Segundo, o algoritmo procede por fases (chegando a três quando são feitas otimizações de espaço), o que sequencializa a detecção.

5.3.2 Algoritmo de Kshemkalyani 94

Uma outra abordagem para detectar *deadlocks* foi apresentada em [KS94]. Nesse trabalho, como em muitas outras técnicas de detecção, a relação de interdependência é modelada no grafo *wait-for* (WFG). Baseado na informação contida nesse grafo, são detectados *deadlocks* generalizados, que correspondem a *deadlocks* no modelo *P-out-of-Q* ou modelo $C_{n,k}$.

O Algoritmo

Em [KS94] é proposto um algoritmo que condensa em uma única fase a obtenção do estado do sistema e a detecção de *deadlocks*. Na verdade, essa fase é composta por dois passos que acontecem de forma concorrente: a varredura do WFG de dentro para fora, a partir de um vértice iniciador (registro de um *snapshot* consistente) e a varredura de fora para dentro (simulação do atendimento dos pedidos para se detectar um *deadlock*).

No segundo tipo de varredura é utilizado um procedimento de redução do grafo. Uma aresta é reduzida se, na simulação, o pedido que ela representa pode ser atendido. Um vértice é reduzido se P das Q requisições que o bloqueia podem ser atendidas durante o processo de redução.

Os eventos são classificados em: eventos computacionais (aplicação) e eventos de controle (detecção de *deadlock*).

Entre os eventos computacionais serão descritos no algoritmo 5.8 o envio e recebimento de algumas mensagens que interessam ao entendimento do algoritmo.

São utilizadas as seguintes estruturas de dados para registrar o estado:

- t_i: tempo local corrente;
- t_block_i: o tempo no qual o vértice i foi última vez bloqueado;
- in_i: conjunto dos vértices que possuem requisições pendentes em i;
- out_i: conjunto dos vértices pelos quais i espera desde t_block_i;
- p_i: o número de respostas necessárias para desbloquear.

O algoritmo de detecção precisa dos seguintes tipos de mensagens: FLOOD (utilizada durante a fase de expansão para percorrer as arestas de entrada e assim registrar o *snapshot*), ECHO (utilizada na fase de redução para simular o atendimento de uma requisição) e SHORT (usada para determinar a terminação do algoritmo).

Em linhas gerais, o algoritmo funciona da seguinte maneira: um vértice iniciador (pode ser qualquer um no sistema) registra seu estado interno e envia mensagens FLOOD para

```
(S1) when send REQUEST
       do
              p_i := valor dependente da aplicação;
              for each (vértice j de quem i depende) do
                     out_i := out_i ∪ {j};
                     send REQUEST(i) to j;
              end for;
              t_block_i := t_i;
       end do;


(S2) when received REQUEST(k)
       do in_i := in_i ∪ {k};
       end do;


(S3) when send REPLY
       do
              in_i := in_i \ {k};
              send REPLY(i) to k;
       end do;


(S4) when received REPLY(j)
       do
              out_i := out_i \ {j};
              p_i := p_i - 1 ;
              if (p_i = 0) then
                     t_block_i := 0;
                     for each (j ∈ out_i) do send CANCEL(i) to j;
                     out_i := ∅;
              end if;
       end do;


(S5) when received CANCEL(k)
       do in_i := in_i \ {k};
       end do;
```

Algoritmo 5.8: Mensagens computacionais da aplicação

todos os vértices que estão no conjunto *out*, sendo que na mensagem é enviado um peso p (a soma de todos os pesos enviados nas mensagens é igual a 1).

Se o vértice que recebe um FLOOD está bloqueado, ele envia FLOOD´s da mesma maneira feita pelo iniciador, com a diferença que será distribuído entre as mensagens o peso p recebido.

Se o vértice está ativo (não depende de nenhum atendimento), ao receber um FLOOD, retorna o peso recebido através de uma mensagem ECHO destinada para quem o enviou FLOOD (redução da aresta). Se essa mensagem FLOOD recebida não é a primeira, o peso recebido é enviado diretamente para o vértice iniciador através de uma mensagem SHORT.

No algoritmo 5.9 são apresentados, de forma mais detalhada, a inicialização de um novo *snapshot* e o tratamento das mensagens FLOOD.

A estrutura de dados para registrar o snapshot consiste de um vetor de registros, Ls_i, cada registro composto pelos seguintes campos:

- *out*: conjunto que armazena identificação dos processos que dependem de P_i. Inicialmente, *out* é igual a $\emptyset$.
- *in*: conjunto que armazena identificação dos processos dos quais P_i depende. É inicializado com $\emptyset$;
- t: registra o tempo da detecção mais recente que o processo têm conhecimento. Inicializada com zero.
- p: campo que registra a quantidade de respostas necessárias para que se tenha uma simulação de atendimento. Inicialmente, $p = 0$.

Ao receber a p-ésima mensagem ECHO (redução do vértice), com um determinado peso w, o vértice envia mensagens de ECHO para os vértices que pertencem ao seu conjunto *in*, distribuindo entre as mensagens o peso w recebido.

Os pesos recebidos são totalizados pelo iniciador de tal forma que quando o total chega a 1 o algoritmo declara que um *deadlock* ocorreu. No caso de não existir tal situação, o grafo será todo reduzido, as p requisições serão atendidas na simulação e será declarado o fim da detecção.

Exemplo

Considere o grafo de dependência apresentado na Figura 5.14. Nele pode-se observar as condições de ativação de todos os processos bloqueados. Os elementos da figura são utilizados conforme visto na Seção 4.1.3.

```
(S6)initiate SNAPSHOT
init := i;              wtinit := 0;
LS_i.t := t_i;          LS_i.out := out_i;
LS_i.in := ∅;           LS_i.p := p_i;
send FLOOD(i, i, t_i, 1/ | out_i |) for each j ∈ out_i;


(S7)when received FLOOD (k, init, t_init, w)
//mensagem válida para um novo snapshot
        do
                if(LS_i.t< t_init) and (k ∈ in_i) then
                        LS_i.out := out_i; LS_i.in := {k};
                        LS_i.t := t_init; LS_i.p := p_i;
                end if;
                if (p_i > 0) then
                        send FLOOD(i, init, t_init, w/ | out_i |)to j ∈ out_i;
                end if;
                if (p_i = 0)then
                        send ECHO(i, init, t_init, w) to k;
                end if;

//FLOOD de um snapshot novo recebido através de uma aresta que deixou de
//existir. Uma resposta já havia sido devolvida ao emissor do FLOOD.
                if (LS_i.t < t_init) and (k ∉ in_i)
                        then send ECHO(i, init, t_init, w) to k;
                end if;

//FLOOD do snapshot atual recebido através de uma aresta que deixou de
// existir. Uma resposta já havia sido devolvida ao emissor do FLOOD.
                if (LS_i.t = t_init) and (k ∉ in_i)
                        then send ECHO(i, init, t_init, w) to k;
                end if;

//mensagem válida para um snapshot atual
                if(LS_i.t = t_init) and (k ∈ in_i) then
                        LS_i.in := LS_i.in ∪{k};
                        if (LS_i.p = 0)
                                then send ECHO(i, init, t_init, w) to k;
                        end if;
                        if (LS_i.p > 0)
                                then send SHORT(init, t_init, w) to init;
                        end if;
                end if;

// FLOOD ultrapassado
                if (LS_i.t > t_init) then descarte a mensagem;
                end if;
        end do;
```

Algoritmo 5.9: O tratamento da mensagem FLOOD

```
(S8) when received ECHO(j, init, t_init, w)
       do
              if ( LS_i.t = t_init ) then
                       LS_i.out:= LS_i.out - {j} ;
                     if ( LS_i.p = 0) then
                            send SHORT(init, t_init, w) to init;
                     end if;
                     if ( LS_i.p > 0) then
                              LS_i.p:= LS_i.p - 1;
                            if ( LS_i.p = 0) then
                                   if (i = init) then
                                          declare que não existe deadlock;
                                          exit;
                                   end if;
                                   send ECHO(i,init,t_init, w/|LS_i.in| to ∀k ∈ LS_i.in;
                            else  // LS_i.p ≠ 0
                                   send SHORT(init, t_init, w) to init;
                            end if;
                     end if;
              end if;
       end do;


(S9) when received SHORT(init, t_init,w)
       do
              if (t_init = t_block_i) and ( LS_i.p = 0) then
                     descarte a mensagem;
              end if;
              if (t_init = t_block_i) and ( LS_i.p > 0) then
                     wtinit := wtinit + w;
                     if (wtinit = 1) then
                            declare deadlock e aborte;
                     end if;
              end if;
       end do;
```

Algoritmo 5.10: O tratamento das mensagens ECHO e SHORT

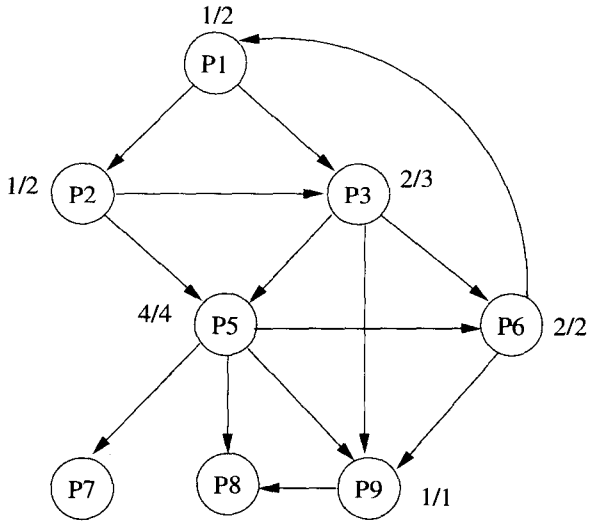


Figura 5.14: As relações de dependência entre os processos

O processo P_1 é o iniciador de uma detecção que começa pela fase de varredura em direção às folhas. Mensagens FLOOD são transmitidas como visto na Figura 5.15.a. O processo iniciador possui um peso igual a um e o divide igualmente pelas arestas que saem dele. No caso do exemplo escolhido tem-se o processo P_1 que envia FLOOD's com pesos iguais a 1/2.

Uma mensagem FLOOD ao chegar a um processo que está esperando por uma combinação de recursos é repassada por todas as arestas de saída, sendo que o peso que foi recebido na mensagem é subdividido pelo número de arestas que vão repassar essa mensagem. Por exemplo, o processo P_3 na Figura 5.15.a recebe o peso 1/2, o subdivide pelo número de arestas de saída e transmite FLOOD's com pesos iguais a 1/6.

Nas ilustrações feitas pode-se observar o recebimento de mensagens FLOOD nas mais variadas situações. Tem-se na Figura 5.15.b o recebimento de uma mensagem quando já foi enviado o recurso pedido e não existe mais a dependência entre P_7 e P_5. Na Figura 5.15.a tem-se a chegada de FLOOD's em nós folhas e não folhas e na Figura 5.17.g o recebimento de FLOOD's subsequentes ao primeiro que foi recebido. Para todos os casos, o tratamento da mensagem pode ser conferido no algoritmo 5.9.

Depois do recebimento de uma mensagem FLOOD, são feitas ações correspondentes à fase de redução. Como já foi dito, uma aresta é reduzida quando é transmitido um ECHO através dela, isso seria equivalente a um atendimento simulado. A redução de um vértice, representado pela área pontilhada como pode-se ver na Figura 5.16.c, dá-se quando o vértice recebe o p-ésimo atendimento dos q recursos que havia solicitado.

Se um FLOOD é recebido por um nó folha, a redução da aresta dá-se com a resposta dada por uma mensagem ECHO que retorna o peso recebido na mensagem FLOOD correspondente.

Quando uma mensagem ECHO não é capaz de reduzir um nó, como acontece a P_5 que só recebe 3 dos 4 atendimentos necessários, o peso recebido em cada mensagem ECHO é retransmitido ao iniciador através de uma mensagem SHORT.

Se um ECHO ocasiona a redução de um determinado vértice, como visto em P_4 na Figura 5.16.d, o processo envia ECHO's por todas arestas de entrada com o peso recebido no ECHO responsável pela redução divido pelo número de processos que dependem do nó reduzido.

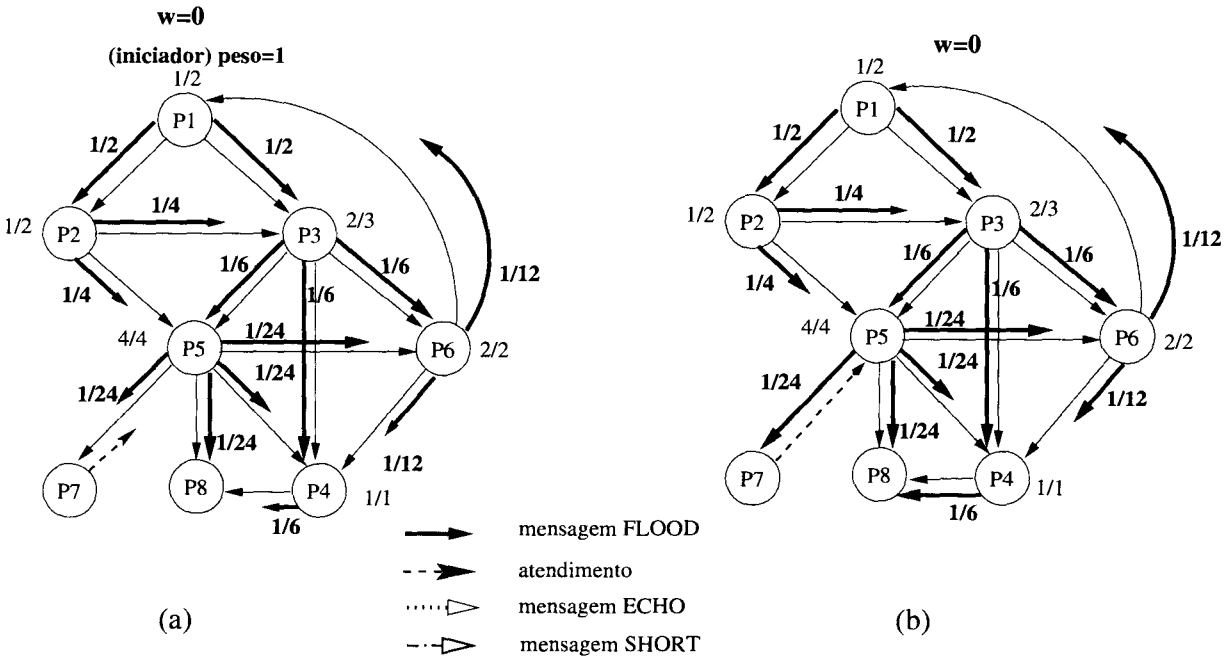


Figura 5.15: Exemplo de uma detecção de *deadlock*

A partir do segundo FLOOD recebido por cada processo (veja na Figura 5.17.e e Figura 5.17.g), o peso recebido é imediatamente retornado ao iniciador através de uma mensagem SHORT, sem haver retransmissão aos possíveis filhos.

Observe que mensagens SHORT e ECHO recebidas pelo iniciador alteram o contador de pesos recebidos que é manipulado somente pelo responsável pelo início da detecção. Cada peso é somado à variável e, se esse contador chegar ao peso inicial que era igual a 1, um *deadlock* é declarado.

No caso de não existir uma situação de *deadlock*, todo o grafo (arestas e vértices) será reduzido. Quando o iniciador receber o p-ésimo ECHO, a detecção terminará declarando que não existe *deadlock*.

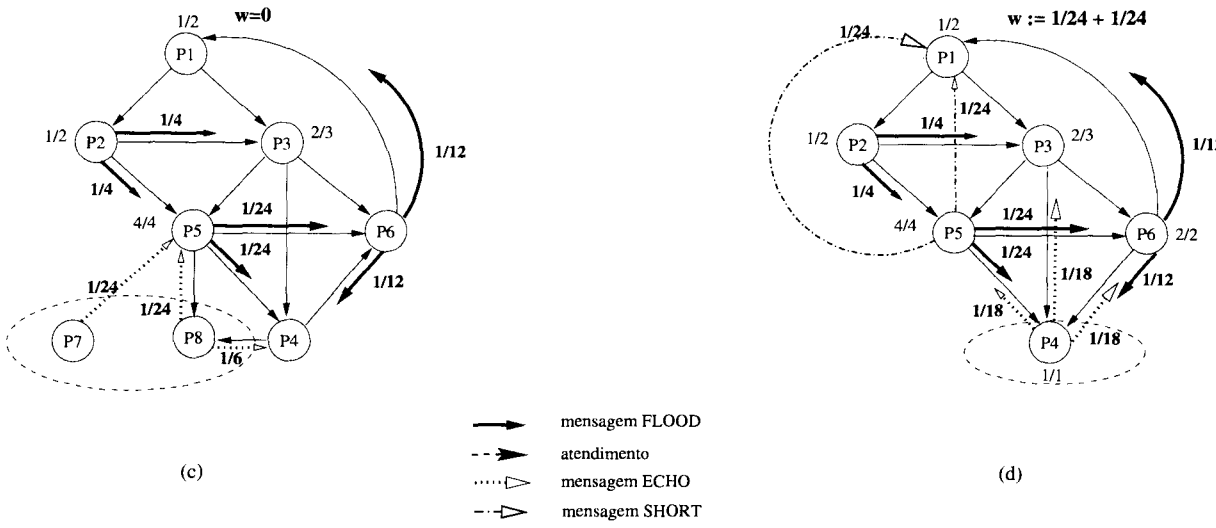


Figura 5.16: Exemplo de uma detecção de *deadlock*

Correção do Algoritmo

Para provar que o algoritmo que foi proposto é correto, os autores mostram que ele satisfaz às seguintes condições:

1. A execução do algoritmo termina.
2. Todo WFG que é alcançável a partir do iniciador é registrado de forma consistente na fase de varredura.
3. As mensagens ECHO reduzem corretamente o WFG registrado.

Para primeira consideração, é útil colocar-se a seguinte invariante do algoritmo em relação ao peso.

$$\sum_{\forall FLOOD} w(FLOOD) + \sum_{\forall ECHO} w(ECHO) + \sum_{\forall SHORT} w(SHORT) + wt_{init} = 1 \qquad (5.1)$$

Quando a variável wt_{init} chega a 1, o algoritmo termina. Enquanto os pesos das mensagens não são computados (não chegaram no iniciador), é considerado que elas estão em trânsito.

Para ter-se uma medida do quanto a terminação demoraria é necessária a introdução do conceito de *hop*. Um *hop* de mensagem se refere a uma transferência lógica de uma mensagem do emissor ao receptor.

Nesse caso, é demonstrado que o algoritmo termina em um número finito de *hops* de mensagens e que esse número é limitado superiormente por $2e$, onde e é o número de arestas no grafo de dependência.

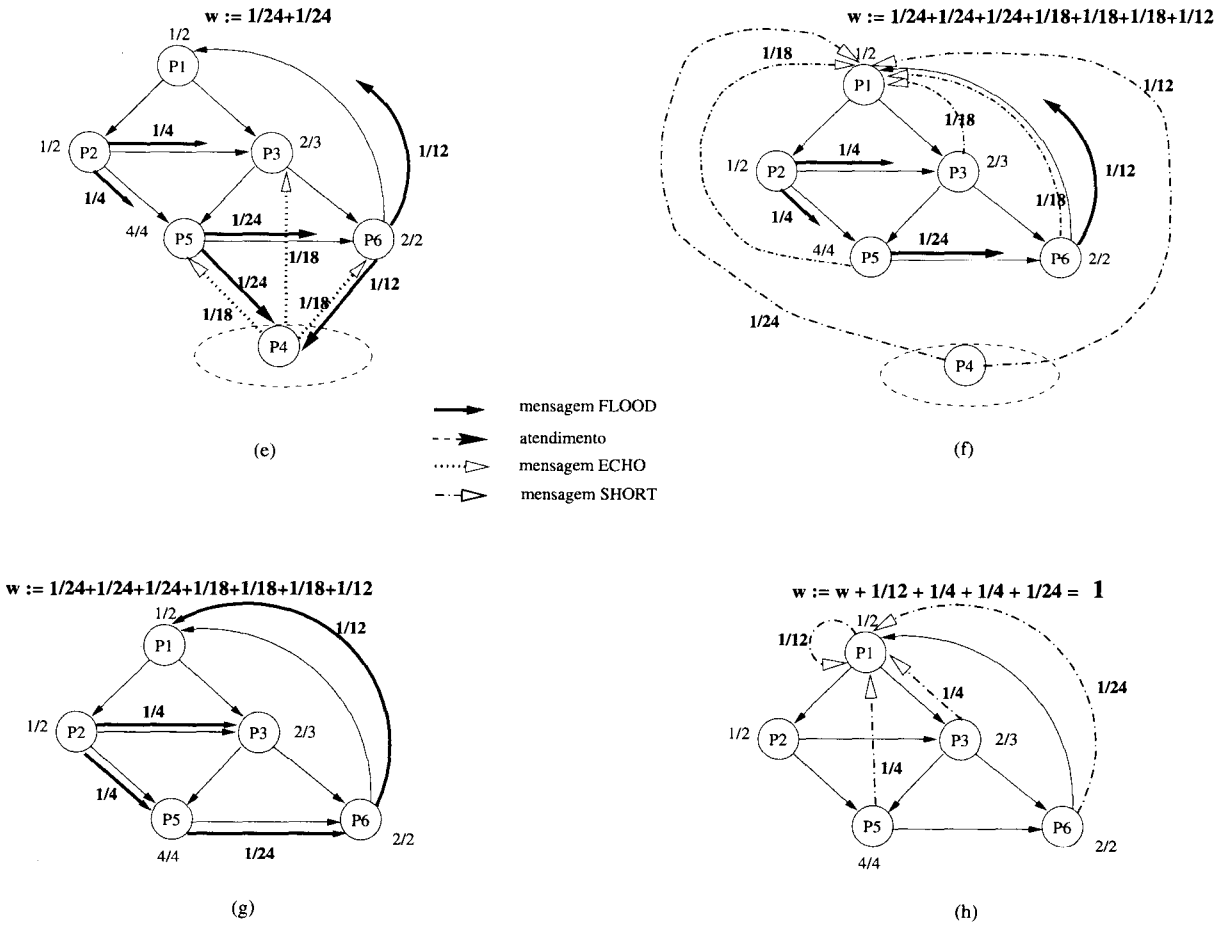


Figura 5.17: Continuação do exemplo de execução do algoritmo

Como é provado no artigo original que, no máximo um FLOOD (expansão) e um ECHO (redução) podem ser enviados por cada aresta do grafo, segue que o algoritmo termina em, no máximo, $2e$ *hops* de mensagens. Esse resultado também se deve ao fato que mensagens SHORT, além de levar um único *hop*, nunca transferem seu peso para mensagens FLOOD ou ECHO.

Para o segundo resultado (a fase de expansão cumpre seu objetivo), os autores relacionam os argumentos que provam que essa fase satisfaz a completeza (todo conjunto alcançável de i registra o seu estado local) e a consistência (se o recebimento de uma mensagem é registrado no estado local, o envio dessa mensagem também o é no estado local do nó que enviou a mensagem). Aos leitores interessados em maiores detalhes sugere-se a consulta a [KS94].

Antes de passar para a terceira demonstração, é preciso definir a função h nos nós do grafo como segue:

$$h(i) \longleftarrow \begin{cases} 0 & \text{se } i \text{ for um nó folha no WFG.} \\ \infty & \text{se } i \text{ está em } \textit{deadlock}. \\ 1 + (p_i\text{–ésimo menor de } \{h(j) | j \in out_i\}) & \text{caso contrário.} \end{cases}$$

Para se conseguir chegar ao terceiro resultado é provado, por indução, que um nó i, com $h(i) < \infty$, realiza redução. Além disso, é demonstrado que um nó i, com $h(i) = \infty$ nunca será reduzido.

Desempenho

Considere um sistema com n nós e e arestas no WFG. O algoritmo apresentado nessa seção possui complexidade de mensagens igual a $4e - 2n + 2l$, onde l é o número de nós folha no WFG.

A argumentação é a seguinte: existem no máximo e FLOOD's e e ECHO's . O número de FLOOD's convertidos para SHORT's é no máximo $e - n + l - af$, onde af é o número de arestas para nós folha. Isso acontece porque $(n - l)$ FLOOD's não podem ser convertidos para SHORT's, já que cada nó interno usa um FLOOD para expandir o WFG e porque af FLOOD's não podem ser convertidos para SHORT's, já que um nó folha sempre envia de volta um ECHO quando chega um FLOOD. Além disso, o número de ECHO's convertidos para SHORT's é no máximo $e - n + l$, pelo fato de um nó não folha no WFG receber um ECHO por cada aresta que representa pendência, um dos ECHO's reduzirá o nó e não será convertido para SHORT.

Dessa forma, o limite superior na complexidade de mensagens do algoritmo é de $4e - 2n + 2l - af < 4e$ mensagens. Isso pode ser aproximado para $4e - 2n + 2l < 4e$ mensagens.

5.3.3 Algoritmo de Brzezinski *et al.* 95

Este trabalho supõe o seguinte modelo do sistema. Um conjunto de nós comunicam-se através de troca de mensagens e estão conectados por canais de comunicação assíncronos, confiáveis, não FIFO e com *buffer* infinito. Os nós estão livres de falhas e não existe um relógio global acessível de maneira instantânea por todos os nós.

Um processo pode estar em dois estados: ativo ou passivo. Somente processos ativos podem enviar e retirar mensagens do *buffer* (mensagens consumidas). Um processo ativo pode se tornar passivo ao pedir por algumas mensagens (recursos) e ao recebê-los, torna-se ativo novamente.

Para cada processo passivo está associado um conjunto de processos pelos quais ele está esperando receber mensagens, o **conjunto dependente** (DS)[3]. Um processo pas-

[3]Do inglês *dependent set*

sivo somente é ativado se a condição de ativação definida sobre o conjunto dependente for satisfeita. A formulação dessas condições depende dos modelos de requisição vistos anteriormente na Seção 4.1.3.

Algoritmo para detecção do maior conjunto em *deadlock*

Nesta seção será apresentado um algoritmo para detecção do maior conjunto em *deadlock*.

Para cada processo da aplicação está associado um processo especial, chamado **controlador**. Esse processo tem duas funções: observar o comportamento do processo associado e cooperar com os outros controladores para a detecção de *deadlock*.

São utilizadas as seguintes variáveis de estado:

- $estado_i$: associada a um processo da aplicação P_i e lida pelo controlador C_i, pode assumir os valores *passivo* e *ativo* ($estado_i = passivo$ é abreviado como $passivo_i$).
- cp_i (continuamente passivo): esta variável é usada por C_i para observar se P_i foi ativado em algum momento.

Para se detectar um *deadlock*, um controlador qualquer C_α inicia a detecção no instante t_i enviando uma ficha (*token*) para o sucessor em um anel virtual que compreende todos os controladores. A ficha carrega o conjunto de todos os processos em potencial *deadlock* (PD). Inicialmente esse conjunto é formado por todos os processos, ou seja, $PD = P$.

Quando um controlador C_i recebe a ficha, ele retira P_i de PD somente se ele verifica que "P_i é potencialmente ativo" (condição a ser detalhada em breve na descrição do teste básico).

Ao receber a ficha, se C_α encontra o valor de PD sem modificações desde a última visita ou se $PD = \emptyset$, ele conclui que a detecção terminou e o resultado está em PD. Se não for esse o caso, a detecção ainda não terminou e a ficha é enviada para mais um percurso no anel.

O teste básico do algoritmo consiste na regra que o controlador C_i executa para determinar se deve retirar P_i de PD na ficha. Geralmente, o controlador faz isso quando, de acordo com seu conhecimento atual, P_i certamente não estava em *deadlock* no instante t_i, isto é, em qualquer instante $t \geq t_i$, o processo estava ativo ou poderia ser ativado por processos conhecidos como não estando em *deadlock*.

Para P_i determinar se está potencialmente ativo ele usa a variável cp_i. Quando a ficha visita P_i pela primeira vez, cp_i é tornado verdadeiro (falso) se P_i estiver passivo (ativo) naquele momento. C_i determina que P_i está potencialmente ativo se cp_i está falso depois da última visita da ficha.

Outra maneira de determinar a potencial ativação é o predicado $fulfilled(CHEGOU_i \cup (P \setminus PD))$. Se ele for verdadeiro durante a visita da ficha no processo P_i, então esse

processo é potencialmente ativo. Isto acontece porque o conjunto $CHEGOU_i \bigcup (P \setminus PD)$ representa o conjunto de processos que enviaram ou poderiam ter enviado mensagens para P_i.

Para manter a consistência da detecção, os canais devem ser monitorizados para que mensagens em trânsito não tornem o predicado $fulfilled_i$ verdadeiro.

Para resolver esse problema, o algoritmo utiliza o mecanismo de confirmação (*acknowledgment*), onde cada controlador deve confirmar cada mensagem tão logo chegue e deve manter um contador ($notack_i$) que controla o número de mensagens enviadas por P_i e ainda não confirmadas.

Se P_i está potencialmente em *deadlock* quando C_i recebe a ficha (isto é, $P_i \in PD \bigwedge cp_i$), então o controlador mantém a ficha até que novas mensagens tornem o predicado $fulfilled_i$ verdadeiro, ou cp_i torne-se falso ou $notack_i$ seja reduzida a zero. As duas primeiras condições implicam que P_i é potencialmente ativo e fazem com que C_i remova P_i de PD. A terceira condição assegura que as mensagens chegaram aos destinos e estão prontas para serem consumidas.

A ficha carrega uma variável lógica, fv[1], que indica se é a primeira passagem da ficha para uma determinada detecção. Isso é utilizado para inicializar a variável cp_i indicando que P_i tem estado continuamente passivo desde da última visita da ficha e para garantir que os canais de saída de processos passivos estão vazios antes da próxima passagem da ficha.

As ações dos controladores estão formalizadas nas sentenças (S1)-(S4). O controlador C_a que inicia a detecção, além de executar esses passos, também deve observar as sentenças (S6) e (S7).

Exemplo

Na Figura 5.18 tem-se o início da execução do algoritmo que detecta o maior conjunto em *deadlock*. Trata-se da primeira passagem da ficha pelo sistema. A variável fv é verdadeira e o conjunto PD é inicializado com a identificação de todos os processos do sistema.

A ficha é repassada por todo o anel, algumas vezes ela é transmitida porque a variável $notack_i$ de cada processo é igual a 0 (caso de P_3, P_4 e P_5) ou porque simplesmente o processo se encontra ativo (situação de P_2).

Quando a ficha passa pela segunda vez pelo processo iniciador (figura 5.19), ele percebe a alteração do conjunto PD e recomeça outra passagem da ficha, que agora carrega a variável fv igual a falso.

Ao receber essa ficha pela terceira vez o iniciador declara há *deadlock* e que PD guarda a identificação de todos os processos que se encontram nesse estado.

[1]Do inglês *first visit*.

```
(S1) when P_i sent (msg) to P_j
        do notack_i := notack_i + 1;
        end do;

(S2) when received (msg) from P_j
        do send (ack) para C_j
        end do;

(S3) when C_i received (ack) from C_j
        do notack_i := notack_i - 1;
        end do;

(S4) when estado_i changes toativo
        do cp_i := false;
        end do;

(S5) when C_i received token (PD,fv)
        do if (P_i ∈ PD) then
                if (fv) then cp_i := (estado_i = passivo);
                end if;
                keep token until ((notcp_i) or (fulfilled_i(CHEGOU_i ⋃ (P \ PD)))or
(notack_i = 0));
                if ((not cp_i) or (fulfilled_i (CHEGOU_i ⋃(P \ PD))))
                        then PD:= PD\{P_i};
                end if;
                cp_i := (estado_i = passivo);
            end if;
            send token(PD,fv) to sucessor;
        end do;

(S6) when C_a inicia detecção
        do
                PD := P;
                fv := true;
                previous := |PD|;
                send token(PD,fv) to sucessor;
        end do;

(S7)when C_a received token(PD,fv)
        do
                if (P_a ∈ PD)
                        then faça as ações correspondentes a (S5);
                end if;
                if (fv or (previous ≠ |PD|) and (PD ≠ 0))
                        then fv := false;
                             previous := PD;
                             send token(PD,fv) to sucessor;
                        else detecção terminada;
                              resultado registrado em PD;
                end if;
        end do;
```

Algoritmo 5.11: Descrição do Algoritmo de Brzezinski et al. 95

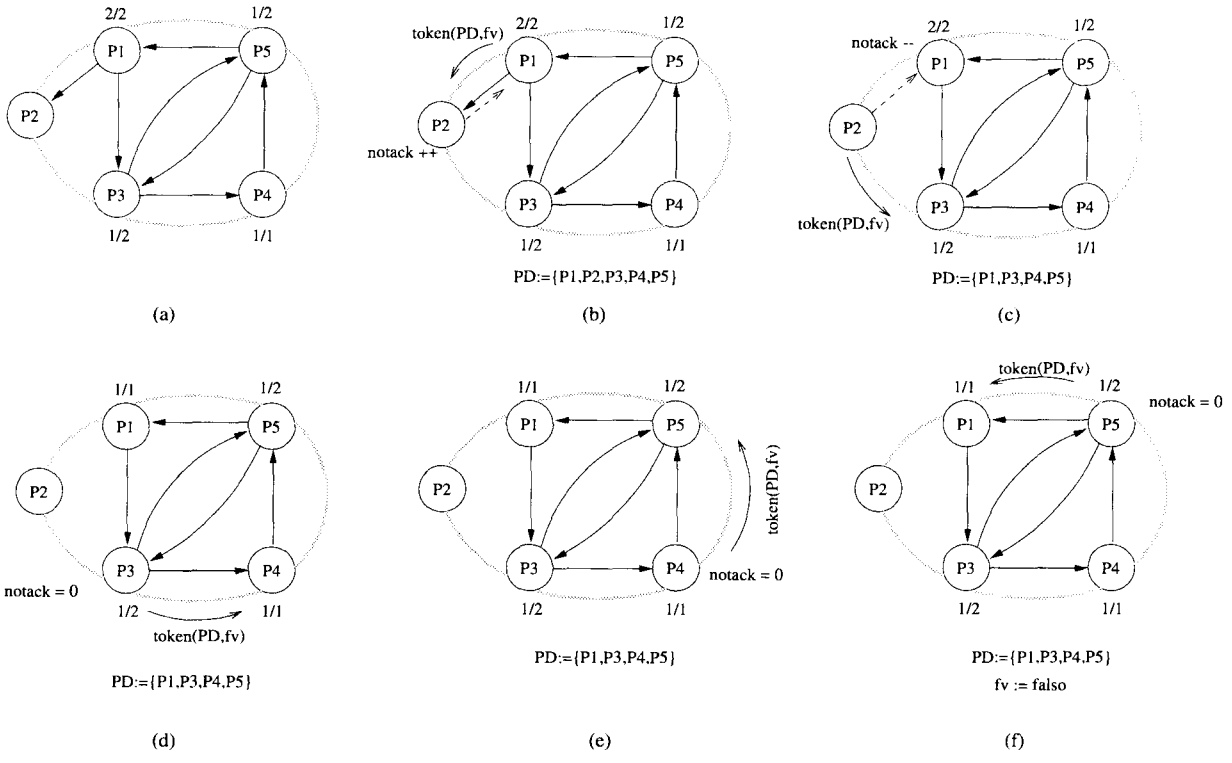


Figura 5.18: Primeira passagem do token pelo sistema

Correção do Algoritmo

Para mostrar a correção, é suficiente que se prove que as seguintes propriedades são satisfeitas pelo algoritmo: segurança e progresso.

Progresso Toda execução do algoritmo termina em um tempo finito.

Consistência Seja t_i e t_t os instantes de tempo do início e do término, respectivamente.

1. Se o algoritmo termina com $PD = \emptyset$, então para qualquer conjunto B, $B \subseteq P$, o predicado *deadlock*(B) estava falso no momento t_i.
2. Se o algoritmo termina com $PD \neq \emptyset$, então o predicado *deadlock*(PD) estava verdadeiro no instante t_t e o predicado *maxdead*(PD) estava verdadeiro no tempo t_i.

A propriedade de progresso é provada com o fato que a ficha se não for repassada imediatamente para o sucessor na estrutura, em algum tempo finito isso deve acontecer. Quando a ficha passa por C_i, ou ela é transmitida para o sucessor ou é mantida até que uma das três condições seja válida: $\neg cp_i \vee fullfiled_i(CHEGOU_i \cup (P \backslash PD)) \vee notack_i = 0$

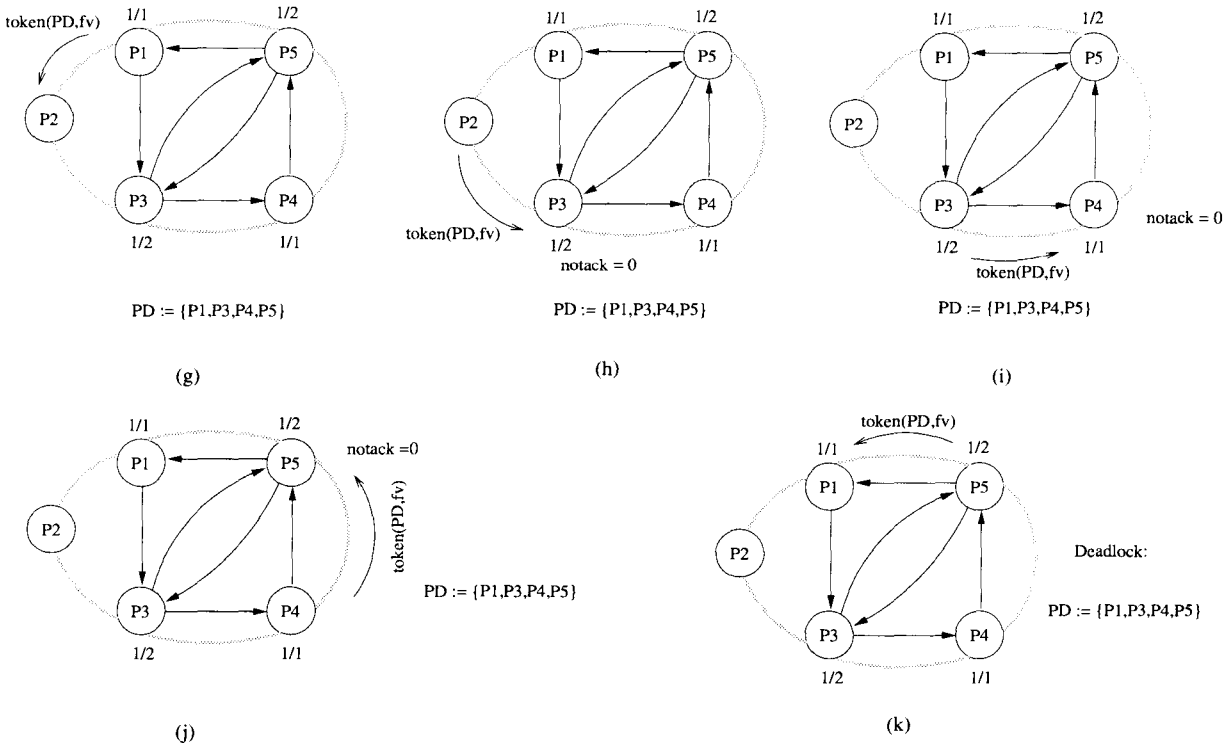


Figura 5.19: Segunda passagem do token pelo sistema

Como um processo passivo não pode enviar mensagens e todas as mensagens enviadas serão reconhecidas em um tempo finito, então $(cp_i \wedge notack_i \neq 0)$ não pode ser válido indefinidamente. Consequentemente, qualquer C_i repassará a ficha e uma volta completa é realizada em tempo finito.

Além disso, também existe o fato que o número de *rounds* é finito. Isso se deve ao fato do tamanho do conjunto PD contido na ficha decrescer a cada passagem da ficha.

A propriedade de consistência é garantida também e encontra-se demonstrada no trabalho. A idéia da prova baseia-se no conteúdo do conjunto PD. Para maiores detalhes, favor consultar [BHRS95, BHRS96].

Se o algoritmo termina com $PD = \emptyset$, não existia situação de *deadlock* no início da detecção. Isso pode ser mostrado por contradição que nenhum processo pertencente a $B \subseteq P$ pode ser removido do conjunto PD que está na ficha. Por essa razão, quando o algoritmo termina com $PD \neq \emptyset$, tem-se que: $B \neq \emptyset \wedge B \subseteq PD$.

Para mostrar que, quando o algoritmo termina com $PD \neq \emptyset$, o predicado $deadlock(B)$ era verdadeiro desde o início da detecção. A prova é feita por construção em [BHRS95, BHRS96]. Se ocorre um *deadlock* no tempo σ, todos os processos $P_i \in PD$ estão passivos e todos os canais de saída estão vazios. Como $deadlock(PD)$ é verdadeiro no tempo $\sigma \leq t_l$, então pelo fato do *deadlock* ser uma propriedade estável, o predicado também será

verdadeiro em t_f.

Múltiplas Detecções

O algoritmo trabalha com uma única ficha que percorre o sistema. Essa ficha pode estar *ocupada* se ela está sendo utilizada em uma detecção, ou pode estar *livre*, quando o processo está ativo e não há necessidade de se iniciar uma detecção de *deadlock*. Nesse caso, ela percorre o anel, indo de controlador em controlador até que encontre algum processo associado que esteja passivo.

Existe uma outra abordagem para tratar do início da detecção: o controlador enviar uma ficha tão logo o processo da aplicação torne-se passivo. Por consequência, dever-se-á estender o algoritmo para lidar, de uma forma consistente, com a múltiplas detecções que estarão em trânsito representadas pelas várias fichas.

A distinção das fichas será feita por um *timestamp* único (j, s), onde j expressa a identificação do iniciador e s é o número de sequência das iniciações da ficha.

Um controlador C_i mantém as variáveis $latest_i(j)$ que registra o maior número de sequência s de ficha iniciada por C_j, e $cp_i(j)$, que denota que P_i tem estado passiva desde a última visita da ficha iniciada por C_j. A variável $cp_i(j)$ é atualizada somente pela ficha com o $timestamp(j, s)$ tal que $s \geq latest_i(j)$.

Execução Paralela

Nesse esquema, assume-se que o conjunto de P processos é completamente conectado de maneira lógica. Assim, é formada uma estrutura virtual em forma de estrela com o iniciador C_α no centro.

O iniciador difunde uma mensagem *query* (formada pelo conjunto PD e flag fv, como no algoritmo básico) para todos os controladores. Em seguida, cada C_i segue as instruções do algoritmo básico e envia o conjunto $PD = PD_i$ atualizado para C_α. Esse controlador, a partir das respostas, constrói um conjunto de dependentes atualizado globalmente $PD := PD_1 \bigcap PD_2 \bigcap \ldots PD_n$.

Roteamento da Ficha

O número de transmissões da ficha pode ser reduzido significativamente ao direcionar a ficha somente para os controladores associados aos processos no conjunto PD ou para o controlador iniciador C_α.

É introduzida a noção de $next(PD)$ que substitui *sucessor*, que é formado pelo anel de controladores C_i tal que $P_i \in PD \bigcup P_\alpha$.

Detecção de um conjunto em *deadlock*

Para se detectar somente um conjunto de processos em *deadlock* (não necessariamente o maior), basta reduzir o tamanho inicial de PD, levando em conta que a ativação de qualquer processo depende diretamente somente dos processos que pertencem ao conjunto dependente.

Além de PD, é utilizado o conjunto V de processos que já foram visitados nesse *round*. O controlador C_α faz $PD := DS_\alpha \setminus \{P_\alpha\}$ e $V := \emptyset$. Temos a seguir as modificações, uma para a implementação baseada em ficha, outra para a paralela:

1. Implementação baseada em ficha: a ficha carrega os conjuntos PD e V. O controlador C_α envia a ficha para $next(PD)$. Cada C_i inicializa sua variável cp_i. Se P_i está ativo, C_i atribui $PD := PD \setminus \{P_i\}, V := V \cup \{P_i\}$. Caso contrário, C_i faz $PD := PD \cup (DS_i \setminus V)$ e $V := V \cup \{P_i\}$. Depois C_i envia a ficha para para $next(PD \setminus V)$. A fase de inicialização é terminada quando o iniciador recebe de volta a ficha.

2. Implementação paralela: C_α envia *queries* para os controladores de todos os processos que compõem DS_α. Em respostas às *queries*, cada C_i inicializa a sua cp_i e devolve o DS_i. C_α coleta as respostas e inclui P_i no conjunto V assim que a resposta de P_i é recebida. Se P_i está ativo ($DS_i = \emptyset$), o iniciador faz $PD := PD \setminus \{P_i\}$. Caso contrário, $PD := PD \cap DS_i$. Se $PD \not\subseteq V$, outra iteração é realizada com os processos $PD \setminus V$, senão a fase de inicialização está terminada.

Depois que essa fase termina, a detecção é continuada com $fv = falso$ e PD tendo o valor obtido na fase de inicialização. As ações posteriores são tomadas de acordo com o algoritmo básico.

Desempenho

Duas medidas são apresentadas: a complexidade de mensagens e a complexidade de tempo.

Primeiro considera-se o algoritmo baseado em ficha. Durante o primeiro *round* modificado, são visitados todos os n processos dos quais P_α depende direta ou indiretamente. Isso resulta em um total de n transmissões.

No pior caso, tem-se que pelo menos um processo é removido de PD. Como a ficha é enviada somente para os processos que compreendem PD, então o número máximo de mensagens necessárias à detecção é $n + (n-1) + \ldots + 2 = (n+2)(n-1)/2$. Dessa forma a complexidade de mensagens é $O(n^2)$. Como o algoritmo é de natureza sequencial, a complexidade de tempo é $O(n^2)$ também.

Na implementação paralela, a fase de inicialização transmite $2(n-1)$ mensagens e gasta o mesmo em unidades de tempo. Durante a fase de inicialização e nas ondas posteriores, pelo menos um processo é retirado de PD. Como as *queries* são enviadas somente para processos em PD, o número máximo de mensagens é $2(n-1)+2(n-2)+\ldots+2\times 2 = (n+1)(n-2)$ e a complexidade de mensagens é $O(n^2)$.

No caso do algoritmo de detecção do maior conjunto em *deadlock*, uma análise similar pode ser feita, exceto que a redução do tamanho inicial de PD não pode ser usada e, consequentemente, o valor de n tem que ser trocado por N, o número total de processos.

Recuperação de Falhas

Assume-se que, quando um nó falha, os outros são notificados em um tempo finito e o sistema subjacente reconfigura os restantes em uma nova estrutura de comunicação.

C_i, ao receber a notificação que o nó f falhou, diminui a variável $notack_i$ pelo número de mensagens que P_i enviou para P_f mas ainda não foram confirmadas. A partir desse momento, C_i descarta qualquer mensagem que chega de C_f. Temos que considerar as seguintes situações em relação à ficha:

A ficha não é perdida Nesse caso, a ficha não estava em trânsito para f e nem estava nesse nó. Não há problema, pois a ficha visitará somente os processos no conjunto dependente, exceto os que falharam. Dessa forma, a detecção pode chegar a $PD = \emptyset$.

Ficha perdida Ou seja, uma das condições descritas para o item anterior é válida. Essa questão é mais intrincada e para ela existem soluções (por exemplo, [LL77]). Ao ser notificado da perda da ficha, C_a aborta a detecção e lança uma nova que se comportará como no caso descrito pelo item anterior.

5.4 Comentários

Este capítulo forneceu uma visão em largura das abordagens para a detecção de *deadlock*, procurando, além de mostrar os mecanismos mais utilizados, apresentar uma classificação dos algoritmos segundo o modelo de requisição adotado. É interessante notar que essa classificação por modelo reflete cronologicamente o desenvolvimento da área.

Para a redação deste capítulo e do anterior, foi feito um grande levantamento bibliográfico na área de *deadlock*, mas foi possível somente incluir os trabalhos mais relevantes de cada abordagem. Aos leitores interessados em referências mais completas, sugere-se a bibliografia anotada [Zob83] sobre trabalhos até a meados de 80. Além dessa bibliografia anotada, sugerem-se os *surveys* [Kna88, Elm86, Sin89]. Alguns livros [Bar96, Ray88] também trazem a descrição de algoritmos, incluindo alguns de publicação mais recente.

Até o final da década de 80 foram criados algoritmos elegantes, mas que apresentavam grande número de erros. Isso mostra que provas rigorosas são imprescindíveis para mostrar a correção dos algoritmos. Na opinião de Knapp[Kna88], é necessária a utilização de técnicas que utilizem o mínimo de argumentos operacionais. Singhal [Sin89] reforça essa idéia criticando as provas informais e intuitivas.

A questão da prova de correção dos algoritmos de detecção de *deadlock* distribuído têm recebido mais atenção da comunidade científica. Atualmente observa-se que as provas de correção têm apresentado um formalismo mais adequado ao problema.

O progresso obtido na questão anterior infelizmente não foi alcançado em outras questões pertinentes à detecção de *deadlock*. A análise de desempenho é uma delas. Não basta obter uma aproximação do número de mensagens necessárias para se detectar um ciclo, deve-se ter uma idéia de como se comporta o número de mensagens geradas pelo algoritmo sob condições normais de execução do sistema (quando não há *deadlock*). Além de estudar a complexidade de mensagens, essas análises precisariam se debruçar sobre a sobrecarga que a detecção traz ao sistema. A complexidade de *bits*, o tempo médio da persistência de um *deadlock* pedem por mais dedicação ao estudo e desenvolvimento de modelos de desempenho e técnicas de simulação.

Além da correção e dos resultados de desempenho de um algoritmo de detecção de *deadlock*, outro parâmetro é muito importante na escolha de uma técnica de detecção. Esse parâmetro é a probabilidade do *deadlock*. Além de auxiliar na escolha, esse estudo poderia originar uma nova classificação baseada na adequação de um algoritmo para um determinado comportamento do sistema. [GHKO81] mostra, em um dos primeiros estudos das ocorrências de *deadlock* que, 90% deles são de tamanho dois e que os *deadlocks* são raros e crescem linearmente com o grau de multiprogramação. Trabalhos nessa área dariam uma contribuição muito valiosa aos projetistas de sistemas distribuídos [AM87, Tri97].

É interessante notar que, apesar de haver uma grande quantidade de algoritmos propostos para a área de *deadlock*, mais especificamente para a detecção, existe uma boa quantidade de trabalho a ser feito.

Capítulo 6

Derivações

Elaborar uma solução para um problema distribuído que atenda aos critérios de segurança e progresso está muito longe de ser uma tarefa trivial. Como visto no Capítulo 4, nem mesmo algoritmos de detecção de *deadlock* publicados na literatura escaparam da contestação posterior, mostrando-se incorretos.

Com o objetivo de encontrar uma modelagem mais adequada, alguns pesquisadores tomam a direção das derivações. Verificando comportamentos similares e características comuns, pretende-se desenvolver um método que facilite o projeto de algoritmos distribuídos para resolver problemas que apresentem algumas similaridades.

Uma característica muito importante presente em certos problemas é a propriedade da estabilidade. Essa propriedade foi explorada em alguns estudos que propõem derivações entre algoritmos para esses tipos de problemas. Um método pode ser encontrado em [TM93]. Nesse trabalho, os autores descrevem como obter algoritmos de detecção da terminação distribuída a partir da transformação de algoritmos para a coleta de lixo. Além desses resultados, algumas considerações são feitas em relação ao problema da aproximação do ínfimo distribuído em [Tel91].

A derivação de algoritmos de detecção de lixo a partir de esquemas de detecção de terminação também foi estudada. Em [TM93] e em [TTvL87] é observado que a detecção de terminação é uma subfase de coletores de lixo *mark-sweep*. No entanto, os autores reconhecem que nem todos os algoritmos de detecção de terminação são adequados à construção de algoritmos de detecção de lixo distribuído.

Com o intuito de fazer um estudo da relação entre a área de detecção de *deadlock* distribuído e a área da detecção da terminação distribuída para resultar em um método de derivação entre os algoritmos, serão apresentados nas próximas seções os esqueletos básicos de cada área, a relação entre eles, o método de derivação obtido da modelagem dos problemas e o exemplo de uma derivação de um algoritmo de detecção da terminação a partir de um esquema de detecção de *deadlock* já conhecido, proposto por Chandy, Misra

e Haas (Seção 5.2.1).

6.1 A Terminação Distribuída

A semântica do problema da terminação distribuída foi descrita em inúmeros artigos sobre o tema e também no Capítulo 2. Essa seção apresenta um esqueleto genérico de uma computação básica.

Recordando a descrição do problema da terminação distribuída, pode-se dizer em poucas palavras o seguinte: considere um conjunto P de processos que se comunicam via troca de mensagens. Um processo pode estar **passivo** ou **ativo**. Somente processos ativos podem enviar mensagens. Um processo ativo pode tornar-se espontaneamente passivo, mas a mudança do estado passivo para o ativo só acontece quando um processo passivo recebe uma mensagem. O esqueleto básico que descreve as possíveis ações da computação básica (envio e recebimento de mensagens) está no algoritmo 6.1.

```
(S) possible only when estado = ativo
        send (msg);

(R) when received (msg) do
        receive (msg);
        estado := ativo;

(T) possible only when estado = ativo
        estado := passivo;
```

Algoritmo 6.1: Ações da computação básica

Qualquer algoritmo de detecção da terminação tem como base o esqueleto da computação básica apresentado e sobre ele são feitas as operações para o controle da detecção. Esse controle, por sua vez, deve obedecer a certos critérios que definem sua correção para que a estatégia proposta seja considerada uma solução ao problema. Relembrando do Capítulo 2, tem-se:

C1T.Segurança Se a terminação é detectada, então ela existe.

C2T.Progresso Se existe terminação, então, em algum tempo finito, ela será detectada.

6.2 O *Deadlock* Distribuído

Como visto no Capítulo 4, um processo pode utilizar um recurso somente na seguinte sequência de operações:requisição, utilização e liberação.

Essa seqüência encontra-se descrita no Algoritmo 6.2. Neste algoritmo foram incluídos os conjuntos IN (arestas de entrada) e OUT (arestas de saída), que mantém a informação sobre quais processos estão requisitando recursos junto a P_i e de quais processos P_i está dependendo, respectivamente.

Com algumas modificações, pode-se observar uma grande semelhança entre o algoritmo 6.1 e este esquema de requisições e atendimentos. Para isso, basta observar que foram acrescentadas ao esqueleto da computação básica as manipulações de IN e OUT, além de considerar *bloqueado* semelhante a *passivo* e *desbloqueado* semelhante a *ativo*.

```
(S)when P received (req r) from Q do
        IN := IN + {Q};
        possible only (estado = desbloqueado and r disponível) do
                send (reply r) to Q;
                IN := IN - {Q};
        end do
end do

(R) when received (reply r) from Q do
        OUT := OUT - {Q};
        estado := desbloqueado;
        utiliza recurso;
        libera recurso;

end do

(T) possible only when estado = desbloqueado do
        OUT := OUT + {Q};
        send (req r) to Q;
        estado := bloqueado;
end do
```

Algoritmo 6.2: Requisições de objetos

Para simplificar o entendimento da construção do esqueleto genérico das ações de requisição de um objeto, primeiramente foi apresentado o esqueleto contido no Algoritmo 6.2 que adequa-se ao modelo de requisição de recurso único, o modelo mais restrito.

O esqueleto que atende ao modelo mais generalizado encontra-se descrito no Algoritmo 6.3. Esse algoritmo utiliza as funções *completa_atendimento*(*recurso*), que verifica se o processo sai do estado bloqueado ao receber um atendimento que espera; e $forma_requisicao(P, R, n)$ que monta uma requisição de acordo com a necessidade do processo e com o modelo de requisição subjacente.

É importante relembrar que os algoritmos que realizam detecção de *deadlock* também têm que obedecer a certos critérios que definem sua correção.

C1D.Segurança Se um *deadlock* é detectado, então ele existe.

```
(S)when received (req r) from Q do
        IN := IN + {Q};
        possible only when (r disponível and estado = desbloqueado) do
                send (reply r) to Q;
                IN := IN - {Q};
        end do;

end do;

(R)whenreceived (reply r) from Q do
        OUT := OUT - {Q};
        when completa_atendimento((reply r)) do
                estado := desbloqueado;
                utiliza recursos envolvidos na requisição;
                libera recursos;
        end do;
end do;

(T)possible only when estado = desbloqueado
        forma_requisição(PROCS,RECS,n);
        OUT := OUT + PROCS;
        for i:= 1 to n do
                send (req REC[i]) to PROCS[i];
        end for;
        estado := bloqueado;
end;
```

Algoritmo 6.3: Requisições de objetos

C2D.Progresso Se um *deadlock* existe, então, em algum tempo futuro, ele será detectado.

6.3 Detecção da Terminação utilizando Detecção de *Deadlock*

Esta seção descreve a modelagem do problema da terminação partindo do problema do *deadlock*. Para tanto, é necessária a definição de um tipo de *deadlock*, denominado neste trabalho de *deadlock D*.

Considere um conjunto P de processos. Deseja-se verificar a existência de um *deadlock* D, isto é, D é um *deadlock* no modelo de comunicação e envolve **todos** os processos do conjunto P. Observe que nem todo *deadlock* obedece a essa especificação, já que poderia ser formado por um subconjunto não próprio de P. Esse tipo de *deadlock* não interessa por não ser capaz de modelar a condição de terminação, que deve ser alcançada por todos os processos do conjunto.

O propósito é detectar a existência ou não de D tal que a seguinte equivalência seja válida:

Teorema 18 $\exists$ *deadlock D* $\Leftrightarrow$ *a condição de terminação é válida.*

($\Rightarrow$)

Hipótese: $\exists$ *deadlock D*.

Tese: a condição de terminação é válida.

Em [CMH83] é utilizada a definição de *deadlock* no modelo de comunicação, chamada neste trabalho de *deadlock D*. Relembrando, um conjunto P de processos está em *deadlock D* se:

1. todos os processos em P estão bloqueados;
2. o conjunto de dependentes de todo o processo em P é um subconjunto de P; e
3. não existem mensagens de atendimento em trânsito entre os processos em P.

Pode-se fazer uma analogia com o problema da terminação da seguinte forma. O que é entendido como *bloqueado* em *deadlock* é equivalente a *passivo* na terminação, pois nessas duas situações os processos não podem enviar mensagens (ou da aplicação ou de atendimento de recurso) e permanecerão nesse estado até o recebimento de alguma mensagem.

Em relação ao item 2 da definição, supõe-se que existe um *deadlock D* e que ele envolve todos os processos do conjunto sobre o qual pretende-se detectar a terminação. Logo, o conjunto de dependentes de todo o processo em P é o próprio P.

Reescrevendo os dois últimos parágrafos, obtém-se:

1. todos os processos em P estão passivos;
2. o conjunto de dependentes de todo processo em P é o próprio P; e
3. não existem mensagens de atendimento em trânsito entre os processos em P.

Pode-se observar que essa é a própria definição da condição de terminação. Portanto, o problema da terminação distribuída está contido no problema do *deadlock*. A terminação pode ser modelada como um *deadlock* no modelo de comunicação.

($\Leftarrow$)

Hipótese: a condição de terminação vale.

Tese: $\exists$ *deadlock D*.

Vamos aplicar algumas regras sobre o conjunto P:

Regra 1: ao se tornar passivo, p_i envia uma requisição do tipo OR para todos os processos p_j, onde $p_j \in P$ e $j \neq i$.

Regra 2: ao receber **pelo menos um** atendimento, o processo é desbloqueado, em outras palavras, torna-se ativo novamente.

Regra 3: uma mensagem básica é uma mensagem de atendimento.

É importante observar que essas regras não alteram em momento algum a hipótese.

Em algum momento, todo processo entra no estado terminado. Todo processo, antes de entrar nesse estado, envia requisições de comunicação para os demais processos, as quais poderão ser atendidas somente por processos ativos.

Observe que uma requisição OR *(a or b or c)* é atendida quando há pelo menos uma resposta. Isso equivale a um processo passivo tornar-se ativo quando recebe uma mensagem.

Portanto, quando a condição de terminação vale, tem-se que:

1. todos os processos em P estão passivos e necessariamente bloqueados (todos fizeram requisições que não foram atendidas);

2. o conjunto de dependentes de todo o processo em P é um subconjunto de P (formado pelo conjunto P exceto o próprio processo) e

3. não existem mensagens (atendimentos) em trânsito entre os processos em P.

O que equivale à definição de um *deadlock* D.□

Para derivar um algoritmo de detecção da terminação deve-se sobrepor à requisição de objetos (algoritmo 6.2) as ações da computação básica ilustradas no algoritmo 6.1. Sobre esse esquema combinado (algoritmo 6.4), está o algoritmo de detecção de *deadlock*. Trocando a detecção de *deadlock* pela notificação de terminação obtém-se o algoritmo que detecta a terminação distribuída.

Parando nesse ponto já é obtido o algoritmo derivado, mas uma modificação é feita a mais: o envio de mensagens de requisições de mensagens feitas por causa da Regra 1 pode perfeitamente ser dispensado no algoritmo obtido para detecção de terminação. Para manter a consistência com a modelagem proposta basta o conjunto OUT registrar os demais nós, sem prejuízo algum na solução final.

Teorema 19 *O algoritmo construído atende aos critérios de correção C1T e C2T.*

Prova

```
(S)possible only when estado=ativo do
        send (msg) to P_j;
end do

(R)when received (msg) do
        receive (msg);
        estado := ativo;
        OUT := ∅;
end do

(T)possible only when estado = ativo do
        for all P_j ∈ P, j ≠ i
                send (req msg) to P_j;
                OUT := OUT + {P_j};
        end for
        estado := passivo;
end do
```

Algoritmo 6.4: As ações básicas aumentadas

No critério de segurança (**C1T**), supõe-se que qualquer processo entra no estado terminado. Quando isso acontece, um *deadlock* D é detectado. Pelo critério de segurança do algoritmo de detecção de *deadlock* (condição **C1D**), se é detectado um *deadlock*, então ele existe. Logo, pelo Teorema 18 a condição de terminação vale.

Para provar o critério de progresso (**C2T**), suponha que a condição de terminação vale. Pelo Teorema 18, existe *deadlock* D, portanto, pela condição de progresso do detector de *deadlock* (critério **C2D**), D será detectado em algum tempo futuro. Logo, os processos entrarão no estado terminado em um tempo finito. □

É importante dizer que nem todos os algoritmos de detecção de *deadlock* servem para essa transformação. No caso dos algoritmos desenvolvidos para os modelos de recurso único e AND, não há garantia de que seja detectado o *deadlock* que envolve o maior número de processos. Esses algoritmos poderiam dar uma resposta positiva da detecção quando ainda não teria sido alcançada a condição de terminação. Veja a ilustração da figura 6.1.

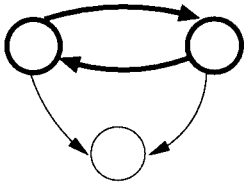


Figura 6.1: *Deadlock* detectado, mas terminação ainda não alcançada.

Os passos da transformação podem ser assim descritos:

1. Sobreponha as ações da computação básica aos atendimentos das requisições.

2. Sobreponha a esse esquema combinado o algoritmo de detecção de *deadlock*.

3. Troque a notificação de *deadlock D* pela notificação da terminação.

6.4 Exemplo de Derivação

Nesta seção será derivado um algoritmo de detecção da terminação distribuída a partir de um esquema de detecção de *deadlock* distribuído. O processo de derivação é apresentado a partir do segundo passo, pois a etapa inicial foi feita na Seção 6.3 que sobrepôs o esqueleto da computação básica ao esqueleto da requisição de objetos, originando o esquema combinado do Algoritmo 6.4.

No segundo passo, o algoritmo de Chandy, Misra e Haas para o modelo de comunicação (Seção 5.2.1) é sobreposto ao esquema combinado. Isso é feito para que as requisições sejam feitas de forma que garantam a coerência do modelo de mapeamento do problema do *deadlock* no problema da terminação distribuída. O resultado da sobreposição é mostrado no Algoritmo 6.5.

Para obter a forma final do algoritmo derivado para detecção da terminação, basta trocar a linha do anúncio do *deadlock* pela detecção do término da atividade do sistema. Além dessa pequena alteração, é proposta uma modificação no algoritmo para melhorar a sua eficiência sem que comprometa a sua correção. As ações tomadas em ($S3$ diferem nos algoritmos no seguinte ponto: para manter a consistência com o esqueleto apresentado, o algoritmo de detecção de *deadlock* sobreposto deve manter a manipulação dos conjuntos IN e OUT através do envio de mensagens de requisição. O mapeamento para o problema da terminação dispensa o envio e inclui de imediato os demais processos no conjunto OUT. O resultado final é apresentado no Algoritmo 6.6.

```
(S1)possible only when estado = ativo do
        send (msg) to P_j;
        for all i
                wait(i) := false;
        end for;
end do

(S2)when received (msg) do
        estado := ativo;
        OUT := ∅;
end do

(S3)possible only when estado = ativo do
        for all P_j ∈ P, j ≠ i
                send (req msg) to P_j;
                OUT := OUT + {P_j}
        end for;
        estado := passivo;
end do

(S4)possible only when estado = passivo do
         inic := i;
         latest(i) := latest(i) + 1;
         wait(i) := true;
         for all P_j ∈ OUT do
                send (QUERY(inic,latest(i),i,j)) to P_j;
         end for
         num(i) := |OUT|;
end do

(S5)when P_k received (QUERY(i,l,j,k)) from P_j and estado = passivo do
        if (l > latest(i)) then
                engager(i) := j;
                wait(i) := true;
                for all P_r ∈ OUT do
                        send (QUERY(i,l,k,r)) to P_r;
                end for;
                num(i) := |OUT|;
        else
                if (wait(i) and l = latest(i))
                        then send (GRANT(i,l,k,j)) to P_j;
                end if
        end if
end do

(S6)when P_k received (GRANT(i,l,r,k)) from P_r and estado = passivo do
        if (l = latest(i) and wait(i)) then
                num(i) := num(i) - 1;
                if (num(i) = 0) then
                        if (i = k) then
                                declare deadlock;
                        else
                                j := engager(i);
                                send (GRANT(inic,l,k,j)) to P_j;
                        end if
                end if
        end if
end do
```

Algoritmo 6.5: Algoritmo de *deadlock* sobreposto ao algoritmo 6.4

```
(S1)possible only when estado = ativo do
        send (msg) to P_j;
        for all i
                wait(i) := false;
        end for;
end do

(S2)when received (msg) do
        estado := ativo;
        OUT := ∅;
end do

(S3)possible only when estado = ativo do
        for all P_j ∈ P, j ≠ i
                OUT := OUT + {P_j}
        end for;
        estado := passivo;
end do

(S4)possible only when estado = passivo do
         inic := i;
         latest(i) := latest(i) + 1;
         wait(i) := true;
         for all P_j ∈ OUT do
                send (QUERY(inic,latest(i),i,j)) to P_j;
         end for
         num(i) := |OUT|;
end do

(S5)when P_k received (QUERY(i,l,j,k)) from P_j and estado = passivo do
        if l > latest(i) then
                engager(i) := j;
                wait(i) := true;
                for all P_r ∈ OUT do
                        send (QUERY(i,l,k,r)) to P_r;
                end for;
                num(i) := |OUT|;
        else
                if (wait(i) and l = latest(i))
                        then send (GRANT(i,l,k,j)) to P_j;
                end if
        end if
end do

(S6)when P_k received (GRANT(i,l,r,k)) from P_r and estado = passivo do
        if (l = latest(i) and wait(i)) then
                num(i) := num(i) - 1;
                if num(i) = 0 then
                        if i = k then
                                declare terminado;
                        else
                                j := engager(i);
                                send (GRANT(inic,l,k,j)) to P_j;
                        end if
                end if
        end if
end do
```

Algoritmo 6.6: Algoritmo de detecção de terminação

Capítulo 7

Conclusões e Trabalhos Futuros

Inicialmente o tema da dissertação concentrava-se no estudo denso dos algoritmos distribuídos de detecção de *deadlock*, mas com o desenvolver da pesquisa nessa área começou-se a dedicar um pouco mais de atenção à propriedade de estabilidade presente em outros problemas distribuídos, como por exemplo, terminação e coleta de lixo.

A partir desse ponto, começou-se a ampliar o escopo do trabalho que passou a englobar o estudo dos algoritmos distribuídos de detecção da terminação. Verificou-se tamanha afinidade entre os problemas da área de detecção de *deadlock* e de detecção da terminação que foi proposto um mapeamento entre essas duas áreas, como também um método de derivação de soluções de detecção de terminação a partir de esquemas de detecção de *deadlock*. Além de ser consistente com o mapeamento, o método preserva no algoritmo derivado as propriedades de segurança e progresso que o algoritmo-base garante.

Nas próximas seções serão descritas as contribuições deste trabalho, bem como as sugestões para trabalhos futuros.

7.1 Algoritmos de Detecção de Terminação Distribuída

Apesar do estudo da área de *deadlock* ter precedido a pesquisa sobre a detecção da terminação, a apresentação é feita em ordem inversa, pois muitas abordagens propostas na área de detecção da terminação foram posteriormente empregadas em algoritmos distribuídos para detecção de *deadlock*.

Os resultados da pesquisa na área de detecção da terminação foram apresentados em dois capítulos: no Capítulo 2 estão a descrição do problema e a discussão de algumas estratégias para resolver o problema. No Capítulo 3 as definições e abordagens são ilustradas na descrição de vários algoritmos distribuídos de detecção do término da com-

putação, apresentando o seu princípio básico, um pseudo-código em um formato padrão, um cenário de execução, a análise de desempenho e a correção do esquema.

Procuramos cobrir as abordagens propostas na literatura sobre o problema, mas restringimos a apresentação àquelas que apresentam princípios utilizados em alguns algoritmos de detecção de *deadlock*. A Tabela 7.1 sumariza os principais aspectos de cada solução.

7.2 Algoritmos de Detecção de Deadlock Distribuído

O estudo da área de *deadlock* apresentou desde a teoria dos grafos subjacente até a classificação dos algoritmos por modelo de requisição e por abordagem. Procurou-se fazer um estudo em largura dessa área e acredita-se que uma visão concisa encontra-se apresentada nos Capítulos 4 e 5.

Além do estudo teórico foi implementado o Algoritmo de Kshemkalyani e Singhal [KS94] para verificar o seu comportamento diante da presença ou não de um *deadlock*. Algumas questões de implementação levaram a sugestão de tratamento de mensagens de algoritmos que utilizam o princípio da recuperação de crédito (Seção 2.3.5).

Por todo texto da dissertação procurou-se apresentar os resultados em uma forma padronizada e o Capítulo 5 não foi exceção. Os algoritmos discutidos seguem a mesma estruturação empregada no Capítulo 3.

A Tabela 7.2 relaciona os principais aspectos dos algoritmos estudados. As colunas *Modelo de Comunicação* e *Canal* foram suprimidas, pois todos os algoritmos trabalham no modelo de comunicação assíncrono e utilizam canais bidirecionais.

7.3 Derivação de algoritmos

As transformações mostram que as dificuldades subjacentes de problemas aparentemente distintos são, de fato, similares ou até idênticas. Dessa forma, pode-se aplicar técnicas e soluções similares.

Além de aplicar resultados já estabelecidos, a área da derivações de algoritmos, mais especificamente para os problemas que apresentam a estabilidade, pode originar novos resultados, pois a pesquisa em uma área do problema pode ter negligenciado uma solução ou que é possível de ser encontrada por uma transformação de uma solução para outro problema.

A esses pontos positivos, soma-se o fato que o método da derivação pode simplificar o projeto e a verificação dos algoritmos. Isso constitui uma ajuda valiosa, principalmente no caso distribuído, onde a elaboração de um algoritmo é uma tarefa intrincada.

Algoritmo distribuído de detecção da terminação	*Modelo de Comunicação*	*Canal*	*Abordagem*	*Complex. de Mensagens*	*Simetria*	*Topologia*	*Pág.*
Dijkstra e Scholten 80	síncrono	bidirecional	Computação difusa	M	assimétrico	Árvore geradora	20
Rana 83	síncrono	unidirecional	*Timestamping*		simétrico	Anel	31
Dijkstra, Feijen e van Gasteren 83	síncrono	unidirecional	Cético	M(N-1)+N	assimétrico	Anel	25
Shavit e Francez 86	síncrono	bidirecional	Computação difusa	$M + W$	assimétrico	Floresta	34
Mattern 89	assíncrono	bidirecional	Recuperação de crédito	N	assimétrico	Estrela	36
Mattern 90	assíncrono	bidirecional	Contagem de mensagem	$O(N^2)$	assimétrico	Árvore geradora	28

Tabela 7.1: Tabela comparativa de alguns algoritmos distribuídos de detecção da terminação

Algoritmo distribuído de detecção de deadlock	*Modelo de Requisição*	*Abordagem*	*Complex. de Mensagens*	*Fases*	*Comentários*	*Pág.*
Menasce e Muntz 79	AND	*Path-pushing*	$O(n^2)$	—	É incorreto.	55
Obermarck 82	AND	*Path-pushing*	$O(n^2)$	—	É incorreto.	60
Chandy, Misra e Haas 83	AND	Caça-aresta	$O(n^2)$	2		63
Chandy, Misra e Haas 83	OR	Computação Difusa	$O(n^2)$	2		69
Bracha e Toueg 87	$C_{n.k}$	Detecção de Estado Global/ Computação Difusa	$O(n^2)$	2, 3		76
Kshemkalyani 94	$C_{n.k}$	Detecção de Estado Global/ Computação Difusa	$O(n^2)$	1	Utiliza *piggybacking*.	86
Brzezinsky, Helary, Raynal, Singhal 95	$C_{n.k}$	Caça-aresta	$O(n^2)$	2		94

Tabela 7.2: Tabela comparativa de alguns algoritmos distribuídos de detecção de *deadlock*

7.4 Sugestões para futuros trabalhos

Seria interessante fazer um estudo que aplicasse a técnica proposta nesta dissertação em variados algoritmos de detecção de *deadlock*. A análise das soluções derivadas enriqueceriam o entendimento tanto do método, como da própria área de detecção da terminação.

O mesmo poderia ser feito tomando a direção contrária: estudar soluções de detecção do *deadlock* que foram obtidas a partir de algoritmos de detecção da terminação. O resultado proposto neste trabalho permite o duplo sentido da transformação, porém esse estudo no sentido inverso levaria a alguns resultados um tanto restritos em relação aos que foram apresentados no capítulo anterior, pois o equivalente à terminação do sistema é um *deadlock* que envolve todos os nós, o que constitui uma situação muito rara (Seção 5.4).

Continuando na questão do estudo da área da detecção do *deadlock*, acredita-se na carência de análises de desempenho do caso médio e que estejam mais comprometidas com requisitos de sistemas distribuídos, como, por exemplo, de gerenciadores de bancos de dados distribuídos. No período desse trabalho foi feito um esforço nessa direção através da implementação de um simulador para avaliar alguns algoritmos de detecção de *deadlock*. O simulador e um algoritmo (Kshemkalyani e Singhal 94) foram implementados, mas a análise dos dados obtidos e o confronto com um segundo algoritmo são trabalhos futuros.

Algumas questões ainda permanecem em relação ao trabalho de generalização. Uma das direções seria a construção de uma derivação entre os algoritmos de detecção de *deadlock* e os de detecção do lixo distribuído. Ao obter-se um resultado positivo, poderia especular-se o desenvolvimento de uma metodologia que unificasse o projeto de algoritmos distribuídos para a classe dos problemas que apresentam propriedades estáveis.

Bibliografia

[AM87] M. J. Agrawal, R.; Carey and L. W. McVoy. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering*, SE-13(12):1348–1363, December 1987.

[Bar96] V. C. Barbosa. *Introduction to Distributed Algorithms*. MIT Press, 1996.

[Bev87] D. I. Bevan. Distributed garbage collection using reference counting. In *Proc.PARLE- Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[BHRS95] J. Brzezinski, J. Helary, M. Raynal, and M. Singhal. Deadlock models and a general algorithm for distributed deadlock detection. *Journal of Parallel and Distributed Computing*, (31):112–125, 1995.

[BHRS96] J. Brzezinski, J. Helary, M. Raynal, and M. Singhal. Erratum: Deadlock models and a general algorithm for distributed deadlock detection. *Journal of Parallel and Distributed Computing*, (32):232, 1996.

[BM93] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report TR UBLCS 93-1, University of Bologna, Laboratory for Computer Science, January 1993.

[BR89] O. Babaoglu and M. Raynal. Sequence-based global predicates for distributed computations: Definitions and detection algorithms. Technical Report 1961, INRIA, July 1989.

[BT83] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. Technical Report 83-558, Cornell University, June 1983.

[BT87] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[CEA71] E. G. Coffman, M. J. Elphick, and Shoshani A. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971.

[Cha82] E. J. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, July 1982.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM82] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 157–164, Ottawa, Canada, August 1982. ACM.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[DFG83] E. W. Dijkstra, W. H. Feijen, and A. J. M. V. Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 16(5):217–219, 1983.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[Elm86] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Record*, 15(3):37–45, September 1986.

[FL82] N. Fischer, M.; Griffeth and N. Lynch. Global states of a distributed system. *IEEE TSE*, 8(3):198–202, May 1982.

[FR82] N. Francez and M. Rodeh. Achieving distributed termination withou freezing. *IEEE Transactions on Software Engineering*, 8(3):287–292, May 1982.

[Fra80] N. Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[GHKO81] J. N. Gray, P. Homan, H. F. Korth, and R. L. Obermarck. A straw man analysis of the probability of waiting and deadlock ina database system. Technical report, IBM Research Lab, 1981.

[Gra78] J. N. Gray. Notes on data base operating systems. In R. M. Grahm R. Bayer and G. Segmuller, editors, *Operating Systems An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[GS80] V. D. Gligor and S. H. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, 6(5):435–440, September 1980.

[Hel89] J. M. Helary. Observing global states of assynchronous distributed applications. Technical Report 1042, INRIA, May 1989.

[HMR93] J.-M. Helary, A. Mostefaoui, and M. Raynal. Determiner un état global dans un systeme reparti. Technical Report 2090, INRIA, November 1993.

[Hol72] R. C. Holt. Some deadlock properties of computer systems. *Computing Surveys*, 4(3):179–196, September 1972.

[Hua89] S. Huang. Termination detection for distributed computations. *Proceedings 9th Int. Conf. Distributed Comput. Syst.*, pages 79–84, 1989.

[IM80] S. S. Isloor and T. A. Marsland. The deadlock problem: An overview. *Computer*, pages 58–70, September 1980.

[Kna88] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, December 1988.

[KS94] A. D. Kshemkalyani and M. Singhal. Efficient deadlock detection and resolution of generalized distributed deadlocks. *IEEE Transactions on Software Engineering*, 20(1):43–53, January 1994.

[KS97] A. D. Kshemkalyani and M. Singhal. Distributed detection of generalized deadlocks. In *Proceedings of the 17th Int. Conf. on Dist. Comput. Syst.*, pages 553–560, Baltimore, Maryland, USA, May 1997. IEEE Computer Society Press.

[Kum85] D. Kumar. A class of termination detection algorithms for distributed computations. In N. Maheshwari, editor, *5th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[LL77] G. Le Lann. Distributed systems:towards a formal approach. *IFIP Congress, Toronto*, pages 155–160, August 1977.

[LR86] I. Lavallee and G. Roucairol. A fully distributed (minimal) spanning tree algorithm. *Information Processing Letters*, 23:55–62, 1986.

[Mat87] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[Mat90] F. Mattern. Assynchronous distributed termination-parallel and symetric solutions with echo algorithms. *Algorithmica*, 5(3):325–340, 1990.

[MC82] J. Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 4(1):37–43, 1982.

[MM79] D. A. Menasce and R. R. Muntz. Locking and deadlock detection in distributed data bases. *IEEE Transactions on Software Engineering*, SE-5(3):195–202, May 1979.

[MS94] K. Marzullo and L. S. Sabel. Efficient detection of a class of stable properties. *Dist. Comp.*, 8:81–91, 1994.

[Obe82] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.

[Ran83] S. P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17:43–46, July 1983.

[Ray88] M. Raynal. *Distributed Algorithms and Protocols*. 1988.

[SG94] A. Silberchatz and P. B. Galvin. *Operating Systems Concepts*. Addison-Wesley Publishing, 4a. edition, 1994.

[Sin89] M. Singhal. Deadlock detection in distributed systems. *Computer*, pages 37–48, November 1989.

[SS91] C. Shih and J. A. Stankovic. Deadlock detection in distributed real-time systems and its application to ada environents. Technical report, Universidade de Massachusetts, 1991.

[SS94] S. Schiper and A. Sandoz. Strong stable properties. *Dist. Comp.*, 8:93–103, 1994.

[Tan95] A. S. Tanenbaum. *Sistemas Operacionais Modernos*. Prentice-Hall do Brasil, 1995.

[Tel91] G. Tel. *Topics in Distributed Algorithms*. Cambridge University Press, 1991.

[Tel94] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[TM93] G. Tel and F. Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[Top84] R. W. Topor. Termination detection for distributed computations. *Information Processing Letters*, 18:33–36, January 1984.

[Tri97] P. Triantafillou. An approach to deadlock detection in multidatabases. *Information Systems*, 22(1):39–55, 1997.

[TTvL87] G. Tel, R. B. Tan, and J. van Leeuwen. The derivation of on-the-fly garbage collection algorithms from distributed termination detection protocols. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Zob83] D. Zobel. The deadlock problem: A classifying bibliography. *Operating Systems Reviews*, 17(4):6–15, October 1983.