

PROJETO E VALIDAÇÃO DE PROTOCOLOS DE COMPUTADOR

Gerard J. Holzmann
Bell Laboratories
Murray Hill, Nova Jersey 07974
PRENTICE-HALL
Englewood Cliffs, New Jersey 07632

Série de software Prentice Hall
Brian W. Kernighan, Conselheiro
Copyright ♥ 1991 da Lucent Technologies, Bell Laboratories, Incorporated.
Este livro foi compilado em Times Roman pelo autor,
usando uma fotocompositora Linotronic 200P e um DEC VAX 8550
executando a 10ª edição do sistema operacional UNIX → .
DEC e VAX são marcas comerciais da Digital Equipment Corporation.
UNIX é uma marca registrada da AT&T.
Todos os direitos reservados.
Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação,
ou transmitido, em qualquer forma ou meio, eletrônico,
mecânica, fotocópia, gravação ou de outra forma,
sem a permissão prévia por escrito do editor.
Impresso nos Estados Unidos da América
10 9 8 7 6 5 4 3 2 1
Prentice-Hall International (UK) Limited, *Londres*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, SA, *México*
Prentice-Hall of India Private Limited, *Nova Delhi*
Prentice-Hall of Japan, Inc., *Tóquio*
Simon & Schuster Asia Pte. Ltd., *Singapura*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

CONTEÚDO

Prefácio	ix
Prefácio	XI
Parte I - Noções básicas	
1. Introdução	
1.1 Primeiros Começos	1
1.2 As primeiras redes	9
1.3 Protocolos como idiomas	12
1.4 Padronização de protocolo	13
1.5 Resumo	15
Exercícios	
Notas Bibliográficas	16
2. Estrutura do Protocolo	

2.1 Introdução	
19	
2.2 Os Cinco Elementos de um Protocolo	
21	
2.3 Um exemplo	
22	
2.4 Serviço e Meio Ambiente	
26	
2.5 Vocabulário e formato	
32	
2.6 Regras de Procedimento	
35	
2.7 Projeto de Protocolo Estruturado	
35	
2.8 Dez Regras de Design	
38	
2.9 Resumo	
39	
Exercícios	
39	
Notas Bibliográficas	
40	
3. Controle de erros	
3.1 Introdução	
43	
3.2 Modelo de Erro	
44	
3.3 Tipos de erros de transmissão	
46	
3.4 Redundância	
46	
3.5 Tipos de Códigos	
47	
3.6 Verificação de paridade	
48	
3.7 Correção de Erro	
48	
3.8 Um Código de Bloco Linear	
52	
3.9 Verificações de redundância cíclica	
56	

Página 4

3.10 Soma de Verificação Aritmética	
63	
3.11 Resumo	
64	
Exercícios	
64	
Notas Bibliográficas	
65	
4. Controle de fluxo	
4.1 Introdução	
66	
4.2 Protocolos de janela	
70	
4.3 Números de Sequência	
74	
4.4 Agradecimentos Negativos	

80	
4.5 Prevenção de Congestionamento	83
4.6 Resumo	86
Exercícios	87
Notas Bibliográficas	88
Parte II - Especificação e Modelagem	
5. Modelos de validação	
5.1 Introdução	90
5.2 Processos, Canais, Variáveis	91
5.3 Executabilidade das Declarações	91
5.4 Variáveis e tipos de dados	92
5.5 Tipos de Processo	93
5.6 Canais de Mensagem	96
5.7 Fluxo de Controle	100
5.8 Exemplos	102
5.9 Procedimentos de modelagem e recursão	104
5.10 Definições de tipo de mensagem	104
5.11 Timeouts de modelagem	105
5.12 Protocolo de Lynch revisitado	106
5.13 Resumo	107
Exercícios	108
Notas Bibliográficas	109
6. Requisitos de correção	
6.1 Introdução	111
6.2 Raciocínio sobre o comportamento	112
6.3 Asserções	114
6.4 Invariantes do Sistema	115
6.5 Deadlocks	117
6.6 Ciclos ruins	118
6.7 Reivindicações temporais	119
6.8 Resumo	125
Exercícios	126

Notas Bibliográficas

127

7. Projeto de protocolo

7.1 Introdução

128

Página 5

7.2 Especificação de Serviço

129

7.3 Suposições sobre o Canal

130

7.4 Vocabulário de Protocolo

131

7.5 Formato da Mensagem

133

7.6 Regras de Procedimento

140

7.7 Resumo

160

Exercícios

160

Notas Bibliográficas

161

8. Máquinas de estado finito

8.1 Introdução

162

8.2 Descrição Informal

162

8.3 Descrição Formal

169

8.4 Execução de Máquinas

170

8.5 Minimização de máquinas

171

8.6 O Problema de Teste de Conformidade

174

8.7 Combinando Máquinas

175

8.8 Máquinas de estado finito estendido

176

8.9 Generalização de Máquinas

178

8.10 Modelos Restritos

181

8.11 Resumo

184

Exercícios

185

Notas Bibliográficas

185

Parte III - Teste de Conformidade, Síntese e Validação

9. Teste de conformidade

9.1 Introdução

187

9.2 Teste Funcional

188

9.3 Teste Estrutural

189

9.4 Derivando sequências UIO

195	
9.5 Tours de transição modificados	196
9.6 Um Método Alternativo	197
9,7 Resumo	199
Exercícios	200
Notas Bibliográficas	200
10. Protocolo de Síntese	203
10.1 Introdução	203
10.2 Derivação de protocolo	203
10.3 Algoritmo de Derivação	208
10.4 Projeto Incremental	210
10.5 Sincronização de local	210
10.6 Resumo	211
Exercícios	212
Notas Bibliográficas	212
11. Validação de protocolo	

Página 6

11.1 Introdução	214
11.2 Método de prova manual	214
11.3 Métodos de validação automatizados	218
11.4 O Algoritmo Supertrace	226
11.5 Detecção de ciclos de não progresso	231
11.6 Detectando Ciclos de Aceitação	234
11.7 Verificação de reivindicações temporais	235
11.8 Gerenciamento de Complexidade	235
11.9 Limite dos Modelos PROMELA	237
11,10 Resumo	238
Exercícios	239
Notas Bibliográficas	240
Parte IV - Ferramentas de Design	
12. Um Simulador de Protocolo	
12.1 Introdução	243

12.2 SPIN - Visão geral	
244	
12.3 Expressões	
245	
12.4 Variáveis	
255	
12.5 Declarações	
265	
12.6 Fluxo de Controle	
275	
12.7 Tipos de processos e mensagens	
282	
12.8 Expansão Macro	
292	
12.9 Opções de SPIN	
293	
12,10 Resumo	
294	
Exercícios	
295	
Notas Bibliográficas	
296	
13. Um validador de protocolo	
13.1 Introdução	
297	
13.2 Estrutura do Validador	
298	
13.3 O kernel de validação	
299	
13.4 A Matriz de Transição	
302	
13.5 O Código Validador-Gerador	
303	
13.6 Visão Geral do Código	
306	
13.7 Simulação Guiada	
308	
13.8 Alguns aplicativos	
310	
13.9 Cobertura no modo Supertrace	
315	
13,10 Resumo	
316	
Exercícios	
316	
Notas Bibliográficas	
317	

Página 7

14. Usando o validador	
14.1 Introdução	
318	
14.2 Um protocolo telegráfico óptico	
318	
14.3 Algoritmo de Dekker	
320	
14,4 Uma validação maior	
322	
14.5 Validação de controle de fluxo	

325	
14.6 Validação da Camada de Sessão	
336	
14.7 Resumo	
349	
Exercícios	
349	
Notas Bibliográficas	
349	
Conclusão	
351	
Referências	
352	
Apêndices	
A. Transmissão de Dados	
367	
B. Linguagem do fluxograma	
380	
C. Relatório de linguagem PROMELA	
383	
D. Fonte do Simulador SPIN	
393	
E. Fonte do validador SPIN	
436	
F. Protocolo de transferência de arquivo PROMELA	
528	
Índice de Nomes	
537	
Índice de Assuntos	
539	

Página 8

Página 9

PREFÁCIO

Os protocolos são conjuntos de regras que governam a interação de processos concorrentes na distribuição sistemas buted. O desenho do protocolo está, portanto, intimamente relacionado a uma série de campos, como sistemas operacionais, redes de computadores, transmissão de dados e dados comunicações. Raramente é escolhido e estudado como uma disciplina em seu próprio direito. Projetar um protocolo logicamente consistente que pode ser comprovado como correto, no entanto, é um tarefa desafiadora e muitas vezes frustrante. Já pode ser difícil nos convencer de a validade de um programa executado sequencialmente. Em sistemas distribuídos, devemos reafilho sobre programas de interação executados simultaneamente. Livros sobre sistemas distribuídos, redes de computadores ou comunicação de dados com frequência não fazem melhor do que descrever um conjunto de soluções padrão que foram aceitas como corretas, por exemplo, por grandes organizações internacionais. Eles não nos dizem porque o soluções funcionam, quais problemas resolvem ou quais armadilhas evitam. Este texto pretende ser um guia para a concepção e análise de protocolo, ao invés de um guia a padrões e formatos. Ele discute questões de design em vez de aplicativos. Dois questões, portanto, estão além do escopo deste texto: controle de rede (incluindo roteamento, endereçamento e controle de congestionamento) e implementação. Não há, no entanto, curto experimentação de textos sobre ambos os tópicos. O problema de design é abordado aqui como um fundamento fundamental e uma questão desafiadora, em vez de um obstáculo prático irritante para o desenvolvimento de sistemas de comunicação confiáveis. O objetivo do livro é familiarizá-lo com todas as questões de validação de protocolo e design de protocolo. A primeira parte do livro cobre o básico. O Capítulo 1 dá uma ideia dos tipos de

problemas que são discutidos. O Capítulo 2 trata da estrutura do protocolo e questões de design. Os capítulos 3 e 4 discutem os fundamentos do controle de erros e controle de fluxo. Os próximos quatro capítulos cobrem modelagem de protocolo formal e técnicas de especificação, começando nos capítulos 5 e 6 com a introdução do conceito de um protocolo de validação modelo de nação, que serve como uma abstração de um projeto e um protótipo de sua implementação. No Capítulo 5, uma nova linguagem concisa chamada PROMELA é introduzida para o Eu

Página 10

PREFÁCIO

XI

descrição dos modelos de validação de protocolo, e no Capítulo 6 é estendido para o especificação dos requisitos de correção do protocolo. No Capítulo 7, usamos PROMELA para discutir uma série de problemas de design padrão no desenvolvimento de um arquivo de amostra protocolo de transferência. A Parte II termina com uma discussão, no Capítulo 8, do modelo finito estendido

máquina de estados, uma noção básica em muitas técnicas de modelagem formal.

A terceira parte do livro concentra-se na síntese de protocolo, teste e tecnologia de validação técnicas que podem ser usadas para combater a complexidade de um protocolo. Ambos os recursos e as limitações das técnicas de design formal são abordadas.

A quarta e última parte do livro fornece uma descrição detalhada do design de dois ferramentas de projeto de protocolo baseadas em PROMELA : um intérprete e um validador automatizado. Com base nessas ferramentas, um gerador de implementação é simples de adicionar. Código-fonte para as ferramentas são fornecidas nos Apêndices D e E. A fonte também está disponível em eletrônico. As informações sobre pedidos podem ser encontradas no Apêndice E.

PLANO DE PALESTRA

O núcleo deste livro está contido nos capítulos 2, 5, 6, 7 e 11. Esses capítulos explorar uma disciplina de design que é suportada pelas ferramentas discutidas nos Capítulos 12 a 14. O restante do texto pretende tornar o livro relativamente independente. Capítulo 3 sobre controle de erros, Capítulo 4 sobre controle de fluxo e Capítulo 8 sobre máquinas de estado finito dar informações básicas que devem fazer parte do conhecimento prático de cada designer de protocolo. Os capítulos 9 e 10 trazem o leitor atualizado com as tecnologias mais recentes especialistas em campos estreitamente relacionados de engenharia de protocolo.

Para um curso de um semestre em desenho de protocolo, a seguinte sequência de capítulos e apêndices são sugeridos: 1, 2, A, 3, B, 4, 5 e C, 6-14. Um curso mais curto, por exemplo incorporado em um curso de semestre completo em sistemas operacionais ou redes de dados, sistema dos capítulos 1, 2, 5, 6, 7-11, 14. O software discutido no livro pode ser usado para projetos de classe na concepção e validação de protocolos de amostra. Sugestões para exercícios são incluídos ao longo do texto.

AGRADECIMENTOS

Muitas pessoas ajudaram na preparação deste livro. Leitores amigáveis que trabalharam nos rascunhos anteriores, incluindo Jon Bentley, Geoffrey Brown, Tom Cargill, John Chaves, Mohamed Gouda, Paul Haahr, Brian Kernighan, David Lee, Doug McIlroy, Sally McKee, Norman Ramsey, Howard Trickey e Colin West.

O software de validação foi desenvolvido ao longo de muitos anos. Ajuda crucial na derivação do algoritmo básico do supertrace foi dada por Doug McIlroy, Rob Pike, Jim Reeds e Ken Thompson. Costas Courcoubetis e Mihalis Yannakakis estendido o software com algoritmos para analisar propriedades de vivacidade.

Também sou grato a Greg Chesson, Tony Dabhura, Sandy Fraser, Joop Goudsblom, Andrew Hume, Mike Lesk, Don Mitchell, Beate Oestreicher, John Peterson, Bjo "rn Pehrson, Dave Presotto, S. Purushothaman, Krishan Sabnani, Ravi Sethi e M. Sullivan para referências e sugestões valiosas.

Página 11

xii

PREFÁCIO

PREFÁCIO

Gerard J. Holzmann

Página 12

INTRODUÇÃO 1

1 Primeiros Começos 1.1
9 As Primeiras Redes 1.2
12 protocolos como idiomas 1.3
13 Padronização de Protocolo 1.4
15 Resumo 1.5
16 exercícios
16 Notas Bibliográficas

1.1 COMEÇOS ANTES

O problema de projetar protocolos de comunicação eficientes e inequívocos existia muito antes de os primeiros computadores serem construídos. Existe uma longa história de tentativas construir sistemas para transferir informações rapidamente a longas distâncias. A partir de um ponto de vista do designer de protocolo, os percalços que foram causados por as comunicações são fascinantes. Claro, os problemas dos primeiros sistemas eram nem sempre documentado tão diligentemente quanto os recursos.

BEACONS E ALARUNS

Qualquer coisa que seja detectável a uma grande distância é um meio potencial de comunicação. Na peça *Agamenon* de 458 aC, por exemplo, Ésquilo descreve em detalhar como os sinais de fogo foram usados, supostamente, para comunicar a queda de Tróia para Atenas a uma distância de mais de 300 milhas. Mas o número de mensagens diferentes que pode ser transferido por um único grande incêndio é limitado. Um relato detalhado desse problema

1. É provavelmente

uma das primeiras descrições explícitas de métodos de transmissão de dados. Polybius começa por explicando por que um método de sinalização é útil em primeiro lugar.

"É evidente para todos que em todas as questões, e especialmente na guerra, o poder de ação no momento certo contribui muito para o sucesso das empresas e sinais de incêndio são os mais eficientes de todos os dispositivos que nos ajudam a fazer isso. Para eles mostram o que tem ocorrido recentemente e o que ainda está em andamento, e por meio deles quem se preocupa em fazê-lo, mesmo que esteja a uma distância de três, quatro ou até mais dias' viagem pode ser informada. De modo que é sempre surpreendente como a ajuda pode ser trazida por meios de disparar mensagens quando a situação assim o exigir."

O uso de sinais de fogo deve ter sido comum nos dias de Políbio. Mas lá foram alguns problemas para resolver.

1. *As Histórias*, Livro X, Capítulo 43. A tradução é de WR Patton e foi publicada por Harvard University Press em 1925.

1

Página 13

2

INTRODUÇÃO
CAPÍTULO 1

"Agora, em tempos anteriores, como os sinais de fogo eram balizas simples, eles eram na maior parte de pouca utilidade para aqueles que os usaram. Pois o serviço deveria ter sido executado por sinais previamente determinados, e como os fatos são indefinidos, a maioria deles desafiou comunicação por sinais de fogo. Para pegar o caso que acabei de mencionar, era possível para aqueles que concordaram com isso para transmitir informações de que uma frota havia chegado a Oreus, Peparethus, ou Chalcis, mas quando se tratava de alguns dos cidadãos tendo mudado de lado ou ter sido culpado de traição ou massacre ocorrido na cidade, ou qualquer coisa desse tipo, coisas que muitas vezes acontecem, mas não podem ser todas previstas - e é principalmente ocorrências inesperadas que requerem consideração e ajuda instantânea - todas tais assuntos desafiavam a comunicação por sinal de fogo. Pois era quase impossível ter um código pré-estabelecido para coisas que não havia meios de predição."

A observação crucial é a parte em negrito. Ao longo deste livro, veremos que é ainda é um problema. São as sequências inesperadas de eventos que levam a falhas de protocolo, e o problema mais difícil no projeto de protocolo é precisamente que devemos tentar esperar o inesperado.

Políbio continua seu relato com uma descrição de um novo método de sinalização que ele acreditado resolveu o problema de comunicação. É extremamente sofisticado, embora apenas parcialmente resolva o problema. O novo sistema usou dois conjuntos de cinco tochas. Por iluminando entre uma e cinco tochas em cada conjunto, um total de 5² personagens podem ser codificado, suficiente para transmitir mensagens arbitrárias como uma sequência de letras codificadas.

Tela

1

0

Tocha

Figura 1.1 - Telégrafo da Tocha

Conforme mostrado na Figura 1.1, as tochas podem ser usadas para enviar um código binário da tocha. UMA

a tocha pode ser tornada visível para o receptor remoto, levantando-a acima de uma tela e poderia ser escondido abaixando-o. Polybius descreve o código da tocha da seguinte maneira.

" Pegamos o alfabeto e o dividimos em cinco partes, cada uma consistindo de cinco letras.

Há uma letra a menos na última divisão, mas não faz diferença prática. Cada das duas partes que estão prestes a sinalizar uma para a outra devem agora preparar cinco tablets e escrever uma divisão do alfabeto em cada tabuleiro, e então chegar a um acordo que o homem que vai sinalizar deve em primeiro lugar levantar duas tochas e esperar até que o outro responda fazendo o mesmo. Isso tem a finalidade de transmitir a cada outro que ambos estão em atenção. Estas tochas foram baixadas, o despachante da mensagem agora levantará o primeiro conjunto de tochas no lado esquerdo indicando qual tablet deve ser consultado, ou seja, uma tocha se for a primeira, duas se for a segundo, e assim por diante. Em seguida, ele levantará o segundo conjunto à direita com o mesmo princípio

Página 14

SEÇÃO 1.1
COMEÇOS ANTES

3

para indicar que letra do tablet o receptor deve escrever. "

Nenhuma melhora real em relação ao telégrafo de Políbio foi feita por quase vinte séculos rias, embora não faltassem alternativas inferiores. Em 1684, o cientista inglês Robert Hooke descreveu um sistema óptico bastante desajeitado que funcionava com grandes personagens de madeira. Os personagens podem ser exibidos em uma estação de sinalização e observado à distância com um telescópio.² Pelo que sabemos, nunca foi colocado em prática.

Em 1796, o alemão G. Huth inventou um sistema igualmente malsucedido que ele chamou " telefone ". A ideia era colocar homens com " tubos de fala " nos telhados e peça-lhes que gritem mensagens entre si. Na verdade, a ideia de Huth já havia sido experimentada antes. Diz-se que Alexandre o Grande (356-323 aC) usou um megafone de 3,5 metros para gritar comandos para seus exércitos das colinas próximas. Não surpreendentemente, Políbio não gaste muito tempo discutindo este sistema.

Outro dispositivo notável foi usado durante a Guerra Revolucionária Americana (1775-1783). Consistia em um poste a partir do qual qualquer combinação de três objetos diferentes pode ser exibido. Com um barril, uma bandeira e uma cesta, 2³ – 1 mensagens diferentes poderia ser transmitido, embora obviamente não em uma sucessão muito rápida.

SISTEMAS ÓPTICOS

O primeiro sistema de telégrafo pré-elétrico de sucesso foi desenvolvido pelos franceses engenheiro Claude Chappe em 1793. Seu sistema consistia em grandes construções de madeira construído no topo de colinas ou torres de igreja e era operado por funcionários públicos equipados com telescópios. O semáforo possuía três partes móveis, regulador e dois indicadores , como ilustrado na Figura 1.2. O regulador tinha cerca de 15 pés de comprimento, os indicadores mediam aproximadamente 7 por 1 ft cada.

Não está claro a partir dos relatórios qual era a sinalização precisa do " alfabeto " ou como foi codificado nas posições de regulador e indicadores. Os braços do semáforo poderiam ser movido apenas em incrementos de 45 ° . Teoricamente, com três partes móveis, cada uma o semáforo pode ser definido em 256 ($8 \cdot 8 \cdot 4$) posições diferentes. Particularmente confuso combinações não foram usadas, por exemplo, posições onde os indicadores duplicam o ângulo do regulador. Alegadamente, cerca de metade das posições de semáforo válidas foram usado para codificar dígitos, sinais de pontuação, letras maiúsculas e minúsculas e outros metade foi usada para códigos de controle especiais. Os funcionários públicos foram contratados para ler o

posição do semáforo das estações vizinhas e copiá-lo em seu próprio semáforo phore para retransmitir mensagens.

No auge de seu sucesso, pouco antes dos telégrafos elétricos assumirem o controle, o sistema de Chappe tem crescido em uma rede completa de não menos que 556 estações de semáforo cobrindo percorrendo mais de 3000 milhas e alcançando quase todas as partes da França. Pouco se sabe

2. O telescópio também foi uma invenção recente na época. Foi descrito por Galileu em *Siderius Nuncius* (The Starry Messenger) em 1610.

4

INTRODUÇÃO
CAPÍTULO 1

sobre os procedimentos operacionais específicos empregados ou os problemas de coordenação que deve ter atormentado os operadores de semáforo. O que, por exemplo, era um semáforo operador deveria fazer quando duas mensagens chegassem simultaneamente do lado oposto instruções?

.....

.....

Indicador

Regulador

Figura 1.2 - Semáforo de Chappe

Quase todos os países tinham uma ou mais variações do telégrafo óptico de Chappe neste período. O almirantado britânico, por exemplo, usou um semáforo de seis venezianas projetado por um Lord George Murray. Cada obturador pode ser aberto ou fechado para transmitir um mensage: um código binário de 6 bits. O uso de mensagens de controle também está documentado para este

sistema. Todas as seis venezianas fechadas foram usadas para sinalizar que *não está pronto*, todas as seis venezianas abertas

significava *pronto para enviar*.

Figura 1.3 - Telégrafo de seis obturadores de George Murray

Um sistema semelhante, usando dez venezianas, foi desenvolvido na Suécia. O sistema sueco está documentado em detalhes em uma publicação de seu inventor, o secretário da chancelaria sueca. tary AN Edelcrantz, chamado *Avhandling om Telegrapher* publicado em 1796. Uma codificação mesa, com base em um sistema simples para atribuir números às posições do obturador, foi incluído.

"Toda a correspondência telegráfica é iniciada com um sinal indicando que você deseja fale, ou um sinal de falar, que é deixado até que o receptor dê o correspondente sinal de alerta. (...) Feito isso, o sinal de falar é retirado e o primeiro sinal em a mensagem é dada. O receptor então tira o sinal de alerta e repete o sinal do remetente para mostrar que leu corretamente. O mesmo procedimento é repetido para todos os sinais na mensagem.

Em 1796, este telégrafo conectou Estocolmo e A ° land. Uma estação telegráfica de obturador ção, construída em Furusund em 1836, sobreviveu e ainda pode ser visitada hoje (ver também

SEÇÃO 1.1
COMEÇOS ANTES

5

Notas Bibliográficas). Os códigos de sinalização usados no sistema sueco incluem códigos para controle de sessão (iniciar, parar), controle de erro, controle de fluxo (repetir), controle de taxa (mais lento, mais rápido), e até mesmo um reconhecimento negativo, que foi denominado apropriado realmente "não consigo ver."

A velocidade de transmissão dos telégrafos ópticos variava. No telégrafo de Chappe o a posição do semáforo foi alterada uma vez a cada 15-20 segundos. Com um subconjunto de 128 possíveis símbolos (ou 7 bits de informação), isso deu uma velocidade de transmissão de aproximadamente

0,5 bits / s. O código de 10 bits do telégrafo sueco foi alterado a cada 8-10 segundos, e o código de 6 bits do sistema britânico a cada 5 segundos, ambos fornecendo uma velocidade de sinalização de aproximadamente 1 bit / seg.

A visibilidade dos semáforos deve ter sido outra preocupação dos operadores. Em uma média de vinte dias por ano, por exemplo, as condições meteorológicas impediam o uso de um semáforo de obturador que conectou cinco cidades da Holanda entre 1831 e 1839.

Depois de 1840, o telégrafo elétrico finalmente provou ser mais rápido, mais confiável e menos visível do que os telégrafos ópticos.

ELETROMAGNETISMO

O princípio de um telégrafo elétrico foi descrito já em 1753 por um misterioso

"CM" em uma carta para a *Scots Magazine*.³ A identidade do autor nunca foi totalmente estabelecida. Algumas fontes dizem que as iniciais são de Charles Marshall

de Renfrew (a carta foi enviada por Renfrew). Outros afirmam que o autor foi alguém chamado Charles Morrison de Greenock. A carta descreve um tele elétrico gráfico com uma série de fios paralelos: um para cada código ou caractere diferente, a ser transmitido. Pequenas bolas de mite foram colocadas no receptor perto dos terminais de cada fio. O remetente pode colocar uma carga de eletricidade estática em um dos fios (por descarga um jarro de Leyden) e fazer com que a bola de fogo correspondente no receptor se mova. Pouco depois de 1830, um novo insight sobre a indução eletromagnética foi obtido por meio de o trabalho de Michael Faraday na Inglaterra e Joseph Henry nos Estados Unidos. No Inglaterra, o princípio foi usado em 1837 por William Cooke na construção do primeiro telégrafo elétrico. Cooke usou uma carga elétrica para desviar a agulha de uma bússola em um pequeno campo magnético no instrumento receptor. A ideia para tal "tele-agulha graph" foi aperfeiçoada por Cooke em cooperação com Sir Charles Wheatstone. Isso foi patenteado em 1837 como um *método de dar sinais e emitir alarmes à distância Locais por meio de correntes elétricas transmitidas por circuitos metálicos*. No Trabalho semelhante nos Estados Unidos foi feito por Samuel Morse e Theodore Vail.

3. The Scots Magazine, 17 de fevereiro de 1753, Vol. XV, pág. 73. A carta era intitulada *Um método expedito para transmitir inteligência*.

Página 17

6
INTRODUÇÃO
CAPÍTULO 1

uma
b
e
h
d
g
eu
f
k
Eu
m
r
v
y
n
s
W
o
t
P
I
2
3
4
5
8
6
7
Cooke's
Eight-Signal Telegraph
Wheatstone's
Dial de incubação

Figura 1.4 - Os primeiros telégrafos de múltiplas agulhas

A primeira patente, datada de 12 de junho de 1837, era para um sistema telegráfico com cinco agulhas. Qualquer combinação de duas das cinco agulhas pode ser desviada para a esquerda ou para a direita, o suficiente para sinalizar vinte letras diferentes. Na Figura 1.4, uma agulha de cinco e um são mostrados telégrafos de duas agulhas. Em ambos os instrumentos, apenas duas agulhas seriam desviadas de cada vez. Juntas, as duas agulhas apontariam para o caractere ou o código sendo transmitido. Um pouco depois, Cooke e Wheatstone também desenvolveram a transmissão de códigos para telégrafos de agulha única que incluíam um pequeno número de códigos de controle, como como *repetir* e *esperar*. O código de *repetição*, por exemplo, foi enviado em um tele de agulha única gráfico como uma sequência de dez cliques da agulha à direita.

William Cooke fez grandes esforços para vender seu sistema às companhias ferroviárias da Inglaterra.

terra como método de controle de tráfego. Em 1842, Cooke publicou um livreto divertido com um título longo, ⁴ que documenta seu lobby. Ele talvez também fosse um pouco otimista sobre os benefícios potenciais:

"... os trens podem prosseguir sem medo, seja no tempo ou fora do tempo, seja no certo ou na linha errada, já que sua velocidade sempre pode ser diminuída logo o suficiente para evitar uma colisão. "

O sistema foi prontamente adotado e usado em várias linhas da Great Western

4. Ferrovias telegráficas ou via única recomendada por segurança, economia e eficiência, sob a guarda e controle do telégrafo elétrico - com referência particular à comunicação ferroviária com Escócia e para as ferrovias irlandesas. (Cooke [1842]).

Página 18

SEÇÃO 1.1
COMEÇOS ANTES

7

Ferrovias na Inglaterra. Os primeiros experimentos mostraram que as despesas operacionais foram apenas um décimo daqueles para telégrafos ópticos, e as velocidades de transmissão eram muito superior.

Infelizmente, uma das primeiras aplicações do telégrafo elétrico foi para proteger trechos de ferrovia notoriamente perigosos, como linhas de via única e túneis.

Muitos acidentes ferroviários desse período foram causados por sutis mal-entendidos entre os sinalizadores usando o novo equipamento.

TREM QUEBRA

A causa de um acidente ferroviário é geralmente investigada e documentada em minutos detalhes, para que não haja falta de material sobre os primeiros problemas de projeto de protocolo. UMA um único exemplo pode ser suficiente para ilustrar como acidentes graves podem resultar apenas de uma combinação inesperada de eventos. Para ter certeza, o acidente a ser descrito poderia foram evitados se um protocolo adequado tivesse sido usado para a comunicação entre os sinaleiros.

Needle Telegraph

Signal Man

Semáforo

Túnel

UMA

B

Figura 1.5 - Túnel Clayton

O acidente ocorreu no túnel Clayton, que deve ter sido um dos melhores seções ferroviárias protegidas na Inglaterra. Em cada extremidade do túnel de 2,4 quilômetros de comprimento, ²⁴

horas por dia, sinaleiros estavam de serviço. Além disso, em 1841, o túnel foi equipado com um novo sistema de sinalização de bloco de intervalo de espaço. Havia semáforos sinais em cada extremidade do túnel, e o sistema de intervalo de bloco garantiu que qualquer trem passando um sinal verde automaticamente definir esse sinal para vermelho. Cabia ao sinal nalmem para redefinir os sinais para verde, mas antes de fazer isso eles eram obrigados a fazer certeza de que os trens que entraram no túnel de um lado realmente emergiram novamente em O outro fim.

Havia duas trilhas no túnel: uma para cada direção. Em todos os momentos, apenas um trem foi permitido por trilha no túnel. Como outra medida de segurança, o túnel tinha sido equipado com um telégrafo de agulha única. Este sistema foi configurado para o troca de um pequeno número de mensagens predefinidas entre os sinalizadores em ambos extremidades do túnel.

Normalmente, depois de permitir que um trem entre em um lado do túnel, o sinalheiro naquele lado transmitiu o *trem de código no túnel* para seu colega. Quando (e se) o trem emergiu do túnel na outra extremidade, seu colega respondeu com o código

Página 19

8
INTRODUÇÃO
CAPÍTULO 1

túnel está livre. Após o recebimento dessa mensagem, o primeiro sinalizador pode redefinir o sinal de entrada para permitir a entrada do próximo trem.

Para tornar o sistema à prova de falhas, um terceiro código de mensagem foi adicionado com o qual um

o sinalizador poderia perguntar ao colega: *o trem saiu do túnel?* A presença de dois sinalizadores garantiram que o túnel poderia ser usado com segurança mesmo que, por qualquer motivo,

o sinal do semáforo em ambos os lados do túnel não funcionou corretamente. Se um semáforo não conseguiu mostrar o vermelho após a passagem de um trem, o sinalizador foi avisado por um sino. Ele poderia então usar bandeiras vermelhas e brancas para sinalizar trens e manter o tráfego.

Mesmo assim, o protocolo acabou por ser especificado de forma incompleta. Aqui está o que aconteceu em

Agosto de 1861.

Um primeiro trem passa pelo semáforo A e não consegue definir o sinal para vermelho. Como esperado, o sino avisa o sinalizador em A (chame-o de sinalizador A). Ele primeiro transmite obedientemente o código *treina no túnel* para seu colega e, em seguida, busca a bandeira vermelha para avisar o próximo trem.

Um segundo trem, entretanto, é muito rápido e já passou pelo sinal verde. Para-felizmente, seu motorista avista a bandeira vermelha bem a tempo quando entra no túnel. Um terceiro trem é avisado a tempo e para totalmente antes do túnel Entrada.

O sinalizador A retorna à sua caixa e novamente sinaliza o *trem no túnel* para indicar que agora existem dois trens no túnel. O protocolo não levou em consideração este evento portanto, o significado de dois *trens* subsequentes em mensagens de *túnel* não foi especificado.

No entanto, uma vez que era improvável que o segundo trem pudesse ultrapassar o primeiro, nenhum problema real existia. O único problema para o sinalizador A era descobrir de seu colega quando os dois trens saíram do túnel, para que o terceiro pudesse entrar.

Para alertar seu colega sobre o problema, o sinalizador A transmite o único outro mensagem apropriada que ele tem: *o trem saiu do túnel?* Neste ponto, não há esperança de recuperação. Mesmo que o sinalizador em B pudesse entender precisamente o que o problema era que ele não tinha como comunicar isso. Depois de ver o primeiro trem emergir do túnel ele responde, em total acordo com suas instruções, *túnel está claro*.

O sinalizador A não pode saber se deve esperar até que dois *túneis* subsequentes *estejam livres* mensagens ou se a mensagem pode ser interpretada literalmente. Ele decide que ambos os trens devem ter saído do túnel e permite que o terceiro trem entre acenando um bandeira branca. O maquinista do segundo trem, porém, viu a bandeira vermelha enquanto entrando no túnel e parou totalmente no meio do túnel. Depois de

alguma deliberação o motorista decide jogar pelo seguro e voltar para fora do túnel.

Na colisão que se seguiu, 21 pessoas morreram e 176 ficaram feridas.

É difícil avaliar quem seria o culpado por esse acidente. Uma vez, por uma combinação de aberrações nação de eventos, tornou-se possível para o segundo trem entrar no túnel antes o primeiro havia saído, não havia como se recuperar. O senso comum de ambos sinalizadores e o maquinista do segundo trem não puderam evitar o acidente. O conjunto das instruções dadas aos sinalizadores estava incompleto. Na época, porém, alguns estavam mais ansiosos para culpar o método de sinalização de bloco relativamente novo ou o telégrafo

instrumentos do que os homens que elaboraram os procedimentos operacionais para o sinalizadores de interações.

Nos primeiros dias das ferrovias, muitos acidentes e quase acidentes foram o resultado de uma total falta de meios de comunicação. Mais tarde, quando as ferramentas certas estivessem disponíveis

capaz, foi descoberto o quanto surpreendentemente difícil pode ser estabelecer inequívocos regras de comunicação. Um historiador de desastres ferroviários (Nock [1967]) descreveu o problema da seguinte forma, muito em linha com as observações anteriores de Políbio:

"Quase se pode ouvir o mesmo comentário sendo feito vez após vez. 'Eu não pude imaginar que isso poderia acontecer. No entanto, a experiência amarga mostrou que poderia, e gradualmente os regulamentos e a prática da engenharia ferroviária foram elaborados.'

O problema era criar um conjunto de regras práticas e de bom senso que fosse eficiente para uso em circunstâncias normais e que permitiu uma recuperação segura de eventos.

1.2 AS PRIMEIRAS REDES

Embora originalmente o telégrafo elétrico fosse usado principalmente para sinalização ferroviária, não demorou muito para que ele se tornasse mais amplamente disponível. Em 1851, as bolsas de valores em Londres e Paris foram conectados por telégrafo, e o primeiro telégrafo público empresas foram fundadas. Em 1875, quase 320.000 milhas de linha telegráfica estavam em Operação. No início, os telégrafos eram operados com instrumentos de agulha ou Chaves de sinalização Morse. O código de sinalização usado com mais frequência foi um modificado Código Morse. O código Morse original usava três elementos de sinalização de duração variável ção: pontos, travessões e traços longos. A versão moderna foi introduzida em 1851, usando um código binário de comprimento variável dos dois elementos de sinalização familiares: pontos e travessões.

Uma primeira melhoria feita a este sistema ainda operado manualmente foi a fita de papel leitor de socos. Em 1858, Wheatstone construiu o *Wheatstone Automatic*, com o qual velocidades de transmissão de 300 palavras por minuto podem ser alcançadas (cerca de 30 bits / s). isto foi usado até muito recentemente. Depois de 1920, teclados especiais de "tele-máquina de escrever" e impressoras foram conectadas diretamente aos fios do telégrafo. O código de 5 bits que foi usado nessas máquinas foi desenvolvido pelo francês Emil Baudot em 1874. Em 1925 redes completas de "telex" (troca de telégrafos) estavam em operação.

No mesmo período, entre 1850 e 1950, dois outros métodos agora familiares de comunicação foram desenvolvidas: telefone e rádio. Elisha Gray e Alexander Graham Bell, por exemplo, entraram com seus pedidos de patente sobre a invenção do telefone⁵ em 1876, e em 1897 Guglielmo Marconi construiu e usou o primeiro rádio telegráfico.

⁵. A patente da Bell, de fato, não mencionava a palavra "telefone" de forma alguma; foi intitulado *Melhorias na Telegraphy*.

PROTOCOLOS MASTER-SLAVE

As demandas pelo rigor de novos protocolos de comunicação aumentaram drasticamente após 1950, quando a execução do protocolo foi automatizada pela primeira vez em grandes redes principais computadores de quadro.

Um dos primeiros computadores programáveis, o ENIAC, foi construído na Universidade da Pensilvânia em 1946. Pesava 30 toneladas. Como sabemos, após a invenção do transistor em 1947 por J. Bardeen, WH Brattain e W. Shockley da AT&T Bell Laboratórios, os sistemas subsequentes rapidamente se tornaram menores e mais rápidos. Apesar do tamanho não é realmente um problema no projeto de protocolo, mas a velocidade é. Ainda hoje, continua a alterar a natureza do problema de design do protocolo.

Os primeiros computadores tiveram que ser conectados a dispositivos periféricos, como fita de papel leitores e teclados de teletipo. Uma vez que os computadores eram inicialmente grandes, caros e escassos, um único mainframe "inteligente" era frequentemente conectado a grandes matrizes de periféricos "burros".

P
T
T
Mainframe
T

Figura 1.6 - Protocolos Mestre-Escravo

No início, os periféricos estavam bem próximos, digamos, na mesma sala que o computador mainframe, conectado por linhas multidrop. Se não houvesse dados para transferir para os periféricos, o mainframe iria "sonhar" os periféricos para ver se algum deles tinha dados para retornar ou um relatório de status para arquivar.

Já em 1956 ocorreram os primeiros experimentos com transmissão de dados a longa distância de computador para computador através de fios telefônicos, causando fundamentalmente diferentes tipos de problemas de controle. Seis anos depois, a primeira transmissão de dados via satélite

(Telstar) ocorreu.

Os primeiros protocolos de comunicação de dados executados em computadores eram de codificação bastante simples

das heurísticas de operações manuais. Os procedimentos foram usados para resolver uma problema de coordenação mestre-escravo funcional. Em todos os momentos, uma das duas partes envolvidas na comunicação estava no controle e era responsável por toda a transferência de dados, tarefas de recuperação, sincronização e gerenciamento de conexão. Muitos dos mais velhos protocols foram projetados com este conceito em mente. Protocolo *Bisync* da IBM, por exemplo, datas desse período. Na década de 1960, com conexões diretas de computadores mainframe

Página 22

SEÇÃO 1.2
AS PRIMEIRAS REDES

11

via redes de dados, o problema de projeto de protocolo tornou-se mais importante. Os dados as velocidades eram mais altas, a carga de tráfego maior e muito da conveniência do master-as relações de escravos foram perdidas. Os mainframes agora estavam falando diretamente uns com os outros, conectados em redes de pares.

M

M

M

M

M

Figura 1.7 - Rede de pares

PROTOCOLOS DE PARES

As primeiras redes de computadores em grande escala foram os sistemas de reserva de companhias aéreas da início dos anos 1960. O sistema SABRE da American Airlines, por exemplo, foi construído em 1961. Em 1969, foi desenvolvida uma grande rede de comutação de pacotes de uso geral, patrocinado pelo Departamento de Defesa dos EUA. Este ARPA (Advanced Research Projects Agency) conectou quase 1200 nós em 1985. A Internet, um sucessor para a rede ARPA, cresceu de cerca de 25.000 nós em 1987 para uma estimativa 250.000 nós no final de 1989.

O número de redes de dados privadas e públicas deve continuar a crescer rapidamente. A tecnologia disponível para a construção desses sistemas é muitas vezes sofisticada, pelo menos na medida em que o hardware e os procedimentos operacionais básicos são em causa. No entanto, embora os sistemas possam agora operar com fibras ópticas e satélite links, os problemas que devem ser resolvidos para utilizar um sistema de comunicação eficaz tivamente são essencialmente os mesmos que nos dias de Políbio.

O problema do projeto do protocolo é estabelecer um acordo sobre o uso de recursos em uma rede de pares. Não está imediatamente claro qual processo é responsável para qual tarefa; essas responsabilidades podem ter de ser negociadas. Se mais de um processo assume erroneamente a responsabilidade por uma tarefa, podendo resultar em confusão. A rede designers da década de 1960 aprenderam da maneira mais difícil que sequências muito improváveis de eventos

realmente acontecem e podem arruinar o melhor design.

Redes inteiras podem ser paralisadas por protocolos defeituosos ou incompletos. Embora um colisão de dois fluxos de dados em um canal de satélite pareça inofensivo em comparação com um de frente colisão de dois trens, em ambos os casos o dano pode ser substancial.

Página 23

12
INTRODUÇÃO
CAPÍTULO 1

1.3 PROTOCOLOS COMO LÍNGUAS

O termo *protocolo* para um procedimento de comunicação de dados foi usado pela primeira vez por RA Scantlebury e KA Bartlett no National Physical Laboratory na Inglaterra, em um memorando que foi escrito em abril de 1967. O memorando foi intitulado *Um protocolo*

para uso na rede de comunicações de dados NPL.

Já sabemos que um protocolo é uma espécie de acordo sobre a troca de informações mação em um sistema distribuído. Uma definição completa de protocolo, na verdade, parece muito com um definição de linguagem.

Ele define um *formato* preciso para mensagens válidas, como os pontos e travessões que compõem o código Morse (uma sintaxe).

Ele define as *regras de procedimento* para a troca de dados (uma gramática).

E define um *vocabulário* de mensagens válidas que podem ser trocadas, com seus significado (semântica).

Iremos chegar a uma definição ligeiramente estendida de um protocolo no próximo capítulo.

Mas observe que a gramática do protocolo deve ser logicamente consistente e completa: em todas as circunstâncias possíveis, as regras devem prescrever em termos inequívocos o que é permitido e o que é proibido. Na prática, este é um requisito difícil de Conhecer.

Embora os protocolos, de uma forma ou de outra, tenham sido usados em comunicações de longa distância

sistemas de comunicação ao longo da história, até recentemente sempre houve um humano operador que poderia ser confiável para tomar decisões de bom senso para resolver problemas detectados. No código telex de 5 bits, existem até dois símbolos especiais para invocar ação humana: o código 10010 significa *quem está aí?*, e o código 11010 toca um Sino.

Ao usar máquinas em vez de operadores humanos, temos a mesma comunicação e problemas de coordenação, mas desta vez os erros podem acontecer mais rápido, e não podemos mais confiar na intervenção humana para se recuperar de casos inesperados.

Um importante requisito oculto do projeto de protocolo agora é óbvio: não apenas deve haver regras para a troca de informações, também deve haver um *acordo* entre o remetente e o destinatário sobre essas regras. Protocolo *Bisync* da IBM , para por exemplo, foi implementado em muitos sistemas diferentes e em cada novo sistema foi embelezado com o inevitável senso comum do implementador de atalhos e melhorias. Essas interpretações ligeiramente diferentes das regras do *Bisync* protocolo descartou qualquer esperança de que duas implementações escolhidas arbitrariamente do mesmo protocolo poderia realmente se comunicar. Em vez de levar a diretrizes mais rígidas para o concepção, especificação e implementação de protocolos, o que levou à instituição de organismos internacionais de normalização.

Página 24

SEÇÃO 1.4
PADRONIZAÇÃO DE PROTOCOLO
13

1.4 PADRONIZAÇÃO DE PROTOCOLO

Muitos organismos de normalização atuam na área de comunicação de dados. Exemplos são o Instituto Nacional de Ciência e Tecnologia (NIST, anteriormente o National Bureau of Standards ou NBS), os Padrões Federais de Telecomunicações Comitê (FTSC) e o Instituto de Engenheiros Elétricos e Eletrônicos (IEEE).

Os dois organismos de normalização mais importantes nesta área, no entanto, são a ISO e o CCITT.

A International Standards Organization (ISO) inclui muitos padrões nacionais órgãos, como o American National Standards Institute (ANSI). ANSI é responsável por padrões importantes, como o código de caracteres ASCII e o Definição da interface RS232. O ISO é organizado em comitês técnicos (TC), cada um organizado em subcomitês (SC) e grupos de trabalho (WG). TC97, para por exemplo, está preocupado com padrões para computadores, TC97 / SC6 lida com telecomunicações e TC97 / SC6 / WG1 trabalha em padrões para proto de enlace de dados cols. O código ASCII é formalmente conhecido como padrão ISO 646. Ao contrário do CCITT, a ISO não é uma organização de tratado e a adesão é voluntária.

O Comite ' Consultatif International Te ' le ' graphique et Te ' le ' phonique (CCITT) é parte da União Internacional de Telecomunicações (UIT). O CCITT é uma ONU organização de tratado que foi formada em 1956 pela união de duas entidades separadas:

o CCIT (sistemas telegráficos) e o CCIF (sistemas telefônicos). Hoje isso inclui muitas das empresas de telefonia pública, como os PTTs europeus e AT&T da América. O Departamento de Estado dos EUA também é membro oficial do organizaçāo. O CCITT está organizado em grupos de estudos (SG) e grupos de trabalho (WP). SGVII, por exemplo, está preocupado com redes de comunicação de dados, e SGVII / WP2 funciona em interfaces de rede. O código telex de 5 bits é oficialmente conhecido como CCITT-Alphabet No. 2. As recomendações de protocolo mais conhecidas publicadas pelo CCITT são X.21 e X.25 (consulte também o Capítulo 2). X.21 tem o duvidoso honra (ver Notas Bibliográficas do Capítulo 11) de ser o primeiro protocolo de referência a ser validado por análise exaustiva de acessibilidade.

Outra organizaçāo que realiza trabalhos importantes nesta área, embora não seja diretamente envolvida com a padronização de protocolo, é a International Federation for Information Processamento (IFIP). Um dos objetivos do IFIP é servir como uma organizaçāo-ponte que conecta o trabalho realizado em órgãos como o CCITT e o ISO. Como o ISO, o IFIP está organizado em Comitês Técnicos (TC), onde cada Comitê Técnico é subdividido em Grupos de Trabalho (GT). TC6, por exemplo, é dedicado a dados comunicações e WG 6.1 estuda *Arquitetura e Protocolos para Rede de Computadores*. O IFIP foi estabelecido em 1960.

Claro, a padronização do protocolo ainda não resolve o problema de design do protocolo em si. Afinal, de que adianta um padrão internacional incompleto ou mesmo defeituoso? Os organismos de padronização enfrentam o mesmo problema de todos os outros protocolos designers, e pode-se dizer que "design por comitê" nem sempre garante obter os melhores resultados.

Antes que este problema possa ser resolvido, precisaremos de métodos convincentes para *projetar e*

Página 25

14
INTRODUÇÃO
CAPÍTULO 1

descrever protocolos e métodos eficazes para *verificar se* qualquer protocolo submetido a um corpo de padronização está correto. Claramente, para projetar e descrever um protocolo, precisamos ser capaz de expressar seus critérios de design, e para verificar um protocolo de forma eficaz, precisamos ser

capaz de verificar se seus critérios de design são atendidos.

O problema de definir um formato comum para os protocolos de especificação em padrões documentos de zação têm sido estudados por muitos anos. Três especificações de protocolo linguagens já foram desenvolvidas: SDL, Lotos e Estelle. São comumente referidos como os três FDTs, ou Técnicas de Descrição Formal.

A especificação e linguagem de descrição (SDL) foi desenvolvida por estudo grupos SGXI e SGX do CCITT. Destina-se especificamente para a especificação e projeto de sistemas de telecomunicações, como comutadores telefônicos. O estudo foi iniciado em 1968. Uma primeira versão tornou-se a Recomendação CCITT Z101-Z104 em 1976, e versões revisadas foram publicadas em 1982 e em 1985. Um final, estável versão foi aprovada em 1987. Existem duas variantes, amplamente equivalentes, do SDL em uso: uma forma gráfica e uma forma de programa. A linguagem de fluxograma usada em a primeira parte deste livro é vagamente baseada na forma gráfica (consulte o Apêndice B). A linguagem de especificações de ordenação temporal (Lotos) está sendo desenvolvida dentro da ISO, TC97 / SC21 / WG1. Lotos também é chamado de "álgebra de processo". baseado no cálculo de sistemas de comunicação (CCS) desenvolvido por Robin Milner da Universidade de Edimburgo. O principal objetivo das álgebras de processo é a especificação formal de comportamentos de processo em um alto nível de abstração. o álgebras definem um conjunto rigoroso de regras de transformação e relações de equivalência que pode permitir a um designer raciocinar formalmente sobre comportamentos. Lotos foi emitido como Padrão internacional ISO IS8807 em fevereiro de 1989.

Estelle é uma segunda técnica de descrição formal sendo desenvolvida dentro de outro subgrupo da ISO TC97 / SC21 / WG1. Um total de três subgrupos de estudo do WG1 técnicas de descrição formal estão ativas desde 1981. (O terceiro subgrupo estuda métodos arquitetônicos.) A linguagem Estelle é baseada em uma nadadeira estendida conceito de máquina de estado ite (consulte o Capítulo 8). Foi emitido como padrão internacional ISO dard IS9074 em julho de 1989.

Lotos é o único FDT desta gama que também aborda especificamente os problemas de design item. Podemos aprender muito com a experiência adquirida aqui. Nenhum dos FDTs, no entanto, também abordaram o problema de que projetos completos devem ser verificáveis em o nível de especificação do protocolo. Devemos ser capazes de verificar, de preferência com automação ferramentas, que um projeto atende aos seus requisitos. Tal como está, a verificabilidade não pode ser garantido para qualquer um dos FDTs. As especificações Lotos e SDL, por exemplo, podem especificar sistemas infinitos, o que torna muitos problemas de verificação formalmente indecisos capaz. Existe uma área ativa de pesquisa para desenvolver ferramentas para subconjuntos das línguas, mas também aqui os problemas a serem resolvidos são formidáveis.

Página 26

SEÇÃO 1.5

RESUMO

15

1.5 RESUMO

O projeto do protocolo não é um problema novo. É tão antigo quanto a própria comunicação. Somente quando a interpretação das regras do protocolo teve que ser automatizada em alta velocidade máquinas, foi descoberto que o projeto de protocolo em si pode ser um problema desafiador item. Os protocolos em desenvolvimento hoje são maiores e mais sofisticados do que nunca antes. Eles tentam oferecer mais funcionalidade e confiabilidade, mas, como resultado, têm aumentado em tamanho e complexidade. O problema que um designer enfrenta agora é fundamental mental: como projetar grandes conjuntos de regras mínimas para a troca de informações, logicamente consistente, completo e implementado com eficiência. O problema pode ser abordado por dois lados.

Dado um problema, como um designer pode resolvê-lo sistematicamente para que o design requisitos são realizados?

Dado um protocolo, como um analisador pode demonstrar convincentemente que ele está em conformidade

aos requisitos de correção?

Neste livro, estudamos o problema fundamental de projetar e analisar protocolos que formalizam interações em sistemas distribuídos. Normalmente, essas serão interações de computadores, mas eles se aplicam igualmente à interação de pessoas com tocha de telegráficos. O problema em todos esses sistemas é chegar a um conjunto inequívoco de regras que permitem iniciar, manter e concluir trocas de informações religiosas habilmente.

DESIGN DISCIPLINE

Primeiro precisamos entender quais são os problemas básicos, e passamos os primeiros capítulos estudando isso. Em seguida, precisamos estabelecer uma disciplina de design, um conjunto de auto-

restrições impostas que podem nos ajudar a evitar problemas. Mas isso não é tudo. Tudo fresco protocolos projetados, não importa o quanto disciplinados seus designers tenham sido, devem ser tratado com suspeita.

Todo protocolo deve ser considerado incorreto até que o contrário seja provado.

Argumentaremos que, para provar a correção ou incorreção dos protocolos, um bom conjunto de ferramentas de design eficientes e automatizadas são indispensáveis.

FERRAMENTAS DE DESIGN

Nem mesmo o melhor conjunto de regras pode evitar todos os erros. Esse é um fato simples da vida. Nós devemos exigir, no entanto, que as regras de protocolo sempre fornecem uma recuperação normal dos erros que ocorrem. Não é bom o suficiente se as regras do protocolo permitirem uma interpretação que evita desastres em circunstâncias inesperadas. Devemos exigir que as regras impedem interpretações que podem levar ao desastre.

Os métodos de design que desenvolvemos neste livro são baseados no conceito de uma *validação modelo*. Um modelo de validação expressa as características essenciais do protocolo, sem entrar em detalhes de sua implementação. Ferramentas automatizadas podem interpretar esses modelos de validação e encontrar as falhas no design com precisão implacável.

Nos próximos capítulos, começaremos a explorar a estrutura geral do protocolo de comunicação e algumas das questões básicas envolvidas no projeto do protocolo.

Página 27

16

INTRODUÇÃO
CAPÍTULO 1
EXERCÍCIOS

1-1. O código de transmissão desenvolvido por Políbio para seu telégrafo de tocha dividiu o Alfabeto grego em cinco grupos. Os primeiros quatro grupos tinham cinco letras cada, e o quinto grupo tinha os quatro restantes.

O telégrafo funcionava com dois grupos de tochas: um era usado para codificar o grupo número, o outro para transmitir o número do caractere dentro desse grupo. A transmissão levou coloque personagem por personagem, levantando e abaixando tochas nos dois grupos. Lá não havia códigos para espaços para separar palavras, nem para qualquer tipo de pontuação. (Punctuação também não foi usada em grego escrito.) Houve, no entanto, uma com adicional mensagem trol para sinalizar o início de uma mensagem: duas tochas levantadas simultaneamente (veja a citação de Políbio na página 2).

Quais são os possíveis problemas de sincronização, na ausência de um acordo adequado sobre a ordem em que as tochas dos dois grupos devem ser baixadas e levantadas?

1-2. Estime a velocidade de transmissão do telégrafo da tocha e compare-a com o sistema de Chappe tem. Quanto tempo leva para transmitir a mensagem "falha de protocolo?"

1-3. Polybius recomendou a compactação de mensagens para reduzir o tempo de transmissão e, assim, o número de erros. Comente sobre esta disciplina. Dica: considere a técnica oposta de aumentar a redundância para proteger contra erros de transmissão e interpretação.

1-4. Se os sinaleiros no túnel Clayton tivessem o caractere completo definido em sua agulha telégrafos, considere como eles poderiam tê-los usado para resolver o problema. O comprimento do túnel tem 1,5 milhas, a velocidade dos trens era de aproximadamente 45 milhas por hora, e a velocidade de transmissão de um telégrafo de agulha é de cerca de 25 símbolos por minuto.

O problema para os sinaleiros era estabelecer o paradeiro do segundo trem. Em o momento crucial em que o segundo trem estava recuando para fora do túnel para onde o terceiro trem estava esperando. O sinaleiro em A presumiu que o segundo trem já havia deixado o túnel nel; o sinaleiro em B não sabia que um segundo trem estava envolvido.

1-5. Tente revisar o protocolo do Túnel Clayton para evitar completamente a possibilidade de o acidente. Não suponha que o número de trens no túnel seja sempre zero ou um, e não presuma que os trens sempre viajam em uma direção.

1-6. O código completo para o telégrafo de agulha tinha uma mensagem de *repetição* que poderia ser usada para solicitar a retransmissão da última mensagem enviada pela outra estação. Considere o que aconteceria se esta disciplina fosse estritamente aplicada e a própria mensagem *repetida* fosse a última mensagem transmitida de ambas as estações.

1-7. (Jon Bentley) Se uma chamada telefônica for encerrada inesperadamente, há um "tele informal protocolo telefônico" que diz que o chamador deve rediscar a chamada. Se a pessoa chamada for desconhecendo este protocolo resulta um problema curioso. Um "Paradoxo do Amante" previne contra tato de ser feito quando ambas as partes tentam estabelecê-lo simultaneamente. O que é falha de protocolo? Suponha que os chamadores sejam máquinas, como as máquinas poderiam ser programado para evitar que o problema se repita ad infinitum? O que acontece com este protocolo se ambas as partes têm um recurso de "interrupção de chamada" (a capacidade de tomar um extra ligar quando já estiver fora do gancho)?

NOTAS BIBLIOGRÁFICAS

O engenheiro francês Claude Chappe nasceu em Bruyères, França, em 1763. Ele originou finalmente ingressou em uma ordem religiosa como monge, mas em 1791 foi forçado a deixar a ordem.

Página 28

CAPÍTULO 1
NOTAS BIBLIOGRÁFICAS

17

Junto com seu irmão Ignace, ele montou uma loja para trabalhar no telégrafo. Unica dele a publicação foi uma nota curta sobre o telégrafo óptico de 1798 (Chappe [1798]). Seu a vida é descrita em um livro de seu irmão, publicado em 1824 (Chappe [1824]). Claude Chappe cometeu suicídio em 1805, supostamente quando outros reivindicaram o crédito por seu invenções.

O telégrafo de veneziana usado na Inglaterra foi projetado por Lord George Murray em 1794.

É descrito em Reid [1886] e Michaelis [1965]. O sistema estava em operação até 1816. O sistema Edelcrantz, e seu código de sinalização, é descrito em Edelcrantz [1796]. Malmgren [1964] e Herbarth [1978] escrevem que o sistema óptico coexistiu com os primeiros telégrafos elétricos por um período de cerca de cinco anos. Herbarth [1978] inclui uma história detalhada das redes de telégrafo óptico que foram construídas na França, Suécia, Inglaterra e Alemanha. Uma foto da estação telegráfica em Furusund pro visitou o logotipo para a 11ª conferência sobre Especificação de Protocolo, Teste e Verificação cação, realizada em Estocolmo em 1991.

Os telégrafos de agulha de Cooke e Wheatstone foram usados para sinalização na Grã-Bretanha ferrovias até meados do século XX. Uma descrição dos primeiros telégrafos, como o instalado no túnel Clayton, pode ser encontrado em Hubbard [1965], março

land [1964], Michaelis [1965], Prescott [1877] e Bennet e Davey [1965]. Somente dois dos instrumentos telegráficos de cinco agulhas mostrados na Figura 1.4 já foram construídos. 1 destes está agora no London Science Museum; o outro está no correio de Berlim Museu.

Ainda não é incomum, embora menos frequente, que os procedimentos de sinalização ferroviária sejam revisado após um acidente grave ter demonstrado que eventos improváveis ocorrem na prática tice. A causa de até mesmo acidentes ferroviários menores é geralmente estudada em grande detalhe e bem documentado; ver, por exemplo, Nock [1967], Rolt [1976], Schneider e Mase [1968] e Shaw [1978].

Muito também se sabe sobre os métodos de sinalização de tambor, às vezes elaborados, usados por Tribos africanas e australianas e os sinais de fumaça e fogo dos índios americanos.

As descrições podem ser encontradas em Mallery [1881] e Hodge [1910]. Tele óptica de Hooke Graph e o " basket telegraph " americano são descritos em Still [1946].

O primeiro uso do termo " protocolo " para sistemas de comunicação de dados foi atribuído para Scantlebury e Bartlett em Campbell-Kelly [1988]. Ele escreve:

"A lembrança de Bartlett é que o termo 'procedimento' tinha sido usado até aquele ponto, mas foi agora contestado com o fundamento de que tinha adquirido um significado especial no Relatório ALGOL. "

O termo tornou-se parte permanente do jargão da informática quando foi adotado no início dos anos 1970 pelos desenvolvedores da rede ARPA (Pouzin e Zimmerman [1978]).

A linguagem de especificação SDL está documentada em CCITT [1988]; veja também Saracco, Smith e Reed [1989], Rockstrom e Saracco [1982] e Saracco e Tilanus [1987]. A construção de um validador automatizado para um subconjunto de SDL é discutida

Página 29

18

INTRODUÇÃO
CAPÍTULO 1

em Holzmann e Patti [1989]. Excelentes introduções a Lotos podem ser encontradas em Brinksma [1987, 1988], Bolognesi e Brinksma [1987] e Eijk, Vissers e Diaz [1989]. Uma visão geral do cálculo para sistemas de comunicação CCS pode ser encontrada em Milner [1980].

Para uma perspectiva diferente de trabalho de padronização de protocolo e desenvolvimento de os três FDTs, ver também Bochmann [1986] e Vissers [1990]. Estelle é descrita em Budkowski e Dembinski [1987].

Página 30

ESTRUTURA DO PROTOCOLO 2

- 19 Introdução 2.1
- 21 Os Cinco Elementos de um Protocolo 2.2
- 22 Um Exemplo 2.3
- 26 Serviço e Meio Ambiente 2.4
- 32 Vocabulário e formato 2.5
- 35 Regras de Procedimento 2.6
- 35 Projeto de Protocolo Estruturado 2.7
- 38 Dez Regras de Design 2.8
- 39 Resumo 2.9
- 39 exercícios
- 40 notas bibliográficas

2.1 INTRODUÇÃO

No primeiro capítulo, vimos alguns exemplos gerais dos problemas de projeto de protocolo lem. Tendo escolhido um meio de transmissão, seja um telégrafo de tocha ou uma fibra óptica, temos que escrever um conjunto de regras para seu uso adequado, definindo como as mensagens são codificado, como uma transmissão é iniciada e encerrada e assim por diante. Dois tipos de erros são difíceis de evitar: projetar um conjunto incompleto de regras ou criar regras que são contraditórios.

Neste capítulo, examinamos maneiras de garantir que o conjunto de regras seja completo e consistente. Requer que sejamos muito precisos na especificação de *todas* as peças relevantes de um protocolo. Também requer alguma disciplina na separação de questões ortogonais, usando modularidade e estrutura.

Vejamos primeiro os tipos gerais de serviços que uma comunicação por computador oferece

o tocol deve ser capaz de fornecer. Suponha que temos dois computadores, *A* e *B*. *A* é con- conectado a um dispositivo de armazenamento de arquivo *d*, e *B* se conecta a uma impressora *p*. Queremos enviar um texto arquivo do armazenamento de arquivo em *A* para a impressora em *B*.

p

d

UMA

B

Figura 2.1 - Servidor de arquivos e servidor de impressão

Obviamente, para serem capazes de se comunicar, as duas máquinas devem usar o mesmo ph- fios elétricos, use codificações de caracteres compatíveis e transmite e escaneie os sinais em os fios mais ou menos na mesma velocidade. Mas, supondo que esses problemas tenham sido resolvido, ainda há mais para o problema do que enviar sinais por um fio.

19

Página 31

20

ESTRUTURA DO PROTOCOLO
CAPÍTULO 2

Um deve ser capaz de verificar se há ou não a impressora está disponível. Deve ser capaz de adaptar a taxa em que está enviando os caracteres para a taxa em que a impressora pode os manuseie. Especificamente, a máquina deve ser capaz de suspender o envio quando o a impressora fica sem papel ou está desligada.

É importante observar que, embora os dados reais fluam em apenas uma direção, de *A* para *B*, precisamos de um canal *bidirecional* para trocar informações de controle. Os dois as máquinas devem ter chegado a um acordo prévio sobre o significado das informações de controle e sobre os procedimentos usados para iniciar, suspender, retomar e concluir as transmissões. No Além disso, se erros de transmissão forem possíveis, as informações de controle devem ser trocadas para proteger a transferência dos dados. Mensagens de controle típicas, por exemplo, são positivas e confirmações negativas que podem ser usadas pelo receptor para avisar o remetente se os dados foram recebidos intactos ou não.

Todas as regras, formatos e procedimentos que foram acordados entre *A* e *B* são chamados coletivamente de *protocolo*. De certa forma, o protocolo formaliza a interação por padronizando o uso de um canal de comunicação. O protocolo, então, pode conter acordos sobre os métodos usados para:

Início e término de trocas de dados

Sincronização de remetentes e receptores

Detecção e correção de erros de transmissão

Formatação e codificação de dados

A maioria dessas questões pode ser definida em mais de um nível de abstração (Figura 2.2).

Em um baixo nível de abstração, por exemplo, qualquer preocupação de sincronização se aplica ao sincronização do relógio do remetente e do receptor que é usado para dirigir ou escanear o linha de transmissão física. Em um nível mais alto de abstração, está preocupado com o sincronização de transferências de mensagens (por exemplo, em controle de fluxo e controle de taxa métodos), e em um nível ainda mais alto, trata da sincronização e coordenação das principais fases do protocolo.

No nível mais baixo, uma definição de formato pode consistir em um método para codificar bits com sinais elétricos analógicos. Um nível acima, pode consistir em métodos para codificar o caracteres individuais de um alfabeto de transmissão em padrões de bits. Próximo personagem os códigos podem ser agrupados em campos de mensagem e os campos de mensagem em frames ou pacotes,

cada um com um significado e estrutura específicos.

Os métodos de controle de erro exigidos em um protocolo dependem das propriedades específicas de o meio de transmissão usado. Esta mídia pode inserir, excluir, distorcer ou até mesmo duplicar e reorganizar mensagens. Dependendo do comportamento específico, o protocolo designer pode conceber uma estratégia de controle de erros.

As descrições de protocolo que discutimos até agora foram bastante informais e frag- mentado. Infelizmente, isso não é incomum. É muito tentador confiar no boa vontade e bom senso do leitor (ou implementador) para preencher os detalhes que foram omitidos, para compreender as suposições ocultas e para eliminar a ambigüidade do língua. Um primeiro passo para um projeto de protocolo mais confiável é formalizar e

estruture as descrições, para tornar explícitas todas as suposições.

Página 32

SECÃO 2.2
OS CINCO ELEMENTOS DE UM PROTOCOLO

21

campos de mensagem

frames / pacotes

sinal elétrico

bits

símbolos / personagens

Figura 2.2 - Amostra de níveis de abstração: formatação

Na próxima seção, começamos este processo considerando quais são os elementos essenciais em uma definição de protocolo são.

2.2 OS CINCO ELEMENTOS DE UM PROTOCOLO

Uma especificação de protocolo consiste em cinco partes distintas. Para ser completo, cada especificação deve incluir explicitamente:

1. O serviço a ser prestado pelo protocolo
2. As suposições sobre o ambiente em que o protocolo é executado
3. O vocabulário de mensagens usado para implementar o protocolo
4. A codificação (formato) de cada mensagem no vocabulário

5. As regras de procedimento que protegem a consistência das trocas de mensagens

O quinto elemento de uma especificação de protocolo é o mais difícil de projetar e também o mais difícil de verificar. A maior parte deste livro é, portanto, dedicada precisamente a isso tópico: o design e a validação de conjuntos inequívocos de regras de procedimento.

Cada parte da especificação do protocolo pode definir uma hierarquia de elementos. O protocolo vocabulário col, por exemplo, pode consistir em uma hierarquia de classes de mensagens. Similarmente, a definição do formato pode especificar como as mensagens de nível superior são construídas a partir de elementos de mensagem de nível inferior e assim por diante.

Conforme observado no Capítulo 1, uma definição de protocolo pode ser comparada a uma definição de linguagem:

contém um *vocabulário* e uma definição de *sintaxe* (ou seja, o formato do protocolo); a

Página 33

22
ESTRUTURA DO PROTOCOLO
CAPÍTULO 2

regras de procedimento definem coletivamente uma *gramática*; e a especificação do serviço define a *semântica* da linguagem.

Existem alguns requisitos especiais que devemos impor a este idioma. Como qualquer um linguagem do computador, a linguagem do protocolo deve ser *inequívoca*. Ao contrário da maioria linguagens de programação, no entanto, a linguagem do protocolo especifica o comportamento de processos atualmente em execução. Esta simultaneidade cria uma nova classe de problemas sutis. Temos que lidar com, por exemplo, *tempo*, *condições de corrida* e possíveis *impasses*. Uma vez que a sequência precisa de eventos nem sempre pode ser prevista, o número de possíveis ordens de eventos pode ser tão esmagador que derrota qualquer tentar analisar o protocolo por análise de caso manual simples.

A próxima seção dá um exemplo informal da definição dos cinco elementos do protocolo mentos e os tipos de erros que podem permanecer em um projeto. Em seguida, consideramos cada um dos cinco elementos principais do protocolo em mais detalhes. O capítulo é concluído com uma discussão das técnicas de projeto de protocolo que podem ajudar a estruturar um projeto, de modo que em última análise, pode ser implementado de forma eficiente e comprovado como correto com ferramentas automatizadas.

2.3 UM EXEMPLO

O seguinte protocolo foi descrito por WC Lynch [1968] como

"... um esquema de aparência razoável, mas inadequado publicado por um dos principais fabricantes de computador em um manual de informações do sistema."

Discutimos este protocolo aqui para ver como podemos identificar os blocos de construção básicos em uma especificação discutida acima. Vamos primeiro considerar a especificação do serviço.

ESPECIFICAÇÕES DE SERVIÇO

O objetivo do protocolo é transferir arquivos de texto como sequências de caracteres em um linha telefônica enquanto protege contra erros de transmissão, assumindo que todos erros de transmissão podem de fato ser detectados. O protocolo é definido para arquivo full-duplex transferência, ou seja, deve permitir transferências em duas direções simultaneamente (ver também Apêndice A). Confirmações positivas e negativas para o tráfego de *A* para *B* são enviadas no canal de *B* para *A* e vice-versa. Cada mensagem contém duas partes: uma mensagem parte sábia e uma parte de controle que se aplica ao tráfego no canal reverso.

PRESSUPOSTOS SOBRE O MEIO AMBIENTE

O "ambiente" em que o protocolo deve ser executado consiste no mínimo em dois usuários do serviço de transferência de arquivos e um canal de transmissão. Os usuários podem ser assumiu simplesmente submeter um pedido de transferência de arquivo e aguardar sua conclusão. o canal de transmissão é considerado para causar distorções arbitrárias de mensagem, mas não para perder, duplicar, inserir ou reordenar mensagens. Vamos assumir aqui que um nível inferior módulo (consulte o Capítulo 3) é usado para capturar todas as distorções e transformá-las em undis mensagens torturadas do tipo *err*.

Página 34

SEÇÃO 2.3
UM EXEMPLO
23

VOCABULÁRIO DE PROTOCOLO

O vocabulário do protocolo define três tipos distintos de mensagens: *ack* para uma mensagem combinado com um reconhecimento positivo, *nak* para uma mensagem combinada com uma nega-confirmação positiva e *err* para uma mensagem com um erro de transmissão. O vocabulário lar pode ser expresso de forma sucinta como um conjunto:

$$V = \{\text{ack}, \text{err}, \text{nak}\} .$$

Cada tipo de mensagem pode ser refinado em uma classe de mensagens de nível inferior, con consistindo, por exemplo, de um subtipo para cada código de caractere a ser transmitido.

FORMATO DE MENSAGEM

Cada mensagem consiste em um campo de controle que identifica o tipo de mensagem e um campo de dados

com o código do personagem. Para o exemplo, assumimos que os campos de dados e controle são de tamanho fixo.

A forma geral de cada mensagem agora pode ser representada simbolicamente como um simples estrutura de dois campos:

{tag de controle, dados}

que em uma especificação semelhante a C pode ser especificado em mais detalhes como segue:

```
controle enum {ack, nak, err};  
struct message {  
    controle de enum  
    tag;  
    dados de char unsigned;  
};
```

A linha que começa com a palavra-chave *enum* declara um tipo de enumeração chamado *controle* com três valores possíveis: um para cada tipo de mensagem usada. A mensagem estruturar-se contém dois campos: uma etiqueta do tipo de *controlo*, e um *dados* campo declarado como um caractere sem sinal (um byte).

REGAS DE PROCEDIMENTO

As regras de procedimento para o protocolo foram descritas informalmente da seguinte forma:

" 1. Se a recepção anterior estava livre de erros, a próxima mensagem no canal reverso carregará um reconhecimento positivo; se a recepção estava errada, carregará um confirmação negativa. "

" 2. Se a recepção anterior teve uma confirmação negativa, ou a anterior a recepção estava errada, retransmita a mensagem antiga; caso contrário, busque uma nova mensagem para transmissão."

Para formalizar essas regras, podemos usar diagramas de transição de estado, fluxogramas, dados algébricos

expressões ou descrições de forma de programa. Nos Capítulos 5 e 6, desenvolvemos um novo linguagem para descrever regras de procedimento como essas em modelos de validação de protocolo. Para o

por enquanto, porém, podemos usar fluxogramas simples, como o mostrado na Figura 2.3. Uma visão geral da linguagem do fluxograma é fornecida no Apêndice B.

24
ESTRUTURA DO PROTOCOLO
CAPÍTULO 2

começar
próximo: o
receber
ack: eu
próximo: o
err: eu
nak: o
nak: eu
ack: o
ack: o

Figura 2.3 - Protocolo de Lynch

A caixa rotulada *receber* simboliza um estado em que a recepção de uma nova mensagem do canal é aguardado. Dependendo do tipo de mensagem recebida, uma de três os caminhos de execução são então escolhidos. A caixa amassada representa o reconhecimento de uma mensagem sábio do tipo que corresponde ao seu rótulo. A caixa pontiaguda indica a transmissão de uma mensagem com o tipo correspondente.

A caixa rotulada a *seguir: o* indica uma ação interna para obter o próximo item de dados (character) a ser transferido. O item de dados é armazenado na variável *o*, que é usada na saída colocar operações. Por exemplo, *ack: o* envia *o* item de dados *o* com uma confirmação positiva da última mensagem recebida. Os dados de entrada são armazenados na variável *i*.

Como podemos esperar, existem alguns problemas com esta descrição que precisam ser considerado.

FLAWS DE DESIGN

Primeiro, temos o problema de que a transferência de dados em uma direção só pode continuar se os dados

transferência na outra direção também ocorre. Poderíamos tentar superar este problema fazendo com que os processos usem mensagens de preenchimento sempre que nenhum dado real for transferido.

Outro problema que deve ser resolvido antes que o protocolo possa ser usado é decidir como uma transmissão de dados deve ser iniciada ou concluída. As duas regras de procedimento especifique a transferência de dados normal, mas não os procedimentos de configuração e encerramento.

SEÇÃO 2.3
UM EXEMPLO
25

Podemos tentar iniciar a transferência de dados fazendo com que um dos dois processos envie um falso mensagem de erro. Observe que se ambas as partes têm permissão para iniciar o protocolo neste forma, é difícil trazer os dois processos em fase. Para encerrar a transferência quando os processos acabaram trocando mensagens de preenchimento, no entanto, requer con- extra mensagens trol.

Uma deficiência mais importante do protocolo é que uma operação essencial foi omitido da especificação. O receptor deve ser capaz de decidir se um item de dados que foi recebido corretamente, e temporariamente armazenado na variável *i*, deve ser aceitos (e, por exemplo, salvos em arquivo). Duplicatas recebidas corretamente de anteriores as mensagens recebidas indevidamente não devem ser aceitas novamente. Este problema parece não ter solução se quisermos manter as duas regras de procedimento listadas acima.

Considere o que pode acontecer se todas as mensagens recebidas corretamente forem aceitas, ou seja, dados anexados a mensagens *ack* e *nak* são aceitos, mas dados anexados a mensagens de *erro* sábios, não. A extensão parece plausível, mas infelizmente não resolve o problema. A seguinte sequência de execução, por exemplo, leva à aceitação de uma mensagem duplicada. Primeiro, o processo *A* inicia a transferência, enviando um deliberado mensagem de erro para *B*. Suponha que *A* tente transmitir os caracteres de *a* a *z*, e que *B* responde transmitindo os caracteres na ordem inversa, de *z* a *a*. Considerar em seguida, a sequência de eventos mostrada no diagrama de sequência de tempo da Figura 2.4. o duas linhas sólidas na figura rastreiam as execuções dos dois processos. O pontilhado linhas mostram transferências de mensagens bem-sucedidas. As linhas tracejadas mostram as transferências de mensagens

que são distorcidos pelo canal. Duas mensagens são distorcidas desta maneira: uma posse reconhecimento tiva de *um* para *B* e uma confirmação negativa a partir de *B* para *A*.

UMA
Próximo
aceitar 'z'
aceitar 'z'
Próximo
B
errar
.....
nak 'z'
.....
ack 'a' err
nak 'z' err
nak 'a'
.....
ack 'z'
.....
ack 'b'
.....
Próximo
aceitar 'a'

Figura 2.4 - Diagrama de sequência de tempo

No final da sequência, quando *A* recebe a última mensagem de *B*, ele não pode dizer se a mensagem é nova ou uma duplicata antiga. A mensagem *nak* que continha isso

Página 37

26

ESTRUTURA DO PROTOCOLO
CAPÍTULO 2

a informação foi corrompida. Na sequência de exemplo, *A* aceita erroneamente a mensagem sábio.

Deve-se notar que, embora o protocolo seja simples, é desproporcionalmente difícil para descobrir o erro. Assumir que o erro, se esquecido na fase de design, mais cedo ou mais tarde revelar-se na prática seria ingênuo. O erro só ocorre no evento raro em que dois erros de transmissão ocorrem em sequência. Como Lynch observou:

"Tais erros, embora raros, ocorrem, e sua raridade tornará extremamente difícil para detectar a falha no sistema. Este esquema inadequado funcionará 'quase' todos os Tempos."

O protocolo de exemplo é simples. A descrição informal é convincente e baseada somente com base nessa descrição, poucos duvidariam da correção do protocolo. Ainda a especificação está incompleta e qualquer implementação simples permite erros durante a troca de dados. Se alguma coisa, este exemplo deve nos convencer que, mesmo para o mais simples dos protocolos, uma boa disciplina de design e análise eficaz ferramentas de calibração são indispensáveis.

Nas próximas seções, retornaremos aos cinco elementos de uma especificação de protocolo definida na Seção 2.2, e considere os métodos de estruturação e critérios de design correspondentes que poderíamos usar. Primeiro, na Seção 2.4, consideraremos a estruturação do serviço especificações e as suposições explícitas que devem ser feitas sobre um protocolo meio Ambiente. Na Seção 2.5, examinamos o vocabulário do protocolo e o formato dos dados, e na Seção 2.6, falaremos com mais detalhes sobre as questões envolvidas no projeto de protocolos de procedimento col.

2.4 SERVIÇO E MEIO AMBIENTE

Para realizar uma tarefa de nível superior, como a transferência de arquivos, um protocolo deve realizar uma série de

funções de nível inferior, como sincronização e recuperação de erros. A realidade específica da execução de um serviço depende das suposições feitas sobre o meio ambiente no qual o protocolo deve ser executado. A recuperação de erros, por exemplo, deve corrigir para o comportamento assumido do meio de transmissão. Detalhes sobre os tipos de suposições que podem ser feitas sobre os canais de transmissão são fornecidas no Apêndice A e no Capítulo 3. Aqui nos concentraremos na estrutura das especificações de serviço adequadas.

O bom senso nos diz que se um problema for muito grande para resolver, devemos partioná-lo em subproblemas que são mais fáceis de resolver ou que já foram resolvidos antes. Programas, e em software de protocolo particular, é então mais convenientemente estruturado em *camadas*. Funções mais abstratas são definidas e implementadas em termos de constructs, onde cada camada esconde certas propriedades indesejáveis da comunicação

canal e o transforma em um meio mais idealizado.

Por exemplo, suponha que queremos implementar um protocolo de transmissão de dados que provides para a codificação de caracteres em tuplas de 7 bits cada, e para alguns rudimentares esquema de detecção de erros para proteger os bytes contra erros de transmissão, para instância pela adição de um bit de paridade a cada byte de 7 bits. Este protocolo então *pro oferece* dois serviços : codificação e detecção de erros. Podemos separar esses dois serviços

Página 38

SEÇÃO 2.4
SERVIÇO E MEIO AMBIENTE
27

em dois submódulos funcionais, um codificador e um módulo de paridade, e invocá-los sequencialmente. Na outra ponta da linha, haverá um decodificador e um verificador de paridade. Para transmissão full-duplex, podemos combinar convenientemente a função do codificador e decodificador em um módulo, digamos P_2 , e da mesma forma podemos combinar o par-somador e verificador em um único módulo P_1 .

P₂
P₁
P₁
P₂

Figura 2.5 - Construindo um canal virtual

A Figura 2.5 ilustra o princípio. O canal (a linha tracejada) é dividido em dois camadas. Com efeito, cada camada fornece um serviço diferente e implementa um separado protocolo. A primeira camada implementa o protocolo P_1 ; a segunda camada implementa o Protocolo P_2 . O formato de dados do protocolo P_2 é um byte de 7 bits. O formato de dados de o protocolo P_1 é um byte de 8 bits.

O protocolo P_2 não vê e não sabe sobre o oitavo bit que é adicionado ao seus bytes. A única coisa que importa é que o canal em que seus bytes de 7 bits viajam seja mais confiável do que o canal bruto no nível inferior. O protocolo P_1 fornece um vir-
canal real para o protocolo P_2 , mas é transparente para o protocolo P_2 . As duas chaves as palavras são *transparentes* e *virtuais*. "Transparente" é algo que existe, mas parece não para. "Virtual" é algo que parece existir, mas não existe.

Para o protocolo P_1 , qualquer formato de dados que é imposto pelo protocolo P_2 é invisível (transparente). No que diz respeito a P_1 , é uma sequência não interpretada de dados, de qual apenas o comprimento é conhecido. Da mesma forma, nem a camada de protocolo P_2 nem P_1 sabe alguma coisa sobre o formato imposto por possíveis camadas superiores da hierarquia (por exemplo, uma camada P_0), ou camadas inferiores (por exemplo, P_3).

P_n
 P_2
 P_1
 P_0
 P_1
 P_2
 P_n
envelope de nível n

Figura 2.6 - Envelopes de dados

Conforme mostrado na Figura 2.6, cada camada pode incluir os dados a serem transmitidos em um novo *envelope de dados*, que consiste em um cabeçalho e / ou trailer, antes de passá-lo para a próxima camada. O formato de dados original das camadas superiores não precisa nem mesmo ser preservado pelo camadas inferiores. Os dados podem ser divididos de forma diferente, em porções maiores ou menores ções, desde que o formato original possa ser restaurado pelo módulo de protocolo receptor.

O princípio do design hierárquico é bem conhecido na programação sequencial, mas é

Página 39

28

ESTRUTURA DO PROTOCOLO
CAPÍTULO 2

relativamente novo para sistemas distribuídos. As vantagens são claras:

Um design em camadas ajuda a indicar a estrutura lógica do protocolo separando tarefas de nível superior de detalhes de nível inferior.

Quando o protocolo deve ser estendido ou alterado, é mais fácil substituir um módulo do que reescrever todo o protocolo.

Em 1980, a International Standards Organization (ISO) reconheceu as vantagens de padronizando uma hierarquia de serviços de protocolo como um modelo de referência para protocolo designers. A recomendação ISO define sete camadas, conforme ilustrado na Figura 2.7.

transmissão média
Interface
camada de link de dados
camada de rede
camada de transporte
camada de sessão
camada de apresentação
camada de aplicação
camada física

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

7
6
5
4
3
2
1
7
6
5
4
3
2
1

Figura 2.7 - Modelo de referência ISO para interconexão de sistemas abertos

As camadas estão listadas abaixo com uma curta frase descriptiva explicando seu lugar em a hierarquia.

1. Camada *física* : transmissão de bits em um circuito físico
2. Camada de *enlace de dados* : detecção e recuperação de erros
3. Camada de *rede* : transferência e roteamento de dados transparentes
4. Camada de *transporte* : transferência de dados de nível superior de usuário para usuário
5. Camada de *sessão* : coordenação de interações em sessões de usuário
6. Camada de *apresentação* : interpretação da sintaxe no nível do usuário por exemplo, para criptografia ou compressão de dados
7. Camada de *aplicação* : ponto de entrada para processos de aplicação como correio eletrônico ou demônios de transferência de arquivos

A primeira camada contém todas as funções de protocolo que se aplicam à transmissão real de

Página 40

SEÇÃO 2.4
SERVIÇO E MEIO AMBIENTE
29

bits em uma conexão física. Ele especifica, por exemplo, se uma conexão é um fio de cobre, um cabo coaxial, um canal de rádio ou uma fibra óptica. O físico meio pode ser um canal *ponto a ponto*, dedicado à comunicação entre dois máquinas específicas, ou pode ser um canal de *transmissão* compartilhado, como a Universidade de Rede *Aloha* do Havaí ou um link Ethernet. Todas as propriedades relevantes de dados brutos canais e dos modems que são usados para conduzi-los (consulte o Apêndice A) são definidos aqui. A primeira camada também define a codificação de bits em, por exemplo, elétrico, ótimo cal ou sinais de rádio. Ele também define e padroniza os requisitos mecânicos de cabos, interruptores e conectores, incluindo atribuições de pinos e semelhantes. Os fisi protocolos da camada cal ocultam todos esses detalhes das camadas subsequentes e transformam a linha física em um link de dados rudimentar.

As próximas três camadas são as mais importantes. Sua função relativa é ilustrado na Figura 2.8. As caixas representam nós de rede ou hosts, os círculos representam processos de nível de usuário em execução nesses hosts, e as linhas representam a conexões vistas em três níveis diferentes de abstração.

A camada de enlace de dados usa o serviço fornecido pela camada física e transforma um link de dados brutos em um confiável, adicionando tratamento de erros. Ele conecta dois hosts, possivelmente de forma flexível, mas não necessariamente, que funciona como nós em uma rede (consulte a Figura 2.8). Isto

transmite os dados em blocos (quadros) e pode fornecer a multiplexação de dent fluxos de dados em um único link de dados. Pode fornecer um serviço de controle de fluxo para garantir que os frames só possam ser recebidos do link na ordem exata em que eles foram enviados, apesar dos erros de canal. Os protocolos que operam no nível do link de dados são conhecidos como *protocolos de nível de link*.

A camada de rede cuida das funções típicas da rede, como o endereçamento e encaminhamento de mensagens. Ele pode tentar evitar gargalos na rede usando adaptativos esquemas de roteamento, ou pode tentar reduzir o congestionamento na rede com controle de taxa métodos. A camada de rede fornece os meios para configurar e liberar o controle da rede

conexões, potencialmente abrangendo vários links de dados, ou saltos, através da rede, por exemplo, do nó *A* ao nó *B* na Figura 2.8.

q
p
.....

B

UMA

Camada de transporte

Camada de rede

Camada de link de dados

Figura 2.8 - Função relativa de três camadas

A camada de transporte liga processos ao nível do utilizador, tais como *p* e *q* na Figura 2.8,

Página 41

30

ESTRUTURA DO PROTOCOLO

CAPÍTULO 2

de forma transparente através de uma rede. Os protocolos da camada de rede e transporte são alguns tempos chamados *protocolos ponta a ponta*, e protocolos de enlace de dados são chamados *salto a salto*. Tanto a rede quanto a camada de transporte podem fornecer um serviço de controle de fluxo, que é agora chamado de ponta a ponta, em vez do controle de fluxo salto a salto que pode ser implementado na camada de enlace de dados. Pode, de fato, fazer uma grande diferença qual desses dois tipos de controle de fluxo são usados (consulte o Capítulo 4, Controle de taxa).

Cada camada na hierarquia define um serviço distinto e implementa um protocolo diferente col. O formato usado por qualquer camada específica é amplamente independente dos formatos usados pelas outras camadas. A camada de rede, por exemplo, envia *pacotes* de dados, o link de dados camada os converte em *quadros*, e a camada física os traduz em *byte* ou *bit* círculos. O receptor decodifica os dados brutos na camada 1, interpreta e exclui o estrutura de quadro na camada 2, para que a camada 3 possa reconhecer novamente a estrutura do pacote. O formato imposto pelas camadas inferiores é transparente para as camadas superiores.

Oficialmente, o modelo esboçado acima é chamado de ISO *Reference Model of Open Systems Interconnection*.¹ Porém, rapidamente se tornou conhecido como o modelo OSI da ISO.

As primeiras camadas do modelo OSI são as usadas com mais frequência. Um protocolo da camada 1 era padronizado pelo CCITT¹ como Recomendação X.21. A recomendação para o a segunda camada é amplamente baseada no protocolo HDLC que mencionamos anteriormente (consulte a Seção

2.5, enchimento de bits). As três primeiras camadas OSI juntas são definidas no CCITT Recomendação X.25. O protocolo X.25 define a interação de um computador, ou DTE para *equipamento de terminal de dados* na terminologia CCITT e um link de rede ou DCE para *dados equipamento de terminação de circuito*. A interação computador-a-computador não é definida até a quarta camada no modelo de referência OSI: a camada de transporte. Um conhecido trans-protocolo de camada de esporte é o protocolo de controle de transmissão (TCP) que foi padronizado pelo Departamento de Defesa dos EUA. O protocolo da camada de rede correspondente é chamado de Protocolo de Internet (IP).

1. Comité Consultatif International Télégraphique et Téléphonique.

Página 42

SEÇÃO 2.4
SERVIÇO E MEIO AMBIENTE

31

entidade

par

UMA

B

par

protocolo

.....

serviço

serviço

forneceu

primitivos

interface

Camada N

N + 1

Figura 2.9 - Camada de protocolo

As funções precisas realizadas em cada camada do modelo OSI e a definição de o protocolo X.25 é de pouco interesse para nós aqui (ver Notas Bibliográficas). Mais importante é o próprio método de estruturação. A estratificação de software é um princípio de design que pode ser poderoso quando usado corretamente, mas prejudica seu propósito quando levado para extremos.

Uma camada define um nível de abstração no protocolo, agrupando intimamente relacionados funções e separando-as das ortogonais. Ao desacoplar camadas, o futuro as alterações feitas em uma camada não precisam afetar o design das outras camadas. A escolha correta dos níveis de abstração exigidos depende necessariamente do protocolo específico sendo projetado.

Uma interface separa níveis distintos de abstração. Uma interface colocada corretamente é pequeno e bem definido. Uma interface mal posicionada causa complexidade desnecessária, isso causa duplicação de código e pode degradar o desempenho.

A Figura 2.9 ilustra os principais conceitos de uma técnica de estratificação. A função do protocolo N camada -ésimo formar uma entidade lógica. No modelo, eles são chamados de *pares entidades*. Por convenção, o limite vertical entre duas camadas adjacentes é chamado de *interface*, e o limite horizontal entre duas entidades em sistemas diferentes é chamado de *protocolo de mesmo nível*. Uma vez que os detalhes de implementação local das interfaces de camada

podem ser facilmente ocultados do ambiente, apenas os protocolos de pares devem ser padronizados entre sistemas.

A interface entre duas camadas adjacentes é definida como uma coleção de *acesso de serviço pontos* implementados pela camada inferior e disponíveis para a camada superior. A informação a ser trocada é formatada de forma incremental pelas várias camadas em *unidades de dados ou envelopes de dados*. Em sequência, a informação é passada do remetente para baixo de a camada mais alta usada, para a camada física, transmitida através do circuito físico real de sistema para sistema, e interpretado passo a passo enquanto é transmitido o protocolo hierarquia novamente para a camada mais alta usada pelo receptor.

Nesta estrutura, podemos reconhecer os primeiros dois elementos do protocolo de cinco partes especificação discutida nesta seção

o *serviço* a ser fornecido pelo protocolo, e as *suposições* feitas sobre seu ambiente

como especificações formais da interface superior e inferior de uma determinada camada de protocolo.

O serviço é fornecido aos protocolos da camada superior ou ao usuário da camada superior.

As suposições feitas são suposições sobre os serviços prestados pela camada inferior protocolos. Na camada de protocolo mais baixa, essas suposições dizem respeito ao serviço básico fornecido pelo meio de transmissão física, ou seja, uma fibra óptica, um fio de cobre ou um telégrafo de tocha.

A hierarquia de protocolo é um excelente exemplo da aplicação da disciplina de design.

Os problemas de design são separados uns dos outros e resolvidos de forma independente. O problema de controle de erro, recuperação de erro, endereçamento e roteamento, controle de fluxo, dados criptografia, etc., pode ser resolvida passo a passo de maneira disciplinada. De um designer ponto de vista, porém, não é predeterminado que todo problema de design é sempre melhor subdividido conforme sugerido na Figura 2.7. As especificidades do sistema de protocolo e o ambiente em que é executado determina como um problema de design pode ser melhor decomposto em problemas menores.

2.5 VOCABULÁRIO E FORMATO

Primeiro, olhamos, em um nível bastante baixo de abstração, em alguma formatação de protocolo métodos. Esses formatos devem estar subjacentes a todas as estruturas de nível superior, por exemplo, estruturas que são usadas para codificar o vocabulário da mensagem do protocolo. Os três principais os métodos de formatação são:

Orientado a bits

Orientado para o caráter

Orientado para contagem de bytes

BIT ORIENTADO

Um protocolo orientado a bits transmite dados como um fluxo de bits. Para permitir que um receptor reconhecer onde uma mensagem (um quadro) começa e termina no fluxo de bits, um pequeno conjunto de padrões de bits exclusivos, ou *sinalizadores*, são usados. Claro, esses padrões de bits podem fazer parte do

dados do usuário também, então algo deve ser feito para garantir que eles sejam sempre interpretados devidamente. Se uma bandeira de enquadramento, por exemplo, é definida como uma série de seis bits um entre eles

em zeros, 01111110, uma série de seis adjacentes nos dados do usuário deve ser interceptada.

Isso pode ser feito inserindo um zero extra após cada série de cinco unidades no usuário dados.

0
dados do usuário
bandeira de enquadramento
bandeira de enquadramento
 $0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0$
 $0\ 1\ 1\ 1\ 1\ 1\ 1\ 0$
 $0\ 1\ 1\ 1\ 1\ 1\ 1\ 0$
Figura 2.10 - Enchimento de bits

Figura 2.10 - Enchimento de bits

O receptor pode agora detectar corretamente a estrutura imposta pelos sinalizadores no bit fluxo inspecionando o primeiro bit após cada série de cinco unidades: se for um zero, deve ser excluído, caso contrário, o padrão que está sendo verificado deve fazer parte de um delimitador de quadro verdadeiro. este *bit* técnica de *enchimento* é usada no protocolo ISO da camada 2 (consulte a Seção 2.6) para *alto nível Data Link Control*, HDLC, que por sua vez é baseado no *Synchronous Data Link* da IBM Protocolo de *controle*, SDLC. Uma vez que a estrutura básica de sinalização de baixo nível está no lugar, ela pode ser usado para apoiar estruturas de nível superior.

CARÁTER ORIENTADO

Em um protocolo orientado a caracteres, alguma estrutura mínima é imposta no fluxo de bits.

Se o número de bits por caractere for fixado em n bits (normalmente 7 ou 8), todas as comunicações ocorrem em múltiplos de n bits. Essas unidades de dados são então usadas para codificar ambos

dados do usuário e códigos de controle. Exemplos de códigos de controle são o início do texto ASCII 2 *STX* e mensagens *ETX* de fim de texto que podem servir como delimitadores e podem ser usadas para coloque os dados do usuário.

DLE
DLE
DLE
DLE
delimitador
delimitador
dados do usuário
STX, DLE, STX, ETX, ...
ETX

Figura 2.11 - Recheio de caracteres

Novamente, se os dados brutos são transmitidos (por exemplo, código de objeto binário), deve-se ter cuidado

considerando que os delimitadores não ocorrem accidentalmente nos dados do usuário. No *Bisync* da IBM protocolo, por exemplo, cada caractere de controle, como *STX* e *ETX*, é precedido por um código extra, o caractere de escape do link de dados *DLE*. Se alguma mensagem de controle, como *STX*, *ETX*, ou mesmo o próprio *DLE*, ocorre literalmente nos dados do usuário, é precedido por um caractere *DLE* extra. O código *DLE* é interpretado pelo receptor como um controle código que desativa qualquer significado especial do primeiro caractere que o segue. O receptor exclui o primeiro código *DLE* que vê no fluxo de caracteres e passa adiante o seguinte caractere não interpretado. Somente se o significado especial de um *STX* ou *ETX* o código não é suprimido por um caractere *DLE* anterior se for interpretado como um delimitador.

2. Código padrão americano para intercâmbio de informações.

CAPÍTULO 2
A técnica é chamada de *enchimento de personagens*.

A Figura 2.11 mostra onde os códigos DLE seriam inseridos em um fluxo que consiste em quatro caracteres de controle subsequentes nos dados do usuário.

BYTE-COUNT ORIENTADO

BIT-ORIENTADO
Os sinalizadores de um protocolo orientado a bits e os caracteres de controle de um protocolo orientado a caracteres.

protocolo são usados para estruturar um fluxo de dados brutos em fragmentos maiores. Uma razão para tal estruturação é para indicar a um receptor onde um fluxo de dados começa e termina. No

Para os protocolos orientados para contagem de bytes, um método ligeiramente diferente é escolhido. Em um lugar conhecido após a mensagem de controle *STX*, o remetente inclui o número preciso de bytes (char- atores) que a mensagem contém. Uma mensagem *ETX* agora é supérflua e a tecnologia técnicas como recheio de bits ou recheio de caracteres não são mais necessários. A maioria dos protocolos em uso hoje são deste tipo. Um exemplo específico é o *Digital Data Communication Protocol de Mensagem deação*, DDCMP.

CABEÇALHOS E TRAILERS

Com os métodos básicos de estruturação que discutimos acima, mais sistemáticos métodos de formatação de dados de nível superior podem ser criados. Até agora, assumimos silenciosamente

a ausência de erros de transmissão. Se um campo de contagem de bytes estiver distorcido, ou um caractere *DLE*

se se perde, essas técnicas falham. Na ausência de uma detecção de erro e erro estratégico de recuperação, portanto, as técnicas são de pouca utilidade.

estratégia de recuperação, portanto, as técnicas são de pouca utilidade. Como veremos em mais detalhes no Capítulo 3, os esquemas de detecção de erro requerem transmissão geração de informações redundantes, normalmente na forma de uma *soma de verificação*. Se controle de fluxo

técnicas são adicionadas, por exemplo, para detectar perda ou reordenação de quadros de texto, um *o campo do número de sequência* é anexado. Se mais de um tipo de mensagem for usado, nós além disso, deve incluir uma indicação do tipo de mensagem que está sendo transferida. E então, se estivermos transmitindo informações redundantes de qualquer maneira, podemos também adicionar outros

dados potencialmente úteis, como o nome do remetente ou a prioridade da mensagem.

Todo esse overhead é mais convenientemente agrupado em estruturas separadas que encapsulam atrasar os dados do usuário: uma mera mensagem de controle *STX*, portanto, se expande em uma estrutura de *cabeçalho*.

e da mesma forma, o *ETX* simples se transforma em um *trailer de mensagem* composta

E da mesma forma, o *ETX* simples se transforma em um *trailer* de mensagem composta. Por razões óbvias, as contagens de bytes são normalmente colocadas nos cabeçalhos das mensagens e as somas são colocadas no trailer. O formato da mensagem pode então ser definido como um conjunto de três elementos:

formato $\equiv \{cabecalho\; dados\; trailer\}$

O cabeçalho e o trailer novamente definem subconjuntos ordenados de campos de controle, que podem ser:

definido como segue:

cabecalho = {*tipo*, *destino*, *número de sequência*, *contagem*}

cabeçalho = {*ípo*, *destino*, *número de sequência*}
trailer = {*checksum*, *endereço de retorno*}

O comprimento do campo de dados é definido pelo último campo do cabeçalho. O destino e o endereço de retorno pode ser novamente definido por subestruturas.

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

ETX
reboque
dados do usuário
STX
cabeçalho
tipo; dest; seqnr; contagem
checksum; endereço de devolução
Figura 2.12 - Formato da mensagem

O campo *tipo* pode ser usado para identificar as mensagens que compõem o vocabulário do protocolo lar. Dependendo da estrutura particular do vocabulário do protocolo, este campo pode ser ainda mais refinado.

2.6 REGRAS DE PROCEDIMENTO

Até este ponto, enfatizamos a similaridade da tarefa de design de protocolo e mau desenvolvimento de software. É hora de examinar uma das diferenças. Um importante aspecto do problema de projeto de protocolo é que as regras de procedimento são interpretadas com atualmente por uma série de processos de interação. O efeito de cada nova regra que adicionamos o conjunto é frequentemente muito maior do que pode ser previsto. Muitas intercalações diferentes no tempo

da interpretação destas regras pelos vários processos será possível. Precisamente devido a essa simultaneidade, um comportamento de protocolo nem sempre é reproduzível. Enganar-Vendo-nos da correção de um design, precisamos de algo melhor do que o informal raciocínio. A ferramenta mais popular para raciocinar sobre protocolos, infelizmente, é o diagrama de seqüência de tempo, como o usado na Figura 2.4. Para ter certeza, a sequência de tempo diagrama é conveniente para relatar um único erro conhecido. Mas é lamentavelmente inadequate para raciocinar sobre o funcionamento de um protocolo em geral. Para permitir isso nós deve, pelo menos, ser capaz de expressar o comportamento de forma inequívoca em um formal conveniente

notação. Tabelas de transição, ou máquinas de estados finitos formais (ver Capítulo 8), para exemplo, pode ser usado para este fim. Além disso, devemos ser capazes de expressar arbitragem requisitos de correção complexos sobre os comportamentos que especificamos (ver Capítulos 5 e 6).

Não existe uma metodologia geral que possa garantir *a priori* o desenho de um unambiguo conjunto de regras de procedimento (discutiremos isso com mais detalhes no Capítulo 10). Tem, no entanto, ferramentas com as quais podemos, mesmo automaticamente, verificar a consistência lógica das regras (ver Capítulo 11) e a observância dos requisitos de correção.

E, claro, há bom senso e boas práticas de engenharia que podem nos ajude a manter as regras do protocolo gerenciáveis. Vemos alguns desses problemas na próxima seção.

2.7 DESIGN DE PROTOCOLO ESTRUTURADO

O desenho do protocolo abrange uma ampla gama de questões. Alguns desses problemas estão bem Entendido; outros, estamos apenas começando a entender. O projeto do protocolo é parcialmente um problema de engenharia que pode ser resolvido pela aplicação de tecnologia bem conhecida niques. Na camada física da hierarquia ISO, por exemplo, sabemos precisamente

36

ESTRUTURA DO PROTOCOLO
CAPÍTULO 2

quais são os comportamentos característicos dos diferentes tipos de portadores de informação, como rapidamente podemos transmitir dados sobre eles e qual será a taxa média de erro de bits resultante. Existem várias técnicas para codificação de dados binários em sinais analógicos que podem ser realizada por vários meios de comunicação, e existem técnicas bem conhecidas para sincronizar transmissores e receptores neste nível. Não temos que reinventar e revalidar essas técnicas para cada novo protocolo e, de fato, podem ser consideradas tão padronizadas dard que não precisamos discuti-los neste livro. Para o leitor interessado, os detalhes estão incluídos no Apêndice A.

Muito mais acima na hierarquia de protocolo, enfrentamos problemas de design de rede: roteamento dados através de redes, o dimensionamento preciso de estruturas de rede, o inter-conexão de várias redes com gateways, e o desenvolvimento de sistemas de alto nível disciplinas para controle e prevenção de congestionamento. Entre este alto *visão de rede* de nível e *visão de baixo* nível de códigos de transmissão e suportes de dados existe um grande território desconhecido, onde existem poucas técnicas que podem nos ajudar através do processo de design. Ainda existe uma gama de controles de erros bem conhecidos e técnicas de controle de fluxo que podem ser usadas para construir links de dados confiáveis, mas isso é apenas

onde começa o verdadeiro problema de projeto de protocolo: o problema real de conceber inequívoca conjuntos completos de regras para a troca de informações em um sistema distribuído.

Antes que essa "área cinzenta" do projeto de protocolo possa se tornar uma verdadeira disciplina de engenharia,

deve ser estabelecido quais são as principais ferramentas de design, quais regras devem ser seguidas reduzidos e que erros devem ser evitados.

O desenvolvimento de uma nova disciplina de engenharia geralmente ocorre em duas fases. Na primeira fase, a nova tecnologia é explorada, e os designers buscam ferramentas que restringem o menos possível na exploração de suas possibilidades. Se as dificuldades forem encontrados, a capacidade das ferramentas é *expandida* para permitir ao usuário lidar com o crescente conjunto de problemas. A tendência nesta primeira fase, então, é remover as restrições em vez de impô-los.

Na segunda fase, após um melhor entendimento da natureza dos problemas desenvolve, um novo conjunto de ferramentas aparece. Essas ferramentas impõem deliberadamente um cuidado

conjunto selecionado de *restrições* sobre o usuário. Essas restrições se destinam a impor um disciplina de design que se baseia na história de erros, chamados coletivamente "experiência," desde a primeira fase de desenvolvimento. No projeto do protocolo, ainda estamos esperando

para fazer a transição para a segunda fase de desenvolvimento. Abaixo discutimos alguns conceitos centrais na nova disciplina de design para protocolos que está surgindo.

Um designer irá aderir à disciplina apenas se em troca, comprovada e reproduzível, um produto mais confiável pode ser obtido. Abaixo, damos uma visão geral do que é provável para se tornar parte de um conjunto geral de princípios de design de som, o que nos permitirá entrar na segunda fase de desenvolvimento no campo da engenharia de protocolo. Isto é importante reconhecer que todas essas notas são variações de dois temas comuns: sim-flexibilidade e modularidade.

SEÇÃO 2.7
DESIGN DE PROTOCOLO ESTRUTURADO

37

SIMPLICIDADE - O CASO PARA PROTOCOLOS LEVES

Um protocolo bem estruturado pode ser construído a partir de um pequeno número de protocolos bem projetados e peças bem compreendidas. Cada peça desempenha uma função e a executa bem. Para compreender o funcionamento do protocolo, deve ser suficiente compreender o funcionamento do as peças a partir das quais é construído e a forma como interagem. Protocolos que são projetados desta forma são mais fáceis de entender e implementar eficientemente e são mais prováveis de serem verificáveis e sustentáveis. Um peso leve

protocolo é simples, robusto e eficiente. O caso para protocolos leves diretamente apóia o argumento de que eficiência e verificabilidade não são ortogonais, mas comple-preocupações mentais.

MODULARIDADE - UMA HIERARQUIA DE FUNÇÕES

Um protocolo que executa uma função complexa pode ser construído a partir de peças menores que interagir de forma simples e bem definida. Cada peça menor é um proto-peso leve col que pode ser desenvolvido, verificado, implementado e mantido separadamente. Orthog-as funções onais não são misturadas; eles são projetados como entidades independentes. O indivíduo módulos duplos não fazem suposições sobre o trabalho um do outro, ou mesmo a presença.

O controle de erros e o controle de fluxo, por exemplo, são funções ortogonais. Eles são os melhores resolvido por módulos leves separados que são completamente inconscientes uns dos outros existência. Eles não fazem suposições sobre o fluxo de dados, exceto o que é estritamente necessários para desempenhar sua função. Um esquema de correção de erros não deve fazer suposições sobre o sistema operacional, endereços físicos, métodos de codificação de dados, velocidades de linha ou hora do dia. Essas preocupações, caso existam, são colocadas em outro módulos, especificamente otimizados para esse fim. A estrutura de protocolo resultante é aberto, extensível e reorganizável sem afetar o funcionamento adequado do componentes individuais.

PROTOCOLOS BEM FORMADOS

Um *formado poços* protocolo não está *especificada em excesso*, isto é, que não contém qualquer inatingível código capaz ou não executável. Um protocolo bem formado também não é *subespecificado* ou incompleto. Um protocolo especificado de forma incompleta, por exemplo, pode causar *não especificado recepções* durante sua execução. Uma recepção não especificada ocorre se uma mensagem chega quando o receptor não espera ou não pode responder a isso.

Um protocolo bem formado é *limitado*: ele não pode ultrapassar os limites do sistema conhecidos, como a capacidade limitada das filas de mensagens.

Um protocolo bem formado é *auto-estabilizante*. Se um erro transitório alterar arbitrariamente o estado do protocolo, um protocolo de autoestabilização sempre retorna a um estado desejável dentro de um número finito de transições e retoma a operação normal. Da mesma forma, se tal protocolo é iniciado em um estado de sistema arbitrário, sempre atinge um dos estados pretendidos dentro do tempo finito.

Um protocolo bem formado, finalmente, é *auto-adaptável*. Ele pode adaptar, por exemplo, a taxa em quais dados são enviados para a taxa em que os links de dados podem transferi-los, e para a taxa em que o receptor pode consumi-los. Um método de *controle de taxa*, por exemplo, pode ser

usado para alterar a velocidade de uma transmissão de dados ou seu volume.

ROBUSTEZ

Como Polybius (Capítulo 1) observou,

"*são principalmente ocorrências inesperadas que requerem consideração e ajuda instantânea.*"

Não é difícil projetar protocolos que funcionem em circunstâncias normais. É o inesperado que os desafia. Isso significa que o protocolo deve estar preparado para lidar apropriadamente com cada ação viável e com cada sequência possível de ações sob todas as condições possíveis. O protocolo deve fazer apenas suposições mínimas sobre seu ambiente para evitar dependências de recursos específicos que podem mudar.

Muitos protocolos de nível de link que foram projetados na década de 1970, por exemplo, não mais funcionam corretamente se forem usados em linhas de dados de velocidade muito alta (no Gigabits / s alcance). Um design robusto aumenta automaticamente com a nova tecnologia, sem a necessidade de mudanças fundamentais. A melhor forma de robustez, então, não é *um design exagerado* por adicionar funcionalidade para novas condições previstas, mas *design mínimo*, removendo suposições não essenciais que poderiam impedir a adaptação a condições imprevistas.

CONSISTÊNCIA

Existem algumas maneiras comuns e temidas pelas quais os protocolos podem falhar. Listamos três dos mais importantes.

Deadlocks - estados em que nenhuma execução de protocolo adicional é possível, para instância porque todos os processos de protocolo estão esperando por condições que nunca podem

ser preenchidas.

Livelocks - sequências de execução que podem ser repetidas indefinidamente, muitas vezes sem sempre fazendo progresso efetivo.

Términos impróprios - a conclusão de uma execução de protocolo sem satisfazer condições de rescisão adequadas.

Em geral, a observância desses critérios não pode ser verificada por uma inspeção manual da especificação do protocolo. Ferramentas mais poderosas são necessárias para prevenir ou detectar eles. Essas ferramentas são discutidas na Parte III.

2.8 DEZ REGRAS DE PROJETO

Os princípios discutidos acima levam a dez regras básicas de projeto de protocolo.

1. Certifique-se de que o problema está bem definido. Todos os critérios de design, requisitos e restrições, devem ser enumeradas antes de um projeto ser iniciado.

2. Defina o serviço a ser executado em cada nível de abstração antes de decidir quais estruturas devem ser usadas para realizar esses serviços (o *que* vem antes de *como*).

3. Projete a funcionalidade *externa* antes da funcionalidade *interna*. Considere primeiro o solução como uma caixa preta e decidir como ela deve interagir com seu ambiente.

Em seguida, decida como a caixa preta pode ser organizada internamente. Provavelmente consiste em caixas pretas menores que podem ser refinadas de maneira semelhante.

4. Mantenha a simplicidade. Protocolos sofisticados são mais problemáticos do que os simples; eles são mais difíceis de implementar, mais difíceis de verificar e geralmente menos eficientes. Existem poucos realmente complexos

Página 50

CAPÍTULO 2
EXERCÍCIOS
39

problemas no desenho do protocolo. Problemas que parecem complexos costumam ser simples problemas amontoados. Nossa trabalho como designers é identificar os problemas mais simples lems, separe-os e resolva-os individualmente.

5. Não conecte o que é independente. Assuntos ortogonais separados.

6. Não introduza o que é imaterial. Não restrinja o que é irrelevante. Um bem o design é "aberto", ou seja, facilmente extensível. Um bom design resolve uma classe de problemas em vez de uma única instância.

7. Antes de implementar um projeto, construa um protótipo de alto nível (Capítulos 5 e 6) e verifique se os critérios de projeto foram atendidos (Capítulos 11 a 14).

8. Implementar o projeto, medir seu desempenho e, se necessário, otimizá-lo.

9. Verifique se a implementação otimizada final é equivalente à de alto nível projeto que foi verificado (Capítulo 9).

10. Não pule as Regras 1 a 7.

A regra violada com mais frequência, claramente, é a Regra 10.

2.9 RESUMO

Um protocolo inclui mais do que um acordo sobre o significado dos sinais para os dados.

No mínimo, o protocolo deve incluir acordos sobre o uso de informações de controle, que é necessário para padronizar o uso do próprio canal.

Para ser completa, a definição de um protocolo deve incluir os cinco elementos principais listados na Seção 2.2. As falhas de protocolo são frequentemente causadas por suposições ocultas sobre eventos ou sobre as possíveis sequências de eventos. É responsabilidade do protocol designer para tornar essas suposições explícitas. Novamente: não é suficiente se um correto a interpretação da especificação é simplesmente possível. É necessário que nenhum a interpretação é possível.

As principais técnicas de estruturação de protocolo são as camadas de software de controle e o estruturação de dados. O modelo OSI é dado como um exemplo dessa abordagem. Cuidado, não é uma receita. Da mesma forma, as dez regras de design são diretrizes, não comandos mentos. Uma abordagem estruturada e sólida para o projeto de regras de procedimento consistentes deve ser sempre uma disciplina auto-imposta.

Nos próximos dois capítulos, cobriremos primeiro os fundamentos do projeto de protocolo, a tecnologia conhecida

técnicas para construir canais confiáveis a partir de canais não confiáveis. O restante do livro é dedicado ao estudo do próprio problema de projeto de protocolo. Não discute

problemas de design de rede, nem a codificação específica ou uso dos padrões de protocolo que estão em uso hoje. Em vez disso, nosso objetivo é discutir como os protocolos podem ser projetados usando uma disciplina simples baseada nas regras fornecidas acima.

EXERCÍCIOS

2-1. Identifique os cinco elementos de protocolo da Seção 2.2 para o telégrafo da tocha de Políbio, discutido no Capítulo 1. Liste pelo menos três casos de incompletude no protocolo.

Página 51

40

ESTRUTURA DO PROTOCOLO

CAPÍTULO 2

2-2. Dê uma descrição informal das regras de procedimento de um protocolo que gerencia os dados transferência de um servidor de arquivos para uma impressora (Seção 2.1). Certifique-se de que o protocolo pode recuperar quando a impressora ficar sem papel ou for desligada.

2-3. Explique quais são os equivalentes de mensagens de controle e dados em uma chamada telefônica. Escrever abaixo uma especificação de protocolo completa (Seção 2.2) para uma chamada telefônica, levando em consideração todos os sinais possíveis e condições de exceção. Considere o caso em que duas pessoas tentam chamem uns aos outros simultaneamente e considerem as melhores regras de procedimento para rediscar após um sinal ocupado.

2-4. Estenda o protocolo de Lynch para evitar o erro de duplicação e mostre com um argumento rigoroso mente que a versão revisada funciona.

2-5. Explique por que uma contagem de bytes é mais convenientemente colocada em um cabeçalho de mensagem (Seção 2.5).

2-6. Explique a diferença entre enchimento de bits e enchimento de caracteres.

2-7. Calcule o comprimento ideal para uma bandeira de enquadramento em um protocolo orientado a bits. Observe que um séries mais longas de uns no sinalizador de enquadramento reduzem a probabilidade de sua ocorrência no dados do usuário e, portanto, a sobrecarga no número de bits recheados, à custa de um maior sobrecarga na própria bandeira de enquadramento. Assuma dados de usuário aleatórios. (Veja Bertsekas e Gal-lager [1987, p. 78-79]).

2-8. Em sua linguagem de programação favorita, escreva uma função que execute *STX - ETX* frame e enchimento de caracteres em um fluxo de bytes arbitrário. Fornece o recebimento correspondente funcionar e testá-lo.

NOTAS BIBLIOGRÁFICAS

Que as mensagens de controle são vitais para uma operação confiável das linhas de comunicação foi já conhecido na época dos telégrafos pré-elétricos. Até mesmo o telégrafo da tocha tinha um *início de mensagem de texto*, e a maioria dos sistemas posteriores tinham pelo menos códigos de controle especiais para

repita e espere. Os mesmos sinais de controle são definidos em quase todos os links de dados em uso hoje. Hubbard [1965], relata mais um tipo de mensagem de controle, desenvolvido por "um inventor francês anônimo" para um dos primeiros sistemas telegráficos eletrostáticos. Ele sugere gerada usando a carga estática da linha telegráfica para acender uma pequena quantidade de arma pó na estação de recepção para *acordar* um atendente dormindo.

O sistema descrito por Marland [1964] ganha o prêmio para as melhores mensagens de controle já inventado. Ele notou um sistema telegráfico que foi descrito na *Mecânica Revista* de 11 de junho de 1825 (Vol. IV, p.148). Neste sistema, os choques eletrostáticos são administrados diretamente ao operador. E, se isso não for suficiente, sugere uma solução original para o problema de um operador de telégrafo sonolento:

"*Deixe o primeiro choque passar por seus cotovelos, então ele estará bem acordado para atender o segundo.*"

Excelentes introduções aos problemas de projeto de protocolo podem ser encontradas em Pouzin e Zimmerman [1978] e em Merlin [1979]. O formalismo para descrever protocolos como uma linguagem abstrata, com vocabulário, gramática formal e sintaxe foi introduzida em Puzman e Porizek [1980].

Talvez a maior importância do artigo de Lynch [1968] é que ele desencadeou um

Página 52

CAPÍTULO 2
NOTAS BIBLIOGRÁFICAS

41

famoso artigo de Bartlett, Scantlebury e Wilkinson do National Physical Laboratório da Inglaterra, definindo um dos protocolos mais simples e conhecidos em uso hoje: *o protocolo de bit alternado*. Discutimos isso no Capítulo 4.

Os símbolos usados no fluxograma na Figura 2.3 são da especificação CCITT linguagem SDL. A linguagem está rapidamente ganhando popularidade como método de especificação

para protocolos de comunicação. Para uma visão geral, consulte, por exemplo, Rockstrom e Saracco [1982] e SDL [1987]. A definição oficial da linguagem SDL está no CCITT [1988]. O fluxograma " linguagem " usado aqui é mais completamente descrito no Apêndice B. A melhor referência à linguagem C, referida brevemente na Seção 2.3, é Ker-nighan e Ritchie [1978, 1988].

As principais ideias de programação estruturada e camadas de software vêm de EW Dijkstra [1968a, 1968b, 1969a, 1969b, 1972, 1976] e N. Wirth [1971, 1974]. Eles estão intimamente relacionados com a técnica de design por refinamento passo a passo Wirth [1971], consulte

também Gouda [1983]. Que o princípio do refinamento gradual era conhecido muito antes o design do programa tornou-se um problema é ilustrado pela seguinte citação da EF Moore.

"*Uma maneira de descrever o que os engenheiros fazem ao projetar máquinas reais é dizer que eles começam com uma descrição geral de uma máquina e a dividem sucessivamente em máquinas cada vez menores, até que os relés individuais ou tubos de vácuo sejam finalmente alcançado.*" (Moore [1956])

As ideias sobre o design do protocolo expressas aqui também são inspiradas por discussões com muitos outros, principalmente Jon Bentley, John Chaves, Peter van Eijk, Rob Pike e Chris Vissers. A importância do conceito de serviço no projeto do protocolo é eloquentemente explicado em Vissers e Logrippo [1985].

O termo *auto-estabilização* também foi cunhado por Dijkstra, ver, por exemplo, [1974, 1986], ver também Kruijer [1979]. Lamport discutiu a autoestabilização em vários artigos, Lamport [1984, 1986]. Multari escreveu sua tese sobre protocolos auto-estabilizadores, Multari [1989]. Outro trabalho pioneiro nesta área é feito na Universidade do Texas em Austin por MG Gouda [1987] e na Cornell University por GM Brown [1989].

O estudo de protocolos de peso leve foi iniciado na década de 1970 por um grupo de pesquisa em o Laboratório de Computação da Universidade de Cambridge, envolvido com o projeto de a Cambridge Ring Network, por exemplo, Needham e Herbert [1982], e um pouco mais tarde por um grupo da AT&T Bell Labs, incluindo Sandy Fraser, Greg Chesson e Bill Marshall, envolvido com o design do hardware e software para o Datakit → interruptor.

O termo protocolo de *peso leve* foi cunhado pelo grupo de Cambridge, que também desenvolveu o primeiro contendor sério nesta classe: o *protocolo de fluxo de bytes* que é usado no Cambridge Ring. O trabalho no Bell Labs levou, em última análise, ao design de o padrão Universal Receiver Protocol (URP), Fraser e Marshall [1989], e seus sucessores dos protocolos de comutação de pacotes PSP e MSP.

Uma descrição completa do modelo OSI pode ser encontrada na ISO [1979]. O X.25

Página 53

42

ESTRUTURA DO PROTOCOLO

CAPÍTULO 2

protocolo, finalmente, está documentado no CCITT [1977] e explicado em, por exemplo, Lindgren [1987] e Stallings et al. [1988]. Mais sobre problemas de rede de dados pode ser encontrado em Tanenbaum [1981, 1988] ou em Stallings [1985].

Página 54

CONTROLE DE ERROS 3

43 Introdução 3.1

44 Modelo de Erro 3.2

46 Tipos de Erros de Transmissão 3.3

46 Redundância 3.4

47 Tipos de Códigos 3.5

48 Verificação de Paridade 3.6

48 Correção de Erro 3.7

52 Código de Bloco Linear A 3.8

56 Verificações de Redundância Ciclífica 3.9

63 Soma de Verificação Aritmética 3.10

64 Resumo 3.11

64 exercícios

65 Notas Bibliográficas

3.1 INTRODUÇÃO

O número de erros causados pela transmissão de dados é normalmente ordens de magnitude

maior do que o número de erros causados por falhas de hardware em um sistema de computador tem. A probabilidade de erro de bit para circuitos internos é geralmente inferior a 10^{-15} . Em um opto-link de fibra óptica, a probabilidade média de erros é de aproximadamente 10^{-9} . Ou seja, no média, um em cada 10^9 bits transmitidos (ou processados) é distorcido, seis ordens de magnitude mais do que para circuitos de hardware. Da mesma forma, em um cabo coaxial, a probabilidade de erros de bits é de aproximadamente 10^{-6} . Para uma linha telefônica comutada, os números são ainda maiores, entre 10^{-4} e 10^{-5} . A diferença de magnitude entre uma probabilidade de erro de 10^{-15} e um de 10^{-4} não deve ser subestimado. Uma taxa de erro de bits de 10^{-15} em uma linha de transmissão seria ser incomensuravelmente pequeno nas taxas de transmissão de hoje. A uma taxa de 9600 bits por segundo lugar, causaria um único erro de bit a cada 3.303 anos de operação contínua. Na mesma taxa de dados, uma taxa de erro de bits de 10^{-4} causa um erro de bit, em média, uma vez por segundo. Dependendo das características da linha e da rede, os dados transmitidos podem ser reordenados, distorcidos ou excluídos e, ocasionalmente, linhas ruidosas podem até inserir novos dados em transmissões. Os erros introduzidos nas transmissões de dados, obviamente, não são inteiramente imprevisível ou inexplicável. Os erros têm duas causas principais, discutidas em mais detalhes no Apêndice A:

Distorção linear dos dados originais, por exemplo, causada por limites de largura de banda itações do canal de dados brutos

Distorção não linear causada por ecos, diafonia, ruído branco e

43

Página 55

44
CONTROLE DE ERROS
CAPÍTULO 3

ruído de impulso

O efeito dessas distorções pode ser remediado, até certo ponto, com isolamento de cabo, filtros de compensação e compensação de hardware. Os erros que permanecem devem ser apanhados em software pelo protocolo de comunicação.

Existem várias maneiras pelas quais as características de erro de uma linha de dados podem ser expresso. O primeiro, e o mais importante, é a taxa média de erro de bits de longo prazo. Mas, uma vez que esta é apenas uma média, existem dois outros fatores em uso:

A porcentagem de tempo em que a taxa média de erro de bit não excede um determinado valor limiar

A porcentagem de segundos sem erros

As duas últimas medidas fornecem uma indicação da qualidade geral de uma linha ou rede.

Para o projeto de um método de controle de erro, geralmente usa-se apenas o bit médio taxa de erro, como uma indicação do desempenho esperado.

Nenhum método de controle de erros pode ser esperado para detectar todos os erros que podem ocorrer. Podemos, no entanto, exigir que um esquema de controle de erro aumente a confiabilidade das transmissões, de preferência ao nível de confiabilidade da operação autônoma de um computador.

Um problema frequentemente esquecido é que um esquema de controle de erro eficaz deve corresponder ao

características de erro dos canais a serem usados. Se um canal produz apenas *inserção* erros, não seria sensato criar um protocolo que proteja contra *exclusões*. Similmente, se um canal produz erros independentes de bit único com uma probabilidade relativamente baixa

bilidade, mesmo o esquema de paridade mais simples (Seção 3.6) pode facilmente superar o que mais métodos sofisticados de controle de erros. E, finalmente, se a taxa de erro do canal é

já inferior ao dos equipamentos periféricos, a inclusão de *qualquer* controle de erros esquema degrada o desempenho desnecessariamente e pode até acabar diminuindo bastante do que aumentar a confiabilidade do protocolo.

3.2 MODELO DE ERRO

Para um canal com uma taxa média de erro de bit de longo prazo de p , é teoricamente mais conveniente se assumirmos uma distribuição aleatória dos erros ao longo da sequência de bits transmitido. A probabilidade de n erros de bit subsequentes em uma mensagem é simplesmente p^n , e a probabilidade de um ou mais erros de bit em uma mensagem de n bits é $1 - (1 - p)^n$. Embora isso ignore o efeito do ruído de impulso, nos dá um bom ponto de partida para o estudo das disciplinas de controle de erros. O modelo formal para um canal deste tipo é o *canal discreto sem memória* mostrado na Figura 3.1.

O canal é denominado discreto porque reconhece apenas um número finito de níveis de sinal. É chamado de sem memória porque a probabilidade de um erro é assumida como ser independente de todas as ocorrências de erros anteriores. Uma vez que assumimos que a probabilidade de um erro de bit é a mesma para ambos os elementos de sinal, o canal na Figura 3.1 também é chamado de *canal simétrico*.

Muitas variações diferentes para este modelo básico são possíveis, acompanhadas por

SEÇÃO 3.2
MODELO DE ERRO

45
0
1
 $q = 1 - p$
0
1
 $q = 1 - p$
 p
 p

Figura 3.1 - Canal discreto sem memória

cálculos cada vez mais complexos para prever o efeito dos métodos de controle de erro. Em um canal assimétrico, por exemplo, a probabilidade de um erro pode depender do sinal valor sendo transmitido. A distribuição de probabilidades de erro também pode ser definida como um processo com memória: se os últimos n bits transmitidos estiverem errados, é muito provável que os próximos também estarão errados. É difícil capturar esse comportamento em uma previsão modelo ativo. O modelo de erro fornecido pelo canal simétrico binário prevê que a probabilidade de uma série de pelo menos n transmissões contíguas de bits livres de erros, chamadas um "intervalo livre de erros" (EFI), é igual a

$$Pr(EFI \in [n]) = (1 - b)^n, \quad n \geq 0 \quad (3.1)$$

onde b é a taxa média de erro de bit de longo prazo.

A probabilidade diminui linearmente com a duração do intervalo. Da mesma forma, a probabilidade de que a duração de uma rajada excede n bits diminui linearmente com n . Para expressar que a probabilidade de um intervalo livre de erros diminui exponencialmente com o seu duração, podemos substituir a fórmula (3.1) por uma distribuição de Poisson:

$$Pr(EFI \in [n]) = e^{-b} \frac{b^n}{n!}, \quad n \geq 0 \quad (3.2)$$

A melhor maneira de verificar a precisão dessa previsão é, obviamente, compará-la contra dados empíricos. Esses estudos indicam de fato que a fórmula (3.2) prevê o erro intervalos livres melhores do que (3.1). Uma correspondência ainda melhor pode ser encontrada se um fator de correção

é adicionado a (3.2). Obtemos, assim, a seguinte aproximação, que se deve a Benoit Mandelbrot (ver notas bibliográficas):

$$Pr(EFI \in [n]) = \frac{e^{-b} b^n}{n!} \left[1 + \frac{b}{2n+2} + \frac{(b^2-1)}{2(n+1)(2n+1)} \right]$$

$$= e^{-b} b^n \frac{n!}{(n+b)!} \quad (3.3)$$

O parâmetro a determina o quanto sério o efeito de agrupamento é previsto.

Quando a é zero, a fórmula (3.3) se reduz à distribuição de Poisson em (3.2). Para não zero a , a probabilidade de intervalos livres de erros mais longos diminui mais do que a probabilidade de intervalos mais curtos. Com o crescimento de a , esse efeito se torna mais pronunciado. Do claro, se as características de erro forem independentes da taxa de bits, elas podem ser expressas em segundos.

Com diferentes valores de parâmetros a e b , funções do tipo (3.2) e (3.3) podem ser usadas para prever a duração dos intervalos livres de erros e a duração das rajadas independentemente diently. Usaremos esse método no Capítulo 7. Para o restante deste capítulo, entretanto, nos restringiremos ao modelo de um canal simétrico binário.

3.3 TIPOS DE ERROS DE TRANSMISSÃO

Muitos tipos diferentes de erros podem ocorrer nas linhas de dados. A transmissão mais importante erros de sessão aparecem como dados

Inserção

Eliminação

Duplicação

Distorção

Reordenando

Os dados inseridos e excluídos podem ser causados pela perda temporária de sincronização entre o remetente e o destinatário. Erros de exclusão também podem ser causados artificialmente por inade- disciplinas de controle de fluxo adequadas. Um receptor, por exemplo, pode ficar sem buffers para conter mensagens recebidas e perder mensagens que não pode armazenar. A duplicação de dados pode até ser realizada intencionalmente, por exemplo, por um remetente que implementa uma retransmissão protocolo. Se os dados forem roteados por meio de redes, potencialmente por muitas rotas diferentes, também pode ocorrer reordenamento de dados.

Problemas de sequenciamento de dados, como exclusão, duplicação e reordenamento, são resolvidos com esquemas de controle de fluxo adequados (Capítulo 4). Mas, em todos os casos em que a distorção de dados

ou a inserção pode ocorrer, não importa qual seja a causa, precisamos de métodos para verificar a consistência dos dados. Discutimos esses métodos abaixo.

3.4 REDUNDÂNCIA

Um método de detecção de erro só pode funcionar aumentando a redundância de mensagens de alguma forma bem definida. Ao verificar a consistência de uma mensagem, o receptor pode em seguida, avalie a confiabilidade das informações que ele contém. Além de detectar erros de transmissão, porém, o receptor também deve ser capaz de corrigir os erros. Lá

Existem duas maneiras de fazer isso:

Controle de erro de encaminhamento

Controle de erro de feedback

Se a redundância for grande o suficiente, o receptor pode ser capaz de reconstruir uma mensagem sábio do sinal distorcido. Este método é denominado controle de erros *direto*. Os códigos de transmissão correspondentes são chamados de *códigos de correção de erros*.

A alternativa é usar um *código de detecção de erros* e providenciar a retransmissão de mensagens corrompidas. Isso é chamado de controle de erro de *feedback*. Um pedido de retransmissão pode ser uma confirmação negativa explícita enviada do receptor para o remetente ou, quando a probabilidade de erro é suficientemente baixa, pode estar implícita na ausência de uma confirmação de dados recebidos corretamente. Nesse caso, o receptor simplesmente ignora quaisquer dados corrompidos e espera que o remetente atinja o tempo limite aguardando o reconhecimento e retransmitir a mensagem.

O objetivo do controle de erros é reduzir a taxa de erros do canal. Nem todos os erros podem ser detectado, então sempre há uma *taxa de erro residual*. Suponha que a probabilidade de um erro de transmissão em uma mensagem é p e que o método de controle de erro captura um fracion f de todos os erros. Para um dado f e p , podemos então calcular a taxa de erro residual

$p \cdot (1 - \frac{1}{n}f)$ e nos convencer de que é, por exemplo, da ordem de 10

$\frac{1}{n}$ ou menos.

Se a probabilidade p for muito próxima de zero, um código de correção de erros geralmente é desaconselhável:

apenas retarda a transferência de dados. Se, por outro lado, p se aproxima de um, a esquema de retransmissão seria uma má escolha: quase todas as mensagens, incluindo o retransmitidos, seriam atingidos. Claro que existem exceções a essas regras. Se p é pequeno, e o custo de retransmissão alto, um esquema de controle de erro direto ainda pode ser rentável. Em outros casos, uma combinação de controle de erro direto e de feedback pode ser um bom compromisso: o receptor corrige os erros que ocorrem com frequência e pergunta o remetente para a retransmissão de mensagens que contêm erros menos frequentes.

Na próxima seção, veremos primeiramente os principais tipos de correção de erros e detecção de códigos que foram desenvolvidos.

3.5 TIPOS DE CÓDIGOS

Os dois tipos básicos de códigos são

Códigos de bloqueio

Códigos de convolução

Em um *código de bloco*, todas as palavras de código têm o mesmo comprimento, e a codificação para cada possibilidade

a mensagem de dados pode ser definida estaticamente. Em um *código de convolução*, a palavra de código produzida depende da própria mensagem de dados e de um determinado número de

mensagens codificadas: o codificador muda seu estado a cada mensagem processada. O comprimento das palavras de código é geralmente constante. Podemos ainda distinguir entre

Códigos Lineares

Códigos cíclicos

Códigos sistemáticos

Os códigos de bloco lineares e cíclicos são os códigos mais comumente usados na comunicação de dados protocolos de instalação. Em um código linear, cada combinação linear de palavras de código válidas (como

uma soma módulo-2) produz outra palavra de código válida. Um código cíclico é um código em que cada deslocamento cíclico de uma palavra-código válida também produz uma palavra-código válida. Uma *sistemática*

código, finalmente, é um código em que cada palavra de código inclui os bits de dados do original mensagem final inalterada, seguida ou precedida por um grupo separado de bits de verificação.

Em todos os casos, as palavras de código são mais longas do que as palavras de dados nas quais se baseiam. E se

o número de bits originais é d e o número de bits adicionais é e , a proporção

$d / (d + e)$ é chamado de *taxa de código*. Melhorar a qualidade de um código muitas vezes significa aumentar

reduzindo sua redundância e, assim, diminuindo a taxa de código. Para reduzir a taxa de erro do canal por um fator de 5,10, por exemplo, pode exigir um código com um taxa de código de 0,5 ou menos.

O restante deste capítulo é organizado da seguinte forma. Seção 3.6, dá uma visão geral introdução aos códigos de verificação de paridade. Na Seção 3.7, estendemos o código para um método de controle de erros. A seção 3.8 discute um código de bloco linear simples, devido a R. Hamming, que oferece proteção contra erros de bit único independentes. Seção 3.9 concentra-se em códigos de bloco cíclicos, usando a popular verificação de redundância cílica como um exemplo. Seção 3.10 discute uma alternativa simples para uma verificação de redundância cílica: o

método de soma de verificação aritmética.

3.6 VERIFICAÇÃO DE PARIDADE

Se a probabilidade de vários erros de bits por mensagem for suficientemente baixa, todos os erros o controle necessário em um canal simétrico binário é um código de verificação de paridade. Para cada mensagem

sábio, adicionamos um único bit que faz a soma do módulo 2 dos bits dessa mensagem igual a um. A sobrecarga é de apenas um bit por mensagem. Se houver um único bit, incluindo o bit de verificação é distorcido pelo canal em que a paridade no receptor sai errada e o erro de transmissão pode ser detectado.

Se definirmos $q = 1 - p$, a probabilidade de uma transmissão livre de erros de n bits de mensagem mais um bit de paridade é $q^{(n+1)}$, e a probabilidade de um único erro de bit em $n + 1$ bits transmitidos é a probabilidade binomial $(n + 1) \cdot p$

q^n . Sob essas premissas (ou seja, um canal sem memória), a taxa de erro residual de uma verificação de paridade de um bit é $1 - q^{(n+1)} - (n + 1) \cdot p$

q^n

Para $n = 15$ e $p = 10^{-4}$

isso deixa uma taxa de erro residual da ordem de 10^{-6}

por mensagem

sábio, ou cerca de 10

por bit.

0.0

0.25

0.50

0.75

1.0

0.0

0.25

0.5

0.75

1.0

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

Taxa de erro / palavra (pontilhada)

Taxa de erro residual (sólido)

(a) - Taxa de erro / bit (p)

0.0

0.1

0.2

0.3

0.0

0.25

0.5

0.75

1.0

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

SEÇÃO 3.7
CORREÇÃO DE ERROS
49

bits.

A taxa de código de um código de correção de erros é em geral menor do que a de um mero código de detecção de erros. Em princípio, portanto, a correção de erros direta é apenas considerado útil quando a comunicação de mensagens de controle de um receptor voltar para um remetente é difícil. A dificuldade pode ser

Um atraso de transmissão muito longo

A ausência de um canal de retorno

Uma alta taxa de erro de bits

Um bom exemplo do primeiro problema é a comunicação entre uma sonda espacial e seu centro de controle remoto na terra. Um sinal de controle, por exemplo, para liberar uma câmera obturador ou para fazer um ajuste de curso, pode levar vários minutos para chegar ao distante sonda. Pode não haver tempo suficiente para repetir um sinal no caso de uma transmissão erro. O sinal passa ou é perdido para sempre.

O segundo problema pode existir em sistemas de transmissão de radiodifusão com um remetente e vários receptores. Um exemplo mais perverso, mas muito real, é quando as sequências de transmissão são armazenadas em um dispositivo de backup e reproduzidas posteriormente. No

momento da transmissão, os dados originais podem não estar mais disponíveis para retransmissão.

O terceiro problema, uma alta taxa de erro de bits, pode significar que mesmo a probabilidade de que um a solicitação de *retransmissão* pode ser recebida corretamente é inaceitavelmente baixa. Em todos os três casos, adicionar redundância a uma mensagem pode ser a única maneira de evitar o irrevogável perda de algumas das mensagens transmitidas.

Mesmo uma única verificação de paridade por palavra de código pode ser facilmente estendida a partir de um único erro

deteção de código em um código de correção de erro único. Cada sequência de sete bits é primeiro estendido com um único bit de paridade que torna o número de bits um em cada seqüência mesmo. O bit de paridade é chamado de verificação de redundância *longitudinal* ou bit LRC. Adicionando uma sequência extra de oito bits a cada série de n códigos, podemos incluir um verificação de redundância *vertical*, ou bit VRC, para o conjunto de bits que ocupam o mesmo bit posição em cada sequência. Por exemplo, com codificação ASCII, para $n = 4$:

LRC

$D = 1000100 \ 0$

$A = 1000001 \ 0$

$T = 1010100 \ 1$

$A = 1000001 \ 0$

$0010000 \ 1 \ VRC$

Um bit VRC com defeito codifica o número da coluna e um bit LRC com defeito o correspondente número da linha para um bit de erro, de modo que qualquer erro de bit único por série de 40 bits transmitidos

realmente pode ser corrigido. Usamos 12 bits de verificação para proteger uma sequência de 28 dados bits, que corresponde a uma *taxa de código* de $28 / (12 + 28) = 0,7$.

Agora, vamos esquecer as verificações de paridade e desenvolver um código de correção de erros a partir de coçar, arranhão. O exemplo a seguir é baseado em JH van Lint [1971].

50
CONTROLE DE ERROS
CAPÍTULO 3
EXEMPLO

Suponha que gostaríamos de padronizar a geração de números aleatórios. o método que escolhemos é nomear uma pessoa imparcial para ser nosso padrão aleatório gerador de números. Ele executa esta tarefa lançando uma moeda padrão A vezes por segundo. Os resultados são transmitidos a todos os quatro cantos da Terra por meio de um padrão canal simétrico binário que opera a uma velocidade máxima de $2A$ bps (bits por

segundo), com uma taxa de erro de bit de 2,10

-□2.

Claramente, o resultado de cada lance da moeda padrão pode ser codificado em um bit de informação mação. A transmissão dos bits brutos pode ser feita a uma taxa de A bps, mas causa o receptores para obter uma média de 2% dos números que se desviam do padrão " aleatório dard. "

A primeira coisa que podemos pensar para resolver este problema pode ser transmitir cada resultado não uma, mas duas vezes, ou seja, codificamos cada resultado em dois bits em vez de um. o receptores agora são capazes de detectar a maioria dos erros de transmissão, mas claramente não há tempo

esquerda para corrigi-los. Um código de correção de erros está em ordem. Agora podemos tentar codificar dois lançamentos da moeda, como um par, em quatro bits de dados, usando a Tabela 3.1.

Tabela 3.1 - Codificação

Código de Resultado

hh
0000
o
1001
ht
0111
tt
1110

{ {
{
{
{
{
{
{

Os receptores usam uma tabela diferente, mostrada na Tabela 3.2, que lhes permite decodificar qualquer palavra de código recebida como uma das quatro mensagens possíveis.

Tabela 3.2 - Decodificação

Códigos Válidos
Resultado

0000 1000 0100 0010
hh
1001 0001 1101 1011
o
0111 1111 0011 0101
ht
1110 0110 1010 1100
tt

{ {
{
{
{
{
{
{

O código é resistente a erros de bit único nos primeiros três bits de cada palavra de código enviada. A primeira coluna da Tabela 3.2 contém a palavra-código original enviada e as três seguintes colunas contêm os códigos que resultam de um erro no primeiro, segundo ou terceiro bit, respectivamente. Erros de vários bits, ou um único erro no quarto bit, ainda levam ao recepção de um número aleatório não padronizado. Quais são as chances de isso acontecer? UMA o código é recebido corretamente se, com probabilidade q^4 , não houver erros ou, com probabilidade $3p$

.

q^3 , tem exatamente um erro entre os três primeiros bits.

Página 62

SEÇÃO 3.7
CORREÇÃO DE ERROS
51

$q^4 + 3p$

.

q^3

= 0.9788

Começamos com uma taxa de erro de 2% por bit, ou seja, 4% de chance de pelo menos um erro em uma série de dois bits. A taxa de erro é reduzida para $1 - 0,9788 = 0,0212$ ou 2,12% para dois bits subsequentes. Usamos quatro bits para codificar dois flips, dando um código taxa de 0.5. Nós desperdiçamos doze das dezesseis palavras de código possíveis para fazer isso redução na taxa de erro, mas ainda estamos transmitindo os códigos tão rápido quanto os resultados são produzidos por nosso gerador de números aleatórios padrão.

Sem alterar a velocidade de sinalização efetiva, ou a taxa de código, poderíamos aumentar o quantidade de desperdício ainda mais, usando oito bits para codificar uma série de quatro bits de dados. Para

selecionar as 24 palavras de código válidas necessárias do intervalo de 28 disponíveis, podemos novamente

tentativa de reduzir a possibilidade de que uma palavra válida seja transformada em outra por erros de transmissão.

DISTÂNCIA DE BARULHO

A diferença entre duas palavras de código pode ser definida como o número de bits em que eles diferem. A diferença mínima entre duas palavras em um código é chamada de *Distância de Hamming*. Se conseguirmos encontrar um código com uma distância de Hamming de n , qualquer combinação de erros de até $n - 1$ bit pode ser detectada. Melhor ainda, qualquer combinação de até $(n - 1)/2$ erros por palavra de código podem ser corrigidos se dissermos ao receptor para interpretar cada palavra de código inválida como a palavra de código válida mais próxima. O receptor vai adivinhar errado para números maiores de erros de bits, mas se a probabilidade destes for suficiente suficientemente baixa, a taxa de erro geral do canal ainda pode ser reduzida.

Formalmente, este método é chamado de *decodificação de máxima verossimilhança*, ou também *mais próximo*

decodificação de vizinho. Aumentando a distância de Hamming, escolhendo cada vez mais palavras de código, devemos ser capazes de aumentar a confiabilidade de um código tanto quanto quer.

A seguinte questão surge agora: isso é verdade para qualquer taxa de transmissão e para algum canal? A resposta pode ser encontrada em um artigo publicado por Claude Shannon em 1948, *A Mathematical Theory of Communication*. Supondo uma largura de banda limitada canal com ruído branco, Shannon provou que apenas para taxas de transmissão de até um certo limite pode a taxa de erro do canal ser arbitrariamente pequena (Apêndice A).

O limite é chamado de *capacidade do canal*.

O argumento de Shannon é baseado na observação de que a quantidade de informações transferido por um canal nunca pode exceder a entropia da fonte de informação nem a entropia do próprio canal causada pelo ruído. Abaixo desse limite é teoricamente sempre possível obter informações confiáveis do canal. Informalmente, Shannon descobriram que quando a relação sinal-ruído fica menor, cada sinal deve durar mais para fazer com que se destaque do ruído, o que por sua vez reduz a velocidade máxima de sinalização que pode ser obtido.

O esforço exigido na codificação dos dados, no entanto, normalmente proíbe a operação de um canal perto do limite teórico. Para uma linha telefônica, por exemplo, com largura de banda

Página 63

52
CONTROLE DE ERROS
CAPÍTULO 3

de 3,1 kHz e uma relação sinal-ruído de 30 dB (ou seja, 8:1), o limite de Shannon é cerca de 30 Kbit / s, que é muito mais do que a taxa máxima usada na prática.

3.8 A CÓDIGO DE BLOCO LINEAR

Vimos na última seção que a redundância de um código determina seu poder de detectar e corrigir erros de transmissão. A redundância pode ser definida como o número de bits usados sobre o mínimo necessário para codificar uma mensagem de forma inequívoca. Para codificar uma de n mensagens igualmente prováveis, por exemplo, requer $\log_2 n$ bits, arredondado para cima

para o valor inteiro mais próximo. Chamamos essa quantidade de m .

$$m = \lfloor \log_2 n \rfloor$$

Podemos proteger esses m bits adicionando c bits de verificação e escolhendo os n códigos usados dos $2^{(m+c)}$ códigos agora disponíveis de tal forma que cada combinação de dois códigos diferem em tantos bits quanto possível.

Tabela 3.3 - Proteção de Paridade

c	0
m	0,00
$m / (m + c)$	1
	0
	0,00
	2
	1
	0,50
	3
	4
	0,57
	4
	11
	0,73
	5
	26
	0,84
	6
	57
	0,90
	7 120
	0,94
	8 247
	0,97

Para poder corrigir todos os erros de bit único, sabemos que precisamos de uma distância de Hamming de pelo menos três entre as palavras de código, mas quantos bits de verificação isso minimamente custa? Para cada palavra de código de $m + c$ bits, existem precisamente códigos $m + c$ que podem resultar

de erros de bit único. Para cada palavra na faixa de 2^m códigos de dados possíveis, portanto, precisamos de $m + c + 1$ palavras para protegê-lo contra erros de bit único. O total o número de palavras no código é $(m + c + 1) \cdot 2^m$, que deve ser igual ao $2^{(m+c)}$ palavras com as quais começamos.

Configuração

$$(m + c + 1) \cdot 2^m$$

$$= 2^{(m+c)}$$

dá

$$m + c + 1 = 2^c$$

permitindo-nos calcular o número mínimo de bits de verificação c para qualquer número de bits de dados m . Para $m = 8$, encontramos um mínimo de $c = 3$. 66 ou 4 bits de verificação por mensagem,

dando uma taxa de código de $8 / (8 + 4) = 0,66$.

verifique os bits c . Os primeiros oito números estão listados na Tabela 3.3, com os correspondentes taxas máximas de código. O mesmo efeito é ilustrado para até 16 checkbits na Figura 3.3.

Com uma boa aproximação, o número de bits de dados que podem ser protegidos aumenta exponencialmente com o número de bits de verificação disponíveis.

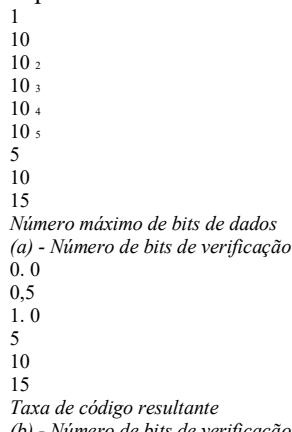


Figura 3.3 - Proteção de Paridade

CÓDIGO DE HAMMING

Um exemplo de código que realiza essa proteção é um código desenvolvido por R. Hamming. No código de Hamming, incluído como um exemplo de correção de erro único perfeito código no artigo de Shannon de 1948, os bits em uma palavra-código são numerados de 1 a $m + c$. O i -ésimo bit de verificação é colocado na posição do bit 2^i para $1 \leq i \leq \log_2(m + c)$.

Os bits de verificação foram colocados na palavra de código de forma que a soma das posições de bit eles ocupam pontos no bit errôneo para qualquer erro de bit único. Pegar um erro de bit único os bits de verificação são usados como bits de paridade.

Quando uma posição de bit é escrita como uma soma de potências de dois, por exemplo, $(1 + 2 + 4)$, também aponta para os bits de verificação que o cobrem. O bit de dados $7 = (1 + 2 + 4)$, por exemplo, é contado nos três bits de verificação nas posições 1, 2 e 4. Um erro de bit único que muda o sétimo bit de dados altera a paridade precisamente dessas três verificações. O receptor pode, portanto, determinar qual bit está com erro somando as posições dos bits de todos os bits de verificação que sinalizaram um erro de paridade. Um erro que muda, por exemplo, o segundo bit afeta apenas aquele único bit e também pode ser corrigido.

<i>o</i>	
<i>o</i>	
<i>o</i>	
<i>o</i>	
.....	
.	
.	
.	
.	
.	
.	
.	
<i>l</i>	
<i>l</i>	
<i>o</i>	
<i>o</i>	
<i>p¹</i>	<i>p²</i>
<i>p⁴</i>	
<i>p⁸</i>	
.	
.	
.	
.	
.	
.	
.	
.	
.....	
.....	

EU:

0

1

-
-

1
1

* * *

•


```

:
:
:
:
:
:
1 + 2 + 4 =
7
dados
enviei
recebido
Verifica
bit de erro

```

Figura 3.4 - Correção de um erro de transmissão

Como exemplo, o código de caractere ASCII para a letra D é 1000100 . Figura 3.4 mostra como os dados e bits de paridade são colocados em um código de Hamming. Se uma transmissão erro muda a posição do bit 7 de 0 para 1, o código chega como o código ASCII para um L 1001100 . Mas, os três primeiros bits de paridade transmitidos agora diferem dos valores que receptor pode calcular e revelar o sétimo bit com defeito.

É claro que não é realmente relevante para o código como tal em que ordem os bits de código são colocado em uma palavra de código. Reorganizando os bits, por exemplo, cada binário Hamming o código pode ser alterado para um código sistemático ou um código cíclico.

REPRESENTAÇÃO DE MATRIZ

Existe um método conveniente para definir os códigos de verificação de paridade de bloco linear na matriz

Formato. Como exemplo, considere um código com três bits de dados, chamados D1 , D2 e D3 , e três bits de verificação, C4 , C5 e C6 . Podemos definir os três bits de verificação como o soma módulo-2 dos bits de dados, por exemplo da seguinte forma:

$$C_4 = D_1 + D_2$$

$$C_5 = D_1$$

$$+ D_3$$

$$C_6 =$$

$$D_2 + D_3$$

Essas três funções podem ser definidas na forma de matriz da seguinte forma:

{

|

□□C 6

C 5

C 4]

|

|

=

|

|

□□0

1

1

1

0

1

1

1

0]

|

|

□□D 3

D 2

D 1]

|

]

Dando um passo adiante, também podemos expressar as três funções de definição da seguinte forma baixos:

$$\begin{aligned} D1 + D2 + \\ + C4 \\ = 0 \\ D1 + \\ + D3 \\ + C5 \\ = 0 \\ D2 + D3 \\ + C6 = 0 \end{aligned}$$

Página 66

SEÇÃO 3.8
UM CÓDIGO DE BLOCO LINEAR

55

o que leva à seguinte representação de matriz.

{

|

□□0

1

1

1

0

1

1

1

0

0

0

1

0

1

0

0

1

0

0

1

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

0

1

0

ele pode encontrar um resultado diferente de zero s .

$$H \cdot (ct + E) = s$$

O vetor s é denominado *síndrome* . Neste código, cada soma módulo-2 de código válido palavras produz outra palavra de código válida. Portanto, se o vetor de erro E acontecer a corresponder a qualquer palavra de código válida, a síndrome é zero e o erro não é detectado.

BURSTS

Até agora, nos concentramos principalmente na detecção e correção de bit único erros de transmissão, supondo que os erros sejam mutuamente independentes. Na prática, sabemos que os erros de transmissão não são mutuamente independentes: eles tendem a ocorrer *rajadas* .

Picos de ruído, ecos e cross-talk afetam séries de bits subsequentes sempre que eles ocorrer. Para uma linha telefônica comutada, a probabilidade média de um erro de bit pode ser 10

- 5 . Mas, se um bit em uma mensagem arbitrária foi distorcido, a probabilidade de que o próximo bit também está errado pode ser tão alto quanto 0. 5. O resultado é que relativamente poucas mensagens

sábios são distorcidos em geral, mas aqueles que são distorcidos são mais gravemente feridos.

Claramente, é bastante inútil desenvolver um esquema de controle de erros que pode perfeitamente detecte e corrija um erro raro de bit único se os erros de rajada forem mais comuns.

Embora a definição do código de Hamming seja relativamente simples, é surpreendentemente difícil para estendê-lo em um código que pode corrigir vários erros de bit por palavra. Para garantir o detecção de números pares de erros de bits por palavra de código, o código de Hamming pode ser estendido com uma única verificação de paridade longitudinal. Uma solução mais geral, no entanto, é mais difícil.

CÓDIGO INTERLEAVING

Um método geral para combater os erros de rajada é a *intercalação de código* , uma intercalação método é alterar a ordem em que os bits são transmitidos através do canal.

Suponha que temos mensagens de n bits cada, protegidas contra erros de bit único.

Página 67

56

CONTROLE DE ERROS
CAPÍTULO 3

Supondo ainda que o tráfego não seja interativo, podemos interceptar erros de burst até um comprimento de k bits, armazenando em buffer cada bloco de k mensagens subsequentes, colocando-as em um

matriz de $k \cdot n$ bits e transmitir os bits nesta matriz coluna por coluna de fileira por fileira. Na extremidade do receptor, a matriz original é restaurada coluna por coluna e leia linha por linha. Um erro de rajada de comprimento k ou menos causa apenas um único bit erro por linha e pode ser corrigido corretamente.

Os verdadeiros códigos de correção de erro duplo, não baseados em esquemas de intercalação, foram publicados pela primeira vez

lished por Hocquenghem [1959] e Bose e Ray-Chaudhuri [1960]. Esses códigos, conhecidos coletivamente como códigos BCH, requerem uma justificativa substancialmente mais teórica do que pode ser dado aqui. Uma outra generalização dos códigos BCH é conhecida como o Código Reed-Solomon. Encontrou aplicação, por exemplo, na codificação digital de som em discos compactos.

Em um estudo realizado na IBM em 1964, verificou-se que em quase todos os casos o feedback o controle de erros pode ser superior ao controle de erros direto, tanto no rendimento quanto no resí-taxas de erro duplas. Portanto, continuamos com uma discussão sobre um código de bloco cíclico que é usado para controle de erro de *feedback* .

3.9 VERIFICAÇÕES DE REDUNDÂNCIA CÍCLICA

O método de verificação de redundância cíclica, ou CRC, também é baseado na adição de séries de bits de verificação para palavras de código. Neste caso, os bits adicionados garantem que, na ausência de erros de transmissão, a palavra de código mais os bits de verificação são divisíveis por um determinado fator.

O método de divisão específico e o fator usado determinam a faixa de transmissão erros que podem ser detectados. Para simplificar a manipulação algébrica de palavras de código, nós pode definir um mapeamento de códigos em polinômios. Uma sequência de N bits pode então ser interpretado como um polinômio de grau máximo $N - 1$:

$i = 0$

©

b_{eu}^{N-1}

.

x_i

onde cada b_i leva o valor do bit na posição i na sequência, com os bits num-reposta da direita para a esquerda. A palavra de código 10011, por exemplo, define polinômio $x^4 + x + 1$

Estamos trabalhando em um sistema binário, então todas as operações, incluindo divisão e multiplicação, são definidos como módulo-2. A adição do Módulo-2 é definida da seguinte forma:

$$0 + 0 = 0 - 0 = 0$$

$$0 + 1 = 0 - 1 = 1$$

$$1 + 0 = 1 - 0 = 1$$

$$1 + 1 = 1 - 1 = 0$$

Em adições mais longas não há transporte e em subtrações não há empréstimo. Em polinomial, portanto, para qualquer i , temos $x^i + x^i = 0$, uma vez que $1 + 1 = 0$ e $0 + 0 = 0$. Para multiplicar duas palavras de código, podemos multiplicar o polinômio correspondente als.

Página 68

SEÇÃO 3.9
VERIFICAÇÕES DE REDUNDÂNCIA CÍCLICA

57

Tabela 3.4 - Um Código Cíclico

Polinômio de palavra de dados multiplicado por
Produz
Palavra de código

0	0	0
0		
$x + 1$		
0		
0	0	0
0	0	1
1		
$x + 1$		
$x + 1$		
0	0	1
0	1	0
x		
$x + 1$		
$x^2 + x$		
0	1	1
0	1	1
$x + 1$		
$x + 1$		
$x^2 + 1$		
0	1	0
1	0	0
x^2		
$x + 1$		
$x^3 + x^2$		
1	1	0
1	0	1
$x^2 + 1$		
$x + 1$		
$x^3 + x^2 + x + 1$		
1	1	1
1	1	0
$x^2 + x$		
$x + 1$		
$x^3 + x$		
1	0	1
1	1	0
1	1	1

$$\begin{array}{r}
 x^2 + x + 1 \\
 x + 1 \\
 x^3 + 1 \\
 1 \ 0 \ 0 \ 1
 \end{array}$$

{
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}

Por exemplo,

$$(x^4 + x + 1) \cdot (x^3 + x^2) = x^7 + x^6 + x^4 + x^2$$

Podemos usar esse mecanismo facilmente para definir um código. Considere, por exemplo, um código com três bits de dados. Codificamos os dados em quatro bits, multiplicando cada palavra de dados com o polinômio $x + 1$, conforme mostrado na Tabela 3.4. O código resultante é uma verificação de paridade

código com uma taxa de código de 3/4. É também um código cílico, mas não sistemático.

Se pudermos adicionar, subtrair e multiplicar polinômios, podemos, é claro, também dividi-los.

Vamos tentar dividir o polinômio $x^7 + x^6 + x^4 + x^2$ por um fator $x^5 + x^2 + 1$.

$$\begin{array}{r}
 x^5 + x^2 + 1 / x^7 + x^6 + x^4 + x^2 \\
 \underline{-} x^2 + x \\
 x^6 + 0 + x^3 \\
 x^7 + 0 + x^4 + 0 + x^2 \\
 \underline{-} x \\
 x^6 + 0 + x^3 + x
 \end{array}$$

Para tornar o polinômio original divisível pelo fator $x^5 + x^2 + 1$, poderíamos simplesmente subtraí-lo residual dele. Mas, embora o receptor seja capaz de detectar erros de transmissão, não seria capaz de recuperar a mensagem original do palavrão de código. Melhor é acrescentar o residual como uma soma de verificação. O fator usado para gerar

um checksum é chamado de *polinômio gerador* do código.

Agora, primeiro multiplicamos o polinômio da mensagem por um fator igual ao mais alto grau do polinômio gerador, neste caso x^5 , para abrir espaço para a soma de verificação. Simplesmente significa deslocar os bits na palavra de código cinco casas à esquerda. Então nós dividimos o polinômio de mensagem pelo polinômio gerador e subtraímos o residual.

Uma vez que o CRC é um código linear, cada padrão de erro E deve ser igual a algum código válido T . Para um código conhecido, esta propriedade pode ser usada para calcular o erro residual. Se P é o polinômio da mensagem e G um polinômio gerador de grau r , o residual R tem grau $r - 1$ e é definido como o restante de

G

P

.

x r

— — —

%

CONTROLE DE ERROS

CAPÍTULO 3

A palavra de código T a ser transmitida é

$T = \square P$

.

$x r \square - \square R$

Um erro de transmissão em vigor adiciona um polinômio de erro E ao código transmitido.

Quando o receptor divide o código pelo polinômio gerador, ele encontra o termo de erro

G

$T + \square E$

$$\frac{\overline{G}}{\overline{T}} =$$

\overline{G}^-

E

$$\frac{\overline{G}^-}{\overline{E}} =$$

E

Um erro de transmissão só é não detectado se o restante da divisão do erro o padrão E pelo polinômio gerador G é zero. Se E for diferente de zero e de um grau inferior do que G , a divisão sempre deixa um resto. Isso significa que todos os erros de burst de comprimento r e menos são detectados perfeitamente. Observe cuidadosamente que isso é independente do

posição da explosão dentro da palavra código T . O padrão de erro E não pode se transformar em um múltiplo de G simplesmente pela multiplicação com um fator x_i (assumindo, é claro, que G é diferente de x_i).

Erros de burst mais longos só não são detectados se o padrão de erro E for um fator de número inteiro vezes

o polinômio gerador. Se assumirmos padrões de erro aleatórios, a probabilidade deste pode ser facilmente calculado. Com bits de código $n + \square r$ transmitidos, há um total de $2^{n + \square r}$ possíveis padrões de erro. O número de múltiplos inteiros de um polinômio gerador de grau r em uma palavra de código de comprimento $n + \square r$ é igual a 2^n . Cada múltiplo pode ser considerado como uma soma finita de n fatores, onde cada fator é obtido por um deslocamento à esquerda do gerador polinomial na palavra de dados. O gerador pode ser deslocado para a esquerda em n bits posições. Cada um desses n fatores está presente ou ausente no múltiplo final, dando 2^n múltiplos possíveis. Isso significa que uma fração

$2^{n + \square r}$

2^n

$$\frac{\overline{2}_r}{\overline{2}_r} =$$

$1_{\underline{\underline{1}}}$

de todos os erros aleatórios são perdidos. Para $r = 16$, isso corresponde a 10 $\square 5$ de todos os padrões de erro andorinhas.

POLINOMIAIS DO GERADOR PADRONIZADO

O problema de projetar um código de verificação de redundância cíclica é claramente encontrar o gerador polinômios que capturam a maior classe de erros de transmissão. Um desses polinômios é conhecido como CRC-12:

$x^{12} + \square x^{11} + \square x^3 + \square x^2 + 1$

Ele gera uma soma de verificação de 12 bits.

O CCITT recomendou o seguinte polinômio gerador para verificação de 16 bits somas, geralmente referidas como CRC-CCITT:

$$x^{16} + x^{12} + x^5 + 1$$

O grau mais alto do polinômio é dezenove, portanto, este código detecta todos os erros de burst até 16 bits de comprimento. Na aritmética do módulo 2, este polinômio também pode ser escrito como

Página 70

SEÇÃO 3.9
VERIFICAÇÕES DE REDUNDÂNCIA CÍCLICA
59

segue:

$$(x + 1) \cdot (x^{15} + x^{14} + x^{13} + x^{12} + x^4 + x^3 + x^2 + x + 1)$$

Agora, é fácil ver que qualquer polinômio multiplicado pelo fator $x + 1$ deve ter um número par de termos (ou seja, bits diferentes de zero). Isso significa que qualquer E com um estranho número de termos, produzido por qualquer número ímpar de erros de transmissão de bit único, não é divisível por $x + 1$ e pode ser detectado. Por esta razão, a maioria dos geradores padrão de polinomiais têm pelo menos um fator $x + 1$. O polinômio CCITT também pode ser mostrado para capturar todos os erros de bit duplo, 99,997% dos erros de burst de 17 bits e 99,998% de todos os bursts erros com mais de 17 bits.

Outro polinômio gerador frequentemente usado é o usado no protocolo *Bisync* da IBM col, conhecido como CRC-16 (que também possui o fator $x + 1$):

$$x^{16} + x^{15} + x^2 + 1$$

Há também um polinômio de soma de verificação de 32 bits, CRC-32, definido por padrões IEEE comitê (IEEE-802):

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

O PADRÃO ANSI FDDI

O checksum CRC-32 de 32 bits também é o polinômio usado no Fibra Distribuída Padrão de interface de dados (FDDI), definido pela ANSI em 1986. No padrão FDDI, entretanto, o cálculo da soma de verificação é um pouco diferente do padrão método explicado acima. O cálculo é o seguinte. Seja p o grau do polinômio de mensagem P , e seja L um polinômio representando uma sequência de 32 bits, tudo com valor um. A soma de verificação é calculada como o *complemento* do restante do G
(L
.
 $x_p + P$). x^{32}

Primeiro, o padrão L é anexado à palavra de código. A palavra resultante é deslocada para a esquerda por 32 bits para abrir espaço para a soma de verificação. A soma de verificação é então calculada como antes e complementado antes da transmissão. O complemento pode ser obtido em módulo-2 aritmética adicionando o padrão L ao resto. Desde o resultado a soma de verificação é obviamente diferente da anterior

G

P

.

x^{32}

uma divisão da palavra de código transmitida T pelo polinômio gerador G não mais retorna zero na ausência de erros. Para realizar a verificação, o receptor FDDI faz um cálculo diferente. Seja M a palavra de código conforme recebida, ou seja,

$$M = T + E$$

O receptor agora verifica se o restante da divisão

Página 71

60
CONTROLE DE ERROS
CAPÍTULO 3

G

(L
.

$x p \square + \square M$). x 32

é igual a
 G
eu
.
 x 32

ou seja, deve ser igual ao padrão L que foi adicionado à soma de verificação no remetente FDDI para invertê-lo antes da transmissão. A adição do padrão L e a inversão de a soma de verificação garante, entre outras coisas, que uma palavra de código transmitida nunca consists de apenas zero bits.

EFICIÊNCIA

A codificação e decodificação de somas de verificação CRC pode ser uma tarefa demorada que pode degradar o desempenho de um protocolo. A implementação é, portanto, tipicamente feito em hardware com registradores de deslocamento ou em software com armazenamento de tabelas de pesquisa valores pré-calculados para partes da soma CRC.

O seguinte programa C, de Don Mitchell, da AT&T Bell Laboratories, gera um tabela de pesquisa para um polinômio de soma de verificação arbitrário. A entrada para a rotina é um octal

número, especificado como um argumento, que codifica o polinômio gerador. Na versão do programa mostrado aqui, cumprimentos de Ned W. Rhodes, Software Systems Grupo, os bits são numerados de zero a $r - 1$, com o bit zero correspondendo à direita a maioria dos bits er o grau do polinômio gerador. (No algoritmo original de Mitchell ritmo, os bits na mensagem e o polinômio do gerador foram invertidos.) O r -ésimo bit em si é omitido da palavra de código, uma vez que está implícito no comprimento.

O uso deste programa requer duas etapas separadas. Primeiro, o programa é compilado e executado para gerar as tabelas de pesquisa. Então, a rotina de geração de checksum pode ser com-empilhados, com as tabelas de pesquisa pré-calculadas no lugar. Em um sistema UNIX → , o programa tor é compilado como

```
$ cc -o crc_init crc_init.c
```

As tabelas de pesquisa para os dois polinômios CRC mais populares agora podem ser produzidas como segue:

```
$ crc_init 0100005 > crc_16.h
$ crc_init 010041 > crc_ccitt.h
```

Este é o texto de `crc_init.c`:

```
principal (argc, argv)
int argc; char * argv [];
{
crc longo sem sinal, poli;
int n, i;
sscanf (argv [1], "% lo", & poly);
```

Página 72

SEÇÃO 3.9 VERIFICAÇÕES DE REDUNDÂNCIA CÍCLICA

```
61
if (poly & 0xffff0000)
{
fprintf (stderr, "polinômio é muito grande \n");
saída (1);
}
printf ("/* \ n * CRC 0% o \ n * / \ n", poli);
printf ("short crc_table estático sem sinal [256] = {\ \ n");
para (n = 0; n < 256; n++)
{
if (n% 8 == 0) printf ("");
")
crc = n << 8;
para (i = 0; i < 8; i++)
{
if (crc & 0x8000)
crc = (crc << 1) ^ poli;
outro
crc << = 1;
CRC & = 0xFFFF;
}
```

```

if (n == 255) printf ("0x% 04X", crc);
outro
printf ("0x% 04X,", crc);
if (n% 8 == 7) printf ("\ n");
}
saída (0);
}

```

A tabela agora pode ser usada para gerar somas de verificação:

```

curto sem sinal
cksum (s, n)
registrar unsigned char * s;
registrar int n;
{
registrar curto sem sinal crc = 0;
enquanto (n--> 0)
crc = crc_table [(crc >> 8 ^ * s++) & 0xff] ^ (crc << 8);
return crc;
}

```

A soma de verificação CRC, usando uma tabela de pesquisa com o algoritmo mostrado acima, é colocado em aproximadamente 1,1 ms de tempo de CPU (para uma mensagem de 512 bits, quando em execução

em um DEC / VAX-750). Para comparação, a seguir está a rotina de soma de verificação de o código uucp do sistema UNIX .

```

cksum (s, n)
registrar char * s;
registrar n;
{
registrar soma curta;
registrar curto não assinado t;
registrar curto x;

```

Página 73

62
CONTROLE DE ERROS
CAPÍTULO 3
soma = -1;
x = 0;
Faz {
if (soma <0) {
soma <<= 1;
soma ++;
} outro
soma <<= 1;
t = soma;
soma += (sem sinal) * s++ & 0377;
x += soma^n;
if ((curto sem sinal) soma <= t) {
soma ^= x;
}
} enquanto (--n> 0);
return (sum);
}

O método é um esquema de hashing simples e um tanto ad hoc. Demora um pouco mais Tempo de CPU para um cálculo de soma de verificação (1,8 ms por chamada), mas a proteção que provides contra erros de transmissão é menor do que a verificação de redundância cíclica.

10
-1
10
-2
10
-3
10
-4
10
-5
0
1 3 5 7 9 11 13 15

(a) - Erros / mensagem de bit único
Fração de erros perdidos

10
-1
10
-2
10
-3
10
-4
10
-5
0
1 3 5 7 9 11 13 15

(b) - Comprimento de Burst (bits)

Fração de erros perdidos

Figura 3.5 - Comparação de métodos de checksum

Uucp Checksum, sólido; CRC-16 Checksum, tracejado

Os dados da Figura 3.5 foram obtidos distorcendo aleatoriamente 164.864 mensagens de 512 bits cada. Em um primeiro teste (mostrado na Figura 3.5a), erros de bit único independentes foram introduzido. Em um segundo teste (Figura 3.5b), erros de burst foram simulados. Checksums foram calculados para as mensagens distorcidas e não distorcidas. Uma mensagem distorcida sage era aceito apenas se sua soma de verificação fosse a mesma da mensagem não distorcida. O CRC-16 captura todos os números ímpares de erros de bits e rejeita adequadamente todos os erros de burst até 16 bits. Os dois métodos têm um desempenho comparável apenas para números de erros de bit único e para erros de rajada com mais de 16 bits (não mostrado). Em todos os outros casos, o método CRC-16 é superior.

SEÇÃO 3.10
ARITHMETIC CHECKSUM

63

3.10 ARITHMETIC CHECKSUM

Cada método de soma de verificação tem uma sobrecarga em bits que é expressa como sua taxa de código.

Ele também tem uma sobrecarga oculta no tempo de CPU que é necessário para calcular a verificação bits de soma, que corroem a taxa de transmissão máxima. Os requisitos de tempo podem ser reduzido usando tabelas de pesquisa, como mostrado acima, ou desenvolvendo um propósito especial hardware para o cálculo da soma de verificação. Em aplicações onde os requisitos para a taxa de erro residual não justifica uma implementação de CRC, pode ser atraente encontrar uma alternativa simples que ainda pode fornecer proteção contra erros graves.

Um método muito interessante deste tipo foi publicado por John Fletcher em 1982. O checksum no algoritmo de Fletcher requer apenas operações de adição e módulo e é trivialmente simples. Aqui está o código de uma versão que foi adotada para a Classe ISO 4 padrão de protocolo de transporte (TP4).

```
curto sem sinal
cksum (s, n)
registrar unsigned char * s;
registrar int n;
{
registrar int c0 = 0, c1 = 0;
Faz {
c0 = (c0 + * s++)% 255;
c1 = (c0 + c1)% 255;
} enquanto (--n> 0);
retorno (curto sem sinal) (c1 << 8 + c0);
}
```

É extremamente simples, mas acaba tendo uma capacidade de detecção de erros respeitável ity. A Figura 3.6 compara o desempenho do algoritmo de Fletcher com o do uucp checksum.

```
10
-1
10
-2
10
-3
10
-4
10
-5
0
1 3 5 7 9 11 13 15
```

(a) - Erros / mensagem de bit único

Fração de erros perdidos

```
10
-1
10
-2
10
-3
10
-4
10
-5
0
1 3 5 7 9 11 13 15
```

(b) - Comprimento de Burst (bits)

Fração de erros perdidos

Figura 3.6 - Comparação de métodos de checksum

Uucp Checksum, sólido; Soma de verificação aritmética, tracejada

Dada a simplicidade do algoritmo, o retorno na capacidade de detecção de erros é certeza muito a pena.

64

CONTROLE DE ERROS

CAPÍTULO 3

3.11 RESUMO

Um módulo funcional na hierarquia do protocolo é o controle de erros. A inclusão de um esquema de controle de erros pode e deve ser transparente para o resto do protocolo. Esta função é transformar um canal com taxa de erro p em um com menor (residual) taxa de erro $p \cdot (1 - f)$, onde f é a fração dos erros que é interceptada pelo erro código.

Um esquema de controle de erro requer sobrecarga que é medida pelo número de redundâncias dadas que são adicionadas a cada palavra de código. Redundância raramente é igual a proteção (ver Exercício 3-1), mas uma pequena quantidade de redundância é um pré-requisito para qualquer erro Esquema de controle.

Com codificação adequada e ao preço de taxas de transferência mais baixas, o receptor pode usar um código de correção de erros para se recuperar dos erros característicos introduzidos pelo canal. Com menor sobrecarga, um desempenho aceitável pode ser alcançado com erro detecção de códigos que dependem de esquemas de controle de fluxo para a retransmissão de dados. Os esquemas de controle de fluxo são estudados no Capítulo 4.

A adequação de um esquema de controle de erro, no entanto, só pode ser avaliada adequadamente quando as características de erro do canal de transmissão, a taxa de transferência necessária (ou seja, taxa de código) e o nível necessário para a taxa de erro residual são conhecidos.

EXERCÍCIOS

3-1. Uma companhia telefônica recentemente considerou executar novas linhas de dados de 56 Kbit / s em uma extremidade taxa de dados final de 9600 bits / s, usando a largura de banda extra para aumentar a confiabilidade. O método escolhido foi transmitir cada byte único cinco vezes em sucessão. Por maioria voto, comparando os cinco bytes sucessivos e escolhendo o mais frequente de cada definido, o receptor decidirá qual byte foi transmitido. Comente sobre o taxa de código e a proteção contra erros de explosão.

3-2. Um esquema simples de controle de erros faz com que o receptor retransmita todas as mensagens que recebe de volta para o remetente. Cada mensagem, então, deve sobreviver a duas transmissões sucessivas para ser aceitaram. Tente construir um protocolo que funcione dessa maneira.

3-3. O protocolo do Exercício 3-2 é modificado para que o receptor apenas retorne uma verificação CRC campo de soma para o remetente por meio de confirmação. A soma de verificação é retornada para cada mensagem recebida, distorcida ou não, e é usada pelo remetente para decidir sobre a retransmissão sion. Comente esta melhoria.

3-4. (Jon Bentley) As duas sentenças " o cachorro corre " e " os cães correm " são válidas em Inglês. As sentenças " o cachorro corre " e " o cachorro corre " são ambas inválidas. Isso seria classificar a gramática do inglês como um feedback ou como um método de controle de erros direto?

3-5. A mensagem 101011000110 é protegida por uma soma de verificação CRC que foi gerada com o polinômio $x^6 + x^4 + x + 1$. O checksum está na cauda (lado direito) do mensagem. (a) Quantos bits é a soma de verificação? (b) Se nenhum erro de transmissão ocorreu, quais seriam os dados originais? (c) Houve algum erro de transmissão?

3-6. Liste as circunstâncias em que um código de correção de erros com uma taxa de código de 0,1 pode ser mais atraente do que um código de detecção de erros com controle de erro de feedback? Considerar taxas de erro e atrasos na propagação de mensagens de ida e volta.

CAPÍTULO 3

NOTAS BIBLIOGRÁFICAS

65

3-7. Outro método para adaptar um único código de correção de erros para proteção contra erros de explosão é usar n códigos de erro para uma sequência de n mensagens, onde a i -ésima palavra de código cobre apenas o i -ésimo bit de cada mensagem. Para proteger contra erros de ruptura de até k bits, este método tenta separar os bits que constituem uma nova "palavra de código," abrangendo n mensagens, por mais de k posições de bits. Calcule os detalhes deste método e aplique-o para uma mensagem de amostra.

3-8. Polinômios de checksum CRC que contêm o fator $x + 1$ capturam todos os números ímpares de bits

erros. Pense em um método para capturar todos os números pares de erros de bits também, por exemplo, por introduzir deliberadamente um erro de bit em uma segunda transmissão e comentar sobre isso esquema. Considere a taxa de código também.

3-9. Como você classificaria o algoritmo de Fletcher? (Consulte a Seção 3.4)

NOTAS BIBLIOGRÁFICAS

Mais informações sobre os vários tipos de erros de transmissão e suas causas podem ser encontrado em, por exemplo, Tanenbaum [1981, 1988], Fleming e Hutchinson [1971], e Bennet e Davey [1965]. Um tratamento orientado a aplicativos de transmissão de dados técnicas podem ser encontradas em Tugal e Tugal [1982].

O código de Hamming foi descrito pela primeira vez por Claude Shannon [1948] como um exemplo de um código perfeito. O artigo de Hamming sobre códigos de correção de erros foi publicado alguns anos depois,

Hamming [1950]. Slepian [1973] oferece uma visão geral da teoria inspirada por Trabalho de Shannon. Uma excelente introdução à teoria da codificação pode ser encontrada em JH van Notas de aula de Lint, Lint [1971].

Outra boa referência funciona na teoria de correção e detecção de erros códigos, incluindo discussões de códigos BCH e Reed-Solomon, são Berlekamp [1968], Kuo [1981], Peterson e Weldon [1972], e MacWilliams e Sloane [1977]. Os resultados do estudo da IBM de códigos de correção e detecção de erros, mencionados na conclusão da Seção 3.7, foram apresentados na IBM [1964]. Um simples método para gerar uma tabela de pesquisa CRC foi descrito em Perez [1983]. Alternativa métodos para gerar somas de verificação CRC-16 e CRC-32 usando tabelas look-ahead podem ser encontrado em Griffiths e Stones [1987]. Métodos para gerar somas de verificação CRC com deslocamento

os registros são dados, por exemplo, MacWilliams e Sloane [1977], Adi [1984] e Stallings [1985].

Os resultados de uma pesquisa realizada em 1969-1970 para medir as características de erro de linhas trocadas foi relatado em Fleming e Hutchinson [1971] e Balkovic et al.

[1971]. Uma visão geral dos resultados das medições nas linhas T1, realizadas por AT&T em 1973 e 1974, é apresentado em Brilliant [1978]. Mais discussões gerais ou várias maneiras de interpretar e analisar os dados de medição podem ser encontradas em, para exemplo, Decina e Julio [1982] ou em Ritchie e Scheffler [1982].

O método de soma de verificação aritmética de Fletcher foi descrito pela primeira vez em Fletcher [1982], e é discutido em Nakassis [1988]. A série do protocolo de transporte ISO era padronizada em ISO [1983]. O modelo preditivo para agrupamento de erros, discutido na Seção 3.2 é descrito em Bond [1987] e é devido a Benoit Mandelbrot [1965] (o inventor de fractais).

CONTROLE DE FLUXO 4

66 Introdução 4.1
70 Protocolos de janela 4.2
74 Números de Sequência 4.3
80 Agradecimentos Negativos 4.4
83 Prevenção de Congestionamento 4.5
86 Resumo 4.6
87 exercícios
88 Notas Bibliográficas

4.1 INTRODUÇÃO

A forma mais simples de um esquema de controle de fluxo apenas ajusta a taxa na qual um remetente produz dados na taxa em que o receptor pode absorvê-los. Esquemas mais elaborados pode proteger contra exclusão, inserção, duplicação e reordenação de dados também.

Mas vamos primeiro examinar a versão mais simples do problema. É usado

Para garantir que os dados não sejam enviados mais rápido do que podem ser processados.

Para otimizar a utilização do canal.

Para evitar o entupimento dos links de transmissão.

O segundo e o terceiro objetivos são complementares: enviar os dados muito devagar é desperdício, mas enviar dados muito rápido pode causar congestionamento. O caminho de dados entre o remetente e o destinatário podem conter pontos de transferência com uma capacidade limitada de armazenamento

mensagens compartilhadas entre vários pares emissor-receptor. Um controle de fluxo prudente esquema impede que um par de monopolizar todo o espaço de armazenamento disponível. Neste capítulo, construímos uma disciplina de controle de fluxo completo em uma sequência de modificações

ções de um modelo básico simples. As regras de procedimento destes protocolos são especificadas com a linguagem de fluxograma introduzida no Capítulo 2 e resumida no Apêndice B. A notação *mesg: o* em uma declaração de entrada ou saída, por exemplo, indica que um mensageiro do tipo *mesg* com campo de dados *o* é recebida ou enviada, respectivamente. O Estado-*seguinte: o* indica a recuperação interna do ítem de dados *o* a ser transmitida no próximo mensageiro de saída. Da mesma forma, *aceitar: i* indica a aceitação (armazenamento) de *i* como corretamente dados recebidos.

A Figura 4.1 ilustra um protocolo sem qualquer forma de controle de fluxo. Observe que é um protocolo *simplex*: pode ser usado para transferência de dados em apenas uma direção (ver Figura 2.1 e Apêndice A).

1. No nível mais baixo, essa sincronização já deve ocorrer para conduzir uma linha física. Veja *Syn-Transmissão crônica e assíncrona* no Apêndice A.
66

Página 78

67
remetente
próximo: o
mesg: o
receptor
receber
mesg: eu
aceitar: eu

Figura 4.1 - Sem controle de fluxo

O protocolo na Figura 4.1 só funciona de forma confiável se o processo do receptor for garantido para ser mais rápido do que o remetente. Se essa suposição for falsa, o remetente pode estourar a entrada fila do receptor. O protocolo viola uma lei básica de design de programa para con sistemas atuais:

Nunca faça suposições sobre as velocidades relativas de processos simultâneos.

A velocidade relativa dos processos simultâneos depende de muitos fatores para basear qualquer decisões de design sobre ele. Além disso, a suposição sobre a velocidade relativa de o remetente e o destinatário geralmente não são apenas perigosos, mas também inválidos. O recebimento de dados é

geralmente um processo mais demorado do que enviar dados. O receptor deve interagir pretenda os dados, decida o que fazer com eles, aloque memória para eles e talvez os encaminhe para o destinatário apropriado. O remetente não precisa encontrar um provedor para os dados que é transmitindo: ele não funciona a menos que haja dados para transferir. E, em vez de alocar memória, o remetente pode ter que liberar memória depois que os dados são transmitidos, usando almente uma tarefa menos demorada. Portanto, o gargalo no protocolo provavelmente ser o processo receptor. É um péssimo planejamento presumir que sempre pode acompanhar o remetente.

A técnica de controle de fluxo mais antiga e menos confiável que pode ser usada para resolver este problema de sincronização não requer negociação prévia entre o remetente e o destinatário sobre o ritmo em que as mensagens podem ser transmitidas. O método usa dois controles mensagens: uma para *suspender* e outra para *retomar* o tráfego. As mensagens às vezes são chamado *x-off* e *x-on*. Suponha, então, que temos um canal livre de erros e um proto vocabulário col dos três tipos de mensagem a seguir:

$$V = \{\text{mesg}, \text{suspender}, \text{retomar}\}$$

2. Os códigos *control-s* e *control-q* em muitos terminais de dados fornecem as mesmas duas funções.

Página 79

68
CONTROLE DE FLUXO
CAPÍTULO 4

onde as mensagens de controle *suspensas* e *retomadas* são usadas para implementar o fluxo de con

disciplina trol. As regras de procedimento do protocolo podem agora ser adicionadas. Nós implementamos

-los aqui com dois processos adicionais, um no remetente e um no receptor, como mostrado na Figura 4.2.

```
remetente
próximo: o
state == go
mesg: o
alternancia
receber
suspenso
estado = espera
currículo
state = go
```

Figura 4.2 - Protocolo X-on / X-off: Processos do Remetente

Depois de receber uma mensagem de suspensão , o processo de alternância no remetente define o valor de um

estado variável para esperar . Ele redefine a variável para seu valor inicial go após a chegada de um retomar a mensagem. O processo do remetente simplesmente espera (na caixa oval) até que o estado tenha o

valor adequado antes de transmitir a próxima mensagem. 3 3. Lembre-se de que a caixa oval indica cates um atraso potencial. O processo aguarda a chegada de uma mensagem quando a caixa é rotulado como recebimento , ou então ele espera até que a condição especificada se torne verdadeira. Cf. Figura

2.1.

O receptor também está dividido em duas partes. Após a chegada de uma mensagem de dados, um contador

Página 80

69
contador / buffer
receber
mesg: eu
mesg: eu
n ++
n > max
falso
receptor
receber
mesg: j
aceitar: j
n—
n < min
falso
verdadeiro
suspenso
verdadeiro
currículo

Figura 4.3 - Protocolo X-on / X-off: Processos do receptor

processo incrementa uma variável n , e após a aceitação da mensagem, um aceitador o processo diminui. As mensagens de dados são passadas do processo de contagem para o processo aceitador por meio de uma fila interna. A contagem lembra o número de mensagens que foram recebidos do remetente e o número que está esperando para ser aceito pelo receptor. Se seu valor aumentar além de algum limite predefinido, um suspender a mensagem é enviada ao remetente. Se cair abaixo de um limite inferior, o currículo a mensagem é enviada, conforme mostrado na Figura 4.3. Para dividir o receptor em dois processos, de claro, só faz sentido se aceitar for uma operação relativamente demorada.

Existem alguns problemas a serem resolvidos. O funcionamento correto do protocolo depende nas propriedades do canal de transmissão. Se uma mensagem de suspensão for perdida ou mesmo atrasado, o problema de estouro se repete. O funcionamento de um protocolo não deve depender no tempo que uma mensagem de controle leva para chegar ao receptor. Pior ainda, se um currículo a mensagem é perdida, o sistema de quatro processos é completamente interrompido.

Temos estes dois problemas para resolver:

Proteja-se contra erros de saturação de uma forma mais confiável.

Proteger contra perda de mensagem.

Um método padrão para resolver o primeiro problema é deixar o remetente esperar explicitamente por a confirmação de mensagens transferidas. Um exemplo é o protocolo Ping-Pong

da Figura 4.4. Esse método costuma ser chamado de protocolo de *parada e espera*. O estouro problema desapareceu, mas o sistema ainda bloqueia se um controle ou dados a mensagem está perdida. O emissor e o receptor estão acoplados de maneira muito forte. Vamos t ser a mensagem

Página 81

70
CONTROLE DE FLUXO
CAPÍTULO 4
remetente
próximo: o
mesg: o
receber
ack
receptor
receber
mesg: eu
aceitar: eu
ack

Figura 4.4 - Protocolo de Ping-Pong

tempo de propagação no canal, uma o tempo que leva para o receptor processar e aceitar uma mensagem recebida, e p o tempo que leva o remetente para preparar uma mensagem para transmissão. Com o esquema acima, o remetente incorre em um atraso de $2t + \square a - \square p$ unidades de tempo para cada mensagem transmitida.

Normalmente $p < \square a$ e, obviamente, t aumenta pelo menos linearmente com a distância entre remetente e destinatário. Observe, no entanto, que a mensagem de confirmação não apenas significa a chegada da última mensagem, também é usado como um *crédito* que o receptor estende-se ao remetente para transmitir a próxima mensagem. Essa ideia leva diretamente a uma solução que pode aliviar o problema de atraso: o protocolo da janela.

4.2 PROTOCOLOS DE JANELA

Em uma fase de configuração de chamada, o receptor pode dizer ao remetente exatamente quanto espaço de buffer ele

está preparado para reservar para mensagens recebidas. O remetente recebe *crédito* por um número fixo de mensagens pendentes. O crédito pode ser atualizado dinamicamente quando a quantidade de alterações de espaço de buffer disponível.

Não vamos nos preocupar com a perda de mensagens ainda e primeiro olhemos para o funcionamento básico de um

protocolo de janela. Cada mensagem recebida é confirmada com um único controle de *confirmação* mensagem em um canal de retorno. Tudo o que temos que fazer é manter a contagem do número de mensagens em trânsito.

O crédito inicial pode ser negociado ou pode ser definido para um número fixo de mensagens sábios W . Para cada mensagem enviada, o remetente diminui seu crédito, e para cada mensagem sábio recebido o receptor estende um novo crédito ao remetente por meio do canal de retorno.

Página 82

SEÇÃO 4.2
PROTOCOLOS DE JANELA
71

O protocolo de exemplo mostrado na Figura 4.5 ilustra isso. A quantidade que $W - \square n$ dá o número de créditos não utilizados.

remetente
 $n < \square W$
verdadeiro
próximo: o
 $n ++$
mesg: o
falso
receber
ack
 $n --$
receptor
receptor
mesg: eu
aceitar: eu
ack

Figura 4.5 - Protocolo de janela para um canal ideal

Seja $a(t)$ o número de mensagens de crédito recebidas pelo remetente no tempo t após o inicialização, seja $m(t)$ o número de mensagens enviadas ao receptor, e seja $n(t)$ o valor de n no tempo t . O número máximo de mensagens que o remetente pode enviar em pé, esperando confirmação, é

$$W - \square n(t) + \square m(t) - \square a(t)$$

onde $W - \square n(t)$ é o número de créditos não utilizados e $m(t) - \square a(t)$ o número de créditos utilizados créditos. Gostaríamos de nos convencer de que

$$W - \square n(t) + \square m(t) - \square a(t) \leq W$$

ou

$$m(t) - \square a(t) \leq n(t)$$

Inicialmente, todas as variáveis nesta desigualdade são zero e a condição é trivialmente verdadeira.

Cada ação de envio no remetente incrementa ambos os lados da desigualdade, lado direito

primeiro, e preserva sua validade. Da mesma forma, com cada ação de recepção, o receptor processa diminui ambos os lados em um, o lado esquerdo primeiro, novamente preservando a correção.

PERDA DE MENSAGEM

O crédito máximo W é chamado de *tamanho da janela* do protocolo. Durante uma transferência, o crédito atual varia entre zero e W , dependendo das velocidades relativas de

Página 83

72

CONTROLE DE FLUXO
CAPÍTULO 4

remetente e destinatário. O remetente só é atrasado quando o crédito é reduzido a zero.

Esta disciplina de controle de fluxo pode otimizar a comunicação em canais com longa transição sentar atrasos, permitindo que o remetente transmita novas mensagens enquanto espera pelo acknowledgement reconhecimento dos antigos.

Os problemas de mensagens perdidas, inseridas, duplicadas ou reordenadas, é claro, ainda existir. Se, por exemplo, um conjunto de mensagens de confirmação for perdido, ambas as partes podem travar: o remetente aguardando as confirmações perdidas, o receptor aguardando pelas mensagens creditadas.

PRAZOS

Para se proteger contra a perda de mensagens essenciais, o remetente deve acompanhar Tempo decorrido. No protocolo de Ping-Pong da Figura 4.4, por exemplo, o remetente pode tentar para prever o pior tempo de resposta para cada confirmação. Se a resposta tiver não chegar dentro desse período, o remetente pode atingir o *tempo limite* e assumir que foi perdido. Na prática, o "pior" tempo de retorno é frequentemente calculado com uma heurística:

$$\text{Pior } t$$

$$= \square T$$

$$+ \square N$$

$$\cdot \sum \Sigma \Sigma \Sigma \Sigma$$

$$\text{var}(T)$$

onde T é o atraso de ida e volta, N geralmente é um e raramente é maior que dois. o atraso de ida e volta é simplesmente o tempo que uma mensagem leva para ir do remetente ao receptor mais o tempo que uma resposta leva para retornar ao remetente (consulte o Exercício 4-12). T

$$\bar{e}$$

$\text{var}(T)$ são, respectivamente, a *média* e o *desvio* de T . O fator N é, portanto, um fator de multiplicação para o desvio padrão do tempo de resposta (a raiz quadrada de a variância).

Em muitos casos, o comportamento do processo do receptor na extremidade de uma transmissão o canal pode ser modelado por um sistema de filas M / M / 1.⁴ Então assumimos que, a partir de do ponto de vista do receptor, a função de distribuição dos tempos entre chegadas de mensagens sábios é um processo de Poisson e o tempo de distribuição para o processamento dessas mensagens sábios é uma função exponencial simples. Para um sistema de filas M / M / 1, pode ser mostrou que a variância do tempo gasto no sistema é o quadrado da média.

Isso significa que, para o nosso canal de transmissão, a variação tanto de sentido único quanto de o atraso de ida e volta também é o quadrado da média, $\text{var}(T) = \square T$

$$\frac{1}{2}$$

. Isso leva ao sim-

regra geral de que uma aproximação para o tempo de retransmissão pode ser obtida dobrando o atraso médio de ida e volta T (assumindo um fator $N = 1$ na estimativa acima companheiro):

Pior t
<<□2. T

Um tempo limite após um erro de exclusão certamente parece simples. Um erro comum, entretanto, é permitir que o remetente e o receptor usem tempos limite. Considere a extensão do protocolo Ping-Pong mostrado na Figura 4.6.

4. A notaçāo é devida a DG Kendall [1951].

Página 84

SEÇÃO 4.2
PROTÓCOLOS DE JANELA

73

remetente
próximo: o
mesg: o
receber
ack
tempo esgotado
receptor
receber
mesg: eu
aceitar: eu
ack
tempo esgotado

Figura 4.6 - Protocolo de Ping-Pong com Timeouts

remetente
Próximo
tempo esgotado
Próximo
Próximo
receptor
tempo esgotado
aceitar
aceitar
mesg
mesg
.....
ack
mesg
.....
ack

Figura 4.7 - Diagrama de sequência de tempo de um erro

A Figura 4.7 mostra o que pode acontecer com este protocolo se ocorrer um erro de exclusão. Ambos o remetente e o destinatário decidem retransmitir a última mensagem enviada quando um erro de exclusão é

assumido. Quando a primeira mensagem de *confirmação* chega ao remetente, ela não pode dizer se reconhece a mensagem perdida ou retransmitida. O remetente acaba correspondendo as mensagens *ack* e *mesg* erradas indefinidamente.

Uma lição a ser aprendida com isso é que o remetente e o destinatário não devem ser capazes para iniciar retransmissões. É suficiente colocar essa responsabilidade com um dos dois processos. Tradicionalmente, este é o processo do remetente, uma vez que apenas o remetente pode

Página 85

74
CONTROLE DE FLUXO
CAPÍTULO 4

saiba com certeza quando novos dados foram enviados. Outra lição é que devemos ser capazes para saber, a partir de uma confirmação, exatamente qual mensagem está sendo confirmada, mesmo se pretendemos enviar apenas uma mensagem de cada vez, como no protocolo Ping-Pong. Nós pode fazer isso adicionando *números de sequência* a cada mensagem de dados e controle. Fazendo então, também obtemos um mecanismo para resolver outras classes de problemas de transmissão em uma maneira bastante direta: erros de duplicação e mensagens fora de seqüência.

Uma vez que os números de sequência necessariamente têm um intervalo restrito,⁵ devemos ter uma maneira de

verifique se os números reciclados não podem atrapalhar o funcionamento correto do protocolo. Nós verá abaixo que se os números de sequência são usados em combinação com uma janela tocol este requisito pode ser cumprido com relativa facilidade. Antes de fazermos essa combinação nação, o protocolo da janela *deslizante*, vamos examinar mais de perto o uso de tempos limite e números de sequência.

4.3 NÚMEROS DE SEQUÊNCIA

A 0

$\overline{B1}$
A 0

$\overline{B0}$
A 1

$\overline{B0}$
A 1

$\overline{B1}$
A 1
A 0
B 0

$\overline{A0}$
B 0

$\overline{A1}$
B 1

$\overline{B1}$

Figura 4.8 - Protocolo de bit alternado original

Como exemplo de uma melhor utilização de um tempo limite e um número de sequência de um bit, podemos

considere uma versão estendida do protocolo de bit alternado (um protocolo famoso, consulte as Notas Bibliográficas). O protocolo continua a vir à tona em tantos disfarces na literatura do protocolo de que vale a pena olhar primeiro para as especificações originais icação de Bartlett, Scantlebury e Wilkinson [1969]. Em seu papel, o protocolo é definido com duas máquinas de estados finitos de seis estados cada, conforme mostrado na Figura 4.8. O protocolo original, portanto, não pode estar em mais de 36 estados diferentes, substancialmente menos do que todas as outras variações que foram estudadas.

Figura 4.8 especifica o comportamento de dois processos, A e B. A notação é de Bartlett, Scantlebury e Wilkinson [1969]. Os rótulos de borda especificam a mensagem trocas. Cada rótulo consiste em dois personagens. O primeiro especifica a origem da mensagem sendo recebida ou transmitida, e a segunda especifica o número da sequência,

5. Existe apenas um número finito de bits para armazená-los nos cabeçalhos das mensagens.

chamado *de bit de alternância* no artigo original. As ações de envio estão sublinhadas. As setas duplas indicam estados onde a entrada deve ser aceita no receptor ou onde uma nova mensagem é buscada para saída no remetente. Entradas erradas, ou seja, mensagens que carregam o número de sequência errado, solicitam uma retransmissão do último mensagem enviada. É relativamente fácil estender o protocolo com tempos limite para permitir recuperação de perda de mensagem. Uma versão do fluxograma desta extensão é mostrada na Figura 4.9.

remetente
próximo: o
mesg: o: s
receber
ack: r
r == s
verdadeiro
s = 1 - s
tempo esgotado
falso
receptor
receber

```

mesg: i: a
ack: a
a == e
verdadeiro
e = 1 - e
aceitar: eu
falso

```

Figura 4.9 - Protocolo de bits alternados com tempos limite

Usamos dois tipos de mensagens, *mesg* e *ack*, com, por exemplo, o formato

{*mesg*, dados, número de sequência}

e

{*ack*, número de sequência}

respectivamente. No fluxograma, *mesg: o*: *s* indica uma mensagem *mesg* com *o* campo de dados *o* e campo de número de sequência *s*.

Também usamos quatro variáveis de bit único: *a*, *e*, *r* e *s*. A variável *s* é usada pelo

76

CONTROLE DE FLUXO
CAPÍTULO 4

remetente para armazenar o último número de sequência enviado, *er* contém o último número de sequência

recebido. O receptor usa *e* para manter o próximo número esperado para chegar e variável *a* para armazenar o último número de sequência real recebido. Todas as variáveis têm um valor inicial zero.

A Figura 4.10 ilustra o que acontece se o erro de exclusão da Figura 4.7 ocorrer no protocolo de bit alternado. O protocolo se recupera do erro quando o remetente processa expira e retransmite a mensagem perdida.

```

remetente
Próximo
tempo esgotado
Próximo
receptor
aceitar
mesg
mesg
...
ack

```

Figura 4.10 - Diagrama de sequência de tempo do erro

Considere também o que acontece se uma confirmação for atrasada o suficiente para o remetente atingir o tempo limite e retransmitir a última mensagem (consulte o Exercício 4-6).

REORDENAGEM DE MENSAGEM

Agora, consideremos a duplicação e reordenação de mensagens, como pode acontecer em, por exemplo, redes de datagramas onde as mensagens podem viajar por diferentes rotas para seu destino. A solução óbvia é codificar a ordem original das mensagens em um número de sequência maior anexado a cada mensagem. Com um campo de 16 bits para os números de sequência, podemos numerar 65.536 mensagens subsequentes. Assumindo uma mensagem comprimento de 128 bits e uma velocidade de linha efetiva de 9600 bps (bits por segundo), poderíamos esgotar os números em 15 minutos. Felizmente, este problema de alcance logo desaponta peras se limitarmos o número máximo de mensagens que podem estar em trânsito em qualquer um tempo: o crédito do remetente. Claramente, o intervalo dos números de sequência deve ser maior do que o crédito máximo usado para que um receptor possa sempre distinguir mensagens duplicadas sábios de originais.

Suponha um intervalo *M* de números de sequência disponíveis e um crédito inicial de *W* mensagens. Presumimos por enquanto que *M* é suficientemente maior do que *W* para evitar confusão

de números de sequência reciclados. O remetente deve fazer alguma contabilidade para cada saída mensagem permanente na janela atual. Usamos dois arrays para esse propósito.

O elemento da matriz booleana *ocupado* [*s*] é definido como *verdadeiro* se uma mensagem com número de sequência *s* era

enviada e ainda não foi confirmada. O segundo *armazenamento de matriz* [*s*] lembra a última mensagem com número de sequência *s* que foi transmitida. Inicialmente, todos os elementos de array *busy* são definidas como *verdadeiras*.

Há muitos problemas a serem resolvidos para que esta versão do protocolo da janela funcione. A tarefa pode ser dividida em três subtarefas: transmitir mensagens, processar ack-agradecimentos e retransmitindo mensagens que permanecem não confirmadas por muito longo. Além das constantes W e M , as seguintes quatro variáveis são usadas, todas com um valor inicial de zero:

```
s , o número de sequência da próxima mensagem a enviar
janela , o número de mensagens não confirmadas pendentes
n , o número de sequência da mensagem não reconhecida mais antiga
m , o número de sequência da última mensagem reconhecida
transmissão
processo
janela <□W
próximo: o
janela ++
ocupado [s] = verdadeiro
loja [s] = o
mesg: o: s
s = (s + 1)% M
retransmissão
processo
janela >□0
ocupado [n]
verdadeiro
tempo esgotado
mesg:
loja [n]: n
falso
janela --
n = (n + 1)% M
reconhecimento
processo
receber
ack: m
ocupado [m] =
falso
```

Figura 4.11 - Processos do remetente, protocolo da janela deslizante

Considere primeiro o processo de transmissão na Figura 4.11. Contanto que todos os créditos não tenham esgotadas, as mensagens podem ser transmitidas. Cada mensagem transmitida incrementa o número de mensagens pendentes e, ao fazer isso, diminui implicitamente o crédito para a transmissão de novas mensagens. Um número de sequência s é atribuído, a mensagem os conteúdos são armazenados na *loja* [s] para possível retransmissão posterior, o sinalizador é definido em

Página 89

ocupado [s] e s é o módulo incrementado do intervalo dos números de sequência M (usando o operador '%').

O processo de confirmação é ainda mais simples. Ele recebe a confirmação de entrada mentos e define o sinalizador de *ocupado* [m] como *falso*. A ordem em que esses reconhecimentos são recebidos é irrelevante.

O processo de retransmissão espera até que haja mensagens em trânsito, verificando se a *janela* é diferente de zero. Cada mensagem enviada deve ser reconhecida e ter seu sinalizador de *ocupado* [n] redefinido para *falso*. O processo de retransmissão espera que isso aconteça

caneta na segunda cláusula de espera. Em caso afirmativo, o tamanho da janela é diminuído e n é incrementado para apontar para a próxima mensagem não confirmada mais antiga. Se o sinalizador de *ocupado* for

não redefinir como *falso* em um período de tempo finito, o processo de retransmissão expira e retransmite a mensagem. A caixa oval atrasa o processo até a condição especificado torna-se *verdadeiro* ou, como no caso atual, até que ocorra um tempo limite (cf. Apêndice B). Da maneira como especificamos aqui, o temporizador de retransmissão repete apenas uma mensagem sábio, a mensagem não reconhecida mais antiga.

O receptor para o protocolo de janela deslizante é dado na Figura 4.12. É dividido em

dois processos. Um processo recebe e armazena as mensagens recebidas em qualquer ordem que podem acontecer para chegar. Um segundo processo aceita e reconhece o mensagens, usando os números de seqüência para restaurar sua ordem adequada. Mensagens não podem ser reconhecidos até que sejam aceitos, para evitar o risco de ficar sem buffers para armazene mensagens se o processo de aceitação for mais lento que o remetente. Nós usamos um array booleano $recv[M]$ para lembrar os números de sequência de mensagens que têm sido recebido, mas ainda não aceito, e um *buffer de matriz* $[M]$ para lembrar o conteúdo dessas mensagens. Existe uma variável extra para manter o controle do protocolo do gress: p , o número de sequência da próxima mensagem a ser aceita. Tem uma inicial valor de zero.

O processo de aceitação é direto. Ele aguarda a bandeira *recebida* da próxima mensagem sóbrio ser aceito para se tornar *verdadeiro*, aceita e reconhece a mensagem, e incrementos p . O receptor verifica se uma mensagem recém-chegada é original ou uma duplicada. Para uma nova mensagem, os sinalizadores *recebidos* são definidos e a mensagem é armazenada

no *buffer de matriz*. Dois sinalizadores devem ser atualizados, um para a mensagem que foi apenas recebido e um para uma mensagem que agora sabemos que não pode mais ser recebida porque está fora da janela atual (consulte o Exercício 4-14).

$recv[m] = \square verdadeiro$
 e
 $recv[(m - \square W + \square M) \% M] = \square falso$
 ou equivalente
 $recv[(m - \square W) \% M] = \square falso$

Uma mensagem duplicada é reconhecida pelo fato de que o sinalizador *recebido* foi definido como *verdadeiro*

Página 90

SEÇÃO 4.3
 NÚMEROS DE SEQUÊNCIA

79

receptor
processo
receber
mesg: i: m
recv [m]
falso
definir bandeiras
buffer [m] = i
verdadeiro
válido (m)
verdadeiro
ack: m
falso
aceitar
processo
recv [p]
aceitar
buffer [p]
ack: p
p
 $(p + 1) \% M$

Figura 4.12 - Processos do receptor, protocolo da janela deslizante

antes. Existem dois motivos possíveis para a chegada de uma mensagem duplicada:
 A mensagem foi recebida, mas ainda não foi confirmada.

A mensagem foi recebida e confirmada, mas a confirmação de alguma forma não alcançou o remetente.

Somente no segundo caso o reconhecimento deve ser repetido. O valor atual da variável p deve ser suficiente para descobrir qual dos dois casos se aplica. Se o a contagem do número de sequência não era módulo M , o teste seria simplesmente:

$válido (m) = (0 < \square p - \square m \delta \square W) \mid | (0 < \square p + \square M - \square m \delta \square W)$

O protocolo da janela garante que uma mensagem retransmitida não possa ter uma sequência número que é mais do que W menor do que a última mensagem que foi confirmada.

80

CONTROLE DE FLUXO
CAPÍTULO 4

O único caso, então, onde podemos ter $m > \square p$ ou $p - \square m > \square W$ é quando p foi embrulhado em torno do máximo M e m não.

TAMANHO MÁXIMO DA JANELA

Se M é o intervalo dos números de sequência, qual é o número máximo de outstanding mensagens W que podemos usar e ainda garantir que o protocolo da janela funcione devidamente? Se todas as mensagens que chegam fora de ordem forem simplesmente rejeitadas pelo receptor,

a resposta seria $M - 1$. Contanto que um número de sequência não seja reciclado antes do última mensagem usando é confirmada, está tudo bem. Isso significa que se as mensagens podem ser recebido fora de ordem, como na Figura 4.12, o tamanho da janela não pode exceder $M/2$ (cf. Exercício 4-9).

Como exemplo, considere o seguinte caso. Seja H o maior número de sequência (módulo M) que o receptor leu e reconheceu. Significa para o receptor que o remetente processou pelo menos uma confirmação para a mensagem W -th precedendo o numerado H (observação 1). O receptor também sabe que, na melhor das hipóteses o remetente processou todas as confirmações até e incluindo aquele para a mensagem com o número H (observação 2).

A observação 1 significa que o remetente pode decidir retransmitir qualquer um dos $W - 1$ mensagens que precedem H e o próprio H . A mensagem mais antiga que poderia ser retransmitida levaria número de sequência ($H - \square W + 1$)% M .

A observação 2 significa que o remetente também pode transmitir até W das mensagens que bem sucedido a mensagem numerados H . A primeira $W - 1$ dessas mensagens pode até mesmo se perder no canal de transmissão para que a mensagem com o número ($H + \square W$)% M é a primeira nova mensagem a chegar.

A mensagem de número mais alto que pode suceder H deve ser distingível da mensagem que pode ser retransmitido anterior número de sequência de número mais baixo H . Isso significa $M > 2W - 1$ ou um tamanho máximo de janela de $W = \square M / 2$.

4.4 RECONHECIMENTOS NEGATIVOS

Até agora, usamos reconhecimentos como um método de controle de fluxo, não de controle de erros trol. Se uma mensagem for perdida ou danificada além do reconhecimento, a ausência de um positivo o reconhecimento faria com que o remetente atingisse o tempo limite e retransmitisse a mensagem. Se a probabilidade de erro for alta o suficiente, isso pode degradar a eficiência de o protocolo, forçando o remetente a ficar ocioso até que possa ter certeza de que um mento não é apenas atrasado, mas positivamente perdido. O problema pode ser aliviado, embora não seja completamente evitado, com a introdução de reconhecimentos *negativos*.

A confirmação negativa é usada pelo receptor sempre que recebe uma mensagem que está danificado no canal de transmissão. Como o receptor pode ser capaz de estabelecer lish que é discutido no Capítulo 3. Quando o remetente recebe uma confirmação negativa , ele sabe imediatamente que deve retransmitir a mensagem correspondente, sem ter que esperar por um tempo limite. O tempo limite em si ainda é necessário, é claro, para permitem a recuperação de mensagens que desaparecem no canal.

As Figuras 4.13 e 4.14 mostram uma extensão do protocolo de bit alternado da Figura

SECÃO 4.4
RECONHECIMENTOS NEGATIVOS

81

remetente
próximo: o
mesg: o: s
receber
tempo esgotado
errar
nak
ack: r
 $r == s$
falso
verdadeiro

$$s = 1 - e$$

Figura 4.13 - Remetente, protocolo de bit alternado estendido

4,9 com confirmações negativas. Neste caso simples, o *nak* não precisa de sequência número. (Veja também Exercício 4-3.)

TERMINOLOGIA

O método de usar confirmações para controlar a retransmissão de mensagens é geralmente referido como um método *ARQ*, onde ARQ significa Repetição Automática Solicitação. Existem três variantes principais:

Pare e espere ARQ

Repetição seletiva ARQ

ARQ Go-back-N contínuo

O protocolo de Ping-Pong da Figura 4.4, possivelmente estendido com confirmação negativa mentos, é classificado como um *ARQ pária-e-espera*. Depois que cada mensagem é enviada, o remetente deve

espere por uma confirmação positiva ou negativa, ou talvez um tempo limite.

O uso de reconhecimentos no protocolo de janela deslizante das Figuras 4.11 e 4.12

é um método *ARQ de repetição seletiva*. Na Figura 4.11 implementou um "um de cada vez" método de repetição seletivo, onde apenas a mensagem não reconhecida mais antiga é retransmitida. Em geral, no entanto, qualquer mensagem que acione uma confirmação negativa ou um tempo limite pode ser retransmitido, independentemente de qualquer outra mensagem pendente.

Página 93

82

CONTROLE DE FLUXO

CAPÍTULO 4

receptor

receber

msg: i: a

ack: a

a == e

verdadeiro

e = 1 - e

aceitar: eu

falso

errar

nak

Figura 4.14 - Receptor, protocolo de bit alternado estendido

O método generalizado é chamado de repetição seletiva "contínua".

A última estratégia, *go-back-N ARQ contínuo*, poderia ser implementada no programa acima tocol fazendo com que o remetente retransmita a mensagem corrompida e todas enviadas posteriormente mensagens. Nesse caso, o design do receptor pode ser simplificado. O aceitar processador da Figura 4.12, por exemplo, agora pode ser excluído e o buffer torna-se supérfluo. Em uma disciplina de retorno-N, o receptor se recusa a aceitar todas as mensagens que chegam fora de serviço e espera que cheguem na sequência adequada. Não vai reconhecer todas as mensagens fora de ordem. Uma confirmação com número de sequência *s* agora pode ser entendido como reconhecendo todas as mensagens até e incluindo *s*. Tal um a confirmação é, portanto, às vezes chamada de *confirmação cumulativa*.

BLOQUEAR RECONHECIMENTO

Uma variação que pode ser usada com a repetição seletiva e a estratégia go-back-N para reduzir o número de mensagens de confirmação individuais que devem ser enviadas de receptor para remetente é conhecido como *confirmação de bloco*. Neste caso, cada confirmação positiva o reconhecimento pode especificar uma gama de números de sequência de mensagens que foram recebido corretamente. A confirmação do bloco pode ser enviada periodicamente ou no pedido do remetente. O reconhecimento de bloco pode ser visto como uma forma estendida de cumulação reconhecimento positivo.

Página 94

SEÇÃO 4.5

EVITAR CONGESTÕES

83

4.5 EVITAÇÃO DE CONGESTÕES

No início deste capítulo, demos duas razões principais para a inclusão do controle de fluxo esquemas em protocolos: sincronização e prevenção de congestionamento.

Até este ponto, ignoramos principalmente a prevenção de congestionamento e nos concentraremos no fim

sincronização final. Uma questão importante em particular ainda não foi discutida: Para um determinado link de dados, como é o tamanho real da janela e o intervalo correspondente de números de sequência escolhidos? É relativamente fácil definir um limite superior na janela tamanho: em algum ponto aumentando ele não pode mais melhorar a taxa de transferência se o canal já está totalmente saturado.

Suponha que leve 0,5 segundos para uma mensagem viajar do remetente ao receptor, e outros 0,5 segundos para que a confirmação retorne ao remetente. O remetente pode então saturar totalmente o canal se continuar enviando dados por 1 segundo. Se os dados taxa do canal é S bps, o remetente deve ser capaz de transmitir S bits antes de precisar para verificar os reconhecimentos. Se houver M bits em cada mensagem transmitida, o melhor tamanho da janela está trivialmente S/M . E, claro, é melhor ter certeza de que $M < S$. Um tamanho de janela maior do que S/M é um desperdício: no momento em que a última mensagem no

janela atual é transmitida, o reconhecimento da mensagem pendente mais antiga sábio deveria ter chegado, e se não chegou, pode ser hora de começar a considerar o retransmissão dessa mensagem.

Existe um perigo no tipo de cálculo que realizamos aqui. Isso reduz o problema de controle de fluxo para um problema de nível de link, enquanto ignora a rede que contém o link de dados. Considere, por exemplo, o link de dados de dois saltos mostrado na Figura 4.15.

Remetente
Transferência de 1 Mbps
Ponto
10 Kbps
Receptor

Figura 4.15 - Link de dois saltos

Existem duas maneiras de definir um protocolo de controle de fluxo para transferências do remetente para o receptor nesta rede de dois links:

Hop-by-hop (também chamado de nó a nó)

De ponta a ponta

Em um protocolo salto a salto, o tamanho da janela é calculado separadamente para cada link a ser tentado

para saturar cada um. O primeiro link é 100 vezes mais rápido que o segundo. Mas se nós sucesso em saturar ambos os canais, só conseguimos criar um problema maior lem. Os dados chegam ao ponto de transferência cerca de 100 vezes mais rápido do que podem ser transmitidos

no receptor. Não importa quanto espaço de buffer o ponto de transferência tenha inicialmente, ele eventualmente ficar sem espaço e, a menos que possa reduzir o fluxo do remetente, ele começará perder mensagens.

A única maneira que o ponto de transferência pode controlar o remetente é se recusando a reconhecer mensagens. O remetente, no entanto, tenta saturar o canal e o fará, também

Página 95

84

CONTROLE DE FLUXO
CAPÍTULO 4

com retransmissões ou com novos dados. Se o número de confirmações cair, o remetente continuará a saturar o canal, retransmitindo dados.

Um esquema de controle de fluxo, então, deve ser projetado para otimizar a utilização de dois recursos separados:

O espaço de buffer nos nós da rede

A largura de banda dos links que conectam os nós

O esquema simples acima falha em ambos os casos: ele desperdiça espaço de buffer na transferência ponto, potencialmente bloqueando outro tráfego que pode ser roteado através desse nó, e desperdiça largura de banda, desencadeando um dilúvio de retransmissões no link do remetente para o ponto de transferência. O uso ideal do caminho de dados de dois links só pode ser alcançada se o remetente oferece dados na taxa de dados do link mais lento no caminho: apenas 1% do ponto de saturação do primeiro link, o que implica algum tipo de feedback esquema do segundo link para o primeiro.

Em um protocolo de ponta a ponta, esse problema não existe. A capacidade de ponta a ponta do o caminho da rede é igual à capacidade do link mais lento e o tamanho da janela pode ser definido adequadamente. O problema é que em uma rede complicada não há esperança de que um O remetente pode prever facilmente onde estará o link mais lento em seu caminho até o receptor. o

a coisa mais segura a fazer seria derivar um tamanho máximo de janela para toda a rede que se baseia em seu link mais lento. Mas isso dificilmente é uma solução inspiradora, não para os homens

ção um desperdício. Além disso, em uma rede maior, a capacidade de um link de dados depende não apenas do hardware, mas também do número de usuários concorrentes. Se dez os usuários começam a transferir arquivos grandes pelo link mais rápido da rede, esse link pode de repente se tornou o mais lento para todos os outros usuários.

Voltando ao problema original, embora tenhamos fingido o contrário até

Nesse ponto, o controle de fluxo não é um problema estático, mas dinâmico. Em um fluxo estático protocolo de controle, um remetente sempre assume que uma mensagem foi perdida ou distorcida se seu reconhecimento não chega com o tempo de atraso da mensagem de ida e volta. o a resposta apropriada do remetente, nesse caso, é retransmitir a mensagem. Pode, no entanto, também significar que a rede está sobrecarregada. A resposta apropriada do remetente deve então *reduzir* a quantidade de tráfego que oferece à rede. O mais simples O método que o remetente tem para fazer isso é diminuir o tamanho da janela.

CONTROLE DE FLUXO DINÂMICO

Um método de *controle de fluxo de janela dinâmico* torna o protocolo auto-adaptável, um dos princípios de design de som que listamos no Capítulo 2. Um método simples e comumente usado método é forçar um remetente a diminuir o tamanho da janela sempre que uma retransmissão tempo limite ocorre. Assim que os tempos limite desaparecerem, o remetente pode ser autorizado a gradualmente

aumente o tamanho da janela de volta ao seu valor máximo. Existem diferentes filosofies sobre os parâmetros precisos a serem usados em tal técnica. Três variedades populares as ações estão listadas abaixo.

Diminua a janela em *um* para cada timeout que ocorrer e aumente em um para *cada* reconhecimento positivo.

Diminua a janela para a *metade* de seu tamanho atual a cada timeout e aumente em

Página 96

SEÇÃO 4.5 EVITAR CONGESTÕES 85

uma mensagem para cada *N* confirmações positivas recebidas.

Diminuir para seu valor *mínimo* de um, imediatamente quando ocorre um tempo limite, e aumente a janela em um para cada *N* confirmações positivas recebidas.

Todos os métodos assumem um tamanho mínimo de janela de um. O tamanho máximo pode ser calculado como antes, ou pode ser definido com um valor heurístico, como o número de saltos em o link através da rede entre o emissor e o receptor. As garantias heurísticas que em operação normal, cada nó intermediário armazena apenas uma mensagem por conexão.

Com todas as três técnicas, assume-se que o protocolo, por padrão, usa seus tamanho máximo calculado da janela. O protocolo de *início lento* desenvolvido por Van Jacob-filho também remove essa suposição: o protocolo começa com o tamanho mínimo da janela de um, e só começa a aumentar o tamanho efetivo da janela quando o primeiro reconhecimento foi recebido. No protocolo de início lento, o atraso de ida e volta é contínuo medido excessivamente e, em vez do tempo limite de retransmissão, é usado como uma medida para aumentar ou diminuir o tamanho da janela.

CONTROLE DE TAXA

Com os esquemas de controle de fluxo de janela dinâmica acima, abordamos mais problemas específicos de projeto de rede, que estão fora do alcance deste livro. De uma rede ponto de vista do operador de trabalho, a melhor técnica para evitar o congestionamento é controlar a quantidade de tráfego que *entra* na rede em condições de sobrecarga, em vez de tentando minimizar os danos para o tráfego que já foi aceito, para exemplo, com tempos limite e retransmissões. Esses métodos são chamados coletivamente *métodos de controle de taxa*. A Figura 4.16 mostra uma taxa de transferência bem conhecida versus carga de tráfego

gráfico que ilustra a necessidade de controle de taxa.

Saturação

Taxa de transferência

Carga oferecida

Joelho

Figura 4.16 - Congestionamento da rede

Idealmente, a taxa de transferência da rede aumenta linearmente com a carga oferecida até que está totalmente saturado. Na prática, os algoritmos de controle de rede consomem um pouco do capacidade de rede e um rendimento um pouco menor. Perto da saturação, uma carga crescente oferecida leva a uma degradação crescente do serviço causada pelo congestionamento da rede. O efeito é comparável a uma rodovia movimentada, onde o tráfego

Página 97

86

CONTROLE DE FLUXO
CAPÍTULO 4

chega lentamente a uma paralisação completa sob cargas de pico. O congestionamento, então, é geralmente

definido como uma condição na rede onde um aumento na carga de tráfego causa um diminuição na taxa de transferência. O melhor ponto para operar a rede é à esquerda de a linha tracejada na Figura 4.16, controlando a carga oferecida diretamente com, para exemplo, um método de controle de taxa. Em alguns estudos, descobriu-se que o ponto ideal é no *joelho* da curva na Figura 4.16: o ponto de saturação da rede sob ideal condições. A otimização é então interpretada como a maximização do rendimento dividido pelo atraso medido da mensagem de ida e volta.

O controle de taxa e o controle de fluxo podem ser aplicados independentemente um do outro. Um standard de controle de taxa padrão é dar ao remetente uma *permissão* para oferecer dados à rede em um número médio específico de bytes por segundo. Ele pode especificar dois parâmetros:

A taxa média de dados R em bytes por segundo

O intervalo de média que é usado para calcular R

No protocolo XTP (consulte as Notas Bibliográficas do Capítulo 2), um terceiro parâmetro é usava:

A taxa máxima de burst de dados

O controle de taxa é importante como uma questão de eficiência e controle de rede. Não pode, como-nunca, afetam a consistência lógica de uma definição de protocolo, que é o foco principal deste livro.

4.6 RESUMO

Problemas como os que discutimos neste capítulo foram descobertos em muitos protocolos da vida real, e os designers de protocolo continuarão a ser confrontados com eles uma e outra vez. Nós os apresentamos aqui em sua forma mais básica, para identificar onde estão as possíveis falhas de projeto.

O controle de fluxo e o controle de erros costumam ser difíceis de distinguir. Um esquema de controle de fluxo

pode ser usado para coordenar a taxa de transmissão de mensagens entre os processos em um sistema distribuído. Pode ser usado para evitar gargalos e para se recuperar de erros de transmissão. As estratégias que exploramos incluem o uso de timeouts, o extensão de mensagens com números de sequência, e o uso de positivo e negativo agradecimentos. Uma extensão lógica do mecanismo de controle de fluxo de janela estática é controle de fluxo de janela dinâmico. Ele permite que os protocolos se tornem auto-adaptáveis, um princípio

ple de design de som. Métodos de controle de fluxo podem ser usados para resolver uma variedade de problemas

lems. Eles podem ser usados em um protocolo de ponta a ponta para sincronizar um remetente e um receptor. Eles podem ser usados em protocolos de nível de link para otimizar o gerenciamento de buffer e utilização da largura de banda. Finalmente, eles podem ser usados para evitar congestionamentos específicos

técnicas para combinar a capacidade de um remetente com a capacidade da rede que transporta o transito.

Ao longo deste capítulo, assumimos que um processo receptor pode estabelecer se as mensagens recebidas devem ser reconhecidas e aceitas, ou devem ser rejeitado devido a erros de transmissão. Consulte o Capítulo 3 para ver como isso pode ser

Página 98

CAPÍTULO 4
EXERCÍCIOS

realizado.

EXERCÍCIOS

4-1. Descreva em detalhes as condições sob as quais um protocolo *X-on / X-off* e um *Ping-Pong* (parar e esperar) o protocolo pode falhar.

4-2. Considere a adequação do protocolo de bit alternado sob perda de mensagem, duplicação, e reordenação.

4-3. Altere o protocolo de bit alternado estendido das Figuras 4.13 e 4.14 também enviando uma confirmação negativa quando uma mensagem é recebida com a sequência errada número. Mostre precisamente o que pode dar errado.

4-4. Estenda o protocolo *X-on / X-off* para transmissões full-duplex. Considere o problema extra Parece que a perda de mensagens de controle agora pode causar.

4-5. Mostra o que acontece se o período de tempo limite no protocolo de bits alternados não for escolhido corretamente.

4-6. Se a mensagem de confirmação no protocolo de bit alternado for atrasada o suficiente para desencadear tempo limite do remetente, uma duplicata *MESG* do remetente é criado, que por sua vez aciona uma mensagem de *confirmação* duplicada e assim por diante. Como você mudaria o protocolo para Resolva esse problema?

4-7. Descreva o seu problema de controle de tráfego favorito (por exemplo, bloqueio de rede, problema de direito de passagem lems, rotatórias) como um problema de protocolo.

4-8. Duas divisões de um exército estão acampadas ao sul e ao norte de uma força guerrilheira isso é ligeiramente mais forte do que qualquer uma das duas divisões separadamente. Juntos, no entanto, as duas divisões podem lançar um ataque surpresa e derrotar seus adversários. O problema para eles é coordenar seus planos de forma que nenhum deles ataque por engano sozinho. Isto é decidiu de antemão que a divisão *A* notificará a divisão *B* do plano de ataque, enviando um mensageiro. O mensageiro, porém, deve passar pelo território controlado pela guerrilha para alcançar seu objetivo.

Este "canal de comunicação" entre *A* e *B* deverá ter uma perda substancial taxa e, pelo menos, um potencial de distorção e inserção de mensagem. Assuma essa mensagem distorção pode ser tratada usando técnicas de codificação adequadas. Existe um controle de fluxo problema causado pelo desaparecimento e reaparecimento de mensageiros detidos. Isto é decidiu que, para confirmar a chegada segura de um mensageiro de *A* a *B*, um segundo mensageiro será enviado de *B* para *A* com uma confirmação. Mas, quando a divisão *B* pode ter certeza de que seu reconhecimento chegou? O reconhecimento tem que sobreviver ao mesmo canal comportamento como a mensagem original. Portanto, o próprio reconhecimento deve ser ack-agora limitado. Mas, nesse caso, o reconhecimento de agradecimentos teria que continue *ad infinitum*. Qual é a falha neste raciocínio? (Este é um problema "folk" em teoria do protocolo; por exemplo, veja Bertsekas e Gallager [1987, pp. 28-29].)

4-9. Em um protocolo de janela deslizante onde as mensagens não são aceitas fora de ordem, mostre o que pode acontecer quando o tamanho da janela *W* é igual ao intervalo dos números de sequência *M* (ver Figura 4.11).

4-10. Mostre como você pode reduzir as dimensões de todas as quatro matrizes no protocolo da Figura 4.11 até o tamanho máximo da janela.

4-11. Considere o seguinte problema em um canal que pode reordenar mensagens. Uma mensagem com o número de sequência *N* é enviado e confirmado pelo receptor, mas a confirmação sofre um atraso muito longo no canal. Ocorre um tempo limite e a mensagem numerada *N* é retransmitido. O novo reconhecimento supera o antigo. A janela do protocolo da janela deslizante avança e, após ter avançado um ciclo completo, uma nova mensagem com número de sequência *N* é transmitida. Por esta altura, o antigo reconhecimento finalmente consegue chegar ao remetente e fica confuso com um novo reconhecimento para o último mensagem enviada. Você pode encontrar uma solução para este problema?

4-12. Um método alternativo para o cálculo de um tempo limite de retransmissão, usado no TCP protocolo, é baseado na seguinte fórmula Stallings [1985, p. 508], Zhang [1986], Karn e Partridge [1987]: $\oplus \cdot (\langle \cdot T_{last} \rangle)$.

T

$\oplus \cdot (1 \ominus \langle \cdot \rangle) T_{last}$, onde T_{last} é a última viagem de ida e volta observada demora. Compare este método com o fornecido neste capítulo. Explique o efeito de parâmetros $\langle \cdot \rangle$ e \oplus .

4-13. O protocolo original de bits alternados, mostrado na Figura 4.8, é apenas parcialmente especificado. Forneça as peças que faltam.

4-14. Considere em detalhes o que poderia acontecer se, na Figura 4.12, *recv /p/* fosse redefinido como *falso* no processo de aceitação imediatamente após o envio de uma confirmação.

NOTAS BIBLIOGRÁFICAS

O "protocolo de bit alternado" apresentado neste capítulo é um dos melhores e mais simples

projetos de protocolo documentados e verificados de maneira mais completa. Foi descrito pela primeira vez em

um artigo de três pessoas do National Physical Laboratory in England, Bartlett, Scantlebury e Wilkinson [1969], em resposta a um artigo de WC Lynch [1968].

Variações do protocolo NPL ainda são populares como um teste decisivo para a validação de novos protocolos

métodos de data e especificação. Cerf e Kahn [1974] primeiro estendeu a alternância protocolo de bits em um protocolo de janela deslizante go-back-N. A estratégia de repetição seletiva é devido a Stenning [1976]. A estratégia de confirmação de bloqueio foi descrita pela primeira vez em Brown, Gouda e Miller [1989].

Uma introdução geral às técnicas de controle de fluxo pode ser encontrada, por exemplo, Pouzin [1976], Tanenbaum [1981, 1988] ou Stallings [1985]. Uma excelente pesquisa e comparação de técnicas de controle de fluxo foi publicado em Gerla e Kleinrock [1980]. Uma tentativa inicial de controle de taxa é descrita em Beeforth et al. [1972]. É distingue entre dois tipos de reconhecimento: um reconhece ao remetente que uma mensagem foi recebida corretamente e não precisa ser retransmitida, e outro sinal informa ao remetente que o espaço do buffer ocupado por essa mensagem foi liberado (por exemplo, porque o pacote foi encaminhado), e que a janela de números de sequência pode avançar um degrau.

Várias versões da Figura 4.16 foram publicadas ao longo dos anos. É discutido em detalhes em, por exemplo, Gerla e Kleinrock [1980] e Jain [1986].

O XTP, ou Express Transfer Protocol é descrito em Chesson [1987]. O protocolo foi projetado para sobreviver a aplicativos em redes de dados de alta velocidade. É promovido por a empresa "Protocol Engines", fundada por Greg Chesson. Outro trabalho importante sobre protocolos para redes de dados de alta velocidade é relatado em Clark [1985] e Clark,

Página 100

CAPÍTULO 4
NOTAS BIBLIOGRÁFICAS
89

Lambert e Zhang [1988]. Os métodos de controle de fluxo de janela dinâmica são descritos em, por exemplo, Gerla e Kleinrock [1980], Jain [1986]. Protocolo de início lento de Jacobson é descrito em Jacobson [1988].

Mais sobre a escolha de intervalos de tempo limite para protocolos de rede podem ser encontrados em Zhang

[1986] e Karn e Partridge [1987]. Para uma introdução ao controle geral de rede questões referem-se a McQuillan e Walden [1977], Tanenbaum [1981, 1988], Cole [1987], ou Stallings [1985, 1988].

Página 101

MODELOS DE VALIDAÇÃO 5

90 Introdução 5.1
91 Processos, Canais, Variáveis 5.2
91 Executabilidade das Declarações 5.3
92 Variáveis e Tipos de Dados 5.4
93 Tipos de Processo 5.5
96 Canais de Mensagem 5.6
100 Controle de Fluxo 5.7
102 Exemplos 5.8
104 Procedimentos de Modelagem e Recursão 5.9
104 Definições de Tipo de Mensagem 5.10
105 Tempo Limites de Modelagem 5.11
106 Protocolo de Lynch revisitado 5.12
107 Resumo 5.13
108 exercícios
109 Notas Bibliográficas

5.1 INTRODUÇÃO

No Capítulo 2, discutimos os cinco elementos principais de uma definição de protocolo: um serviço especificação, suposições explícitas sobre o ambiente, o vocabulário do protocolo, definições de formato e regras de procedimento. A maioria desses elementos pode ser estruturada em um forma hierárquica. A especificação do serviço, por exemplo, pode ser dividida em camadas, cada nova camada construída sobre as que estão abaixo dela e fornecendo um nível superior serviço ao usuário. Para realizar o serviço em uma determinada camada, um conjunto consistente de

as regras de transferência devem ser derivadas e descritas em alguma linguagem formal. O design de um conjunto completo e consistente de regras de procedimento, no entanto, é um dos problemas mais difíceis lems no projeto do protocolo.

UMA LINGUAGEM DE VALIDAÇÃO DE PROTOCOLO

Neste capítulo, introduzimos uma notação para a especificação e verificação de pro- regras de cedência. Uma vez que o foco está nas regras de procedimento, essas especificações fornecem apenas uma descrição parcial de um protocolo. Chamamos essa descrição parcial de protocolo *modelo de validação*. A linguagem que usamos para descrever os modelos de validação é chamada PROMELA.

Nosso objetivo é modelar protocolos o mais sucintamente possível, a fim de ser capaz de estudar a sua estrutura e verificar a sua integridade e consistência lógica. Fazendo isso, nós abstrair deliberadamente de outras questões de design de protocolo, como formato de mensagem. UMA modelo de validação define as interações de processos em um sistema distribuído. Faz não resolve os detalhes da implementação. Não diz como uma mensagem deve ser transmitida ented, codificado ou armazenado. Ao simplificar o problema desta forma, podemos isolar e 90

Página 102

SEÇÃO 5.3 EXECUTABILIDADE DE DECLARAÇÕES

91

concentre-se na parte mais difícil: a concepção de um conjunto completo e consistente de regras para governar as interações em um sistema distribuído.

Este capítulo fornece uma introdução ao uso de PROMELA para especificar o sistema comportamento em modelos de validação formal. O próximo capítulo discute métodos específicos para definir os critérios precisos de correção que podem ser aplicados à validação de esses modelos. Um breve manual de referência para PROMELA pode ser encontrado no Apêndice C. O Capítulo 7 dá um exemplo de uma aplicação séria da PROMELA no projeto de um arquivo protocolo de transferência. Capítulo 12 discute o projeto de um intérprete / simulador para o linguagem, e o Capítulo 13 discute como este software pode ser estendido com um analisador automatizado para modelos PROMELA .

5.2 PROCESSOS, CANAIS, VARIÁVEIS

Descrevemos as regras de procedimento como programas formais para um modelo abstrato de um sistema. Claro, queremos que este modelo seja o mais simples possível, mas suficientemente poderoso para representar todos os tipos de problemas de coordenação que podem ocorrer em sistemas. No que diz respeito ao poder descritivo, bastaria definir apenas um tipo de objeto: a máquina de estado finito. A máquina de estado pode modelar todos os outros objetos que possamos estar interessados, incluindo variáveis finitas e canais de mensagem (filas FIFO limitadas). Embora tal modelo possa ser suficiente, não é muito complicado

veniente para trabalhar. Portanto, definimos modelos de validação diretamente em termos de três tipos específicos de objetos:

processos

canais de mensagem

variáveis de estado

Para fins de análise, cada um desses objetos pode ser traduzido em um estado finito máquina por um processo de tradução simples que é considerado no Capítulo 8. Por enquanto, no entanto, podemos fingir ter o luxo de trabalhar diretamente com esses objetos de nível. Todos os processos são, por definição, objetos globais. Variáveis e canais representam dados que podem ser globais ou locais para um processo.

5.3 EXECUTABILIDADE DE DECLARAÇÕES

Na PROMELA não há diferença entre condições e afirmações. Mesmo isolado

As condições booleanas podem ser usadas como instruções. A execução de uma declaração é condi- internacional em sua *executabilidade*. Todas as instruções PROMELA são executáveis ou bloqueadas, dependendo dos valores atuais das variáveis ou do conteúdo dos canais de mensagem.

A executabilidade é o meio básico de sincronização. Um processo pode esperar por um evento acontecer ao esperar que uma instrução se torne executável. Por exemplo, em vez de escrevendo um loop de espera ocupado:

```
enquanto (a! = b) pula  
/* aguarde a == b */
```

podemos obter o mesmo efeito na PROMELA com a declaração
(a == b)

A condição só pode ser executada (aprovada) se for mantida. Se a condição não

92

MODELOS DE VALIDAÇÃO

CAPÍTULO 5

segure, blocos de execução até que isso aconteça. Operadores aritméticos e booleanos em condições como estes são os mesmos que em C. Como veremos abaixo, as atribuições às variáveis são sempre executável.

5.4 VARIÁVEIS E TIPOS DE DADOS

As variáveis no PROMELA são usadas para armazenar informações globais sobre o sistema como um todo ou informação que é local para um processo específico, dependendo de onde a declaração para a variável é colocada. Uma variável pode ser uma das seguintes seis tipos de dados predefinidos:

bit , bool , byte , short , int , chan .

Os primeiros cinco tipos nesta lista são chamados de tipos de dados básicos. Eles são usados para especificar

objetos que podem conter um único valor por vez. O sexto tipo especifica o canal da mensagem nels. Um canal de mensagem é um objeto que pode armazenar uma série de valores, agrupados em estruturas definidas pelo usuário. Discutimos os tipos de dados básicos primeiro. Canais de mensagem são

discutido separadamente na Seção 5.6.

As declarações

```
bool  
bandeira;  
int  
Estado;  
byte  
msg;
```

definir variáveis que podem armazenar valores inteiros em três intervalos diferentes. O escopo de a variável é global se for declarada fora de todas as declarações de processo e local se for declarado em uma declaração de processo. A Tabela 5.1 resume os tipos de dados básicos, tamanhos e os intervalos de valores correspondentes em computadores DEC / VAX.

Tabela 5.1 - Tipos de dados básicos

Nome	Tamanho (bits)	Uso	Alcance
mordeu			
1			
não assinado			
0..1			
bool			
1			
não assinado			
0..1			
byte			
8			
não assinado			
0..255			
curto			
16			
assinado			
-2 ¹⁵ ..2 ¹⁵ -1			
int			
32			
assinado			
-2 ³¹ ..2 ³¹ -1			

Os nomes bit e bool são sinônimos para um único bit de informação. Um byte é um quantidade não assinado que pode armazenar um valor entre 0 e 255. curtas e int s são quantidades assinadas que diferem apenas na faixa de valores que podem conter.

ARRAYS

As variáveis podem ser declaradas como arrays. Por exemplo,
estado do byte [N]
declara uma matriz de N bytes que pode ser acessada em instruções como

Página 104

SEÇÃO 5.5
TIPOS DE PROCESSO

93

estado [0] = estado [3] + 5 * estado [3 * 2 / n]

onde n é uma constante ou variável declarada em outro lugar. O índice de uma matriz pode ser qualquer expressão que determina um valor inteiro único. O intervalo válido de índices é 0 .. N-1 . O efeito do uso de um valor de índice fora desse intervalo é indefinido; a maioria provavelmente causará um erro de tempo de execução.

Até agora vimos exemplos de uma declaração de variável e de dois tipos básicos de declarações: condições booleanas e atribuições. Declarações e atribuições são sempre executável .

5.5 TIPOS DE PROCESSO

Para executar um processo, temos que ser capazes de nomeá-lo, definir seu tipo e instanciá-lo. Vejamos primeiro a definição e nomenclatura dos processos. Todos os tipos de processos que podem ser instanciados são definidos em declarações proctype . O seguinte, para

instância, declara um processo com uma variável local chamada estado .

proctipo A () {estado do byte; estado = 3}

O tipo de processo é chamado A . O corpo da declaração, entre parênteses, consiste em declarações e declarações de variáveis locais ou canais. A declaração acima contém uma declaração de variável local e uma única instrução: uma atribuição de o valor 3 para o estado variável .

O ponto e vírgula é um *separador de instrução* (não um terminador de instrução, portanto, não há ponto e vírgula após a última declaração). PROMELA define duas instruções diferentes separadores: uma seta, -> , e um ponto e vírgula , ; . Os dois separadores são equivalentes. A flecha às vezes é usado como uma forma informal para indicar uma relação causal entre dois estados mentos. Considere o seguinte exemplo.

```
estado do byte = 2;  
proctipo A () {(estado == 1) -> estado = 3}  
proctipo B () {estado = estado - 1}
```

Neste exemplo nós declaramos dois tipos de processo, A e B . O estado variável é agora um global, inicializado com o valor 2. O tipo de processo A contém duas instruções, separadas por um seta. A declaração do tipo de processo B contém uma única instrução que diminui o valor da variável de estado por 1. Uma vez que a atribuição é sempre executável, os processos do tipo B sempre pode terminar sem demora. Processos do tipo A , no entanto, são atrasado até que o estado da variável contenha o valor adequado.

O PROCESSO INICIAL

Uma definição de proctype apenas declara o comportamento do processo, ela não o executa. Inicialmente, apenas um processo é executado: um processo do tipo init que deve ser declarado explicitamente em todas as especificações PROMELA . O processo init é comparável ao função main () de um programa C padrão. O menor PROMELA possível especificação é

Página 105

94
MODELOS DE VALIDAÇÃO
CAPÍTULO 5
init {skip}

onde skip é uma instrução nula. Apenas um pouco mais complicado é o PROMELA equivalente ao famoso programa " hello world " de C:

```
init {printf ("olá mundo \ n")}
```

Mais interessante, no entanto, o processo inicial pode inicializar variáveis globais, criar canais de mensagens e processos de instância. Uma declaração init para os dois processos sistema na Seção 5.5, por exemplo, pode ser o seguinte.

```
init {executar A (); executar B ()}
```

Este processo init inicia dois processos, que serão executados simultaneamente com o init processo a partir de então. No caso acima, o processo init termina após iniciar o segundo processo, mas não precisa ser assim. Run é um operador unário que instancia um

cópia de um determinado tipo de processo (por exemplo, A). Não espera que o processo termine minate. A instrução run é executável e retorna um resultado positivo apenas se o processo pode ser efetivamente instanciado. Não é executável e retorna zero se isso não puder ser feito, por exemplo, se muitos processos já estiverem em execução. Desde PROMELA modelos de sistemas de estado finito, o número de processos e canais de mensagem é sempre limitado. O valor preciso do limite depende do hardware e, portanto, indefinido na PROMELA . O valor retornado pela execução é um número de processo em tempo de execução, ou

pid . Como run é definido como um operador, run A () é uma expressão que pode ser embutido em outras expressões. Seria, portanto, válido, embora talvez não também útil, para usá-lo em uma expressão composta, como

```
i = executar A () && (executar B () || executar C ())
```

Uma vez que a comunicação entre os processos é definida em canais nomeados, o processo números (pids) são geralmente irrelevantes. Há uma exceção importante que descartamos cuss no Capítulo 6, Seção 6.7.

O operador run pode passar valores de parâmetro para o novo processo, por exemplo, como segue baixos:

```
proctipo A (estado do byte; conjunto curto)
{
(estado == 1) -> estado = conjunto
}
```

```
init {executar A (1, 3)}
```

Apenas canais de mensagens, discutidos posteriormente, e instâncias dos cinco tipos de dados básicos podem

ser passados como parâmetros. Matrizes e tipos de processo não podem ser transmitidos.

Executar pode ser usado em qualquer processo para gerar novos processos, não apenas no processo inicial.

Um processo em execução desaparece quando termina (ou seja, atinge o final do corpo de sua declaração de tipo de processo), mas não antes de todos os processos que instanciou (seus " filhos ") foram encerrados primeiro.

Voltando ao exemplo anterior, observe que usando run , podemos criar qualquer número de exemplares dos tipos de processo A e B . Se, no entanto, mais de um processo simultâneo é

Página 106

SEÇÃO 5.5
TIPOS DE PROCESSO
95

permitti ler e escrever o valor de uma variável global um conjunto bem conhecido de podem ocorrer problemas (ver notas bibliográficas). Considere, por exemplo, o seguinte sistema de dois processos, compartilhando acesso ao estado da variável global .

```
estado do byte = 1;
proctipo A () {(estado == 1) -> estado = estado + 1}
proctipo B () {(estado == 1) -> estado = estado - 1}
init {executar A (); executar B ()}
```

Se um dos dois processos termina antes de seu concorrente ter iniciado, o outro processo será bloqueado para sempre na condição inicial. Se ambos passarem na condição simultaneamente- Obviamente, ambos podem terminar, mas o valor de estado resultante é imprevisível. Pode ser 0, 1 ou 2.

Muitas soluções para este problema têm sido consideradas, desde a abolição do variáveis globais para o fornecimento de instruções de máquina especiais que podem garantir um seqüência de teste e conjunto indivisível em uma variável compartilhada. O exemplo abaixo foi um dos as primeiras soluções publicadas. É devido ao matemático holandês T. Dekker. isto concede a dois processos acesso mutuamente exclusivo a uma seção crítica arbitrária em seu código, manipulando três variáveis de estado globais. As primeiras quatro linhas da PROMELA especificação abaixo são definições de macro de estilo C. As duas primeiras macros definem verdadeiro para ser um valor constante igual a 1 e falso para ser uma constante 0. Da mesma forma, Aturn e Bturn são definidos como constantes booleanas.

```
1 #define true
1
2 #define false
0
3 # define Aturn
1
4 #define Bturn
0
```

```

5
6 bool x, y, t;
7
8 proctipo A ()
9 {x = verdadeiro;
10
t = Bturn;
11
(y == falso || t == Aturn);
12
/* seção Crítica */
13
x = falso
14}
15 proctipo B ()
16 {y = verdadeiro;
17
t = Aturn;
18
(x == falso || t == Bturn);
19
/* seção Crítica */
20
y = falso
21}
22 init {executar A (); executar B ()}

```

As condições nas linhas 11 e 18 são usadas para sincronizar os processos. Eles podem só ser executado se eles segurar. O algoritmo pode ser executado repetidamente e é

Página 107

96
MODELOS DE VALIDAÇÃO
CAPÍTULO 5

independente das velocidades relativas dos dois processos.

SEQUÊNCIAS ATÔMICAS

Na PROMELA, há outra maneira de evitar o problema de *teste e ajuste : atômica* sequências. Uma sequência de instruções entre parênteses prefixadas com a chave palavra atômica indica que a sequência deve ser executada como uma unidade indivisível, não intercalado com quaisquer outros processos. É um erro se houver qualquer afirmação, exceto o primeiro, pode bloquear em uma sequência atômica. O processo de execução será abortado em Aquele caso. Aqui está o exemplo anterior, reescrito com duas sequências atômicas.

```

estado do byte = 1;
proctipo A () {atômico {(estado == 1) -> estado = estado + 1}}
proctipo B () {atômico {(estado == 1) -> estado = estado - 1}}
init {executar A (); executar B ()}

```

Neste caso, o valor final do estado é 0 ou 2, dependendo de qual processo executa cutes. O outro processo será bloqueado para sempre.

As sequências atômicas podem ser uma ferramenta importante na redução da complexidade de uma validação

modelo. Uma sequência atômica restringe a quantidade de intercalação permitida que pode efetivamente tornar tratáveis modelos de validação complexos, sem perda de generalidade.

O exemplo abaixo ilustra isso.

```

proctype nr (short pid, a, b)
{
int res;
atômica {
res = (a * a + b) / 2 * a;
printf ("resultado% d:% d \n", pid, res)
}
}
init {run nr (1,1,1); run nr (1,2,2); executar nr (1,3,2)}

```

O processo init inicia três cópias do tipo de processo nr . Cada processo com puta algum número e imprime. As manipulações das variáveis dentro destes os processos são todos locais e não podem afetar o comportamento dos outros processos. Definindo o corpo do processo como uma sequência atômica reduz drasticamente o número de casos que precisariam ser considerados em uma validação (Capítulo 11), sem alterar os possíveis comportamentos dos processos de alguma forma. Geralmente é trivial identificar sequências de instruções que podem ser reescritas com sequências atômicas.

5.6 CANAIS DE MENSAGEM

Os canais de mensagens são usados para modelar a transferência de dados de um processo para outro. Eles são declarados local ou globalmente, assim como variáveis dos tipos de dados básicos,

usando a palavra-chave `chan`. Por exemplo,
`chan a, b; chan c [3]`
declara os nomes `a`, `b`, e `c` como identificadores de canal, o último como uma matriz. A declaração `nel` também pode ter um campo inicializador:

Página 108

SEÇÃO 5.6
CANAIS DE MENSAGEM
97

`chan a = [16] de {short}`
inicializa o canal `a`. O inicializador diz que o canal pode armazenar até 16 mensagens do tipo `curto`. Similarmente,
`chan c [3] = [4] de {byte}`
inicializa uma matriz de 3 canais, cada um com uma capacidade de 4 slots de mensagem, cada slot consistindo em um campo de mensagem do tipo `byte`.
Caso as mensagens a serem passadas pelo canal possuam mais de um campo, a declaração parece o seguinte:
`chan qname = [16] de {byte, int, chan, byte}`
Desta vez, definimos um único canal que pode armazenar até 16 mensagens, cada consistindo em 4 campos: um valor de 8 bits, um valor de 32 bits, um nome de canal e outro de 8 bits valor.
A declaração
`qname! expr`
envia o valor da expressão `expr` para o canal que acabamos de criar, ou seja, acrescenta o valor para a cauda do canal.
`qname? msg`
recupera uma mensagem do chefe do canal e a armazena na variável `msg`.
Os canais transmitem mensagens na ordem do primeiro a entrar, primeiro a sair. Nos casos acima, apenas um único valor é passado pelo canal. Se vários valores forem transferidos por mensagem, eles são especificados em uma lista separada por vírgulas
`qname! expr1, expr2, expr3`
`qname? var1, var2, var3`
Se mais parâmetros são enviados por mensagem do que o canal de mensagens pode armazenar, os parâmetros redundantes são perdidos. Se menos parâmetros forem enviados, o canal de mensagem pode armazenar, o valor dos parâmetros restantes é indefinido. Da mesma forma, se o receber operação tenta recuperar mais parâmetros do que os disponíveis, o valor do extra parâmetros é indefinido; se receber menos do que o número de parâmetros que foi enviada, a informação extra é perdida.
Por convenção, o primeiro campo de mensagem é frequentemente usado para especificar o tipo de mensagem (um constante). Uma notação alternativa e equivalente para as operações de envio e recebimento é, portanto, especificar o tipo de mensagem, seguido por uma lista de campos de mensagem incluídos em parênteses. Em geral:
`qname! expr1 (expr2, expr3)`
`qname? var1 (var2, var3)`
A operação de envio é executável apenas quando o canal endereçado não está cheio. A operação de recepção, da mesma forma, só é executável quando o canal não está vazio.

Página 109

98
MODELOS DE VALIDAÇÃO
CAPÍTULO 5

Opcionalmente, alguns dos argumentos na operação de recebimento podem ser constantes:
`qname? cons1, var2, cons2`
Neste caso, uma outra condição sobre a executabilidade da operação de recebimento é que o valor de todos os campos da mensagem que são especificados como constantes correspondem ao valor do campos correspondentes na mensagem que está na cabeça do canal.
Aqui está um exemplo que usa alguns dos mecanismos apresentados até agora.
`proctipo A (chan q1)
{
chan q2;
q1? q2;
q2! 123
}`

```

proctipo B (chan qforb)
{
int x;
qforb? x;
printf ("x =% d \ n", x)
}
iniciar
{
chan qname [2] = [1] de {chan};
chan qforb = [1] de {int};
execute A (qname [0]);
execução B (qforb);
qname [0]! qforb
}

```

Observe que o canal `qforb` não é declarado como uma matriz e, portanto, não precisa de um índice na operação de envio no final do processo inicial. O valor impresso pelo o processo do tipo `B` será 123.

Um operador unário predefinido `len (qname)` leva o nome de um canal `qname` como um operando e retorna o número de mensagens que ele contém atualmente. Observe que se `len` é usado como uma condição, ao invés do lado direito de uma atribuição, não é executado capaz se o canal estiver vazio: retorna um resultado zero, o que por definição significa que a declaração é temporariamente não executável.

As operações de envio e recebimento não podem ser avaliadas sem potenciais efeitos colaterais. Com condições postas como

```

(qname? var == 0)
ou
(a> b && qname! 123)

```

são, portanto, inválidos na PROMELA. Para a operação de recebimento, no entanto, há um notação alternativa, usando colchetes ao redor da cláusula atrás da pergunta marca. Por exemplo,

```
qname? [ack, var]
```

é avaliada como uma condição e pode ser combinada com outras expressões booleanas. isto retorna um resultado positivo (1) se a instrução de recebimento correspondente

Página 110

SEÇÃO 5.6
CANAIS DE MENSAGEM

99
qname? ack, var

seria executável, ou seja, se realmente houver um reconhecimento de mensagem no cabeçalho do canal.

Caso contrário, retorna zero. Não tem efeito colateral; especificamente, não remove o mensagem do canal.

Observe cuidadosamente que em sequências não atômicas de duas declarações, como

```
(len (qname)> 0) -> qname? msgtype
```

ou

```
qname? [msgtype] -> qname? msgtype
```

a segunda instrução não é necessariamente executável após a primeira ter sido executada cortado. Pode haver condições de corrida se o acesso aos canais for compartilhado entre vários processos. Em ambos os casos, um segundo processo pode roubar a mensagem logo após o o atual determinou sua presença. PROMELA não impede, e de fato não pode impedir o usuário de escrever essas especificações. Pelo contrário, estes são precisamente os tipos de problemas que queremos modelar em nossa linguagem de validação.

COMUNICAÇÃO RENDEZVOUS

Até agora, falamos sobre comunicação assíncrona entre processos via canais de mensagens criados em declarações como

```
chan qname = [N] de {byte}
```

onde `N` é uma constante positiva que define o tamanho do buffer. Usando um tamanho de canal de zero, como em

```
porta chan = [0] de {byte}
```

define uma porta de encontro que só pode passar, e não armazenar, mensagens de byte único.

As interações de mensagens por meio dessas portas de encontro são síncronas, por definição. Vigarista- Considere o seguinte exemplo:

```
# define msgtype 33
nome do chan = [0] de {byte, byte};
nome do byte;
proctipo A ()
```

```

{
  nome! msgtype (124);
  nome! msgtype (121)
}
proctipo B ()
{
  estado de byte;
  nome? msgtype (estado)
}
iniciar
{
  atômico {run A (); executar B ()}
}

```

Página 111

100
MODELOS DE VALIDAÇÃO
CAPÍTULO 5

As duas instruções de execução são colocadas em uma sequência atômica para garantir que as duas os processos começam simultaneamente. Claro, eles não precisam terminar simultaneamente, e eles não precisam ser executados até a conclusão antes que a sequência atômica termine.

O nome do canal é uma porta de encontro global. Os dois processos são executados de forma síncrona sua primeira declaração: um aperto de mão na mensagem `msgtype` e uma transferência do valor 124 para o `estado` da variável local . A segunda declaração no processo A não é executável, porque não há correspondência operação de recebimento no processo B .

Se o nome do canal for definido com uma capacidade de buffer diferente de zero, o comportamento é diferente.

Se o tamanho do buffer for de pelo menos dois, o processo do tipo A pode completar sua execução antes mesmo de seu par começar. Se o tamanho do buffer for um, a sequência de eventos é tão segue. O processo do tipo A pode completar sua primeira ação de envio, mas bloqueia no segundo, porque o canal agora está cheio. O processo do tipo B pode então recuperar a primeira mensagem e termine. Neste ponto, A torna-se executável novamente e termina, deixando sua última mensagem como um resíduo no canal.

As portas síncronas podem ser declaradas como arrays, assim como os canais assíncronos. Ren-
A comunicação dezvoux é binária: apenas dois processos, um emissor e um receptor, podem ser sincronizado desta forma. Veremos um exemplo de uma maneira de explorar isso para construir um semáforo abaixo. Mas, primeiro, vamos apresentar mais algumas estruturas de fluxo de controle.

5.7 FLUXO DE CONTROLE

Nas entrelinhas, já apresentamos três maneiras de definir o fluxo de controle:
concatenação de instruções dentro de um processo, execução paralela de processos e sequências atômicas. Existem três outras construções de fluxo de controle no PROMELA para serem discutido:

Seleção de caso

Repetição

Saltos incondicionais

SELEÇÃO DE CASO

A construção mais simples é a estrutura de seleção. Usando os valores relativos de dois variáveis a e b para escolher entre duas opções, por exemplo, podemos escrever

```

E se
:: (a! = b) -> opção1
:: (a == b) -> opção2
fi

```

A estrutura de seleção contém duas sequências de execução, cada uma precedida por um duplo colón. Apenas uma sequência da lista é executada. Uma sequência pode ser selecionada apenas se sua primeira instrução for executável. A primeira afirmação é, portanto, chamada de *guarda* . No exemplo acima, os guardas são mutuamente exclusivos, mas não precisam ser. Se mais do que um guarda é executável, uma das sequências correspondentes é selecionada em dom. Se todos os guardas não forem executáveis, o processo será bloqueado até que pelo menos um deles possa

ser selecionado. Não há restrição sobre o tipo de declaração que pode ser usada como um guarda. O exemplo a seguir usa instruções de entrada:

Página 112

SEÇÃO 5.7

```

CONTROLE DE FLUXO
101
#define um 1
#define b 2
chan ch = [1] de {byte};
proctipo A () {ch! a}
proctipo B () {ch! b}
proctipo C ()
{
E se
:: ch? a
:: ch? b
fi
}
init {atômico {executar A (); executar B (); executar C ()}}

```

Este exemplo define três processos e um canal. A primeira opção na seleção estrutura de ção do processo do tipo C é executável se o canal contém uma mensagem a , onde a é uma constante com valor 1, conforme definido em uma definição de macro no início de o programa. A segunda opção é executável se contiver uma mensagem b , onde b é um constante. Qual mensagem estará disponível depende das velocidades relativas do processos.

Um processo do seguinte tipo aumenta ou diminui o valor da variável

```

conte uma vez.
contagem de bytes;
contador proctype ()
{
E se
:: contagem = contagem + 1
:: contagem = contagem - 1
fi
}

```

REPETIÇÃO

Uma extensão lógica da estrutura de seleção é a estrutura de repetição. Podemos modificar o programa acima para obter um programa cíclico que aumenta aleatoriamente ou diminui a variável.

```

contagem de bytes;
contador proctype ()
{
Faz
:: contagem = contagem + 1
:: contagem = contagem - 1
:: (contagem == 0) -> pausa
od
}

```

Apenas uma opção pode ser selecionada para execução por vez. Depois que a opção for concluída, a execução da estrutura é repetida. A maneira normal de terminar a repetição estrutura é com uma instrução break . No exemplo, o loop pode ser quebrado quando o

102
MODELOS DE VALIDAÇÃO
CAPÍTULO 5

a contagem chega a zero. Não é necessário terminar, pois as outras duas opções permanecem em execução

capaz. Para forçar o encerramento, poderíamos modificar o programa da seguinte maneira:

```

contador proctype ()
{
Faz
:: (contagem! = 0) ->
E se
:: contagem = contagem + 1
:: contagem = contagem - 1
fi
:: (contagem == 0) -> pausa
od
}

```

SALTOS

Outra maneira de quebrar o loop é com um salto incondicional: o infame goto declaração. Isso é ilustrado na seguinte implementação do algoritmo de Euclides para encontrar o maior divisor comum de dois números positivos:

```

proctipo Euclides (int x, y)
{

```

```

Faz
:: (x> y) -> x = x - y
:: (x <y) -> y = y - x
:: (x == y) -> ir para pronto
od;
feito:
pular
}

```

O goto neste exemplo salta para um rótulo denominado done . Um rótulo só pode aparecer antes de uma declaração. Acima, queremos pular para o final do programa. Neste caso, um salto de instrução fictícia é útil: é um marcador de posição que sempre é executável e tem nenhum efeito. A instrução goto em si é sempre executável.

5.8 EXEMPLOS

O exemplo a seguir especifica um filtro que recebe mensagens de um canal em e divide-os em dois canais grandes e pequenos, dependendo dos valores anexados. A constante N é definida como 128, e o tamanho é definido como 16 em duas macro definições.

```

#define N
128
#define o tamanho 16
chan em
= [tamanho] de {curto};
chan grande = [tamanho] de {curto};
chan pequeno = [tamanho] de {curto};

```

SEÇÃO 5.8
EXEMPLOS

```

103
protoype split ()
{
carga curta;
Faz
:: na? carga ->
E se
:: (carga >= N) -> grande! carga
:: (carga <N) -> pequena! carga
fi
od
}
init {executar divisão ()}

```

Um tipo de processo que mescla os dois fluxos de volta em um, provavelmente em um ordem, e escreve-lo de volta para o canal em Pode ser especificado como

```

protoype merge ()
{
carga curta;
Faz
::
E se
:: grande? carga
:: pequena? carga
fi;
na! carga
od
}

```

Com a seguinte modificação no processo init , os processos de divisão e fusão podem desempenhar ativamente suas funções para sempre.

```

iniciar
{
em! 345; em! 12; em! 6777; em! 32; em! 0;
execute split (); execute merge ()
}

```

Como um exemplo final, considere a seguinte implementação de um semáforo Dijkstra, usando comunicação binária de rendezvous.

```

#define p
0
#define v
1
chan sema = [0] de {bit};
protoype dijkstra ()
{
Faz
:: sema! p -> sema? v
od

```

}

```
104
MODELOS DE VALIDAÇÃO
CAPÍTULO 5
usuário proctype ()
{
    sema? p;
    /* seção Crítica */
    sema! v
    /* seção não crítica */
}
iniciar
{
atômica {
    execute dijkstra ();
    executar usuário (); executar usuário (); run user ()
}
}
```

O semáforo garante que apenas um processo do usuário pode entrar em sua seção crítica em um Tempo. No exemplo, cada processo do usuário acessa sua seção crítica apenas uma vez. E se acesso repetido pode ser solicitado, o semáforo não necessariamente impediria um processo de monopolizar o acesso à seção crítica.

5.9 PROCEDIMENTOS DE MODELAGEM E RECURSÃO

Os procedimentos, mesmo os recursivos, podem ser modelados como processos. O valor de retorno pode ser passado de volta ao processo de chamada por meio de uma variável global ou por meio de uma mensagem. O seguinte

O programa abaixo ilustra isso.

```
fato de proctype (int n; chan p)
{
    resultado interno;
    E se
        :: (n <= 1) -> p! 1
        :: (n >= 2) ->
            filho chan = [1] de {int};
            executar fato (n-1, filho);
            filho? resultado;
            p! n * resultado
        fi
    }
    iniciar
    {
        resultado interno;
        filho chan = [1] de {int};
        executar fato (7, criança);
        filho? resultado;
        printf ("resultado:% d \ n", resultado)
    }
}
```

O fato de processo (n, p) calcula recursivamente o factorial de n , comunicando o resultado para seu processo pai através do canal p .

5.10 DEFINIÇÕES DE TIPO DE MENSAGEM

Vimos como as constantes podem ser definidas usando macros de estilo C. Como uma forma leve de açúcar sintático, PROMELA permite definições de tipo de mensagem do formulário

```
mtype = {ack, nak, err, próximo, aceitar}
```

SEÇÃO 5.11
TEMPO LIMITE DE MODELAGEM
105

A definição é equivalente à seguinte sequência de definições de macro.

```
#define ack
1
#define nak
2
#define err
3
#define próximo
4
#define aceitar 5
```

Uma definição formal de tipo de mensagem é a maneira preferida de especificar os tipos de mensagem uma vez que adia qualquer decisão sobre os valores específicos a serem usados. Ao mesmo tempo,

torna os nomes das constantes, ao invés dos valores, disponíveis para uma implementação , o que pode melhorar o relatório de erros. Só pode haver um tipo de mensagem definição por especificação.

5.11 TEMPO LIMITE DE MODELAGEM

Já discutimos dois tipos de declarações com um significado predefinido em PROMELA : pular e quebrar . Outra instrução predefinida é o tempo limite . O tempo limite declaração permite que um processo aborte a espera por uma condição que não pode mais torna-se verdade, por exemplo, uma entrada de um canal vazio. O tempo limite fornece um escapar de um estado de suspensão. Pode ser considerada uma condição artificial predefinida que torna-se verdadeiro apenas quando nenhuma outra instrução no sistema distribuído é executável. Observe que ele não carrega nenhum valor: ele não especifica um intervalo de tempo limite, mas um tempo limite possibilidade. Nós deliberadamente abstraímos de considerações de tempo absolutas, o que é crucial no trabalho de validação, e não especificamos como o tempo limite deve ser implementado mentado. Um exemplo simples é o seguinte processo que envia uma mensagem de redefinição para um canal denominado guarda sempre que o sistema pára.

```
proctype watchdog ()  
{  
Faz  
:: timeout -> guard! reset  
od  
}
```

O tempo limite, conforme definido aqui, não modela erros causados por tempos limite prematuros em um sistema real. Se for necessário, pode ser alcançado redefinindo a palavra-chave em um macro, por exemplo como segue.

```
#define timeout 1  
/* sempre ativado, atraso arbitrário */  
Mais exemplos são fornecidos no Capítulo 7.
```

TIPOS DE DECLARAÇÃO

Com exceção de declarações assert e primitivas de reivindicação temporal (ver Capítulo 6) agora discutimos todos os tipos básicos de declarações definidas na PROMELA :

Atribuições e condições

Seleções e repetições

Enviar e receber

Declarações Goto e Break

Tempo esgotado

Página 117

106

MODELOS DE VALIDAÇÃO

CAPÍTULO 5

Observe que run e len não são instruções, mas operadores unários que podem ser usados em atribuições e condições.

A instrução skip foi introduzida como um preenchimento para satisfazer os requisitos de sintaxe. Não é formalmente parte da linguagem, mas uma *pseudo-declaração* , um sinônimo para outro estado com o mesmo efeito. Trivialmente, o salto é equivalente à condição (1) ; isto é sempre executável e não tem efeito.

5.12 PROTOCOLO DE LYNCH REVISITADO

Agora que temos uma linguagem, vamos tentar descrever o protocolo de exemplo de Capítulo 2. A versão abaixo é baseada nas Figuras 2.1 e 2.3, com a extensão de teste para aceitar mensagens e para inicializar a transferência de dados discutida na Seção 2.4.

```
mtype = {ack, nak, err, próximo, aceitar}  
transferência de proctipo (chan in, out, chin, chout)  
{  
byte o, i;  
em? próximo (o);  
Faz  
:: chin? nak (i) -> out! accept (i); chout! ack (o)  
:: chin? ack (i) -> out! accept (i); em? próximo (o); chout! ack (o)  
:: queixo? err (i) -> chout! nak (o)  
od  
}  
iniciar  
{  
chan AtoB = [1] de {byte, byte};  
chan BtoA = [1] de {byte, byte};  
chan Ain = [2] de {byte, byte};
```

```

chan Bin = [2] de {byte, byte};
chan Aout = [2] de {byte, byte};
chan Bout = [2] de {byte, byte};
atômica {
transferência de corrida (Ain, Aout, AtoB, BtoA);
executar a transferência (Bin, Bout, BtoA, AtoB)
};
AtoB! Err (0)
}

```

Os canais `Ain` e `Bin` devem ser preenchidos com mensagens de token do tipo `next` e arquivos variáveis (por exemplo, valores de caracteres ASCII) por processos de fundo não especificados: os usuários do serviço de transferência. Da mesma forma, esses processos do usuário podem ler os dados recebidos

dos canais `Aout` e `Bout`. Os processos são inicializados em um estado atômico e começou com a mensagem de `erro` fictício.

Como último exemplo, abaixo está uma lista no PROMELA de um procedimento extremamente complexo para calcular a função de Ackermann, que é definida recursivamente como

```

ack (0, b) = b + 1
ack (a, 0) = ack (a-1, 1)

```

Página 118

SEÇÃO 5.13
RESUMO

107

```

ack (a, b) = ack (a-1, ack (a, b-1))
A versão PROMELA é a seguinte.
/ ***** Função de Ackermann ***** /
proctype ack (abbreviatura a, b; chan ch1)
{
chan ch2 = [1] de {short};
short ans;
E se
:: (a == 0) ->
ans = b + 1
:: (a! = 0) ->
E se
:: (b == 0) ->
execute ack (a-1, 1, ch2)
:: (b! = 0) ->
execute ack (a, b-1, ch2);
ch2? ans;
execute ack (a-1, ans, ch2)
fi;
ch2? ans
fi;
ch1! ans
}
iniciar
{
chan ch = [1] de {short};
short ans;
execute ack (3, 3, ch);
ch? ans;
printf ("ack (3,3) =% d \ n", ans);
afirmar (0)
/* uma parada forçada, (Capítulo 6) */
}

```

Parece bastante simples? Leva 2.433 instâncias de processo para produzir a resposta.

A resposta, aliás, é 61.

5.13 RESUMO

Introduzimos uma notação para descrever regras de procedimento de protocolo em um linguagem de especificação e modelagem chamada PROMELA. Neste capítulo, temos discutido recursos PROMELA discutidos para descrever apenas o comportamento do sistema. No próximo capítulo nós

discutir os recursos de linguagem restantes que estão especificamente relacionados ao especificação dos critérios de correção.

A linguagem de modelagem de validação tem vários recursos incomuns que a tornam adequada para modelagem de sistemas distribuídos. Toda a comunicação entre os processos ocorre via mensagens ou variáveis compartilhadas. Comunicação síncrona e assíncrona são modelados como casos especiais de um mecanismo geral de passagem de mensagens.

Cada instrução no PROMELA pode potencialmente modelar o atraso: é executável ou não, na maioria dos casos dependendo do estado do ambiente do processo em execução.

108

MODELOS DE VALIDAÇÃO
CAPÍTULO 5

A interação e coordenação de processos estão, portanto, na base da linguagem. A semântica da linguagem faz um mapeamento da linguagem do fluxograma usado em a primeira parte do livro para programas PROMELA direto. Provavelmente é bom tenha em mente que PROMELA é uma linguagem de modelagem, não uma linguagem de programação. Não existem tipos de dados abstratos e apenas alguns tipos básicos de variáveis. Uma validação modelo é uma abstração de uma implementação de protocolo. A abstração mantém o fundamentos da interação do processo para que ele possa ser estudado isoladamente. Suprime detalhes de implementação e programação. Uma visão geral do idioma pode ser encontrada no Apêndice C.

Nos próximos capítulos, encontraremos um bom uso para a linguagem. No Capítulo 7, é usado no projeto de um protocolo de transferência de arquivos. Na Parte IV, mostramos como desenvolver o software para

analisar modelos de protocolo escritos em PROMELA .

EXERCÍCIOS

5-1. Suponha que a instrução run A () não seja executável, por exemplo, porque havia muitos processos em execução. Você pode dizer se b = run A () é executável?

5-2. Se a declaração (qname? Var == 0) fosse permitida na PROMELA , qual seria seu efeito?

Dica: considere os efeitos colaterais da operação de recepção.

5-3. Revise os dois programas da Seção 5.6 para incorporar o uso de mensagens do tipo eot para significar o fim de um fluxo de entrada.

5-4. Reescreva a declaração dos tipos de processo fact () e ack () para usar uma variável global em vez de mensagens para comunicar o resultado do cálculo de um processo filho para seu pai.

5-5. Reescreva o programa fact () para retornar o enésimo número de Fibonacci, $f(n) = f(n-1) + f(n-2)$, em vez de um fatorial. Por definição, $f(0) = 0$ e $f(1) = 1$.

5-6. Reescreva seu programa para gerar números de Fibonacci para reduzir o número de processos que são necessários. (Dica: torne o programa recursivo individualmente; cada processo cria no máximo um filho.)

5-7. Estenda o modelo do protocolo de Lynch com dois processos de usuário que usam o serviço de transferência vice.

5-8. Estenda o mesmo programa com um tipo de processo modelando um canal de transmissão defeituoso entre os dois usuários.

5-9. Escreva um programa PROMELA que realiza uma classificação por bolha nos elementos de um canal que é inicializado com mensagens do tipo int , cada uma carregando um valor. Uma classificação de bolha é feita por digitalizando uma lista de números repetidamente, trocando qualquer par de números adjacentes que estão fora de serviço.

5-10. Escreva um programa PROMELA que classifique inteiros construindo uma árvore binária de processos. Cada o processo contém um inteiro na sequência. Tem um processo pai e até duas crianças processos dren, esquerda e direita. Os inteiros entram no classificador por meio do processo na raiz de a árvore. Todos os processos seguem a mesma disciplina. Se o próximo inteiro recebido for maior do que aquele em poder do receptor, ele é encaminhado para a esquerda. Se for menor, é encaminhado para o direito. Os processos filhos são criados apenas quando necessário. Quando o último inteiro tiver processados, os valores armazenados na árvore devem ser recuperados na ordem certa e

CAPÍTULO 5
NOTAS BIBLIOGRÁFICAS
109

impresso.

5-11. Com processos distribuídos, é relativamente fácil projetar gerentes de recursos que podem, por exemplo, fornecer aos programas do usuário acesso mutuamente exclusivo a dispositivos ou serviços. Escrever um servidor de impressão de amostra que " possui " um canal de exibição e permite processos para enviar um sequência de mensagens para ele (por exemplo, qualquer coisa até um eot) em fragmentos, sem a possibilidade capacidade de interrupção por outros processos.

5-12. Reescreva o exemplo usando o semáforo Dijkstra com comunicação de rendezvous em uma solução que usa comunicação assíncrona entre o processo de monitoramento e o usuário processos.

5-13. Considere em detalhes porque os tipos de dados C reais ou duplos não são definidos no PROMELA .

5-14. (Paul Haahr) Muitos processadores usam " níveis de prioridade de interrupção " para garantir que alguns dispositivos são manipulados antes de outros (por exemplo, os discos são geralmente tratados como mais urgentes que os teclados). O nível de prioridade atual é armazenado em um registro de CPU especial, geralmente um de 3 ou 4 bits inteiro. Durante a operação normal, o nível de prioridade é zero. Cada dispositivo de hardware que

pode interromper a CPU é atribuído um nível de prioridade. Quando ocorre uma interrupção de hardware, se o processador está atualmente funcionando em um nível inferior ao do dispositivo, o processador começa a executar o manipulador de interrupções apropriado; se não, o dispositivo espera até a prioridade cai e depois interrompe. Quando o manipulador de interrupção começa, o nível da prioridade é definido para o prioridade do dispositivo. Ele é redefinido para o nível anterior quando o manipulador é encerrado. O processador também pode definir o nível de prioridade independentemente dos manipuladores de interrupção, por exemplo, com um

instrução `sp1 (x)`. Isso pode ser usado para evitar que um manipulador de interrupção seja interrompido no meio - quando suas estruturas de dados não estão necessariamente em um estado adequado - por outra interrupção que usará as mesmas estruturas. Ele também é usado por sistemas operacionais para assegurar exclusão mútua. Por exemplo, se um dispositivo (digamos um disco) interrompe no nível seis, o driver de dispositivo que executa o disco deve definir o nível de prioridade temporariamente para seis antes de usar estruturas de dados que podem ser alteradas por uma interrupção de disco. Modele a prioridade de interrupção esquema no PROMELA para três processos, modelando o comportamento de uma CPU, um processo de disco e um processo terminal. (Dica: use uma matriz para modelar a pilha de níveis de prioridade.)

5-15. Modifique o modelo de validação do protocolo de Lynch para modelar a possibilidade de transmissão erros de sessão.

NOTAS BIBLIOGRÁFICAS

PROMELA é uma extensão de uma linguagem menor chamada Argos que foi desenvolvida em 1983 para validação de protocolo, por exemplo, Holzmann [1985]. A sintaxe de PROMELA expressões, declarações e atribuições é vagamente baseado na linguagem C, Kernighan e Ritchie [1978]. A linguagem foi influenciada significativamente pelo comando "guardado linguagens de mando" de EW Dijkstra [1975] e CAR Hoare [1978]. Existem, como sempre, diferenças importantes. A linguagem de Dijkstra não tinha primitivas para interação de processos ção. A linguagem de Hoare era baseada exclusivamente na comunicação síncrona. Além disso na linguagem de Hoare, o tipo de declarações que podem aparecer nos guardas de um opção foi restrita. A semântica das instruções de seleção e ciclo em PROMELA também é bastante diferente de outras linguagens de comando protegidas: o estados não são abortados quando todos os guardas são falsos, mas eles bloqueiam, fornecendo assim a sincronização necessária.

O problema da exclusão mútua (ou "seção crítica"), referido brevemente neste

Página 121

110
MODELOS DE VALIDAÇÃO
CAPÍTULO 5

capítulo, foi estudado por muitos anos. A seguinte série de artigos intrigantes documenta algumas das melhorias que foram feitas: Dijkstra [1965], Knuth [1966], deBruyn [1967], Dijkstra [1968], Eisenberg e McGuire [1972], Lamport [1974, 1976, 1986]. Discussões mais elaboradas também podem ser encontradas em Bredt [1970] ou Holzmann [1979].

Página 122

REQUISITOS DE CORREÇÃO 6

- 111 Introdução 6.1
- 112 Raciocínio sobre o comportamento 6.2
- 114 Asserções 6.3
- 115 Invariante do Sistema 6.4
- 117 Deadlocks 6.5
- 118 Ciclos ruins 6.6
- 119 Reivindicações Temporais 6.7
- 125 Resumo 6.8
- 126 exercícios
- 127 Notas Bibliográficas

6.1 INTRODUÇÃO

No Capítulo 5, desenvolvemos uma linguagem para modelar comportamento em sistemas distribuídos. A linguagem é deliberadamente definida em um alto nível de abstração para nos permitir focar no design e não nas questões de implementação. Os programas que podem ser escritos em esta linguagem são, portanto, chamados de *modelos de validação*. Os detalhes necessários para converter um modelo de validação em uma implementação poderia provavelmente ser preenchido com pouco esforço, talvez até mecanicamente. Mas o objetivo principal da PROMELA é validação, não implementação.

Para validar um design, precisamos ser capazes de especificar precisamente o que significa para um

design para ser correto. Um projeto pode ser comprovado como correto apenas em relação a critérios de correção. Três critérios razoavelmente padronizados foram listados no Capítulo 2: o ausência de deadlocks, livelocks e encerramentos impróprios. É, por exemplo, nunca o suficiente para apenas "saber" que um design está livre de impasses.

Um bom design é comprovadamente livre de impasses.

Existem muitas propriedades de protocolo que alguém pode estar interessado em provar para qualquer determinado design. Mas os problemas com os quais estamos lidando são complexos. Não é muito difícil mostram que o problema de verificar até mesmo as propriedades de protocolo mais simples, como ausência de impasse, é PSPACE difícil (ver notas bibliográficas), mesmo para um finito modelo de estado. Na tentativa de provar a exatidão de um protocolo, devemos estar cientes desses limites de complexidade. Uma vez que é nosso objetivo desenvolver uma metodologia de validação

que podem ser aplicados a protocolos de tamanho arbitrário, precisamos desenvolver métodos para identificando a complexidade de dois lados diferentes:

Precisamos de um formalismo para especificar os requisitos de correção que não é assim inerentemente complexo que uma análise eficaz para modelos maiores se torna impossível.

Precisamos de um método para reduzir a complexidade dos modelos que estão além do gama de nossos métodos de validação.

111

Página 123

112
REQUISITOS DE CORREÇÃO
CAPÍTULO 6

O segundo ponto, redução, é discutido nos Capítulos 8 e 11 (ver Seção 8.9, Generalização de Máquinas e Seção 11.7, Gerenciamento da Complexidade). O primeiro ponto será abordado aqui. Diz que quanto mais expressiva tornamos nossa notação para especificando os requisitos de correção, menos útil será na prática. O conjunto de Os critérios de correção que podem ser expressos no PROMELA são, portanto, escolhidos com cuidado. Este conjunto não é deliberadamente restrito a um único mecanismo todo-poderoso. De várias níveis independentes de complexidade são suportados. O mais simples, usado com mais frequência requisitos, como ausência de impasse, são expressos de forma direta e verificado independentemente de outras propriedades. Eles podem ser analisados mecanicamente com algoritmos rápidos e econômicos, mesmo para sistemas muito grandes. Um pouco mais complicado tipos de requisitos, como ausência de livelocks, são expressos de forma independente, e carregam um preço independente em despesas computacionais quando validado mecanicamente. Os requisitos mais sofisticados são inevitavelmente também os mais caros para Verifica. No Capítulo 11, discutimos os algoritmos mais conhecidos para a validação automatizada identificação de cada tipo de requisito de correção e quantificar o tamanho dos sistemas que eles pode validar.

Na próxima seção, daremos uma visão geral dos tipos de critérios de correção que podem ser expresso para modelos PROMELA . Cada uma das seções que se seguem elabora um dos essas propriedades em detalhes. Mostra as estruturas de linguagem PROMELA que são necessárias para expressar cada propriedade e dar alguns exemplos de seu uso.

6.2 RAZÃO SOBRE O COMPORTAMENTO

Nós formalizar critérios de regularidade como afirmações sobre o comportamento de um PROMELA validado modelo de ção. Dois tipos gerais de afirmações são, então, que um determinado comportamento é ou Inevitável ou Impossível

Uma vez que o número de comportamentos possíveis de qualquer modelo PROMELA é finito, como-sempre, uma reivindicação de qualquer tipo define implicitamente uma reivindicação complementar e equivalente do outro tipo. Portanto, basta suportar apenas um.

Todos os critérios de correção que podem ser expressos no PROMELA definem comportamentos que são considerados impossíveis .

Para afirmar que um determinado comportamento é inevitável, por exemplo, podemos afirmar que todos os desviantes comportamentos são impossíveis. Da mesma forma, se uma afirmação de correção afirma que uma condição é invariavelmente verdadeiro, a afirmação de correção afirma que é impossível para a afirmação ser violado, independentemente do comportamento do sistema.

Antes de continuarmos, no entanto, teremos que ser mais precisos sobre os termos que estão usando. O que, por exemplo, é um comportamento e como podemos fazer afirmações sobre ele? O *comportamento* de um modelo de validação é definido completamente pelo conjunto de todos os executivos sequências de execução que ele pode executar, onde uma *sequência de execução* é simplesmente finita e ordenada conjunto de estados. Um *estado*, por sua vez, é completamente definido pela especificação de todos os valores para variáveis locais e globais, todos os pontos de fluxo de controle de processos em execução, e o conteúdo de todos os canais de mensagens. Dizemos que um modelo de validação pode *chegar a* um determinado

Página 124

SEÇÃO 6.2
RAZÃO SOBRE O COMPORTAMENTO

113

estado pela execução de instruções PROMELA, usando a semântica definida anteriormente de executabilidade. Um modelo de validação também pode *ser colocado* em um determinado estado por uma atribuição

mento dos valores apropriados para variáveis, pontos de fluxo de controle e canais.

Claro, não apenas qualquer coleção arbitrária de estados também é uma sequência de execução válida.

Um conjunto finito e ordenado de estados é considerado *válido* para um

determinado modelo M PROMELA se ele

satisfaz os dois critérios a seguir:

O primeiro estado da sequência, ou seja, o estado com número ordinal 1, é o inicial estado do sistema de M , com todas as variáveis inicializadas em zero, todos os canais de mensagem vazio, com apenas o processo `init` ativo e definido em seu estado inicial.

Se M é colocado no estado com número ordinal i , há pelo menos um executável declaração que pode trazê-lo para o estado com número ordinal $i + 1$.

Vamos considerar dois tipos especiais de sequências, chamadas de terminação e cíclica sequências.

Uma sequência de execução está *terminando* se nenhum estado ocorrer mais de uma vez na sequência, e o modelo M não contém instruções executáveis quando colocado no último estado da sequência.

Uma sequência de execução é dita *cíclica* se todos os estados, exceto o último, forem separados tinto, e o último estado da sequência é igual a um dos primeiros estados.

As sequências cíclicas definem execuções potencialmente infinitas. Todos terminantes e cílicos sequências de execução que podem ser geradas executando um modelo PROMELA, juntos definir o *comportamento* do *sistema* desse modelo. A união de todos os estados incluídos no sistema Este comportamento é chamado de *conjunto de estados alcançáveis* do modelo.

PROPRIADES DOS ESTADOS

As reivindicações de exatidão para modelos PROMELA podem ser construídas a partir de proposições simples,

onde uma proposição é uma condição booleana no estado do sistema. A proposta ções podem se referir a todos os elementos de um estado do sistema: variáveis locais e globais, pontos de controle de fluxo de processos de execução arbitrários e o conteúdo da mensagem canais. Algumas das notações para isso foram discutidas no Capítulo 5; o restante recursos serão introduzidos em breve.

As proposições definem implicitamente uma rotulagem de estados. Em qualquer estado, uma proposição é verdadeira ou falsa. Os critérios de correção, então, podem ser expressos em termos de estados, por exemplo, definindo explicitamente em quais estados uma determinada proposição é necessária para

aguarde. Alguns desses requisitos podem ser especificados na PROMELA com, por exemplo, declarações de *asserção* que estão embutidas no modelo. Este mecanismo por si só, como-nunca, não é suficiente. Se mais de uma proposição for usada, podemos querer expressar uma exigência de correção como uma ordem temporal de proposições, ou seja, especificando o ordem em que as proposições devem ser mantidas (com a verdade de uma proposição imediatamente ou eventualmente seguindo a verdade de outro). Alternativamente, o a ordenação temporal pode definir a ordem em que as proposições nunca devem ser válidas. Como indicadas acima, essas duas alternativas para definir ordenações temporais são completas

mentário. Apenas a segunda alternativa é suportada no PROMELA . O formalismo para apoiar este é um novo recurso de linguagem chamado de *reivindicação temporal* .

Página 125

114

REQUISITOS DE CORREÇÃO

CAPÍTULO 6

REIVINDICAÇÕES TEMPORAIS

Na formalização de reivindicações temporais, especificamos uma ordem de proposições. Isto é importante notar que a semântica dessas ordens de *proposições* é diferente de a semântica de ordens de *instrução* em outro lugar em um modelo PROMELA . Dentro definições de tipo `proc` , uma ordenação sequencial de duas declarações implica que a segunda instrução deve ser executada após o primeiro terminar. Uma vez que não temos permissão para fazer quaisquer suposições sobre as velocidades relativas de processos em execução simultânea, a única interpretação válida da palavra *depois* na frase acima é *eventualmente depois* . Reivindicações de correção, no entanto, podem ter que ser mais específicas. Em um temporal afirmar que uma ordem sequencial de duas proposições define uma consequência *imediata* . Mostraremos mais tarde como outros tipos de relações temporais também podem ser especificados com este mecanismo.

Os tipos de requisitos de correção que fazemos podem ser diferentes para rescisão e sequências cíclicas, assim como os algoritmos de que precisaremos para verificar essas afirmações. Um requisito importante que se aplica à terminação de sequências é, por exemplo, ausência de impasse. Nem todas as sequências de terminação, no entanto, correspondem a impasses. Teremos que ser capazes de expressar quais propriedades o estado final em um sequência deve ter para torná-la aceitável como um término sem deadlock sequência. Para sequências cíclicas, finalmente, devemos ser capazes de expressar condições como a ausência de livelock.

VISÃO GLOBAL

O restante deste capítulo é dedicado a uma discussão mais detalhada da zação de critérios de correção na PROMELA . Ele irá apresentar os últimos idiomas estruturas que lidam especificamente com a validação:

A instrução `assert ()` , que pode ser usada para expressar ambas as asserções locais e invariantes do sistema global

Três tipos de rótulos que podem ser usados para definir uma pequena classe de reivindicações de correção para sequências finais e cíclicas

A formalização de reivindicações temporais gerais

A notação que pode ser usada em asserções e em reivindicações temporais para se referir a os estados de fluxo de controle e as variáveis locais de processos arbitrários em execução

Começamos dando uma olhada mais de perto na especificação das propriedades de correção de estados.

6.3 ASERÇÕES

Os critérios de exatidão muitas vezes podem ser expressos como condições booleanas que devem ser satisfeita sempre que um processo atinge um determinado estado. A declaração PROMELA afirmar (*condição*)

é sempre executável e pode ser colocado em qualquer lugar em um modelo PROMELA . A condição pode ser uma expressão booleana arbitrária. Se a condição for verdadeira, a declaração tem nenhum efeito. A validade da declaração é violada, no entanto, se houver pelo menos um

Página 126

SEÇÃO 6.4

INVARENTE DO SISTEMA

115

sequência de execução em que a condição é falsa quando a declaração `assert` torna-se executável.

Considere o seguinte exemplo do Capítulo 5.

```
estado do byte = 1;
proctipo A () {(estado == 1) -> estado = estado + 1}
proctipo B () {(estado == 1) -> estado = estado - 1}
init {executar A (); executar B ()}
```

Poderíamos tentar afirmar que quando um processo do tipo A () completa o valor da variável o `estado` deve ser 2 e quando um processo do tipo B () é concluído, ele deve ser 0. Isso pode ser expressa da seguinte forma.

```

estado do byte = 1;
proctipo A ()
{
  (estado == 1) -> estado = estado + 1;
  afirmar (estado == 2)
}
proctipo B ()
{
  (estado == 1) -> estado = estado - 1;
  afirmar (estado == 0)
}
init {executar A (); executar B ()}

```

As afirmações são, obviamente, falsas, e um validador automatizado demonstraria que rapidamente.

6.4 INVARENTES DO SISTEMA

Uma aplicação mais geral da declaração `assert` é formalizar invariantes do sistema, ou seja, condições booleanas que, se verdadeiras no estado inicial do sistema, permanecem verdadeiras em *todo* alcance

estados do sistema capazes, independentemente da sequência de execução que leva a cada Estado. Para expressar isso no PROMELA, basta colocar o sistema invariante por si mesmo em um processo de monitoramento separado.

```
monitor proctype () {assert (invariante)}
```

Uma vez que uma instância do monitor de tipo de processo tenha sido iniciada (o nome é imaterial), com uma instrução de execução regular, é executado independentemente do resto do sistema tem. Ele pode decidir avaliar a afirmação a qualquer momento; sua declaração `assert` é executável precisamente uma vez para cada estado do sistema.

Para uma aplicação simples deste tipo de afirmação de correção, considere o `dijkstra` modelo de validação de semáforo, apresentado no Capítulo 5.

116
REQUISITOS DE CORREÇÃO
CAPÍTULO 6

```

#define p
0
#define v
1
chan sema [0] de {bit};
proctype dijkstra ()
{
Faz
:: sema! p -> sema? v
od
}
usuário proctype ()
{
sema? p;
/* seção Crítica */
sema! v
/* seção não crítica */
}
iniciar
{
atômica {
execute dijkstra ();
executar usuário (); executar usuário (); run user ()
}
}

```

O semáforo garante o acesso mutuamente exclusivo dos processos do usuário aos seus

Seções. Podemos modificar os processos do usuário da seguinte forma, para contar o número de processos na seção crítica em uma contagem de variável global.

```

contagem de bytes;
usuário proctype ()
{
sema? p;
contagem = contagem + 1;
pular; /* seção Crítica */
contagem = contagem-1;
sema! v;
pular
/* seção não crítica */
}

```

O seguinte sistema invariante agora pode ser usado para verificar a operação correta do

semáforo:

```
monitor proctype () {assert (contagem == 0 || contagem == 1)}  
Uma instanciação do monitor deve ser incluída no processo inicial, para permitir que  
execute a verificação de exatidão.
```

```
iniciar  
{  
atômica {  
executar dijkstra (); monitor de execução ();  
executar usuário (); executar usuário (); run user ()  
}  
}
```

Página 128

SEÇÃO 6.5
DEADLOCKS
117

6.5 DEADLOCKS

Em um sistema de estado finito, todas as sequências de execução terminam após um número finito de transições de estado, ou eles retornam a um estado visitado anteriormente. Nem todos terminam sequências de transmissão, no entanto, são necessariamente deadlocks. Para definir o que é um impasse em um modelo PROMELA é, devemos ser capazes de distinguir o fim esperado, ou *adequado*, estados dos inesperados. Os estados finais inesperados incluirão não apenas estados de deadlock, mas também muitos estados de erro que são o resultado de uma lógica incompleta da especificação do protocolo. O exemplo clássico deste último é o *não especificado recepção*.

O estado final em uma sequência de execução de término deve satisfazer minimamente o seguinte dois critérios a serem considerados um estado final adequado:

Cada processo que foi instanciado foi encerrado

Todos os canais de mensagens estão vazios

Mas nem todos os processos terminam necessariamente. Pode ser perfeitamente válido, por exemplo, para os processos do servidor permanecem ativos após o término dos processos do usuário. Devemos ser capazes, lá-

portanto, para identificar estados de processos individuais nas definições de *proctipo* como fim adequado

estados. No PROMELA, isso pode ser feito com rótulos de *estado final*. No exemplo do semáforo do Capítulo 5, por exemplo, podemos escrever

```
proctype dijkstra ()  
{  
fim:  
Faz  
:: sema! p -> sema? v  
od  
}
```

para definir que qualquer processo do tipo *dijkstra* é considerado em um estado final adequado quando está no estado rotulado como *final*.

Se houver mais de um estado final adequado em uma única definição de *proctype*, todos os nomes dos rótulos ainda devem ser únicos. Um rótulo de estado final é definido como qualquer nome de rótulo

que tem um *final* de prefixo de três caracteres. Portanto, é válido usar variações como *enddne*, *end0*, *end_war*. Podemos agora revisar o primeiro critério da definição de um adequado estado final:

Cada processo que foi instanciado foi encerrado ou atingiu um estado marcado como um estado final adequado

Qualquer estado final em uma sequência de execução final que não satisfaça os dois critérios critérios para estados finais adequados são automaticamente classificados como estados finais impróprios. A

a afirmação de correção implícita que é feita sobre todos os modelos de validação será que o os comportamentos que eles definem não incluem estados finais impróprios.

Consulte o Capítulo 14 e o Apêndice F para alguns exemplos do uso de etiquetas de estado final em uma validação real.

Página 129

118
REQUISITOS DE CORREÇÃO

6.6 CICLOS RUINS

Duas propriedades de sequências cíclicas podem ser expressas em PROMELA, correspondendo a dois tipos padrão de requisitos de correção. Ambas as propriedades são baseadas no marcação explícita de estados em um modelo de validação. A primeira propriedade especifica que Não há comportamentos infinitos de apenas estados não marcados ou seja, o sistema não pode circular infinitamente pelos estados não marcados. Os estados marcados são chamados *de estados de progresso*, e as sequências de execução que violam o acima correto reivindicação de ness são chamados *de ciclos de não progresso*. A segunda propriedade é o oposto do primeiro. É usado para especificar que Não há comportamentos infinitos que incluem estados marcados As sequências de execução que violam essa reivindicação são chamadas de *livelocks*. Discutimos cada tipo de reivindicação de correção em mais detalhes abaixo.

CICLOS SEM PROGRESSO

Para reivindicar a ausência de ciclos de não progresso, devemos ser capazes de definir o sistema estados dentro do modelo PROMELA que denotam progresso. Esses estados de progresso são definido muito como rótulos de estado final.

Um rótulo de estado de progresso marca um estado que *deve* ser executado para que o protocolo faça progresso. Um exemplo pode ser o incremento de um número de sequência ou a entrega de dados para um receptor. No exemplo do semáforo, podemos rotular a passagem bem-sucedida de um teste de semáforo como "progresso". Simplesmente marcando-o como um estado de progresso, podemos expressar o critério de correção de que a passagem da guarda do semáforo não pode ser adiado infinitamente longo, por exemplo, por um ciclo de execução infinito que não passa o estado de progresso.

```
proctype dijkstra ()  
{  
fim:  
Faz  
:: sema! p ->  
progresso:  
sema? v  
od  
}
```

Um validador automatizado pode confirmar prontamente que de fato esta afirmação não pode ser violada. Se mais de um estado carrega um rótulo de estado de progresso, variações com um prefixo comum são novamente válido: *progress0*, *progressisslow*, e assim por diante.

LIVELOCKS

Suponha que quiséssemos expressar o oposto de uma condição de progresso, por exemplo, queremos formalize que algo não pode acontecer com uma frequência infinita. Podemos expressar propriedades assim com a terceira e última classe de rótulos PROMELA especiais. Em adição ao rótulos de estado final e estado de progresso introduzidos anteriormente, agora definimos o estado de aceitação

rótulos. Um *rótulo de estado de aceitação* é qualquer rótulo que comece com a sequência de caracteres "Aceitar." Marca um estado que *pode não* ser parte de uma sequência de estados que podem ser repetido infinitamente frequentemente.

Por exemplo, se substituirmos o rótulo de estado de progresso em *proctype dijkstra ()* por um

```
proctype dijkstra ()  
{  
fim:  
Faz  
:: sema! p ->  
aceitar:  
sema? v  
od  
}
```

que afirmam que é impossível percorrer uma série de *p* e *v* operações. Nós saber, é claro, que essa afirmação é falsa. Podemos provar que é falso manualmente, ou podemos usar um validador automatizado para fornecer um contra-exemplo.

Mais uma vez, todas as variações, tais como `aceitador`, `aceitável`, e `accept_yo`, são permitidos. Em princípio, poderíamos fazer o melhor uso de um estado de aceitação se pudéssemos usá-lo para expressar comportamentos completos que devem ser impossíveis, ao invés de apenas o ausência de um estado designado em todos os ciclos. Faremos exatamente isso usando o rótulos para definir os estados de aceitação de autômatos de declarações especiais que modelam o erro comportamentos. (Isso também explica a escolha do termo "aceitar"). Esses autômatos expressar reivindicações temporais gerais.

6.7 RECLAMAÇÕES TEMPORAIS

Até este ponto, falamos sobre a especificação dos critérios de correção com asserções e com três tipos especiais de rótulos que podem ser usados para marcar estados finais, estados de progresso e estados de aceitação. Tipos poderosos de critérios de correção podem já ter sido expressa com essas ferramentas, mas até agora nossa única opção é adicioná-los definições de `proctype`. Suponha que, dentro desta estrutura, queremos expressar o reivindicação temporal " todo estado em que a propriedade **P** é verdadeira é seguido por um estado em qual propriedade **Q** é verdadeira. " Observamos antes que duas interpretações diferentes do termo "seguido por" são possíveis, dependendo se os dois estados devem *imediatamente* atamente ou *eventualmente* se seguirão. É básico para a semântica da PROMELA que não quaisquer suposições podem ser feitas sobre o tempo relativo das execuções do processo. Isso significa que, até agora, a única interpretação legítima do termo acima é que dois passos "eventualmente" seguem uns aos outros (incluindo "imediatamente" como um caso especial). Isso, no entanto, nos deixa com o problema de expressar os outros tipos de propriedades.

Para isso, precisamos de um tipo diferente de primitiva de validação.

As reivindicações temporais definem ordenações temporais de *propriedades* de estados. Para expressar o requisito de que " todo estado em que a propriedade **P** é verdadeira seja seguido por um estado em qual propriedade **Q** é verdadeira, " poderíamos escrever

$P \rightarrow Q$

Mas não é tão simples. Existem dois empecilhos.

Uma vez que todos os nossos critérios de correção são baseados em propriedades que afirmam ser *impossível*, as reivindicações temporais que usamos também devem expressar *ordenações* de propriedades isso é impossível.

As reivindicações temporais são definidas em sequências de execução *completas* (encerrando ou

Página 131

120

REQUISITOS DE CORREÇÃO
CAPÍTULO 6

cíclico). Mesmo que um prefixo da sequência seja irrelevante, ele ainda deve ser representado como uma seqüência de proposições trivialmente verdadeira.

O requisito acima deve ser expresso, portanto, como

`nunca {faça :: pular :: quebrar od -> P ->! Q}`

ou seja, independente da sequência inicial de eventos, é impossível para um estado em cuja propriedade **P** é verdadeira para ser seguida por um estado em que a propriedade **Q** é falsa. A reivindicação é *correspondida*, e a propriedade de correção correspondente, portanto, violada, se e quando o corpo da reivindicação termina.

A notação PROMELA para uma reivindicação temporal é

`Nunca { ... }`

onde os pontos contêm os detalhes da reclamação.

As declarações temporais podem ser preparadas com rótulos de estado de progresso ou estado de aceitação para capturar

mais tipos de erros do que apenas uma correspondência completa de um comportamento de encerramento. o

`nunca` reivindicações podem ser expressas, por exemplo, como máquinas especiais de estado finito que percorrer um estado de aceitação se o comportamento indesejável for reconhecido.

Suponha que quiséssemos expressar a propriedade temporal de que a condição 1 nunca pode permanecer verdadeiro por muito tempo. Para detectar violações desta propriedade, devemos encontrar um

representação em uma reivindicação temporal de todos os comportamentos em que a condição 1 pode ser falsa

inicialmente, torna-se verdadeiro eventualmente e permanece verdadeiro. É expresso da seguinte forma

`Nunca {
Faz`

```

:: pular
:: condiçãol -> quebra
od;
aceitar: fazer
:: condiçãol
od
}

```

Esta reivindicação (não encerrada) é *correspondida* e a propriedade de correção correspondente violado, se e quando o ciclo de aceitação for detectado. A parte complicada é lembrar a inclusão do `salto` (uma condição que é sempre verdadeira) no primeiro loop. Observe que sequências onde o valor de verdade da `condição 1` primeiro muda de verdadeiro para falso alguns vezes são permitidos pela reivindicação. A afirmação em si é simplesmente uma máquina de estados finitos, com uma proposição definida em cada Estado. Para cada transição de estado em outro lugar no modelo de validação, ou seja, pelo execução de uma instrução PROMELA, a máquina de reclamação deve alterar seu estado e sair uma proposição para a próxima. Para corresponder a uma reivindicação temporal, em cada estado em uma sequência de estados, a proposição no estado correspondente na máquina de reivindicações deve ser verdadeira. E se nós escrevemos erroneamente, por exemplo

Página 132

SEÇÃO 6.7 REIVINDICAÇÕES TEMPORAIS

121

```

Nunca {
Faz
:: pular
:: condiçãol -> quebra
od;
aceitar:
condição1
}

```

a declaração contém apenas duas transições de estado após a `condição 1` se tornar verdadeira. Isto reivindicação é completamente correspondida se houver pelo menos uma sequência de execução na qual `condição1` é válida em dois estados subsequentes.

As máquinas de estado finito especificadas nas reivindicações acima contêm três estados cada: o estado inicial, o estado rotulado como `aceitar` e o estado final normal. O primeiro, correto, verificação da reivindicação especifica que seria um erro (um livelock) se a máquina pudesse permanecer no segundo estado, infinitamente longo. A segunda versão especificou que seria um erro se o terceiro estado (terminal) for alcançável.

ESPECIFICANDO REIVINDICAÇÕES TEMPORAIS

O corpo de uma reivindicação temporal é definido apenas como corpos de protótipo PROMELA. Isto significa que todas as estruturas de fluxo de controle, como seleções `if-fi`, repetições `do-od`, e `goto` jumps, são permitidos. Há, no entanto, uma diferença importante:

Cada declaração dentro de uma reivindicação temporal é (interpretada como) uma condição.

Especificamente, isso significa que as declarações dentro de reivindicações temporais devem estar livres de efeitos colaterais. Para referência, as declarações PROMELA com efeitos colaterais são: assignments, afirmações, envia, recebe e instruções `printf`.

As declarações temporais são usadas para expressar comportamentos do sistema que são considerados indesejáveis

ou ilegal. Dizemos que uma reivindicação temporal é *correspondida* se o comportamento indesejável pode ser

percebidos, e assim nossa reivindicação de correção pode ser violada.

A aplicação mais útil de reivindicações temporais é em combinação com aceitação rótulos. Existem então duas maneiras de *corresponder* a uma reivindicação temporal, dependendo se o comportamento indesejável define as sequências de término ou de execução cíclica.

Para uma sequência de execução final, uma reivindicação temporal é correspondida apenas quando pode terminar (atinge a chave de fechamento) Ou seja, a reivindicação pode ser violada se a chave de fechamento do corpo PROMELA da reivindicação está acessível.

Para uma sequência de execução cíclica, a reivindicação é correspondida apenas quando uma existe um ciclo de aceitação. Os rótulos de aceitação dentro de reivindicações temporais são do usuário definido, não há padrões. Isso significa que, na ausência de rótulos de aceitação

nenhum comportamento cíclico pode ser correspondido por uma reivindicação temporal. Também significa que verificar uma reivindicação temporal cíclica, os rótulos de aceitação devem ocorrer apenas dentro do reclamar e não em qualquer outra parte do código PROMELA . Reclamações nunca , usadas em combinação com rótulos de estado de aceitação, podem expressar também o ausência de ciclos de não progresso. As reivindicações são, portanto, mais gerais do que rótulos de estado de progresso. A despesa (complexidade) de encontrar ciclos de não progresso

Página 133

122
REQUISITOS DE CORREÇÃO
CAPÍTULO 6

diretamente com rótulos de estado de progresso, no entanto, é menor do que o custo da validação de uma reivindicação que especifica a mesma propriedade. Para obter o benefício total das reivindicações temporais, devemos ser capazes de nos referir ao fluxo de controle estados e os valores das variáveis dos processos em execução. Como exemplo, considere o seguinte versão seguinte do protocolo de bit alternado.

Página 134

SEÇÃO 6.7
REIVINDICAÇÕES TEMPORAIS

```

123
1 /* bit alternado - versão com perda de mensagem */
2
3 # define MAX 3
4
5 mtype = {msg0, msg1, ack0, ack1};
6
7 chan
remetente = [1] de {byte};
8 chan
receptor = [1] de {byte};
9
10 proctype Sender ()
11 {byte qualquer;
12 novamente:
13
Faz
14
:: receptor! msg1;
15
E se
16
:: sender? ack1 -> break
17
:: remetente? qualquer /* perdido */
18
:: tempo esgotado
/* retransmit */
19
fi
20
od;
21
Faz
22
:: receptor! msg0;
23
E se
24
:: remetente? ack0 -> pausa
25
:: remetente? qualquer /* perdido */
26
:: tempo esgotado
/* retransmit */
27
fi
28
od;
29

```

```

vá de novo
30}
31
Receptor 32 proctype ()
33 {byte qualquer;
34 novamente:
35
Faz
36
:: receptor? msg1 -> remetente! ack1; quebrar
37
:: receptor? msg0 -> remetente! ack0
38
:: receptor? qualquer / * perdido * /
39
od;
40 P0:
41
Faz
42
:: receptor? msg0 -> remetente! ack0; quebrar
43
:: receptor? msg1 -> remetente! ack1
44
:: receptor? qualquer / * perdido * /
45
od;
46 P1:
47
vá de novo
48}
49
50 init {atomic {run Sender (); execute Receiver ()}}

```

Os processos receptor e remetente se comunicam por meio de canais de mensagens denominados receptor e remetente. Cada canal pode conter uma mensagem do tipo byte. A perda de mensagens é modelada explicitamente nos processos do remetente e do receptor com um

Página 135

124
REQUISITOS DE CORREÇÃO
CAPÍTULO 6

cláusula que pode roubar uma mensagem recebida antes de ser processada (linhas 15, 23, 36 e 42). Podemos querer expressar a reclamação "é sempre verdade que quando o remetente transmite uma mensagem, o receptor acabará por aceitá-la." Nossa primeiro trabalho é novamente encontre a propriedade indesejável correspondente que pode ser expressa em uma reivindicação temporal. Para ser capaz de especificar isso, no entanto, precisamos ser capazes de nos referir a estados dentro do processos do emissor e do receptor. A notação necessária é usada nas seguintes formalidades da reclamação.

```

Nunca {
Faz
:: pular / * permitir qualquer atraso * /
:: receptor? [msg0] -> ir para aceitar0
:: receptor? [msg1] -> ir para aceitar1
od;
aceitar 0:
Faz
::! Receptor [2]: P0
od;
aceitar1:
Faz
::! Receptor [2]: P1
od
}

```

A afirmação acima é uma máquina de quatro estados: o estado inicial, os dois estados que foram etiquetada e o estado final normal. Pelo menos uma das três condições deve ser verdadeira no estado inicial do sistema. A reivindicação permanece neste estado enquanto o receptor do canal estiver vazio. Se contiver uma mensagem

`msg0` ou `msg1` mudará o estado para `aceitar0` ou `aceitar1`, dependendo da mensagem que foi correspondida. Assim que a transição para, para instância, o estado `aceita 0` foi feito, a reivindicação só pode permanecer nesse estado se o processo receptor nunca aceitará uma mensagem com o mesmo número de sequência, ou seja, se o processo receptor nunca passa do estado rotulado como `P0`.

Pode haver muitas instanciações do tipo de processo `Receiver`, então precisamos de alguma forma

de especificar exatamente qual instanciação particular queremos dizer quando nos referimos ao estado de um processo. Este é o único momento em que precisamos ser capazes de nos referir ao *índice de associação* ou o `pid` de um processo. O `pid` de um processo é o número que é retornado pelo operador `run`, quando um processo é instanciado. Os `pid`s são atribuídos em a ordem em que os processos são iniciados, mas podem ser reciclados quando os processos morrem. O processo inicial sempre tem `pid` zero e seu número nunca é reciclado. Um `pid` pode geralmente ser facilmente inferido do texto do programa. Uma vez que o processo receptor é o segundo processo que é instanciado neste sistema, seu `pid` é dois. Podemos nos referir ao receptor processa inequivocamente como Receptor [2]. A condição em que o receptor é atualmente no estado rotulado como `P0` é expresso como Receptor [2]: `P0`. A condição é falso sempre que o segundo processo instanciado está em qualquer estado diferente de

1. A notação para uma inspeção livre de efeitos colaterais do conteúdo de um canal de mensagem foi introduzida em Capítulo 5 na página 98.

Página 136

SEÇÃO 6.8
RESUMO
125

etiqueta `P0` do tipo de processo Receptor .

A notação Receiver [2]: `P0` é um caso especial de uma construção mais geral que permite que as reivindicações temporais se refiram às condições internas dos processos assíncronos definido dentro de um modelo PROMELA. Uma referência ao valor atual da variável local qualquer no processo receptor, por exemplo, é escrito Receiver [2] .any . Isso pode ser usado em expressões arbitrárias, como afirmar (Receptor [2]. qualquer <0)

Em referências de processo, dois pontos são usados para se referir a rótulos (ou seja, estados de fluxo de controle) e um

período é usado para se referir a variáveis locais. No primeiro caso, o valor retornado é um booleano. No segundo caso, é o valor inteiro da variável especificada.

A declaração de exemplo pode ser comprovada para o protocolo de bit alternado conforme especificado, usando

a semântica padrão de tempo limite . Isso significa que a reivindicação é correspondida, desde que os temporizadores de retransmissão se comportam como pretendido: não haverá retransmissão a menos que um

mensagem foi perdida. Para verificar também o caso bastante perverso em que o tempo limite pode disparar em

a qualquer momento, independentemente da perda de mensagem, podemos adicionar a definição de macro

```
#define timeout skip
```

Agora, a reivindicação temporal pode ser violada. Contra-exemplos são facilmente produzidos com um validador automatizado, como o discutido nos Capítulos 11-14.

6.8 RESUMO

Neste capítulo, apresentamos todos os recursos de linguagem restantes do validador de modelagem de código PROMELA . Todos estão diretamente relacionados com a especificação do requisitos de correção de um modelo. Eles são

Declarações de asserção

Rótulos de estado final, estado de progresso e estado de aceitação

A reivindicação temporal nunca

A notação para se referir aos estados de fluxo de controle e valores de variáveis locais de processos em execução dentro de asserções e reivindicações temporais

A ordem em que introduzimos as ferramentas para expressar propriedades de correção corresponde aproximadamente a um nível crescente de sofisticação na execução de validações.

Nos estágios iniciais de um projeto, é improvável que um usuário use mais do que afirmações e talvez rótulos de estado final. Nos estágios finais de um projeto, quando todas as falhas iniciais foi corrigido e uma avaliação qualitativa mais precisa do projeto pode ser feita,

validações com reivindicações temporais explícitas podem ser desenvolvidas. Em muitos casos, este nível de sofisticação em uma validação nunca é necessária e todas as propriedades necessárias podem ser estabelecido sem ele.

É quase impossível verificar manualmente os requisitos de correção, como os

Nós discutimos. O comportamento de até sistemas de protocolo muito simples é de um

complexidade impressionante que nenhum designer pode esperar para avaliar com precisão. Ferramentas são necessários não apenas para expressar os requisitos de correção de um projeto de protocolo, mas

Página 137

126

REQUISITOS DE CORREÇÃO

CAPÍTULO 6

também para verificar-los de forma confiável. Nos próximos capítulos, mostramos como podemos usar uma linguagem

como PROMELA para projeto de protocolo sistemático e como sistemas automatizados podem ser desenvolvidos para apoiar validações eficientes de declarações de correção.

EXERCÍCIOS

6-1. A afirmação usada para verificar a exclusão mútua no exemplo do semáforo foi formalizada como um sistema invariante global. Encontre outro lugar para a afirmação realizar o mesmo verificar sem um processo de monitor extra.

6-2. Encontre a reivindicação temporal nunca que expresse o mesmo requisito de correção que o definição de protótipo "comum"

monitor proctype () { (invariante) -> assert (invariante) }

Explique a diferença na semântica do ponto e vírgula (ou seta).

6-3. Considere o algoritmo de Dekker do Capítulo 5. Faça um novo modelo PROMELA, onde os processos acessam repetidamente suas seções críticas. Expresse o requisito de correção que dois processos não podem estar em suas seções críticas simultaneamente, com

Uma afirmação

Um sistema invariante

Rótulos de estado final

Uma reivindicação temporal nunca (combinada com rótulos de estado de aceitação)

Rótulos de estado final podem ser usados neste problema, por exemplo, forçando o sistema a um estado final impróprio quando a exclusão é violada. Você tem permissão para apresentar variáveis globais de "estado", por exemplo, para contar o número de processos dentro da seção crítica.

6-4. Repita o Exercício 6-3, mas desta vez não use nenhuma variável global nas asserções ou nas reivindicações temporais. Defina rótulos extras "normais" para definir pontos de fluxo de controle no orgãos do programa que podem ser monitorados em afirmações e reivindicações.

6-5. Para o modelo do Exercício 6-4, expresse o requisito de correção de que nenhum processo pode monopolizar o acesso à seção crítica. Expresse a afirmação de que dentro do infinito após a tentativa de acesso, o acesso é concedido. Usar

Rótulos de estado de progresso

Rótulos de estado de aceitação

Uma reivindicação temporal nunca combinada com rótulos de estado de aceitação

(Três soluções diferentes.) Adicione variáveis e rótulos conforme necessário.

6-6. Todas as "fórmulas lógicas temporais proposicionais de tempo linear" (ver Notas Bibliográficas) podem ser expressas na PROMELA com rótulos de estado de aceitação e reivindicações temporais. Ilustrar isso, encontre as representações PROMELA dos seguintes requisitos

Eventualmente, a proposição p sempre permanece verdadeira

p é sempre verdadeiro até que q se torne verdadeiro

Eventualmente, p é sempre verdadeiro até que q se torne verdadeiro

Você pode descobrir um padrão na modelagem dessas fórmulas em PROMELA temporal reivindicações? Pode ser automatizado?

6-7. Encontre uma maneira de traduzir reivindicações temporais arbitrárias de volta em fórmulas lógicas temporais.

6-8. Em reivindicações temporais, considere se a seguinte construção é equivalente a afirmar (0) aceitar: fazer :: pular od

Página 138

CAPÍTULO 6
NOTAS BIBLIOGRÁFICAS

127

NOTAS BIBLIOGRÁFICAS

A definição das sequências de término e de execução cíclica para raciocinar sobre o comportamento de um sistema distribuído é descrito em Owicky e Lampert [1982], Manna e Wolper [1984] e Snepscheut [1985]. A formalização da correção exige-mentos em uma notação que é baseada em fórmulas lógicas temporais foi explorada pela primeira vez em Pnueli [1977]. Logo foi aplicado também ao estudo de sistemas concorrentes. Cedo aplicações são o sistema de validação francês Cesar, Queille [1982], e Clarke's sistema de verificação de modelos, Clarke [1982], Browne, Clarke, Dill e Mishra [1986]. Formalmente, as reivindicações temporais definidas neste capítulo descrevem B não determinístico

..
uchi

autômato, uma classe especial do -automata descrito em Manna e Pnueli [1987]. No Wolper [1981] foi mostrado que B

.. autômatos uchi têm o poder expressivo de um tipo estendido de fórmulas lógicas temporais.

A dureza PSPACE é uma medida da complexidade dos algoritmos. Informalmente, um problema é difícil, sabe-se que não há uma solução eficiente. Para uma discussão, ver Garey e Johnson [1979].

Página 139

PROTOCOLO DESIGN 7

128 Introdução 7.1
129 Especificação de Serviço 7.2
130 Suposições sobre o Canal 7.3
131 Vocabulário de Protocolo 7.4
133 Formato de Mensagem 7.5
140 Regras de Procedimento 7.6
160 Resumo 7.7
160 exercícios
161 Notas Bibliográficas

7.1 INTRODUÇÃO

Até agora, nossa discussão sobre o design do protocolo cobriu a especificação e estruturação de métodos, princípios de design e soluções para um conjunto de problemas de protocolo padrão. Isto é hora de ver como podemos aplicar tudo isso a um problema real de design. Por exemplo, neste capítulo, empreendemos a concepção de um protocolo para a transferência de arquivos entre dois máquinas assíncronas.

Nosso objetivo neste capítulo é usar um método disciplinado para especificar os cinco elementos do protocolo (Capítulo 2, Seção 2.2). Para as regras de procedimento de protocolo, nosso objetivo é construir um protótipo de alto nível com critérios de correção explícitos. Mais tarde, seremos, então, capazes de mostrar de forma convincente que os critérios de design estão satisfeitos ou violados, usando ferramentas automatizadas.

Tentaremos fazer isso aplicando as regras de design listadas no Capítulo 2, Seção 2.8. O mais importante entre aqueles para fins de validação é a Regra 7:

Antes de implementar um design, construa um protótipo de alto nível e verifique se os critérios de design são atendidos.

Nosso protótipo é um modelo de validação, conforme definido nos Capítulos 5 e 6. Uma verificação que os critérios de design sejam atendidos é feita no Capítulo 14, com as ferramentas que são desenvolvidas

nos capítulos 11 a 13.

Fazemos duas suposições cruciais sobre o processo de design.

O projeto do protocolo é um processo iterativo. O projeto provavelmente não está correto na primeira vez que é escrito, e muito provavelmente não será totalmente correto na segunda ou pela terceira vez.

Pior, cada vez que uma fase de design for concluída, nós, os designers, estaremos contraconvencido de que está livre de erros. Um passo-a-passo manual do código pode revelar o grande erro menor, mas não se pode esperar que revelem também os sutis. Quase por definição, vamos ignorar os casos inesperados que podem causar erros.

128

Página 140

A construção de um modelo de validação formal (um protótipo) e um modelo rápido e imparcial verificador de exatidão automatizado é, portanto, indispensável para todos, exceto o mais simples designs. Os capítulos 11 e 13 discutem como um verificador de correção para modelos PROMELA pode ser construído. No Capítulo 14, a ferramenta de validação é aplicada ao design deste capítulo, seja para verificar se ele atende às especificações ou para revelar onde está falho.

Nossa preocupação neste capítulo é o design e a correção. Na discussão que se segue é importante ter em mente que estamos construindo um modelo de validação, não um implemento. O modelo é uma abstração e, como tal, é uma ferramenta de design em si.

Uma lista completa do protocolo desenvolvido aqui pode ser encontrada no Apêndice F. A validação do projeto com uma ferramenta chamada SPIN é discutida no Capítulo 14.

UM PROTOCOLO DE TRANSFERÊNCIA DE ARQUIVOS

O protocolo de transferência de arquivos que desenvolvemos pode ser classificado de várias maneiras diferentes.

É um protocolo *ponto a ponto*, ou seja, possui um emissor e um receptor. Protocolos com mais de um receptor às vezes são chamados *de protocolos multiponto* ou de transmissão.

O protocolo de transferência de arquivos fornece um serviço *ponta a ponta* entre dois usuários em dois máquinas diferentes, possivelmente se comunicando por meio de muitas máquinas intermediárias.¹

As regras de procedimento para o protocolo são desenvolvidas como uma sequência de modelos de validação

que podem ser verificados em suas propriedades de exatidão individualmente ou em combinação. A suposição em toda esta parte do projeto é que as ferramentas adequadas são disponíveis para verificar a consistência dos modelos de validação intermediários, e que eles podem ser refinados e ajustados com a ajuda dessas ferramentas.

Primeiro, vamos ter certeza de que o problema está completamente definido. Nós especificamos explicitamente

os outros quatro elementos do protocolo: a especificação do serviço, as suposições sobre o meio ambiente (o canal de transmissão), o vocabulário do protocolo e o formato da mensagem.

7.2 ESPECIFICAÇÕES DE SERVIÇO

O protocolo deve implementar um serviço confiável de transferência de arquivos de ponta a ponta. Este serviço

incluir estabelecimento e término de conexão, recuperação de erros de transmissão, e uma estratégia de controle de fluxo para evitar que o emissor transborde para o receptor. O protocolo deve ser capaz de transferir arquivos de texto ASCII, um de cada vez, com a probabilidade de um erro de bit não detectado sendo menor do que um modesto 1 em 10⁸ bits transmitidos. Nós exigem que o usuário seja capaz de abortar uma transferência de arquivo em andamento, e que o protocolo seja

capaz de se recuperar da perda de mensagens.

1. Cfr. a hierarquia OSI no Capítulo 2, Seção 2.6. Apenas as camadas 4 a 7 fornecem serviço ponta a ponta.

7.3 PRESSUPOSTOS SOBRE O CANAL

O protocolo deve ser projetado para transferência full-duplex de mensagens por voz, linhas telefônicas comutadas. Ter uma linha dedicada, em vez de uma conexão de rede, podemos ignorar com segurança problemas de rede específicos, como roteamento, controle de congestionamento,

atrasos na fila, etc., e foco no controle de simultaneidade. Para torná-lo interessante, nós suponha que as transferências ocorram entre Nova York e Los Angeles, uma distância de aproximadamente 4500 km. Dado que o tempo de propagação de um sinal elétrico em um cabo tem cerca de 30.000 km / s (Apêndice A), o tempo mínimo para uma mensagem para a viagem do emissor ao receptor leva aproximadamente 0,15 segundos.

A largura de banda e a relação sinal-ruído média de uma linha telefônica de nível de voz permitem para transmitirmos confortavelmente a uma velocidade de sinalização de 1200 bps, usando um modem padrão

(cf. Capítulo 3 e Apêndice A). Assumimos que a transmissão é orientada a caracteres, usando codificação de caracteres ASCII (Capítulo 2, Seção 2.5).

A maioria dos erros nas linhas telefônicas é causada por picos de ruído, ecos e conversas cruzadas. os erros de bit resultantes não são distribuídos uniformemente: eles vêm em rajadas. Portanto, para descrever as características do erro, devemos conhecer algumas funções de distribuição de erro. No Capítulo 3 (página 45), fornecemos vários métodos para prever a probabilidade de erro intervalos livres e durações de burst. Para os intervalos sem erros, estamos principalmente interessados na duração média, não nas distribuições de erro precisas, então vamos assumir uma sim-distribuição de Poisson ple

$$Pr(EFI = n) = f$$

.

e

$$- \frac{1}{f} \cdot n, n \geq 0$$

onde $1/f$ nos dá a duração média do intervalo sem erros. Nós vamos usar $f = 8,10$

\square 6, que corresponde a um intervalo sem erros médio de 125.000 bits transmissões (ou cerca de 100 segundos).

Para as rajadas de erro, estamos interessados em uma previsão mais precisa da probabilidade que uma determinada rajada dura pelo menos n transmissões de bits. Usamos a função de Mandelbrot:

$$Pr(Burst \geq n) =$$

$$\int_0^{\infty} e^{-(n-g)(1-a)t} dt$$

$$= e^{-ng(1-a)}$$

onde a e g são parâmetros. Os valores dos parâmetros que escolhemos aqui irão determinar mina o tipo de método de controle de erro que será necessário. Os valores precisos devem ser derivado de medições nos canais telefônicos usados. Eles são, no entanto,

amplamente irrelevante para a discussão que se segue. Assumiremos $a = 0,9$ e

$g = 0,009$, uma escolha que corresponde à nossa intuição sobre o comportamento dos erros de burst. A duração média de um erro de explosão que pode ser calculado para esses valores de parâmetro é tempos de transmissão de aproximadamente 12 bits (10 ms). A Figura 7.1 mostra como a probabilidade de

uma rajada depende de seu comprimento, usando a previsão acima. O eixo y é logarítmico.

Se conseguirmos reduzir a taxa de erro residual para não mais do que 1 em 10^8 transmitidos bits, a 1200 bps, isso dá uma expectativa de não mais do que um erro não detectado para a cada 23 horas de operação contínua.

Página 142

0
10
20
30
0,01
0,1
1

$Pr(Burst \geq n)$

Burst Length n

$a = 0,9, g = 0,009$

Figura 7.1 - Probabilidade de erros de ruptura

7.4 VOCABULÁRIO DE PROTOCOLO

Considere o protocolo como uma caixa preta. Para desempenhar sua função, o protocolo deve comunicar-se com seu ambiente. Ele troca mensagens com o sistema remoto por meio de um link de dados e com o usuário local e um servidor de arquivos local via mensagem interna canais.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....



Dados Ligaçāo

Figura 7.2 - Ambiente de protocolo

Sem entrar em detalhes sobre o protocolo em si ainda, vamos ver quais tipos de são necessárias mensagens para construí-lo.

A caixa preta aceita dois tipos de mensagens do usuário local. A primeira mensagem tipo é usado para iniciar uma transferência de arquivo. Ele pode ter um único parâmetro: `transferir (file_descriptor)`

O segundo tipo de mensagem pode ser usado pelo usuário para interromper uma transferência em andamento.

abortar

A mensagem de transferência deve acionar uma mensagem para o servidor de arquivos local para verificar se

o arquivo a ser transferido realmente existe e, em caso afirmativo, qual é o tamanho do arquivo. Nós chamamos

aquela mensagem

abrir (file_descriptor)

Se o arquivo pode ser aberto, seu tamanho é determinado e uma conexão é feita com o sistema remoto. Para comunicar a solicitação de conexão do local para o remoto sistema usamos a mensagem

conectar (tamanho)

No lado remoto, uma solicitação de conexão recebida novamente aciona uma mensagem para o arquivo servidor, desta vez para verificar se um novo arquivo de determinado tamanho pode ser armazenado. A mensagem

mensagem
porque isso pode ser
criar (tamanho)

Precisamos pelo menos mais duas mensagens do servidor de arquivos para indicar se um aberto ou uma solicitação de criação pode ser aceita ou deve ser rejeitada:
aceitar (tamanho)

e
rejeitar (status)

Em resposta a uma solicitação aberta, a mensagem de aceitação retorna o tamanho do arquivo para a chamada

processo. Se a solicitação for rejeitada, um parâmetro inteiro pode ser usado para transportar informações sobre o motivo. Os tipos de mensagem aceitar e rejeitar também podem ser usados para informar o usuário local sobre o sucesso ou falha de uma transferência de arquivo de saída. E da mesma forma, eles podem ser usados pelo sistema local para informar um sistema remoto se um

a transferência de arquivos recebidos é aceita.

Para transferir um arquivo, quatro coisas devem acontecer em uma ordem específica:

Estabelecimento de uma conexão com o servidor de arquivos local

Estabelecimento de uma conexão com o sistema remoto

Transferência de dados

Encerramento ordenado da conexão

Cada fase do protocolo tem seu próprio vocabulário de mensagens e suas próprias regras para interfingindo-os. Cada fase pode ter de ser subdividida em etapas ainda menores.

A segunda fase, por exemplo, consistirá em duas etapas: (1) uma inicialização do protocolo de controle de fluxo e (2) um aperto de mão com o sistema remoto usando a conexão e aceitar ou rejeitar mensagem. A sincronização do local e do remoto

protocolos de camada de controle de fluxo são necessários para garantir que eles concordam com os números de sequência a serem usados. Podemos usar a mensagem

sincronizar

e seu reconhecimento

sync_ack

para este propósito.

Na terceira fase do protocolo, precisamos de mensagens para recuperar os dados do arquivo servidor e transmiti-los para o sistema remoto. Por exemplo, podemos usar uma mensagem sábio

dados (cnt, ptr)

para transferir bytes cnt de informação, disponível em um buffer de dados que é identificado pelo segundo parâmetro. Obviamente, esta não é necessariamente a forma como as interações com o servidor de arquivos deve ser implementado em um design final. Por enquanto, no entanto, basta que ele modele com precisão os fundamentos dessas interações.

Outra mensagem,

eof

pode ser usado pelo servidor de arquivos para indicar o fim de um arquivo. A conclusão correta de um

Página 144

a transferência de arquivos, com a mensagem eof , pode ser confirmada pelo sistema remoto com um simples mensagem gle

fechar

Até agora, a única suposição que fizemos sobre o servidor de arquivos é que ele reconhece seis mensagens: abertos ,criar ,aceitar ,rejeitar ,dados ,e EOF . Para evitar a sincronização problemas de instalação entre o sistema local e o servidor de arquivos local, assumimos que o acima de seis mensagens são trocadas por comunicação de encontro (ou seja, eles são equivalentes a chamadas de procedimento local). Isso garante que, por exemplo, dados não utilizados as mensagens de ou para o servidor de arquivos não se acumulam. Após a conclusão bem-sucedida de uma solicitação de abertura local e de criação remota , as mensagens de dados são usadas para transferir dados do servidor de arquivos local para o servidor de arquivos remoto, usando o protocolo a ser desenvolvido.

Precisamos de uma mensagem extra para implementar uma disciplina de controle de fluxo simples que ack-

nowledges recebeu dados corretamente:

ack

O vocabulário completo do protocolo consiste então em treze tipos distintos de mensagens.

Nove dessas mensagens podem ser trocadas pelas duas máquinas remotas: aceitar ,ack , perto , conexão , dados , EOF , rejeitar , sync_ack ,e sincronização . Os outros quatro abortam ,

criar , abrir ,e transferência ,são apenas mensagens internas.

7.5 FORMATO DE MENSAGEM

Agora deve ser decidido qual é o formato correto para as mensagens acima. Temas- Os sábios requerem, no mínimo, um campo de tipo e um campo de dados opcional. Para implementar um fluxo

disciplina de controle, as mensagens enviadas para o sistema remoto também devem levar sequência números e, para implementar o controle de erros, eles devem conter um campo de soma de verificação. Desde o

canal de transmissão é orientado por bytes, podemos formatar cada mensagem como uma sequência de

bytes. Obviamente, gostaríamos que essas sequências fossem as mais curtas possíveis. Deixe-nos ver como podemos calcular o tamanho mínimo necessário, sabendo apenas o que sabemos até agora sobre o protocolo e o canal de transmissão a ser utilizado.

No nível mais alto, uma mensagem pode ser codificada em uma série de bytes, indicando seu tipo e o valor de seus parâmetros. Antes que a mensagem seja enviada, o controle de fluxo acrescenta um número de sequência e o controle de erro acrescenta um campo de soma de verificação. No nível mais baixo, pouco antes de a mensagem ser colocada na linha de transmissão, um driver de linha acrescenta os delimitadores de mensagem, para permitir que o sistema remoto reconheça onde um a mensagem começa e para. Os padrões ASCII de 8 bits STX (início do texto) e ETX (fim do texto) pode ser usado para este fim. Com o enchimento de bytes (Capítulo 2), má interpretação delimitadores de mensagem que fazem parte dos próprios dados podem ser evitados.

O receptor remoto remove os caracteres STX e ETX e remove os caracteres empalhados ters. O controle remoto de erro retira e interpreta a soma de verificação e o fluxo remoto tira de controle e interpreta o número de sequência. Se tudo estiver bem, a mensagem finalmente chega ao seu par como a série original de bytes novamente.

A melhor posição dos vários campos de mensagem na sequência de bytes depende de alguns

Página 145

extensão na codificação das rotinas de protocolo no nível mais baixo (a camada física) na hierarquia. Colocar o campo de soma de verificação no final de uma mensagem tem a vantagem prova de que o remetente pode computá-lo instantaneamente enquanto transmite o corpo da mensagem sábio. Também pode haver uma pequena diferença no desempenho, dependendo de onde o campo do tipo de mensagem é colocado. Ele pode ser colocado na frente da mensagem, em um lugar fixo atrás da STX símbolo, ou no final, em um lugar fixo antes da ETX symbol. Uma vez que temos mensagens de comprimento variável, a posição do símbolo ETX é menor previsível do que o do símbolo STX . Colocando o campo de tipo no início da mensagem O sage pode, portanto, facilitar a análise de uma mensagem recebida.

Assim, chegamos ao formato de mensagem mostrado na Figura 7.3, onde o campo *Data* é ausente nas mensagens de controle.

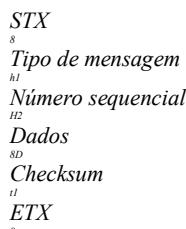


Figura 7.3 - Formato da mensagem

Na Figura 7.3, um símbolo que representa o comprimento de cada campo em bits é indicado no canto inferior direito. D bytes de informação no campo de dados corresponde à largura do campo de bits $8D$. Como são os números $h1$, $h2$, D , e $t1$ determinado?

O CAMPO TIPO DE MENSAGEM

Este é o mais fácil dos quatro números de calcular. Temos 13 tipos diferentes de mensagem. Como $2^3 < 13 \leq 2^4$, 4 bits para o campo de tipo são suficientes.

$h1 = \boxed{4}$

O CAMPO DE NÚMERO DE SEQUÊNCIA

Agora, vamos derivar uma largura apropriada para o campo de número de sequência $h2$. Temos O tempo de propagação do sábio é de 0,15 segundos, o suficiente para transmitir 180 bits. Se uma mensagem

seria devolvida pelo receptor sem atrasos no processamento, o remetente poderia transmitir 360 bits antes de receber a mensagem de retorno. Gostaríamos de projetar o protocolo de tal forma que o remetente pode saturar completamente o canal sempre que o receptor consome os dados tão rápido quanto o remetente os produz. Devemos, portanto certifique-se de que o remetente não terá que esperar ociosamente por confirmações quando pode estar enviando mensagens. Isso significa que o remetente deve ser capaz de ser pelo menos 360 bits à frente do receptor em circunstâncias normais. O número de mensagens a que isso corresponde depende do número de bytes no campo de dados e no cabeçalho e trailer.

Se usarmos um método ARQ contínuo de repetição seletiva (Capítulo 4), o mais flexível

método de controle de fluxo que discutimos, o número máximo de mensagens pendentes

sábios com um campo de número de sequência de h_2 bits é 2

$h_2 = 1$

. Com uma sequência de um bit

número, apenas uma mensagem pode ficar pendente por vez, forçando o remetente a permanecer

Página 146

inativo por pelo menos 0,3 segundos, aguardando o reconhecimento dessa mensagem (duas vezes o tempo de propagação da mensagem). Com um número de sequência de dois bits, duas mensagens podem

ser excelente, o que significa que o remetente pode transmitir uma mensagem enquanto aguarda o reconhecimento por outro. Se essa mensagem tiver pelo menos 360 bits, nenhum tempo será perdido. Abaixo, veremos que é realmente vantajoso usar um campo de dados maior que 360 bits. Então nós temos

$h_2 \geq 2$

Assumindo novamente que o receptor é pelo menos tão rápido quanto o remetente, poderíamos organizar o

controle de fluxo da seguinte forma. Todas as mensagens na janela atual são enviadas antes do reconhecimento da mensagem pendente mais antiga é verificado. Se nenhum reconhecimento é recebida nessa altura, a mensagem mais antiga é retransmitida. Se um reconhecimento-for recebido, a janela desliza para cima e uma nova mensagem pode ser transmitida ted. O remetente nunca fica ocioso e o canal está saturado.

Observe com atenção que se o receptor for mais lento do que o remetente, é prudente não tentar saturar o canal: o receptor precisa de tempo para alcançar o emissor. Para este caso podemos incluir um cronômetro. Se nenhum reconhecimento para o quadro pendente mais antigo é recebido no momento em que o remetente verifica se há um, o remetente agora espera pelo menos um adicional

período de tempo limite tradicional para a confirmação chegar. O valor apropriado para o a contagem de tempo limite pode ser estimada ou ajustada dinamicamente com um *controle de taxa* método. Devemos nos restringir ao caso ideal, com um canal totalmente saturado nel.

O CAMPO DE DADOS

O comprimento do campo de dados é expresso na contagem de bytes D e é variável por dados mensagem. Não é nossa vantagem deixar uma mensagem muito longa, pois a expectativa o fato de uma mensagem conter erros de bit aumenta trivialmente com seu comprimento. No outro lado, se tornarmos as mensagens muito pequenas, a sobrecarga t do cabeçalho e do trailer torna-se muito grande. A duração média de um intervalo livre de erros foi estimada em aproximadamente 125.000 bits de transmissão, então certamente não devemos fazer uma mensagem mais do que isso. Em algum lugar entre esses 125 Kbits e os 360 bits que derivamos anteriormente, deve haver um comprimento ideal para o campo de dados. Podemos aproximar isso ótimo da seguinte forma.

TAMANHO DE DADOS OTIMAL

Seja t o comprimento do overhead da mensagem de dados em bits (cabeçalho mais trailer) e d o comprimento do campo de dados, também em bits: $d = 8D$. Além disso, seja a a duração de um ACK mensagem de controle, p_d a probabilidade de uma mensagem de dados ser distorcida ou perdida, e p_a a probabilidade do mesmo erro para um reconhecimento.

Vamos primeiro considerar o caso em que $p_d = p_a = 0$. A transmissão de uma mensagem requer uma mensagem de dados e uma mensagem de confirmação, um total de $d + t + a$ bits.

2. Consulte também "taxas de código" definidas no Capítulo 3, Seção 3.5.

Página 147

A sobrecarga é $t + a$, e a eficiência do protocolo é

$E =$

$d + t + a$

d

Portanto, na ausência de erros, é melhor escolher d o maior possível.

Agora considere o caso em que p_d e p_a são diferentes de zero. A probabilidade de que nem o

mensagem de dados nem seu reconhecimento é atingido por um erro de transmissão é $(1 - p_d)(1 - p_a)$. Portanto, a probabilidade de que a mensagem deve ser retransmitida é

$$p_r = \square(1 - \square(1 - \square p_d))(1 - \square p_a)$$

A probabilidade de que é preciso i transmissões subsequentes para obter a mensagem de dados através, $i - 1$ retransmissão e uma transmissão bem-sucedida, é

$$p_i = \square(1 - \square p_r) p_r$$

$$_{eu-1}$$

O número esperado de transmissões por mensagem R é então dado por

$$R =$$

$$_{i=1}$$

④

$$ip_i$$

$$=$$

$$_{i=1}$$

④

$$i(1 - \square p_r) p_r$$

$$_{eu-1}$$

$$= \square(1 - \square p_r)$$

$$_{i=1}$$

④

$$ip_r$$

$$_{eu-1}$$

$$= \square(1 - \square p_r)$$

$$_{j=0}$$

④

$$i = \square j$$

④

$$p_r$$

$$_{Eu}$$

$$= \square(1 - \square p_r)$$

$$_{j=0}$$

④

$$1 - \square p_r$$

$$p_r$$

$$j$$

$$= \frac{1}{\square}$$

$$=$$

$$1 - \square p_r$$

$$1$$

A eficiência relativa E , na presença de erros, é então

$$E =$$

$$R(d + \square t + \square a)$$

$$d$$

O valor ideal para d pode ser encontrado definindo a derivada de E em relação a d para zero: $\text{TM} \square E / \text{TM} \square d = 0$. Alternativamente, podemos simplesmente preencher o valor conhecido ou aproximado

valores para R , t e a , e gráfico E como uma função de d .

Com os valores que derivamos anteriormente, e contando 8 bits cada para as mensagens STX e ETX delimitador sábio, nós temos

$$a = \square h_1 + \square h_2 + \square t + 16$$

$$= \square 4 \square + \square 2 \square + \square 16 \square + \square 16 \square = \square 38$$

e trivialmente

$$t = \lceil a \rceil = 38$$

Presumimos anteriormente que um intervalo sem erros duraria em média 125 Kbits.

Isso corresponde a $(125,10^3) / (d + \lceil t \rceil)$ mensagens ou $(125,10^3) / a$ confirmações.

Uma aproximação de primeira ordem pode então ser obtida tomando

$$p_d = (d + 38) / (125,10^3)$$

e

$$p_a = 38 / (125,10^3) = 3,4 \cdot 10^{-4}$$

Agora temos

$$R =$$

$$1 - \lceil p_r \rceil$$

$$1 - \frac{1}{(1 - \lceil p_d \rceil)(1 - \lceil p_a \rceil)}$$

=

$$(1 - \lceil p_d \rceil)(1 - \lceil p_a \rceil)$$

1

Substituindo na última expressão para E dá:

$$E =$$

$$d + \lceil t \rceil + \lceil a \rceil$$

$$d \cdot (1 - \lceil p_d \rceil)(1 - \lceil p_a \rceil)$$

$$= \frac{d}{d + 76}$$

$$d = d \cdot (d + 38) / (125,10^3) (1 - 38 / (125,10^3))$$

Na Figura 7.4, a eficiência do protocolo E é traçada como uma função do comprimento do campo de dados da mensagem d .

2500

3000

3500

0,9506

0,9508

0,951

0,9512

E

d

Figura 7.4 - Eficiência do protocolo versus tamanho dos dados em bits

Há claramente um valor ótimo para d . A eficiência máxima de 95,13% é alcançado para um tamanho de dados de 3004 bits ou 376 bytes. Portanto, agora derivamos o terceiro valor D para a Figura 7.3.

Na realidade, é claro, a probabilidade de dados perdidos e distorcidos deve ser derivada de funções de distribuição de erros. Devíamos ter resolvido as equações mais difíceis

$$p_d =$$

$$i = 0$$

④

$d + 38$

$$\Pr_{i=0} (\text{EFI} = i) p_d =$$

⑤

38

$$\Pr_{i=0} (\text{EFI} = i)$$

Para ver quão grande é o erro que cometemos, definimos $d = 3004$ e calculamos

$$p_d =$$

$$i = 0$$

©

$3004 + 38$

$$Pr(EFI = \square_i) = 240,42 \cdot 10$$

$\square 4$

e

$$p_a =$$

$i = 0$

©

38

$$Pr(EFI = \square_i) = 3,039 \cdot 10$$

$\square 4$

As aproximações anteriores fornecem o mesmo valor de d

$$p_d = (3004 + 38) / (125,10^3)$$

$$= 243,36 \cdot 10$$

$\square 4$

e

$$p_a = 38 / (125,10^3)$$

$$= 3,04 \cdot 10$$

$\square 4$

Nossa primeira estimativa está dentro de 1,2 por cento dos valores recalculados. Uma vez que não temos razão para confiar no valor preditivo da função de distribuição de erro com esse grau de precisão, decidimos pela primeira estimativa.

O CAMPO CHECKSUM

Tudo o que resta é derivar o valor para t_1 , a largura do campo de checksum. O canal tem erros de exclusão e distorção, mas não se espera que produza inserções ou reordenamento de mensagens. A taxa de erro é baixa o suficiente para que não precisemos de um erro código de correção. Para um comprimento de mensagem razoável, bem abaixo de 125 Kbits, a maioria das mensagens

sábios passam sem erros de transmissão. Devemos, no entanto, ser capazes de corrigir para os erros característicos do canal: erros de burst. Isso torna um redun cílico dancy verifique uma boa escolha. A duração média de um erro de ruptura foi assumida como 10 msec, afetando uma sequência de 12 bits. O grau do polinômio gerador portanto, pelo menos, deve ser maior que 12 para detectar esses erros.

A taxa de erro de bit residual alvo é 10

$\square 8$. Para poder verificar se podemos cumprir este requisito, escolhendo o polinômio de soma de verificação CRC-CCITT de 16 bits, temos para observar a distribuição das durações dos erros de burst. Sabemos que o CRC-CCITT a soma de verificação captura todos os erros de bit único e duplo, todos os números ímpares de erros de bit, todos erros de burst de até 16 bits de comprimento, 99,997% dos erros de burst de 17 bits e 99,998% de todos erros de rajada com mais de 17 bits.

Presumimos que os erros de burst ocorrem em média uma vez a cada 100 segundos. Uma explosão de aproximadamente 12 bits de comprimento, então corresponde a uma taxa média de erro de bits de longo prazo de 10

$\square 4$.

Dos bursts com mais de 16 bits, dois ou três em cada 10⁵ bursts irão não detectado. Se cada rajada tivesse mais de 16 bits, isso significaria que duas ou três

Página 150

erros de rajada por $100 \cdot 10^5$ segundos iriam passar, correspondendo a um longo prazo taxa média de erro de bit de aproximadamente 10

$\square 6$ (há mais de dez bits em cada burst).

Portanto, podemos apenas realizar uma taxa de erro de bit residual de 10

$\square 8$ se erros de burst

mais de 16 bits ocorrem menos de uma vez em cada 10² bursts, ou menos de uma vez a cada 10.000 segundos. Usando a função de distribuição de probabilidade, encontramos (cf. Figura 7.1):

$$Pr(Burst \geq 17) = 0,009 \cdot e$$

$\square 0,009,17$

$$= 0,007$$

O resultado está dentro do alcance do alvo. Podemos escolher a soma de verificação de 16 bits e ter

nosso último valor:

$$t_1 = \square 16$$

Para as mensagens de controle, parece um exagero incluir até mesmo uma soma de verificação de 16 bits para

uma mensagem que contém apenas dois pequenos números. Observe, no entanto, que um erro de explosão pode

apague a mensagem completa, então com a mesma redundância um código de correção de erros não poderia ter um desempenho melhor.

Como um aparte, o formato da mensagem, com todas as larguras de campo que agora derivamos, pode ser definido em C como segue.

```
struct {
    tipo sem sinal: 4;
    sem sinal semej: 2;
```

```
sem sinal seqno: 2;
dados de caracteres não assinados [376];
soma de verificação de char unsigned [2];
} mensagem;
```

Usamos campos de bits para os campos no cabeçalho da mensagem e caracteres não assinados (8 bits largura) para o campo de dados e a soma de verificação. Os delimitadores de mensagem STX e ETX eram omitido. Esta não é, necessariamente, a maneira pela qual um compilador C faz com que os bits sejam armazenados na memória. Pode ocorrer algum preenchimento para alinhar bits-

campos com limites de palavra ou byte.

EFEITOS DE ARREDONDAMENTO

Na Figura 7.4, vemos que a eficiência do protocolo não é muito sensível às variações na o tamanho dos dados próximo ao ótimo. Também estamos presos aos valores peculiares para h_1 e h_2 que calculamos anteriormente. Múltiplos de 8 bits seriam mais convenientes para o receptor para processar. Vamos ver como a eficiência seria afetada se usássemos o mais próximo múltiplo de 8 para todas as larguras de campo, conforme mostrado na Tabela 7.1.

Tabela 7.1 - Arredondamento

Os valores recalculados para E em função de d para esses novos valores são indicados por

a curva inferior na Figura 7.5. O efeito de adicionar 10 bits de overhead é uma redução de E por menos de um por cento, o que deve ser considerado insignificante em comparação com os erros introduzidos por aproximações anteriores. O novo ótimo é alcançado por um

comprimento de dados de 3370 bits, ou cerca de 422 bytes (46 bytes a mais do que antes).

2500
3000
3500
4000
4500
0,944
0,946
0,948
0,95
E
d

Figura 7.5 - Degradação da eficiência do protocolo por arredondamento

Após o arredondamento, o formato da mensagem pode ser escrito sem campos de bits

```
struct {  
    tipo de char unsigned;  
    unsigned char seqno;  
    dados de caracteres não assinados [422];  
    soma de verificação de char unsigned [2];  
} mensagem;
```

7.6 REGRAS DE PROCEDIMENTO

Agora estamos prontos para começar com a descrição das regras de procedimento. Nesta parte de o design, poucas coisas podem ser calculadas ou medidas. No entanto, as regras devem ser completo e consistente. Antes de nos comprometermos diretamente com uma implementação, nós gostaria de ser capaz de projetar e depurar as regras de uma forma intermediária, para instância como um modelo de validação. O PROMELA foi projetado exatamente para este propósito. Primeiro, olhamos para algumas das abstrações que temos que fazer na modelagem da mensagem sábios. Em seguida, olhamos para as camadas do protocolo e derivamos uma estrutura global aproximada ture. Finalmente, da camada mais alta para a mais baixa, refinamos cada camada e compomos bine-os no design final. Os requisitos de exatidão para cada camada são para- malizado usando a notação desenvolvida no Capítulo 6.

ABSTRAÇÃO

A codificação precisa das mensagens, derivada nas seções anteriores, também contém muita informação para o problema que enfrentamos agora. Manipulando mensagens neste nível de detalhe complicaria desnecessariamente o design, descrição e validação de as regras de procedimento. Para derivar as regras de procedimento, construímos um modelo do

Página 152

sistema de comunicação em que consideramos apenas a semântica do protocolo, não sua sintaxe precisa. Para o modelo, por exemplo, o conteúdo dos arquivos transferidos são irrelevantes. O campo de dados de comprimento variável da Figura 7.3, portanto, não precisa ser representado, nem (como argumentaremos abaixo) o campo de soma de verificação. Usamos um tempo de mensagem

placa de apenas dois campos no modelo de validação.

fld1, fld2

O primeiro campo representa o tipo de mensagem genérica, por exemplo, *dados*, e o segundo campo carrega um parâmetro, como um número de sequência. Ambos os campos podem ser do tipo PROMELA byte.

O que estamos projetando a partir deste ponto é um modelo de validação, não uma implementação ção. Se fizermos nosso trabalho direito, porém, deve ser simples remover as abstrações do modelo, refine-o quando necessário e obtenha uma implementação.

CAMADAS

Para estruturar o design, nós o dividimos em várias camadas. O usuário interage com um protocolo da camada de apresentação. Abaixo disso, colocamos uma camada de controle de sessão, e abaixo

que uma camada de enlace de dados que impõe uma disciplina geral de controle de fluxo. O link de dados é

a linha de dados física, equipada com modems, para codificação de dados binários em analógico- sinal, e fornecendo detecção de erro em cada mensagem transmitida usando CRC- Soma de verificação CCITT. O link de dados com o qual trabalhamos a partir deste ponto, portanto, pode perder, mas não distorcer mensagens.

Do utilizador

Processo
Presente-
ação
Sessão
Fluxo
Ao controle
Protegido contra erros
Link de dados
Servidor de arquivos

Figura 7.6 - Hierarquia de protocolo

Cada camada nesta hierarquia é gerenciada por um ou mais processos PROMELA, como indicado com o protocolo "pipeline" na Figura 7.6.

AMBIENTE DE PROTOCOLO

Nosso primeiro trabalho na construção de um modelo de validação é tornar explícitas todas as suposições relevantes

informações sobre o comportamento do ambiente, conforme ilustrado nas Figuras 7.2 e 7.6. O ambiente consiste em três entidades:

Processo do usuário
 Servidor de arquivos
 Link de dados

As suposições mínimas que devemos fazer sobre o comportamento de cada um desses três são formalizado em código PROMELA.

Considere primeiro o protocolo de nível de usuário. Pode haver dois processos de usuário: um em cada

Página 153

fim do link de dados. Os usuários podem enviar uma solicitação de transferência a qualquer momento, passando um

descritor de arquivo para a camada de apresentação do protocolo. A qualquer momento após a transferência

solicitação, o usuário de origem também pode decidir abortar uma transferência. Assumimos que o usuário então espera por uma resposta das camadas de protocolo inferiores, sinalizando o conclusão bem-sucedida ou malsucedida da transferência. Podemos modelar essas suposições com o seguinte processo PROMELA.

```

proctype userprc (bit n)
{
  use_to_pres [n]! transferência;
  E se
  :: pres_to_use [n]? aceitar -> ir para Concluído
  :: pres_to_use [n]? rejeitar -> ir para Concluído
  :: use_to_pres [n]! abortar -> goto abortado
  fi;
  Abortado:
  E se
  :: pres_to_use [n]? aceitar -> ir para Concluído
  :: pres_to_use [n]? rejeitar -> ir para Concluído
  fi;
  Feito:
  pular
}
  
```

O argumento binário *n* identifica o usuário e os canais que ele acessa. Tema-
sage *transfer* normalmente carrega um parâmetro que aponta para o arquivo a ser
transferido (por exemplo, um descritor de arquivo). Para uma validação, no entanto, o valor desse
parâmetro

é irrelevante e, portanto, não está presente no modelo. Claramente, a correção
do protocolo de transferência de arquivos não deve depender dos descritores de arquivo específicos
usados.

Os canais de mensagem que usamos no modelo podem ser definidos globalmente. Cada flecha em
A Figura 7.6 corresponde a dois desses canais, um para cada lado do protocolo. Usando
um esquema de nomenclatura simples para indicar quais camadas cada canal conecta, podemos
definir os seguintes tipos de canais. Existem dois usuários, e *QSZ* é um tamanho de fila
de pelo menos um (cf. Figura 7.6).

```

chan use_to_pres [2] = [QSZ] de {byte};
chan pres_to_use [2] = [QSZ] de {byte};
chan pres_to_ses [2] = [QSZ] de {byte};
chan ses_to_pres [2] = [QSZ] de {byte, byte};
chan ses_to_flow [2] = [QSZ] de {byte, byte};
chan flow_to_ses [2] = [QSZ] de {byte, byte};
  
```

```
chan dll_to_flow [2] = [QSZ] de {byte, byte};  
chan flow_to_dll [2] = [QSZ] de {byte, byte};
```

Os canais para a comunicação síncrona entre o protocolo da camada de sessão e o servidor de arquivos são definidos com capacidade de buffer zero, da seguinte forma:

```
chan ses_to_fsrv [2] = [0] de {byte};  
chan fsrv_to_ses [2] = [0] de {byte};
```

Isso traz o total para dez tipos diferentes de canais de mensagem, com uma cópia sendo instanciado para cada lado da conexão. Os canais usados para o protocolo superior as camadas podem ser definidas com um formato de mensagem mais simples do que as camadas inferiores. o

Página 154

O campo do número de sequência, por exemplo, é usado apenas pela camada de controle de fluxo. Em seguida, devemos formalizar nossas suposições sobre o comportamento do servidor de arquivos. Novamente, nenhuma decisão de design foi feita ainda. O objetivo é apenas tornar explícito o que deve ser minimamente assumido sobre o comportamento externo do processo do servidor de arquivos. A transferência de arquivos recebidos começa com uma mensagem de `criação`. O servidor de arquivos responde com uma `rejeição` ou uma mensagem de `aceitação`. No que diz respeito ao modelo de validação, ambas as escolhas são igualmente prováveis, portanto, podem ser modeladas como não determinísticas. Se o a solicitação for aceita, zero ou mais mensagens de dados seguem, e o servidor de arquivos retrocede em seu estado inicial após a recepção do `eof` final ou, no caso de um aborto, o `fechar` mensagem. Em uma primeira aproximação, desconsiderando o que dissemos anteriormente sobre abstração, podemos tentar descrever este comportamento em um modelo de validação PROMELA como segue.

```
proctype fserver (bit n)  
{  
    int fd, tamanho, ptr, cnt;  
    Faz  
    :: ses_to_fsrv [n]? criar (tamanho) ->  
    / * entrando * /  
    E se  
    / * escolha não determinística * /  
    :: fsrv_to_ses [n]! rejeitar  
    :: fsrv_to_ses [n]! aceitar ->  
    Faz  
    :: ses_to_fsrv [n]? dados (cnt, ptr)  
    :: ses_to_fsrv [n]? eof -> pausa  
    /* abortar */  
    :: ses_to_fsrv [n]? fechar -> quebrar  
    od  
    fi  
    :: ses_to_fsrv [n]? open (fd) -> / * de saída * /  
    ...  
    od  
}
```

Mas estamos no caminho errado com este modelo. Observe que as variáveis e mensagens locais parâmetros de sálvia `fd`, `tamanho`, `cnt`, e `ptr` realmente fornecer detalhes indesejados no modelo. Estamos projetando as regras de procedimento para a interação do servidor de arquivos com o sessão-protocolo da camada de sessão. Queremos especificar como esses dois módulos interagem, ou seja, os tipos de mensagens que eles vão trocar e as expectativas inerentes sobre a ordem em que os diferentes tipos de mensagens chegarão e serão enviados. Importante especificar neste nível de abstração é *quando* os dados podem ser passados, não *quais* dados serão ser passado. O tamanho e o conteúdo dos dados transferidos são, portanto, ainda irrelevantes em este nível de modelagem (cf. Capítulo 14 sobre a generalização formal de modelos).

A melhor maneira de modelar o comportamento relevante do servidor de arquivos para os dados de entrada é

```
proctype fserver (bit n)  
{  
    Faz  
    :: ses_to_fsrv [n]? criar ->  
    / * entrando * /
```

Página 155

```

E se
:: fsrv_to_ses [n]! rejeitar
:: fsrv_to_ses [n]! aceitar ->
Faz
:: ses_to_fsrv [n]? dados
:: ses_to_fsrv [n]? eof -> pausa
:: ses_to_fsrv [n]? fechar -> quebrar
od
fi
:: ses_to_fsrv [n]? abrir ->
/* extrovertido */
...
od
}

```

O modelo para dados de saída é semelhante. Depois que o servidor recebe uma mensagem `aberta`, pode responder com uma mensagem de aceitação ou `rejeição`. Se a solicitação for aceita, uma série de mensagens de `dados` é transferida, seguida por uma única mensagem `eof`. O arquivo a transferência pode ser abortada novamente por uma mensagem de `fechamento` da camada de sessão para o arquivo

`servidor`.

```

:: ses_to_fsrv [n]? abrir ->
/* extrovertido */
E se
:: fsrv_to_ses [n]! rejeitar
:: fsrv_to_ses [n]! aceitar ->
Faz
:: fsrv_to_ses [n]! dados
:: fsrv_to_ses [n]! eof -> pausa
:: ses_to_fsrv [n]? fechar -> quebrar
od
fi

```

A última parte do ambiente é o link de dados. Novamente, devemos tornar explícito todos nossas suposições sobre o seu comportamento, na medida em que se relaciona com o protocolo que estamos

projetoando.

O link de dados é considerado protegido por um protocolo de detecção de erros. Os detalhes do cálculo da soma de verificação pode ser encontrado no Capítulo 3, Seção 3.7, mas para esta parte do problema de design esses detalhes são irrelevantes. O cálculo da soma de verificação é um computadoração, e não um padrão de interação. Seria tolice tentar modelá-lo em detalhes em PROMELA como se fosse uma regra de procedimento.

Tudo o que nos interessa aqui é o comportamento externo do link de dados. O link de dados, então, pode omitir mensagens arbitrariamente das sequências que ele passa, usando alguns den oracle para decidir o destino de cada mensagem. A escolha pode ser modelada como um não um determinístico.

```

proctype data_link ()
{tipo de byte, seq;

```

```

Faz
:: flow_to_dll [0]? tipo, seq ->
E se
:: dll_to_flow [1]! tipo, seq
:: pular / * perder * /
fi
:: flow_to_dll [1]? tipo, seq ->
E se
:: dll_to_flow [0]! tipo, seq
:: pular / * perder * /
fi
od
}

```

Concluímos agora a formalização de todas as premissas sobre o meio ambiente em que o protocolo deve ser usado. Agora estamos prontos para projetar os três proto-principais camadas col: a apresentação, a camada de sessão e a camada de controle de fluxo.

7.6.1 CAMADA DE APRESENTAÇÃO

A camada de apresentação fornece a interface para o usuário. Seu trabalho é cuidar do detalhes da transferência do arquivo, prevendo, por exemplo, o reenvio da solicitação para transferência de arquivos quando ocorre um erro não fatal. Podemos antecipar cinco razões diferentes para que uma solicitação de transferência de arquivo de saída falhe.

1. O sistema local está ocupado servindo uma transferência de arquivo de entrada.
2. O servidor de arquivos local rejeita a solicitação, por exemplo, porque o arquivo a ser transferido não existe.
3. O servidor de arquivos remoto rejeita a solicitação, por exemplo, porque não pode permitir cante espaço suficiente.
4. Há uma colisão entre uma solicitação de transferência de arquivo de entrada e de saída.
5. A transferência do arquivo foi abortada pelo usuário.

Duas dessas cinco causas possíveis de uma falha são transitórias (1 e 4) e podem desaparecer pêra se a solicitação de transferência for repetida. A camada de apresentação pode ainda tentar proteger-se contra solicitações repetidas de aborto dos processos do usuário, filtrando qualquer duplicatas.

Para começar a projetar a camada de apresentação, podemos desenhar um estado provisório diagrama. A camada de apresentação, então, pode receber dois estados principais: um estado IDLE e um ocupado ou estado de TRANSFERÊNCIA. O processo passa de IDLE para TRANSFER no recepção da solicitação de transferência do usuário. Ele retorna ao estado IDLE após conclusão bem sucedida da transferência ou detecção de um erro fatal. Este esboço inicial é mostrado na Figura 7.7.

Página 157

```
OCIOSO
TRANSFERIR
transferir (fd)
aceitar
rejeitar
abortar
abotar
```

Figura 7.7 - Diagrama de transição de estado parcial: camada de apresentação

Usando a Figura 7.7 como orientação, podemos preencher os detalhes relevantes em um modelo de validação do seguinte modo.

```
#define FATAL
1
/* tipos de falha */
#define NON_FATAL
2
/* Repetivel
*/
#define COMPLETE
3
/* sucesso
*/
protoype presente (bit n)
{
    status de byte, uabort;
OCIOSO:
Faz
:: use_to_pres [n]? transferir ->
uabort = 0;
ir para TRANSFERÊNCIA
:: use_to_pres [n]? abortar ->
pular
/* ignore */
od;
TRANSFERIR:
pres_to_ses [n]! transferência;
Faz
:: use_to_pres [n]? abortar ->
E se
:: (! uabort) ->
uabort = 1;
pres_to_ses [n]! abortar
:: (uabort) ->
pular
fi
:: ses_to_pres [n]? aceitar ->
goto DONE;
:: ses_to_pres [n]? rejeitar (status) ->
E se
:: (status == FATAL || uabort) ->
ir para FALHA
:: (status == NON_FATAL && ! uabort) ->
ir para TRANSFERÊNCIA
fi
od;
FEITO:
```

```
pres_to_use [n]! aceitar;
goto IDLE;
```

Página 158

```
FALHOU:
pres_to_use [n]! rejeitar;
ir para o IDLE
}
```

A solicitação de transferência de arquivos é repetida até ser bem-sucedida ou até acionar um erro fatal. A principal suposição que a camada de apresentação faz sobre o proto da camada de sessão col é que ele acabará respondendo a uma solicitação de transferência com um aceite ou um rejeitar mensagem. Também assume que a camada de sessão pode aceitar uma mensagem de aborto a qualquer momento.

Requisitos de correção : como um requisito de correção para a apresentação camada iremos identificar seus estados finais válidos. Há apenas um estado final válido, o IDLE Estado. Adicionando o prefixo " end ", isto é, substituindo o nome IDLE por endIDLE , nós pode especificar o requisito de que a camada de apresentação deve estar no estado fornecido quando uma transferência de protocolo termina. Desde que nossas suposições sobre a camada de sessão e a camada do usuário são verdadeiras, não é muito difícil mostrar que esse requisito foi satisfeito. A camada de apresentação pode ser bloqueada apenas em um pequeno número de estados não executáveis mentos. As suposições sobre as camadas de usuário e sessão garantem que nenhum dos essas declarações podem permanecer inexequíveis para sempre. Mas um argumento informal, como aquele acima, não é uma prova. No Capítulo 14, usamos um provador automatizado para mostrar que as suposições que fizemos e as conclusões que tiramos delas não são apenas convincente, mas também correto.

7.6.2 CAMADA DE SESSÃO

Anteriormente, decidimos que o protocolo teria quatro fases:

Inicialização do servidor de arquivos

Estabelecimento de conexão

Transferência de dados

Conclusão de chamada

As mensagens no vocabulário de protocolo que examinamos anteriormente, é claro, não não relacionado; eles implicam uma certa ordem. Uma solicitação de transferência deve preceder uma conexão

confirmação de conexão, assim como o estabelecimento da conexão deve preceder a transferência de dados. Nós

pode tornar algumas dessas relações implícitas explícitas com o esboço mostrado na Figura 7.8.

As transições são rotuladas com números que se referem a alguns, embora não todos, dos mensagens que são trocadas. As mensagens recebidas são prefixadas com um ponto de interrogação, mensagens enviadas com um ponto de exclamação. As setas tracejadas indicam o sequência de eventos esperada para uma transferência de arquivo de saída bem-sucedida; as linhas pontilhadas

indica uma transferência de arquivo de entrada normal. Escrevendo o diagrama de transição de estado nos forçou a fazer mais algumas escolhas de design. Na fase de estabelecimento da chamada, por exemplo, uma solicitação de conexão pode vir do sistema remoto diretamente após o o processamento de uma solicitação de conexão local foi iniciado. Na Figura 7.8, escolhemos rejeitar ambos os pedidos quando ocorrer tal colisão de chamadas. A fase de conclusão da chamada é inserido quando um arquivo foi transferido com sucesso, e a mensagem eof foi enviei. Alternativamente, a transferência pode ser interrompida quando o usuário envia o cancelamento

Página 159

*Iniciar
Local
Conectar
Controlo remoto
Conectar
Ligar
Colisão
Controlo remoto
Fechando
Local*

```

Fechando
Dados
Transferir
!1
?2
-----
?2
?3
?4
!3
-----
!4
!4
!6
?6
-----
?5
!5
-----
```

Figura 7.8 - Diagrama de transição de estado parcial: camada de sessão

1: transferir, 2: conectar, 3: aceitar, 4: rejeitar, 5: fechado, 6: eof
 mensagem. Nos parágrafos a seguir, consideramos cada fase do protocolo separadamente e preencha os detalhes.

FASE DE ESTABELECIMENTO DE CHAMADA

A camada de sessão fica entre a apresentação e a camada de controle de fluxo. Depois de a mensagem de transferência local da camada de apresentação chega, a camada de sessão deve executar uma série de tarefas cruciais. Deve tentar abrir o arquivo para leitura no local sistema, ele deve criar um arquivo para gravação no sistema remoto, e entre ele deve estabelecer uma conexão com o sistema remoto. Qualquer uma dessas três tarefas pode falhar. O servidor de arquivos local cuida de todo o acesso aos arquivos.

Isso leva às seguintes descrições para a fase de estabelecimento da chamada. Primeiro lá são apenas duas mensagens que podem acionar uma transferência de arquivo: a mensagem de transferência local e a mensagem de conexão remota. Todo o resto deve ser ignorado pela sessão protocolo. Em vez de especificar explicitamente todas as mensagens não válidas, podemos fazer o camada de sessão receber uma mensagem de tipo arbitrário e verifica apenas se há uma correspondência com o mensagem esperada, que no estado IDLE é uma mensagem de transferência vinda de a camada de protocolo superior ou uma mensagem de conexão da camada inferior.

```

sessão proctype (bit n)
{
mordeu
alternancia;
byte
tipo, status;
OCIOSO:
```

Página 160

```

Faz
:: pres_to_ses [n]? digite ->
E se
:: (tipo == transferência) ->
ir para DATA_OUT
:: (tipo! = transferência)
/* ignore */
fi
:: flow_to_ses [n]? tipo ->
E se
:: (digite == conectar) ->
ir para DATA_IN
:: (digite! = conectar)
/* ignore */
fi
od;
DATA_OUT:
...
DATA_IN:
...
}
```

Parece melhor separar as fases de transferência de dados para arquivos de entrada e saída. O caso mais fácil é a preparação para um arquivo de entrada. Existe apenas uma interação com o servidor de arquivos local.

```

DATA_IN:
/* preparar servidor de arquivos local */
```

```

ses_to_fsrv [n]! criar;
Faz
:: fsrv_to_ses [n]? rejeitar ->
ses_to_flow [n]! rejeitar;
ir para o IDLE
:: fsrv_to_ses [n]? aceitar ->
ses_to_flow [n]! aceitar;
quebrar
od;
... transferência de dados de entrada ...
... fechar conexão etc. ...

```

Uma transferência de saída é feita em três etapas, cada uma das quais pode falhar:

Handshake em uma solicitação aberta local com o servidor de arquivos

Inicialização da camada de controle de fluxo

Handshake com o sistema remoto em uma solicitação de conexão

A primeira etapa pode ser especificada da seguinte maneira.

```

DATA_OUT:
/* 1. preparar arquivo local */
ses_to_fsrv [n]! open;
E se
:: fsrv_to_ses [n]? rejeitar ->
ses_to_pres [n]! rejeitar (FATAL);
ir para o IDLE

```

Página 161

```

:: fsrv_to_ses [n]? aceitar ->
pular
/* Continuar */
fi;

```

A segunda etapa é mais difícil. Devemos ter certeza de que não podemos aceitar accidentalmente um mensagem sync_ack antiga de uma tentativa de inicialização anterior. Usamos um bit número de sequência para resolver esse problema (consulte o Exercício 7-12).

```

/* 2. inicializar o controle de fluxo */
ses_to_flow [n]! sincronizar, alternar;
Faz
:: flow_to_ses [n]? sync_ack, digite ->
E se
:: (digite != alternar) /* ignorar */
:: (tipo == alternar) -> pausa
fi
:: tempo limite -> /* falhou */
ses_to_fsrv [n]! close;
ses_to_pres [n]! rejeitar (FATAL);
ir para o IDLE
od;
toggle = 1 - alternar;

```

Na terceira e última etapa, devemos considerar a possibilidade de uma colisão de chamadas.

```

/* 3. preparar arquivo remoto */
ses_to_flow [n]! conectar;
E se
:: flow_to_ses [n]? aceitar ->
pular
/* sucesso */
:: flow_to_ses [n]? rejeitar ->
ses_to_fsrv [n]! close;
ses_to_pres [n]! rejeitar (FATAL);
ir para o IDLE
:: flow_to_ses [n]? conectar ->
ses_to_fsrv [n]! close;
ses_to_pres [n]! rejeitar (NON_FATAL);
ir para o IDLE
:: timeout -> /* foi desconectado? */
ses_to_fsrv [n]! close;
ses_to_pres [n]! rejeitar (FATAL);
ir para o IDLE
fi;
... transferência de dados de saída ...
... fechar conexão etc. ...

```

Quando uma colisão de chamada é detectada, ambas as partes fecham seus arquivos e retornam ao inicial estado, deixando para as camadas de apresentação fazer outra tentativa de transferir seus arquivos. A prudência determina que incluamos uma cláusula de tempo limite em cada estado onde aguarde eventos que só podem ser fornecidos pelo sistema remoto. No caso de todas as comunicações o cátion é perdido, por exemplo, na perda da portadora, ele nos dá um caminho de volta ao estado inicial.

FASES DE TRANSFERÊNCIA DE DADOS

O design das fases de transferência de dados é relativamente simples. Para saída transferências, obtemos dados do servidor de arquivos e os transferimos para o controle de fluxo camada. As únicas coisas que podem complicar o design são as mensagens que podem interagir interromper a transferência: um tempo limite ou uma mensagem de cancelamento do usuário.

```
Faz
/* dados de saída */
:: fsrv_to_ses [n]? dados ->
ses_to_flow [n]! dados
:: fsrv_to_ses [n]? eof ->
ses_to_flow [n]! eof;
status = COMPLETO;
break /* goto call termination */
:: pres_to_ses [n]? abortar ->
/* abortos do usuário */
ses_to_fsrv [n]! close;
ses_to_flow [n]! close;
status = FATAL;
break /* goto call termination */
od;
```

Para transferências de arquivos de entrada, obtemos mensagens de dados da camada de controle de fluxo e

transfira-os para o servidor de arquivos. Apenas o usuário remoto pode abortar uma transferência de entrada.

Quando o usuário remoto aborta, o sistema local recebe uma mensagem de fechamento do sistema remoto (veja acima). Isso leva ao seguinte modelo PROMELA .

```
Faz
/* dados recebidos */
:: flow_to_ses [n]? dados ->
ses_to_fsrv [n]! dados
:: flow_to_ses [n]? eof ->
ses_to_fsrv [n]! eof;
quebrar
:: pres_to_ses [n]? transfer -> /* desculpe, ocupado */
ses_to_pres [n]! rejeitar (NON_FATAL)
:: flow_to_ses [n]? fechar ->
/* abortos de usuário remoto */
ses_to_fsrv [n]! close;
quebrar
:: timeout -> /* foi desconectado? */
ses_to_fsrv [n]! close;
ir para o IDLE
od;
```

Todas as operações necessárias de enchimento de caracteres e codificação de bytes (Capítulo 2, Seção 2.5) são novamente irrelevantes para o modelo de validação. Podemos presumir com segurança que eles acontecem

caneta em outro lugar, por exemplo, no modem que nos conecta à linha física.

FASE DE RESCISÃO DE CHAMADA

Concluímos o design de alto nível da fase de transferência de dados adicionando o processamento de mensagens de terminação de chamadas. A fase de terminação de chamadas só pode ser acessada a partir do

fase de transferência de dados, conforme mostrado acima. Primeiro, vamos considerar o término normal de

uma sessão de transferência de arquivos de saída.

```
/* fechar conexão, transferência de saída */
Faz
:: pres_to_ses [n]? abortar /* tarde demais, ignorado */
:: flow_to_ses [n]? fechar ->
E se
:: (status == COMPLETO) ->
ses_to_pres [n]! aceitar
:: (status != COMPLETO) ->
ses_to_pres [n]! rejeitar (status)
fi;
quebrar
:: tempo limite -> /* desconectado? */
ses_to_pres [n]! rejeitar (FATAL);
quebrar
```

```

od;
ir para o IDLE
O código para responder ao término de uma transferência de arquivo de entrada é mais simples:
/* fechar conexão, transferência de entrada */
ses_to_flow [n]! close; /* confirme */
ir para o IDLE
Requisitos de correção : o principal requisito de correção para a sessão
protocolo da camada de controle é que satisfaça as suposições feitas pela apresentação
protocolo de camada: sempre responde a uma mensagem de transferência , dentro de uma quantidade
finita de
vez, com uma mensagem de aceitação ou rejeição . Da mesma forma, uma conexão remota
a mensagem deve ser seguida por uma mensagem de aceitação ou rejeição para o controle remoto
camada de apresentação, dentro de um período de tempo finito. Podemos formalizar ambos
requisitos em uma reivindicação temporal, definindo o comportamento que deve estar ausente. UMA
a primeira tentativa é mostrada abaixo.
Nunca {
Faz
::! pres_to_ses [n]? [transferir]
&&! flow_to_ses [n]? [conectar]
:: pres_to_ses [n]? [transferir] ->
ir para aceitar 0
:: flow_to_ses [n]? [conectar] ->
ir para aceitar 1
od;
aceitar 0:
Faz
::! ses_to_pres [n]? [aceitar]
&&! ses_to_pres [n]? [rejeitar]
od;
aceitar1:
Faz
::! ses_to_pres [1-n]? [aceitar]
&&! ses_to_pres [1-n]? [rejeitar]
od
}

```

onde n é zero ou um. Uma validação automatizada do protocolo da camada de sessão

Página 164

com base nesta afirmação é discutida no Capítulo 14. Também podemos identificar o estado IDLE em a camada de sessão como um estado final válido, novamente simplesmente substituindo o nome por endIDLE .

Para completar o projeto, agora fornecemos a camada de enlace, que vem em dois partes: uma camada implementando uma disciplina de controle de fluxo e outra fornecendo controle de erro trol.

7.6.3 CAMADA DE CONTROLE DE FLUXO

Podemos modelar um protocolo de janela deslizante completo, conforme discutido no Capítulo 4, diretamente

codificação das Figuras 4.11 e 4.12. Para tornar as coisas interessantes, vamos nos restringir para uma codificação com um único processo em vez de cinco. Neste caso, não podemos fugir com uma abstração de parâmetros como o tamanho da janela e o intervalo de sequência números usados, sem perder informações essenciais sobre o funcionamento deste proto camada col. Todos os dados no PROMELA são inicializados em zero por padrão. Discutimos o modelo passo a passo.

```

#define true
1
/* Por conveniência */
#define false 0
#define M
4
/* números de sequência de intervalo */
#define W
2
/* tamanho da janela: M / 2
*/

```

Como vimos no Capítulo 4, e provaremos no Capítulo 11, o número máximo de mensagens permanentes em um protocolo deste tipo é igual a metade do intervalo da sequência números.

```
prototype fc (bit n)
```

```

{
bool
ocupado [M];
/* mensagens pendentes
*/
byte
q;
/* seq # msg mais antiga não confirmada */
byte
m;
/* seq # última mensagem recebida */
byte
s;
/* seq # próxima mensagem a enviar */
byte
janela;
/* nº de mensagens pendentes */
byte
tipo;
/* tipo de mensagem
*/
mordeu
recebeu [M];
/* manutenção do receptor */
mordeu
x;
/* variável de rascunho
*/
byte
p;
/* seq # da última mensagem confirmada */
byte
I_buf [M], O_buf [M];
/* buffers de mensagem */

```

O modelo fc contém código para ações independentes do remetente e do receptor na íntegra comunicações duplex. Algumas das tarefas domésticas, então, têm a ver com manter faixa de mensagens enviadas e algumas com mensagens recebidas no canal de retorno.

As mensagens enviadas são armazenadas em um buffer de mensagem `O_buf`, indexado por sua sequência

número. O armazenamento em buffer das mensagens de saída permite que o protocolo retransmita mensagens antigas

sábios quando não são reconhecidos. Um array booleano `ocupado` é usado para lembrar quais slots na matriz de dados de saída estão livres e quais são ocupados.

O corpo principal da camada de controlo de fluxo é um único `do` loop que verifica a existência de saída mensagens da camada de sessão, adiciona números de sequência, encaminha as mensagens para a camada de controle de erros e mantém o controle de seu reconhecimento. Em cláusulas separadas

Página 165

ele verifica se há mensagens recebidas da camada de controle de erro, tira a sequência números e encaminha os restantes para a camada de sessão. O envio de mensagens pode ser modelado como segue. Uma mensagem só é enviada se estiver disponível, ou seja, se a mensagem canal `ses_to_flow` não está vazio, e se a camada de protocolo inferior está livre para aceitar ele, ou seja, se o canal de mensagens `flow_to_dll` não estiver cheio.

Faz

```

:: (janela <W && len (ses_to_flow [n]) > 0
&& len (flow_to_dll [n]) <QSZ) ->
ses_to_flow [n]? type;
janela = janela + 1;
ocupado [s] = verdadeiro;
O_buf [s] = tipo;
flow_to_dll [n]! tipo, s;

```

Há uma pequena dança extra a ser feita se a mensagem enviada for uma mensagem do tipo sincronização,

que é usado pela camada de sessão para redefinir o protocolo da camada de controle de fluxo. Naquilo caso, todos os sinalizadores de ocupado são apagados e o número de sequência retorna a zero.

E se

```

:: (digite! = sincronizar) ->
s = (s + 1)% M
:: (tipo == sync) ->
janela = 0;
s = M;
Faz

```

```

:: (s > 0) ->
s = s-1;
ocupado [s] = falso
:: (s == 0) ->
quebrar
od
fi

```

Quando uma mensagem de confirmação chega, seu número de sequência `m` aponta para a mensagem que é sendo reconhecido. O status dessa mensagem, mantida na matriz `ocupada [m]`, é redefinido para zero, o que significa que o slot ficou livre. No código do receptor discutido abaixo, está incluído da seguinte forma:

```

:: dll_to_flow [n]? digite, m ->
E se
:: (tipo == ack) ->
ocupado [m] = falso
...

```

Se a mensagem que foi confirmada for a mensagem pendente mais antiga, a janela pode deslizar um ou mais entalhes para cima e abrir espaço para mais mensagens a serem transmitidas. Isso é modelado com duas cláusulas condicionais independentes no loop de execução externo. O avanço da janela é protegido com uma cláusula de tempo limite para proteção contra mensagens perdidas.

Página 166

```

:: (janela > 0 && ocupado [q] == false) ->
janela = janela - 1;
q = (q + 1) % M
:: (tempo limite && janela > 0 && ocupado [q] == verdadeiro) ->
flow_to_dll [n]! O_buf [q], q

```

Resta apenas a modelagem das cláusulas para a parte receptora do controle de fluxo camada. As mensagens são recebidas em uma cláusula genérica

```

:: dll_to_flow [n]? digite, m ->

```

seguido por uma mudança no tipo de mensagem. Os dados de entrada são armazenados em buffer para proteger contra mensagens que são recebidas fora de ordem, por exemplo, como resultado de perda de mensagem e retransmissão, e para permitir que a camada de controle de fluxo encaminhe essas mensagens para o camada de sessão na ordem certa. A matriz booleana `recebida` é usada para acompanhar quais mensagens chegaram e quais estão pendentes.

Vejamos primeiro o processamento de mensagens `sync` e `sync_ack`.

```

E se
...
:: (tipo == sync) ->
m = 0;
Faz
:: (m < M) ->
recebido [m] = 0;
m = m + 1
:: (m == M) ->
quebrar
od;
flow_to_dll [n]! sync_ack, m
:: (tipo == sync_ack) ->
flow_to_ses [n]! sync_ack, m

```

A mensagem de sincronização tem como objetivo inicializar a camada de controle de fluxo. Neste caso, a mensagem

sage vem de um par remoto e deve ser reconhecido com um `sync_ack` quando a reinicialização for concluída. Se uma mensagem `sync_ack` chegar do peer remoto, é passado para o protocolo da camada de sessão. A mensagem `sync_ack` ecoa o número da sessão da mensagem de sincronização (consulte a página 150, parte inferior).

Todas as outras mensagens são consideradas mensagens de dados:

```

:: (digite != ack && type! = sync && type! = sync_ack) ->
E se
:: (recebido [m] == verdadeiro) ->
x = ((0 < pm && pm <= W)
|| (0 < p-m + M && p-m + M <= W));
E se
/* ack foi perdido? */
:: (x) -> flow_to_dll [n]! ack, m
:: (! x) /* else ignorar */

```

fi

Página 167

```
:: (recebido [m] == falso) ->
I_buf [m] = tipo;
recebido [m] = verdadeiro;
recebido [(m-W + M)% M] = falso
fi
```

Quando uma mensagem de dados chega, devemos primeiro verificar se ela foi ou não recebida antes (cf. página 79). Se foi recebido antes, `recebemos [m] == verdadeiro`, e devemos verificar se já foi reconhecido. As mensagens são apenas ack-agora eliminados após terem sido passados para a camada de sessão e terem liberado o buffer espaço que eles detinham. Uma mensagem que não foi recebida antes é armazenada, e o sinalizador apropriado é definido na matriz `recebida`.

A mesma cláusula é usada para redefinir o sinalizador `recebido` como falso para a mensagem que é uma tamanho total da janela longe da última mensagem recebida: apenas neste ponto no proto col podemos ter certeza de que esta mensagem não pode ser transmitida novamente. O reconhecimento para essa mensagem deve ter sido recebido ou não poderíamos ter recebido a mensagem atual.

Se a mensagem atual foi recebida antes, uma verificação no número de sequência nos diz se foi previamente reconhecido ou não. Se fosse, o fato de ter sido retransmitido indica que a confirmação foi perdida e precisa ser repetida. Mais um a cláusula permanece: aquela que codifica o processo de aceitação da Figura 4.12.

```
:: (recebido [p] == verdadeiro && len (flow_to_ses [n]) < QSZ
&& len (flow_to_dll [n]) < QSZ) ->
flow_to_ses [n]! I_buf [p];
flow_to_dll [n]! ack, p;
p = (p + 1)% M
od
}
```

A camada de controle de fluxo agora está completa. Demora algum tempo para nos convencermos de que Realmente funciona. Embora o protocolo tenha apenas cerca de cem linhas de código, o comportamento que ele especifica pode ser complexo, especialmente na presença de erros de transmissão.

Requisitos de correção: o principal requisito de correção para este protocolo camada é que ele transfere mensagens sem exclusões e reordenações, apesar do comportamento do canal de transmissão subjacente. Para expressar, ou verificar, esta exigência-poderíamos rotular cada mensagem transferida pelo remetente e verificar no receptor que nenhum rótulo é perdido e que a ordem relativa dos rótulos não é perturbada. Em um forma, tal rótulo atua como apenas outro número de sequência.

Dado um protocolo de controle de fluxo com um tamanho de janela w e uma gama de números de sequência

M , quantos rótulos distintos seriam minimamente necessários para verificar a exatidão requerimento? A resposta, devida a Pierre Wolper (ver Notas Bibliográficas), é surpreendente. O número é independente de w e M : três rótulos diferentes são suficientes para qualquer protocolo. Considere o seguinte experimento de verificação. Transmitimos sequências de mensagens sages através da camada de controle de fluxo, carregando apenas o rótulo e nenhum outro dado. os três diferentes tipos de etiquetas são chamados arbitrariamente vermelho, branco, e azul. Para convenience chamamos as mensagens correspondentes vermelho, branco, e azul mensagens como

Página 168

bem. Para construir uma série de sequências de teste, uma mensagem vermelha e uma azul são colocados aleatoriamente em uma seqüência infinita de mensagens brancas. Se o controle de fluxo protocolo pode perder uma mensagem, ele poderá perder a mensagem vermelha ou azul em às pelo menos uma das sequências de teste. Da mesma forma, se o protocolo puder reordenar duas mensagens

sábios, ele será capaz de mudar a ordem da mensagem vermelha e azul em pelo menos uma sequência. As sequências de teste podem ser geradas por um protocolo de camada de sessão falso módulo.

```
proctype test_sender (bit n)
{
Faz
:: ses_to_flow [n]! branco
```

```

:: ses_to_flow [n]! vermelho -> pausa
od;
Faz
:: ses_to_flow [n]! branco
:: ses_to_flow [n]! blue -> break
od;
Faz
:: ses_to_flow [n]! branco
:: pausa
od
}

```

O modelo de teste correspondente, que recebe as mensagens de teste na extremidade remota do protocolo, pode ser definido como segue.

```

proctype test_receiver (bit n)
{
Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? vermelho -> quebra
od;
Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? blue -> break
od;
Faz
:: flow_to_ses [n]? branco
od
}

```

O requisito de correção agora pode ser formalizado com uma reivindicação temporal ou mesmo mais simplesmente adicionando algumas afirmações ao processo do receptor.

```

proctype test_receiver (bit n)
{
Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? vermelho -> quebra
:: flow_to_ses [n]? blue -> assert (0)
/* erro */
od;

```

Página 169

```

Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? vermelho -> assert (0)
/* erro */
:: flow_to_ses [n]? blue -> break
od;
Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? vermelho -> assert (0)
/* erro */
:: flow_to_ses [n]? blue -> assert (0)
/* erro */
od
}

```

Isso completa o design das três camadas principais do protocolo. Nós projetamos um pro
modelo de tocol que tem detalhes suficientes para nos permitir expressar e verificar um conjunto de
requisitos de correção sobre o protocolo. Não é uma implementação. O problema,
no entanto, não precisa surgir antes de o próprio projeto ter sido validado (Capítulo
14). Só então, a tarefa de derivar uma implementação eficiente do design
tornam-se relevantes. Apenas damos uma olhada rápida em alguns dos problemas aqui.

UM LADO NA IMPLEMENTAÇÃO

Não é relevante para o design do modelo de validação, mas apenas por curiosidade, podemos
veja como poderíamos implementar a análise de mensagens no nível mais baixo do
protocolo, logo abaixo da camada de controle de erro, ou talvez até mesmo combinado com ele. Em C,
por exemplo, os bytes obtidos da linha podem ser armazenados, sem processamento, em um
buffer bruto que é grande o suficiente para conter uma mensagem de dados completa. Nós assumimos
que

os drivers de linha cuidam do recheio e recheio de caracteres, então temos uma mensagem pura
sage delimitado por bytes de controle STX e ETX .

Se usarmos o protocolo com os campos no cabeçalho da mensagem arredondados para o
múltiplo de oito (Tabela 7.1), o tamanho máximo do buffer necessário é de 428 bytes (422 dados
bytes mais sobrecarga de 6 bytes). Os bytes são lidos no buffer em um loop apertado que

pode ser codificado da seguinte forma. Usamos a união de um buffer bruto, uma mensagem de dados e um

mensagem sem dados. O código a seguir assume que o byte de ordem inferior de um número, como a soma de verificação, é enviado antes do byte de ordem superior.

```
typedef struct DATA_MSG {
    tipo de char unsigned;
    unsigned char seqno;
    cauda de carvão sem sinal [424];
    /* inclui soma de verificação */
} DATA_MSG;
typedef struct NON_DATA_MSG {
    tipo de char unsigned;
    unsigned char seqno;
    soma de verificação de char unsigned [2];
} NON_DATA_MSG;
```

Página 170

```
União {
    unsigned char raw [426];
    /* dados + cabeçalho + trailer */
    DATA_MSG
    dados;
    NON_DATA_MSG
    non_data;
} no;
```

Assumindo que a proteção contra a ocorrência accidental de mensagens STX e ETX os delimitadores sábios são fornecidos pelo remetente inserindo (preenchendo) um DLE (link de dados escape) antes de todos os dados que correspondem a um dos três caracteres especiais STX , ETX , ou DLE , podemos escrever a rotina de varredura e desempacotamento de bytes da seguinte maneira.

```
recv ()
{
    unsigned char c;
    char não assinado * start = in.raw;
    unsigned char * stop = start + 428;
    char sem sinal * ptr = start;
    Varredura:
    para (++);
    /* para sempre */
    {
        if ((c = line_in ()) == STX)
        /* próximo byte */
        ptr = inicio;
        /* reset ptr */
        else if (c == ETX)
        goto check;
        /* tem msg */
        else if (ptr < stop)
        {
            if (c == DLE)
            /* escapar
            */
            * ptr ++ = line_in ();
            outro
            * ptr ++ = c;
            /* loja
            */
        }
    }
    Verifica:
    ...
}
```

Quando o marcador ETX for visto, devemos verificar se há uma mensagem válida calculando a soma de verificação, por exemplo, chamando a rotina CRC discutida no Capítulo 3. Se na verdade, o byte de ordem inferior da soma de verificação é enviado primeiro, um recálculo da verificação

a soma de todos os dados brutos recebidos, *incluindo* o campo de soma de verificação, deve sair zero na ausência de erros de transmissão:

```
Verifica:
if (cksum (in.raw, ptr-start) == 0)
/* boa mensagem, aceite */
outro
/* mensagem distorcida, ignore */
goto scan;
```

/ * retomar a varredura de mensagens * /

Quando a soma de verificação é diferente de zero, o conteúdo do buffer deve ser descartado; caso contrário isto pode ser copiado para um local seguro e passado para a camada de protocolo superior.

Página 171

7.7 RESUMO

Neste capítulo, discutimos um projeto de protocolo, levando a uma hierarquia de validação de modelos de dados escritos em PROMELA. A função de cada camada nesta hierarquia é amplamente independente das funções desempenhadas pelas outras camadas. Cada camada adiciona alguma funcionalidade e ajuda a transformar o canal físico subjacente em um vírus tual com um comportamento progressivamente mais idealizado.

Com a derivação dos modelos de validação, o processo de design não está completo. O comportamento do modelo terá que ser formalmente validado, talvez revisado e, finalmente, implementado. Encontrar recepções não especificadas, segmentos de código não executáveis ou deadlocks é quase impossível de fazer apenas pela inspeção. O normal, esperado sequências de eventos podem ser verificadas facilmente. Os erros, no entanto, geralmente se escondem no combinações inesperadas de eventos: sequências de execução que têm baixa probabilidade de ocorrer. No Capítulo 14, consideraremos a validação dos modelos que temos derivado.

Se a implementação não deriva automaticamente do modelo validado, devemos adicionar outra etapa de validação para garantir que a implementação e o modelo são equivalentes: teste de conformidade (consulte o Capítulo 8, Seção 8.6 e o Capítulo 9).

EXERCÍCIOS

7-1. Compare a hierarquia de protocolo derivada neste capítulo com o modelo de referência OSI discutido no Capítulo 2, Seção 2.6. Quais camadas estão faltando?

7-2. Descreva precisamente em que enquadramento de ordem (usando símbolos STX e ETX), preenchimento de bytes, e cálculos de checksum devem ser executados no receptor e no remetente.

7-3. Qual camada teria que ser modificada para incluir um método de controle de taxa neste protocolo?

Qual camada teria que ser modificada para incluir um método de controle de fluxo dinâmico?

Descreva uma versão simples de cada método.

7-4. Considere a possibilidade de recalcular o tamanho de dados ideal D em tempo de execução e usando o informações obtidas para fazer o protocolo se adaptar a características de canal que mudam dinamicamente terísticas.

7-5. Complete o diagrama de transição de estado da Figura 7.8, adicionando transições e estados para todas as mensagens no vocabulário do protocolo.

7-6. Redesenhe o protocolo para uma taxa de transmissão de 2400 bps.

7-7. Redesenhe o protocolo para um canal com taxa de erro zero e propagação de sinal zero demora.

7-8. (Doug McIlroy) Estenda o protocolo para permitir que a transferência de arquivos seja retomada de onde saiu off após a perda da transportadora. Considere a opção de usar um parâmetro de status em aceitar e rejeitar mensagens enviadas pela camada de apresentação para indicar quanto do arquivo foi bem-sucedido transferido com sucesso. Dica: não pode haver confirmação para os últimos dados recebidos e armazenados no sistema remoto no momento da perda da portadora.

7-9. Existe a possibilidade de que as duas partes no protocolo de transferência de arquivos possam ser capturadas em um ciclo de execução, ou *livelock*, em que ambos continuam a enviar mensagens, mas nenhuma das partes consegue chegar à fase de transferência de dados? Se verdadeiro, sugira uma maneira de alterar o protocolo para evitar este problema.

Página 172

7-10. Redesenhe o processo do servidor de arquivos para permitir a comunicação assíncrona com o servidor de ação.

7-11. Tente mesclar as duas camadas de enlace de dados em uma camada, realizando o controle de fluxo e controle de erros.

7-12. Mostre que o número da sessão de alternância de um bit protege contra a interpretação errônea de mensagens sync_ack. Dica: compare com o protocolo de bit alternado.

7-13. Suponha que as suposições do canal sejam alteradas para incluir a possibilidade de dados reordenar no canal de transmissão. O método de configuração de conexão de um bit de O exercício 7-12 ainda funciona? Dica: uma solução padrão para esta versão generalizada do problema é conhecido como *handshake de três vias*, consulte, por exemplo, Stallings [1985, p. 490].

7-14. Considere o que precisaria ser alterado na implementação (consulte "Um aparte em Implementação") para transmissões em que o byte de ordem superior de uma quantidade de 16 bits é enviado antes do byte de ordem inferior.

7-15. Expressse o requisito de correção para a camada de controle de fluxo em uma reivindicação temporal. Dica: primeiro expresse uma exigência sobre o comportamento correto; em seguida, inverta-o para especificar todos os inválidos

comportamentos. Use um do-loop rotulado com um rótulo de estado de aceitação como a especificação de o estado de erro que é alcançado (e nunca encerrado) sempre que o comportamento correto é violado.

7-16. Verifique as dez regras de design do Capítulo 2 e verifique como elas foram aplicadas a este Projeto. Critique o design onde as regras foram violadas.

NOTAS BIBLIOGRÁFICAS

A estratificação de software é um conceito que se deve principalmente a EW Dijkstra (ver Bibliotecas gráficas, Capítulo 2). Mais sobre a derivação de parâmetros de protocolo, como o comprimento de dados ideal, pode ser encontrado em Field [1976], Tanenbaum [1981, 1988], ou Arthurs, Chesson e Stuck [1983].

Em espírito, pelo menos, o design empresta muitas ideias do mundo dos protótipos leves cols (ver Capítulo 2). A importância de uma colocação criteriosa das informações de controle no cabeçalho ou no trailer de um protocolo, foi enfatizado pela primeira vez por Greg Chesson, um dos iniciadores do projeto de protocolo leve.

Em muitos padrões, todas as informações de controle são, por padrão, colocadas em um único controle bloco que é colocado no cabeçalho de cada mensagem. O termo "protocolo de trailer" era cunhado por Chesson em um esforço para mostrar que a disciplina oposta, de colocar todas as informações de trol no trailer e nenhuma, exceto talvez um endereço de roteamento, no cabeçalho, pode levar a codificações mais eficientes do remetente e do destinatário. Para instância, tanto a contagem de bytes quanto a soma de verificação de uma mensagem podem ser calculadas em

a mosca pelo remetente. O receptor pode calcular somas de verificação de forma semelhante em tempo real (iniciar-
em uma bandeira de controle STX), e depois de localizar a bandeira ETX correspondente, o receptor pode encontrar a contagem de bytes por perto. A soma de verificação pode ser verificada imediatamente.
O critério de correção que especificamos para a camada de controle de fluxo foi primeiro discussed in Wolper [1986]. Também é discutido em Aggarwal, Courcoubetis e Wolper [1990].

Página 173

MÁQUINAS DE ESTADO FINITO 8

162 Introdução 8.1
162 Descrição Informal 8.2
169 Descrição Formal 8.3
170 Execução de Máquinas 8.4
171 Minimização de Máquinas 8.5
174 O Problema de Teste de Conformidade 8.6
175 Máquinas Combinadoras 8.7
176 Máquinas de Estado Finito Estendido 8.8
178 Generalização de Máquinas 8.9
181 Modelos Restritos 8.10
184 Resumo 8.11
185 exercícios
185 Notas Bibliográficas

8.1 INTRODUÇÃO

Em um baixo nível de abstração, um protocolo é frequentemente mais facilmente entendido como um estado máquina. Os critérios de design também podem ser facilmente expressos em termos de desejáveis ou indesejáveis estados de protocolo compatíveis e transições de estado. De certa forma, o estado do protocolo simboliza o suposições que cada processo no sistema faz sobre os outros. Define o que ações que um processo pode realizar, quais eventos ele espera que aconteçam e como isso acontecerá responder a esses eventos.

O modelo formal de uma máquina de estados finitos em comunicação desempenha um papel importante na

três áreas diferentes de projeto de protocolo: validação formal, síntese de protocolo e teste de conformidade. Este capítulo apresenta os conceitos principais. Primeiro o finito básico modelo de máquina de estado é discutido. Existem várias maneiras, igualmente válidas, de estender este modelo básico em um modelo para comunicar máquinas de estado finito. Nós selecionamos um desses modelos e formalizá-lo em uma definição de um comunicante finito generalizado máquina de estado. O modelo pode ser aplicado prontamente para representar as especificações PROMELA e construir um validador automatizado.

Existem muitas variações do modelo básico de máquina de estado finito. Em vez de listar

todos eles, concluímos este capítulo com uma discussão de dois dos mais interessantes exemplos: a Rede de Petri e a Rede FIFO.

8.2 DESCRIÇÃO INFORMAL

Uma máquina de estado finito é geralmente especificada na forma de uma tabela de transição, bem como aquele mostrado na Tabela 8.1 abaixo.

162

Página 174

163

Tabela 8.1 - Mealy ¹

Doença
Efeito

Estado atual no próximo estado

q0

-

1

q2

q1

-

0

q0

q2

0

0

q3

q2

1

0

q1

q3

0

0

q0

q3

1

0

q1

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{

{
{
{
{
{
{
{

Para cada estado de controle da máquina, a tabela especifica um conjunto de regras de transição. Existe uma regra por linha na tabela e geralmente mais de uma regra por estado. A tabela de exemplo contém regras de transição para estados de controle chamados q_0 , q_1 , q_2 e q_3 . Cada regra de transição tem quatro partes, cada parte correspondendo a uma das quatro colunas na mesa. Os dois primeiros são condições que devem ser satisfeitas para a regra de transição para ser executável. Eles especificam

O estado de controle em que a máquina deve estar

Uma condição no "ambiente" da máquina, como o valor de um sinal de entrada

As duas últimas colunas da tabela definem o efeito da aplicação de uma transição regra. Eles especificam

Como o "ambiente" da máquina é alterado, por exemplo, como o valor de um mudanças de sinal de saída

O novo estado que a máquina atinge se a regra de transição for aplicada

No modelo de máquina de estado finito tradicional, o ambiente da máquina consiste de dois conjuntos de sinais finitos e separados: sinais de entrada e sinais de saída. Cada sinal tem um intervalo arbitrário, mas finito, de valores possíveis. A condição que deve ser satisfeita definido para que a regra de transição seja executável é então formulada como uma condição no valor de cada sinal de entrada, e o efeito da transição pode ser uma mudança dos valores de os sinais de saída. A máquina da Tabela 8.1 ilustra esse modelo. Tem uma entrada sinal, denominado In, e um sinal de saída, denominado Out.

Um travessão em uma das duas primeiras colunas é usado como uma abreviação para indicar um "não condição de cuidado" (que sempre avalia o valor booleano verdadeiro). Uma regra de transição, então, com um traço na primeira coluna aplica-se a todos os estados da máquina, e uma transição regra de ção com um traço na segunda coluna se aplica a todos os valores possíveis da entrada sinal. Traços nas duas últimas colunas podem ser usados para indicar que a execução de um regra de transição não muda o ambiente. Um traço na terceira coluna significa

1. Este exemplo apareceu pela primeira vez em dois artigos seminais sobre máquinas de estado finito, publicado por George H. Mealy [1955] e Edward F. Moore [1956].

Página 175

164

MÁQUINAS DE ESTADO FINITO

CAPÍTULO 8

que o sinal de saída não muda e, da mesma forma, um traço na quarta coluna significa que o estado de controle permanece inalterado.

Em cada estado particular da máquina, pode haver zero ou mais regras de transição que são executáveis. Se nenhuma regra de transição for executável, diz-se que a máquina está em um *fim estado*. Se precisamente uma regra de transição é executável, a máquina faz um determinístico mover para um novo estado de controle. Se mais de uma regra de transição for executável, um não-determinante

uma escolha minimalista é feita para selecionar uma regra de transição. Uma escolha *não determinística* neste

contexto significa que o critério de seleção é indefinido. Sem mais informações qualquer opção deve ser considerada igualmente provável. A partir daqui, vamos chamar as máquinas que podem fazer tais escolhas *máquinas não determinísticas*. 2 A Tabela 8.2 ilustra a exceto Duas regras de transição são definidas para o estado de controle q_1 . Se o sinal de entrada for um, apenas a primeira regra é executável. Se o sinal de entrada for zero, no entanto, ambas as regras serão executável e a máquina se moverá para o estado q_0 ou para o estado q_3 .

Tabela 8.2 - Não Determinismo

Estado atual no próximo estado

q1
-
0
q0
q1
0
0
q3

O comportamento da máquina na Tabela 8.1 é mais facilmente compreendido quando representado graficamente na forma de um diagrama de transição de estado, conforme mostrado na Figura 8.1.

q0
q3
q1
q2
- / 1
1/0
- / 0
0/0
1/0
0/0

Figura 8.1 - Diagrama de transição de estado

Os estados de controle são representados por círculos e as regras de transição são especificadas como bordas direcionadas. Os rótulos de borda são do tipo c / e , onde c especifica a transição condição (por exemplo, o conjunto necessário de valores de entrada) e e o efeito correspondente (por exemplo, um nova atribuição ao conjunto de valores de saída).

2. Os autômatos formais não determinísticos (NFA) da teoria dos autômatos são freqüentemente definidos de forma diferente. (Vejo por exemplo, Aho, Sethi e Ullman [1986, p. 114].) Ao contrário de nossas máquinas não determinísticas, um NFA pode ser em mais de um estado ao mesmo tempo.

A definição acima de uma máquina de estados finitos é intuitivamente a mais simples. Tem muitas variantes deste modelo básico que diferem na forma como o ambiente do máquinas é definido e, portanto, na definição das condições e dos efeitos das regras de transição. Para sistemas de estado verdadeiramente finitos, é claro, o ambiente deve ser finito (é também possível, por exemplo, definir o ambiente como outra máquina de estado finito). Se este requisito for eliminado, obtemos o modelo da *Máquina de Turing* bem conhecido. É usado extensivamente na ciência da computação teórica como o modelo de escolha, por exemplo, no estudo da complexidade computacional. A máquina de Turing pode ser vista como uma generalização do modelo de máquina de estados finitos, embora o trabalho de Turing seja anterior ao de Mealy e Moore por quase duas décadas.

O "ambiente" no modelo da máquina de Turing é uma fita de comprimento infinito. A fita consiste em uma sequência de quadrados, onde cada quadrado pode armazenar um de um conjunto finito de símbolos de fita. Todos os quadrados da fita estão inicialmente em branco. A máquina pode ler ou escrever.

um quadrado de fita de cada vez, e pode mover a fita para a esquerda ou direita, também por um quadrado de uma

Tempo. Inicialmente, a fita está vazia e a máquina aponta para um quadrado arbitrário. A condição de uma regra de transição agora consiste no estado de controle do estado finito máquina e o símbolo da fita que pode ser lido a partir do quadrado que a máquina atualmente aponta para. O efeito de uma regra de transição é a saída potencial de uma nova fita símbolo no quadrado atual, um possível movimento para a esquerda ou direita e um salto para um novo estado de controle.

A fita é geralmente suficiente para modelar uma memória de acesso aleatório, seja ela ineficiente

1. A Tabela 8.3 ilustra esse tipo de máquina de estados finitos.

Tabela 8.3 - Busy Beaver

卷之三

Esta máquina tem dois sinais de saída: um é usado para substituir o quadrado atual em a fita com um novo símbolo, e um é usado para mover a fita para a esquerda ou direita um quadrado. O estado q_3 é um estado *final*.

3. Esta mesa é a entrada clássica de Tibor Rado no movimentado jogo do castor. O objetivo do jogo é criar um Máquina de estados finitos N-state (aqui $N = 3$) que, quando iniciada em uma fita vazia (ou seja, com todos os quadrados zero) atinge um estado final conhecido em um número finito de etapas, deixando a mais longa sequência possível de etapas na fita.

É bastante difícil definir uma extensão desta variante do modelo com um prático método para modelar a interação controlada de várias máquinas de estado finito. A escolha óbvia seria deixar uma máquina ler uma fita que foi gravada por outra, mas isso não é muito realista. Além disso, o número infinito de estados potenciais para o ambiente significa que muitos problemas se tornam intratáveis computacionalmente. Para o estudo de problemas de projeto de protocolo, portanto, devemos explorar outras variantes do modelo de máquinas de estados finitos.

COMUNICAÇÃO DE MÁQUINAS DE ESTADO FINITO

Considerando o que acontece se permitirmos a sobreposição dos conjuntos de sinais de entrada e saída de um máquina de estado finito do tipo mostrado na Tabela 8.1. Com toda a justiça, não podemos dizer o que vai acontecer sem primeiro considerar em mais detalhes o que é um " sinal ".

Assumimos que os sinais têm uma faixa finita de valores possíveis e podem alterar o valor apenas em momentos precisamente definidos. A máquina executa um algoritmo de duas etapas. Na primeira etapa, os valores do sinal de entrada são inspecionados e uma transição executável arbitrária regra de ação é selecionada. Na segunda etapa, a máquina muda seu estado de controle em acordo com essa regra e atualiza seus sinais de saída. Essas duas etapas são repetidas para sempre. Se nenhuma regra de transição for executável, a máquina continuará circulando seu algoritmo de duas etapas sem mudança de estado, até uma mudança no sinal de entrada valores, efetuados por outra máquina de estados finitos, torna possível uma transição. Um sinal, então, tem um estado, muito parecido com uma máquina de estados finitos. Pode ser interpretado como uma variável

que só podem ser avaliados ou atribuídos em momentos precisamente definidos.

que se podem ser avaliados ou atribuídos em momentos precisamente definidos. O comportamento da máquina da Tabela 8.1 agora está totalmente definido, mesmo se assumirmos um feedback da saída para o sinal de entrada. Neste caso, a máquina fará um loop através da seguinte sequência de três estados para sempre: q_0, q_2, q_1 . Em cada etapa, o máquina inspeciona o valor de saída que foi definido na transição anterior. o comportamento da máquina é independente do valor inicial do sinal de entrada.

Podemos construir sistemas elaborados de máquinas interagindo desta forma, conectando o sinal de saída de uma máquina para os sinais de entrada de outra. As máquinas devem compartilhar um "relógio" comum para seu algoritmo de duas etapas, mas eles não são de outra forma cronizado. Se sincronização adicional for necessária, ela deve ser realizada com um sistema sutil tempo de handshaking nos sinais que conectam as máquinas. Este problema, como vimos no Capítulo 5, tem três características perceptíveis: é um problema difícil, foi resolvido, e, do ponto de vista do projetista do protocolo, é irrelevante. A maioria dos sistemas fornece um designer com primitivas de sincronização de alto nível para construir um protocolo. Um exame-

ple de tais primitivas de sincronização são as operações de envio e recebimento definidas em PROMELA.

ACOPLAMENTO ASSÍNCRONO

No projeto de protocolo, as máquinas de estado finito são mais úteis se puderem modelar diretamente o fenômenos em um sistema de computador distribuído. Existem dois diferentes e igualmente formas válidas de fazer isso, com base em uma comunicação assíncrona ou síncrona modelo. Com o modelo assíncrono, as máquinas são acopladas via FIFO limitado

Página 178

167

(primeiro a entrar, primeiro a sair) filas de mensagens. Os sinais de uma máquina são agora objetos abstratos

mensagens chamadas. Os sinais de entrada são recuperados das filas de entrada e a saída sinais são anexados às filas de saída. Todas as filas e os conjuntos de sinais ainda estão finito, portanto, não desistimos da finitude de nosso modelo.

A sincronização é obtida definindo os sinais de entrada e saída para serem condicionais sobre o estado das filas de mensagens. Se uma fila de entrada estiver vazia, nenhum sinal de entrada

está disponível nessa fila, e as regras de transição que exigem uma não são executáveis.

Se uma fila de saída estiver cheia, nenhum sinal de saída pode ser gerado para essa fila, e o as regras de transição que produzem um são igualmente não executáveis.

Deste ponto em diante, restrinjimos os modelos que estamos considerando para aqueles sem mais de um evento de sincronização por regra de transição; ou seja, uma única regra pode especificar um entrada ou saída, mas não ambas. A razão para essa restrição é dupla. Primeiro simplifica o modelo. Não temos que considerar a semântica de complicadas composições de eventos de sincronização que podem ser inconsistentes (por exemplo, duas saídas para o mesmo

fila de saída que pode acomodar apenas um dos dois). Em segundo lugar, ele modela o real comportamento de um processo em um sistema distribuído mais de perto. Observe que a execução de uma regra de transição é um evento atômico do sistema. Na maioria dos sistemas distribuídos, um único a operação de envio ou recebimento é garantidamente um evento atômico. Portanto, é apropriado não devemos assumir ainda outro nível de interligação em nosso modelo de sistema básico.

Tabela 8.4 - Remetente

Estado
No
Fora
Próximo estado

q0

-

mesg0

q1

q1

ack1

-

q0

q1

ack0

-

q2

q2

-

mesg1

q3

q3

ack0

-

q2

q3

ack1

-

q0

Como um exemplo de acoplamento assíncrono de máquinas de estado finito, Tabelas 8.4 e 8.5 fornecer modelos de tabela de transição para uma versão simples do protocolo de bit alternado (ver também Capítulo 4, Figura 4.8). A possibilidade de uma retransmissão após um tempo limite não é modelado na Tabela 8.4. Poderíamos fazer isso com transições espontâneas, adicionando dois regras:

Estado em
Próximo estado

q1
- mesg0
-
q3
- mesg1
-

A tabela pode modelar a *possibilidade* de retransmissões desta forma, embora não seus *probabilidade*. Felizmente, este é exatamente o poder de modelagem que precisamos em um sistema que deve analisar protocolos independentemente de quaisquer suposições sobre o tempo ou velocidade de

processos individuais (ver também Capítulo 11).

Tabela 8.5 - Receptor

卷之三

A última mensagem recebida pode ser aceita como correta nos estados q_1 e q_4 . Um estado transiente q_3 é necessário para lidar com o caso de tempo limite. O diagrama de posição das Tabelas 8.4 e 8.5 é dado na Figura 8.2. A opção de tempo limite no campo *Time Out* do remetente produziria um loop automático extra nos estados q_1 e q_3 .

```
q0  
q1  
q2  
q3  
q5  
q4  
! mesg0  
? ack1  
? ack0  
! mesg1  
? ack1  
! mesg0  
? ack0  
! mesg1
```

Remetente (Tabela 8.4)

q2
q0
q1
q3
q5
q4
! ack0
? mesg0
? mesg1
! ack1
? mesg0
! ack0
? mesg1
! ack1

Receptor (Tabela 8.5)

Figura 8.2 - Diagramas de transição de estado, tabelas 8.4 e 8.5

Ainda não temos valores de parâmetro nas mensagens. No modelo acima, o valor do bit alternado é, portanto, marcado no nome de cada mensagem.

ACOPLAMENTO SÍNCRONO

O segundo método para máquinas de acoplamento é baseado em um modelo síncrono de comunicação, como a discutida brevemente no Capítulo 5. As condições de transição são agora as "seleções" que a máquina pode fazer para comunicação. Novamente nós permitimos apenas um evento de sincronização por regra de transição. A máquina pode selecionar um entrada ou um sinal de saída para o qual uma regra de transição é especificada. Para fazer um movimento, um sinal deve ser selecionado precisamente por duas máquinas simultaneamente, em uma máquina como

as máquinas fazem a transição correspondente simultaneamente e mudam sua seleção de acordo com os novos estados que alcançam.

As Tabelas 8.6 e 8.7 fornecem um exemplo de máquinas de estado finito sincronizadas.

A máquina na Tabela 8.6 pode fazer apenas uma seleção de entrada P no estado q0 e uma fora coloque a seleção V no estado q1 .

Tabela 8.6 - Usuário

Estado no próximo estado

q0
P
-
q1
q1
-
V
q0

{
}
{
}
{
}
{
}
{
}
{
}
{
}
{
}

A segunda máquina é quase igual à primeira, mas tem as entradas e saídas trocados (Tabela 8.7).

Tabela 8.7 - Servidor

Estado no próximo estado

q0
-
P
q1
q1
V
-
q0

{
}
{
}
{
}
{
}
{
}
{
}
{
}
{
}

Se criarmos duas máquinas do tipo 8.6 e combiná-las com uma máquina do tipo

8.7, podemos ter certeza de que, para todas as execuções possíveis, as duas primeiras máquinas não podem ambos estejam no estado q_1 ao mesmo tempo. Observe que a comunicação síncrona foi definido como binário: exatamente duas máquinas devem participar, uma com uma determinada entrada seleção e a outra com a seleção de saída correspondente. Normalmente, um parâmetro o valor será passado do emissor para o receptor no handshake síncrono. O valor que transferência, no entanto, ainda não está no modelo.

Podemos novamente considerar a comunicação síncrona como um caso especial de assíncrono comunicação nous com uma capacidade de fila de slots zero (consulte também os Capítulos 5 e 11). No restante deste capítulo, nos concentramos, portanto, no caso mais geral de uma acoplamento assíncrono de máquinas de estados finitos.

8.3 DESCRIÇÃO FORMAL

Vamos agora ver se podemos organizar as definições informais discutidas até agora. Um com a máquina de estados finitos de comunicação pode ser definida como um demônio abstrato que aceita entradas símbolos, gera símbolos de saída e muda seu estado interno de acordo com algum plano predefinido. Por enquanto, esses símbolos ou "mensagens" são definidos como abstratos objetos sem conteúdo. Vamos considerar as extensões necessárias para incluir valor transferência na Seção 8.8. Os demônios da máquina de estado finito se comunicam via Filas FIFO que mapeiam a saída de uma máquina na entrada de outra. Deixe-nos primeiro defina formalmente o conceito de fila.

Uma fila de mensagens é tripla (S, N, C) , onde:
 S é um conjunto finito chamado vocabulário da fila,

Página 181

170
 MÁQUINAS DE ESTADO FINITO
 CAPÍTULO 8

N é um número inteiro que define o número de slots na fila, e C é o conteúdo da fila, um conjunto ordenado de elementos de S . Os elementos de S e C são chamados de mensagens. Eles têm nomes exclusivos, mas outros objetos abstratos indefinidos sábios. Se mais de uma fila for definida, exigimos que o os vocabulários da fila podem ser desconexos. Seja M o conjunto de todas as filas de mensagens, um sobreescrito $1 \delta \square m \delta | \square M |$ é usado para identificar uma única fila, e um índice $1 \delta \square n \delta \square N$ é usado para identificar um slot na fila. C_n e, em seguida, é o n mensagem -ésimo no m fila -ésimo. Um sistema vocabulário V pode ser definido como a conjunção de todos os vocabulários da fila, mais um nulo elemento que indicamos com o símbolo Σ . Dado o conjunto de filas M , numeradas de 1 a $|\square M |$, o vocabulário do sistema V é definido como

$$V = \\ m = 1$$

*

$$|\square M |$$

$$S_m \square * \square \Sigma$$

Agora, vamos definir uma máquina de estados finitos em comunicação.

Uma máquina de estados finitos em comunicação é uma tupla (Q, q_0, M, T) , onde Q é um conjunto finito e não vazio de estados, q_0 é um elemento de Q , o estado inicial, M é um conjunto de filas de mensagens, conforme definido acima, e T é uma relação de transição de estado. A relação T leva dois argumentos, $T(q, a)$, onde q é o estado atual e a é um ação. Até agora, permitimos apenas três tipos de ações: entradas, saídas e uma ação nula Σ . A executabilidade dos dois primeiros tipos de ações está condicionada ao estado do filas de mensagens. Se executados, ambos mudam o estado de precisamente uma mensagem fila. Além disso, é imaterial, pelo menos para nossos propósitos atuais, o que o pré A definição precisa de uma ação de entrada ou saída é. A relação de transição T define um conjunto de zero ou mais estados sucessores possíveis no conjunto Q para o estado atual q . Este conjunto conterá precisamente um estado, a menos que o não determinismo é modelado, como na Tabela 8.2. Quando $T(q, a)$ não é definido explicitamente, assumimos

$T(q, a) = .$

$T(q, \Sigma)$ especifica transições espontâneas. Uma condição suficiente para essas transições para ser executável é que a máquina esteja no estado q .

8.4 EXECUÇÃO DE MÁQUINAS

Considere um sistema de máquinas de estado finito P , com conjuntos de mensagens sobrepostos filas. A união dos conjuntos de todas as filas de mensagens é novamente chamado M . Este sistema de a comunicação de máquinas de estado finito é executada aplicando as seguintes regras, assumindo apenas acoplamento assíncrono. Os elementos da máquina de estados finitos *que* são referido com um sobreescrito i .

ALGORITMO 8.1 - EXECUÇÃO FSM

1. Defina todas as máquinas em seu estado inicial e inicialize todas as filas de mensagens para esvaziar:

$\square(i), 1 \square \delta \square i \square \delta \square P \square \square q_i = \square q_0$
Eu

Página 182

SEÇÃO 8.5
MINIMIZAÇÃO DE MÁQUINAS

%

$\square(i), 1 \square \delta \square i \square \delta \mid \square M \square \mid \square C_i \square =$

2. Selecione uma máquina arbitrária i e uma regra de transição arbitrária T_i com $T_i(q, i, a) = .$ e a é executável

e executá-lo.

3. Se nenhuma regra de transição executável permanecer, o algoritmo termina.

A ação a pode ser uma entrada, uma saída ou pode ser a ação nula Σ . Seja $1 \square d(a) \delta \mid \square M \mid$ seja a fila de destino de uma ação a , e seja $m(a)$ a mensagem enviada ou recebida, $m(a) \square S_{d(a)}$. Além disso, deixe N_i representar o número de slots na mensagem fila i . Em um sistema assíncrono, por exemplo, as três regras a seguir podem ser usado para determinar se a é executável.

$a = \Sigma$

(1)

ou

a é uma entrada e $m(a) = \square C_1$

$d(a)$

(2)

ou

a é uma saída e $|\square C_{d(a)}| < \square N_{d(a)}$

(3)

O Algoritmo 8.1 não termina necessariamente.

8.5 MINIMIZAÇÃO DE MÁQUINAS

Considere a máquina de estados finitos mostrada na Tabela 8.8, com o estado correspondente diagrama de transição na Figura 8.3.

Tabela 8.8 - Receptor-II

Doença
Efeito

Estado atual
No
Próximo estado

q0
mesg1

-

q1

q0

mesg0

-

q2

q1

-

ack1

q0

q2

-

Embora esta máquina tenha três estados a menos do que a máquina da Tabela 8.5, é certamente que não se comporta de maneira diferente. Duas máquinas são consideradas *equivalentes* se eles podem gerar a mesma sequência de símbolos de saída quando oferecidos o mesmo sequência de símbolos de entrada. A palavra-chave aqui é *can*. As máquinas que estudamos podem fazer escolhas não determinísticas entre as regras de transição se mais de uma for executável ao mesmo tempo. Este não determinismo significa que mesmo duas máquinas iguais *podem* se comportar de maneira diferente quando forem oferecidos os mesmos símbolos de entrada. A regra para equivalência é

que as máquinas devem ter opções equivalentes para estar em estados equivalentes.

Página 183

172
MÁQUINAS DE ESTADO FINITO
CAPÍTULO 8
q1
q0
q2
? mesg1
! ack1
? mesg0
! ack0

Figura 8.3 - Diagrama de transição de estado para a Tabela 8.8

Os estados dentro de uma única máquina são considerados equivalentes se a máquina puder ser iniciada em qualquer um desses estados e gerar o mesmo conjunto de possíveis sequências de resultados quando oferecido qualquer sequência de teste de entradas. A definição de um apropriado relação de equivalência para estados, entretanto, deve ser escolhida com algum cuidado. Considerar o seguinte processo PROMELA:

```

o seguente processo PR
proctipo A ()
{
E se
:: q? a -> q? b
:: q? a -> q? c
fi
}

```

Sob a noção padrão de equivalência de linguagem que muitas vezes é definida para determinar máquinas de estados finitos minimalistas, isso seria equivalente a

```
maquinas de c  
proctipo B ()  
{  
q? a;  
E se  
:: q? b
```

```

:: q? c
fi
}

```

uma vez que o conjunto de todas as sequências de entrada (o idioma) aceito por ambas as máquinas é o mesmo. Ele contém duas sequências, de duas mensagens cada:

```
{q? a; q? b, q? a; q? c}
```

O comportamento dos dois processos, no entanto, é muito diferente. A sequência de entrada $q? a; q? b$, por exemplo, é sempre aceito pelo processo B , mas pode levar a um não especificado recepção no processo A . Para máquinas de estado finito de comunicação não determinística, portanto, os processos A e B não são equivalentes. As definições fornecidas abaixo irão suprir porta essa noção.

Na discussão a seguir de equivalência, minimização de estado e composição de máquina ção, vamos nos concentrar exclusivamente no conjunto de estados de controle Q e no conjunto de transições

T das máquinas de estado finito. Especificamente, o "estado" interno da mensagem filas no conjunto M são consideradas parte do ambiente de uma máquina e não tributação ao estado da própria máquina. Que esta é uma suposição segura precisa de alguns motivação. Considere, como um caso extremo, uma máquina de estado finito em comunicação que acessa uma fila de mensagens privadas para armazenar informações de estado interno. Pode fazer isso por anexando mensagens com informações de estado na fila e recuperando essas informações mação mais tarde. A fila de mensagens é interna e aumenta artificialmente o número de estados da máquina.

Página 184

SEÇÃO 8.5
MINIMIZAÇÃO DE MÁQUINAS
173

Quando consideramos a fila de mensagens como parte do ambiente de uma máquina em as definições que se seguem, ignoramos o fato de que a informação que é recuperada de tal fila privada é sempre fixo (ou seja, só pode ter sido colocado no fila pela mesma máquina em um estado anterior). Se dissermos que dois estados deste máquina são equivalentes se respondem às mesmas mensagens de entrada da mesma maneira, na verdade, colocamos uma exigência *mais forte* sobre os estados do que o estritamente necessário. Nós exigem, por exemplo, que os dois estados respondam de forma semelhante a mensagens que nunca poderia estar em uma fila privada para o estado determinado. Para suprimir informações de estado que poderia estar implícito no conteúdo da fila de mensagens, portanto, não relaxa o requisitos de equivalência. Como veremos, isso leva a algoritmos mais simples.

Usando esta abordagem, o conjunto de estados de controle de uma máquina de estado finito em comunicação

pode ser minimizado, sem alterar o comportamento externo da máquina, por substituição ing cada conjunto de estados equivalentes com um único estado. Mais formalmente, podemos dizer que esta relação de equivalência define uma partição dos estados em um conjunto finito de disjuntos classes de equivalência. A menor máquina equivalente a dada terá como muitos estados como a máquina original tem classes de equivalência.

Podemos agora definir um procedimento para a minimização de um estado finito arbitrário máquina com $|Q|$ estados.

ALGORITMO 8.2 - MINIMIZAÇÃO FSM

1. Defina uma matriz E de $|Q| \times |Q|$ valores booleanos. Inicialmente, cada elemento $E[i, j]$ de a matriz é definida com o valor verdadeiro da seguinte condição, para todas as ações a :

$$T(i, a) \rightarrow T(j, a)$$

Dois estados não são equivalentes, a menos que as relações de transição de estado correspondentes sejam definido para as mesmas ações.

2. Se a máquina considerada contém apenas escolhas determinísticas, T define um único Estado sucessor para todos os *verdadeiros* entradas de série E . Altere o valor de todas essas entradas $E[i, j]$ para o valor de

$$\square(a), E[T(i, a), T(j, a)]$$

Isso significa que os estados não são equivalentes, a menos que seus sucessores também sejam equivalentes. Quando $T(i, a)$ e $T(j, a)$ podem ter mais de um elemento, a relação é mais complicado. O valor de $E[i, j]$ agora é definido como *falso* se um dos dois seguintes as condições são *falsas* para qualquer ação a .

$$\square(p), p \square \square T(i, a) \square \square \square(q), q \square \square T(j, a) \text{ e } E[p, q]$$

$$\square(q), q \square \square T(j, a) \square \square \square(p), p \square \square T(i, a) \text{ e } E[q, p]$$

Isso significa que os estados i e j não são equivalentes, a menos que para cada sucessor possível estado p do estado i há pelo menos um estado sucessor equivalente q do estado j , e vice

versa.

3. Repita a etapa 2 até que o número de entradas *falsas* na matriz E não possa mais ser aumentado. O procedimento sempre termina, pois o número de entradas do array é finito e cada entrada só pode ser alterada uma vez, de *verdadeiro* para *falso*, na etapa 2. Quando o

174

MÁQUINAS DE ESTADO FINITO

CAPÍTULO 8

procedimento termina, as entradas da matriz definir uma compartimentação do $|Q|$ estados em classes de equivalência. O estado i , $1 \leq i \leq |Q|$, é equivalente com todos os estados j , $1 \leq j \leq |Q|$,

com $E[i, j] = \text{verdadeiro}$.

Tabela 8.9 - Equivalência

q0 q1 q2 q3 q4 q5

q0 1

0

0

1

0

0

q1 0

1

0

0

0

1

q2 0

0

1

0

1

q3 1

0

0

1

0

0

q4 0

0

1

0

1

q5 0

1

0

0

0

1

{
}
}
}
}
}
}
}
}
}
}

Se aplicarmos este procedimento à máquina de estados finitos na Tabela 8.5, obtemos o estável matriz de valores para E mostrado na Tabela 8.9 após uma única aplicação dos dois primeiros

passos. Um na tabela representa o valor booleano *true*. Da tabela, vemos que pares de estados (q_0, q_3), (q_1, q_5) e (q_2, q_4) são equivalentes. Podemos, portanto, reduzir Tabela 8.5 para a máquina de estado finito de três estados que foi mostrada na Tabela 8.8. Isto é necessariamente a menor máquina que pode realizar o comportamento da Tabela 8.5. O procedimento acima pode ser otimizado observando, por exemplo, que a matriz E é simétrica: para todos os valores de i e j devemos ter $E(i, j) = E(j, i)$. Trivialmente, cada estado é equivalente consigo mesmo.

8.6 O PROBLEMA DE TESTE DE CONFORMIDADE

O procedimento para testar a equivalência de estados também pode ser aplicado para determinar a equivalência de duas máquinas. O problema é então determinar que cada estado em uma máquina tem um equivalente na outra máquina e vice-versa. Claro, as máquinas não precisam ser iguais para serem equivalentes.

Uma variante desse problema é de grande importância prática. Suponha que temos um formal especificação do protocolo, na forma de máquina de estado finito, e uma implementação desse especificação. As duas máquinas devem ser equivalentes, ou seja, a implementação, visto como uma caixa preta, deve responder aos sinais de entrada exatamente como a máquina de referência seria. Não podemos, no entanto, saber nada com certeza sobre a verdadeira estrutura interna da implementação. Podemos tentar estabelecer a equivalência por sistema sondando tematicamente a implementação com sequências de entrada de teste e comparando as respostas com as da máquina de referência. O problema agora é encontrar apenas o conjunto certo de sequências de teste para estabelecer a equivalência ou não equivalência das duas máquinas. Este problema é conhecido na teoria da máquina de estados finitos como a *falha problema de detecção* ou teste de conformidade. O Capítulo 10 revisa os métodos que têm sido desenvolvidos para resolver este problema.

Levando isso um passo adiante, podemos também querer determinar a estrutura interna de uma máquina de estado finito desconhecida, apenas testando-a com um conjunto conhecido de sinais de entrada.

e observando suas respostas. Este problema é conhecido como problema de *verificação de estado*. Sem qualquer conhecimento adicional sobre a máquina, esse problema infelizmente não foi resolvido.

Observe, por exemplo, que na Figura 8.1 o estado q_3 não pode ser distinguido do estado q_2 por qualquer sequência de teste que comece com um símbolo de entrada um. Da mesma forma, o estado q_1 não pode ser distinguido do estado q_3 por qualquer sequência começando com zero. Desde todo a sequência de teste deve começar com um ou zero, não pode haver um único teste sequência que pode nos dizer com segurança em qual estado esta máquina está.

8.7 MÁQUINAS DE COMBINAÇÃO

Ao recolher duas máquinas de estado finito separadas em uma única máquina, a complexidade de validações formais com base em modelos de máquina de estado finito podem ser reduzidas. O algoritmo abaixo é referido no Capítulo 11 na discussão de um protocolo incremental, método de validação, e no Capítulo 14 na discussão de métodos para stepwise abstração.

O problema é encontrar uma tupla (Q, q_0, M, T) para a máquina combinada, dados dois máquinas (Q_1, q_0) , (M_1, T_1) e (Q_2, q_0) , (H_2, t_2) .

ALGORITMO 8.3 - COMPOSIÇÃO FSM

- Defina o conjunto de produtos dos dois conjuntos de estados das duas máquinas de estado. Se o primeiro máquina tem estados $|Q_1|$ e a segunda máquina tem estados $|Q_2|$ o conjunto de produtos contém $|Q_1| \cdot |Q_2|$ estados. Inicialmente, nomeamos os estados da nova máquina por concatenar os nomes de estado das máquinas originais em uma ordem fixa. Isso define conjunto Q da máquina combinada. O estado inicial q_0 da nova máquina é a combinação q_0

¹ q_0

² dos estados iniciais das duas máquinas originais.

- O conjunto de filas de mensagens M da máquina combinada é a união dos conjuntos de filas das máquinas separadas, $M_1 * M_2$. Os dois conjuntos originais não precisam ser

disjuntar. O vocabulário V da nova máquina é o vocabulário combinado de M_1 e M_2 , e o conjunto de ações um é a união de todas as ações que as máquinas individuais podem executar.

3. Para cada estado $q_1 q_2$ em Q , defina a relação de transição T para cada ação a como o escolha não determinística das relações correspondentes de M_1 e M_2 separadamente, quando colocados nos estados individuais q_1 e q_2 . Isso pode ser escrito:

$$\square(q_1 q_2) \square \square(a), \square \square T(q_1 q_2, a) = \square T_1(q_1, a) * \square T_2(q_2, a)$$

A máquina combinada agora pode ser minimizada usando o Algoritmo 8.2. Algoritmo 8.3 pode ser facilmente adaptado para combinar mais de duas máquinas.

O maior valor da técnica de composição da última seção é que ela permite para simplificar comportamentos complexos. Em validações de protocolo, por exemplo, poderíamos certificar proveito de um método que nos permite juntar duas máquinas em uma.

Um método seria compor duas máquinas usando o Algoritmo 8.3, remover todos suas interações internas, ou seja, a interface original entre as duas máquinas, e minimizar a máquina resultante.

Há duas peças faltando em nossa estrutura de máquina de estado finito para nos permitir

Página 187

176

MÁQUINAS DE ESTADO FINITO
CAPÍTULO 8

aplique composições e reduções desta forma. Primeiro, o modelo de máquina de estado finito que desenvolvemos até agora, não podemos representar facilmente os modelos PROMELA. No próximo segundo

ção, mostramos como o modelo básico de máquina de estado finito pode ser estendido o suficiente para modelo PROMELA modela elegantemente. A segunda peça que está faltando em nosso quadro trabalho é um método para remover ações internas de uma máquina sem perturbar seu comportamento externo. Discutimos esses métodos na Seção 8.9.

8.8 MÁQUINAS DE ESTADO FINITO ESTENDIDO

Os modelos de máquina de estado finito que consideramos até agora ainda falham em dois aspectos importantes: a capacidade de modelar a manipulação de variáveis convenientemente e a capacidade de modelar a transferência de valores arbitrários. Essas máquinas foram definidas para trabalhar com objetos abstratos que podem ser anexados e recuperados de filas e eles só são sincronizados no acesso a essas filas.

Fazemos três alterações neste modelo básico de máquina de estado finito. Primeiro, apresentamos uma primitiva extra que é definida como uma fila: a variável. Variáveis têm nomes simbólicos e contêm objetos abstratos. Os objetos abstratos, neste caso, são valores inteiros. A principal diferença de uma fila real é que uma variável pode conter apenas um valor de cada vez, selecionado a partir de um intervalo finito de valores possíveis. Qualquer número de valores podem ser anexados a uma variável, mas apenas o último valor anexado pode ser recuperado.

A segunda mudança é que agora usaremos as filas especificamente para transferir inteiros valores, em vez de objetos abstratos indefinidos. Em terceiro e último lugar, apresentamos uma série de operadores aritméticos e lógicos para manipular o conteúdo das variáveis.

Tabela 8.10 - Variável de estado finito

Estado atual no próximo estado

q0
s0
-
-
q0
s1
-
q1
q0
s2
-
q2
q0
rv

- r0
r0 - 0 q0

q1
s0

-
q0
q1

s1

q1
s2

q2
q1

-
r1

r1
-1

q1

s0
-

q0
q2
s1

-
q1
a2

s2

q2
rv

-
r2
r2

-
2
a²

1

THE JOURNAL OF CLIMATE

A extensão com variáveis, desde que tenham um intervalo finito de valores possíveis, não aumenta o poder computacional das máquinas de estado finito com Filas FIFO. Uma variável com um intervalo finito pode ser simulada trivialmente por um estado finito máquina. Considere a máquina de seis estados mostrada na Tabela 8.10, que modela uma variável com a faixa de valores de zero a dois. A máquina aceita quatro entradas diferentes mensagens. Três são usados para definir a pseudo variável para uma das três possíveis valores. A quarta mensagem, rv , é usada para testar o valor atual do pseudo variável. A máquina irá responder à mensagem rv retornando um dos três possíveis valores ble como uma mensagem de saída.

Assim, às custas de um grande número de estados, podemos modelar qualquer variável finita sem estender o modelo básico, como uma máquina de estados finitos de propósito especial. A extensão com variáveis explícitas, portanto, não é mais do que uma conveniência de modelagem.

Lembre-se de que as regras de transição de uma máquina de estados finitos têm duas partes: uma condição e um efeito. As condições das regras de transição agora são generalizadas para incluir expressões booleanas sobre o valor das variáveis e os efeitos (ou seja, as ações) são generalizado para incluir atribuição a variáveis.

Uma máquina de estado finito estendida agora pode ser definida como uma tupla (Q, q_0, M, A, T), onde A é o conjunto de nomes de variáveis. Q, q_0 e M são conforme definidos anteriormente. O estado de transição relação de ção T permanece inalterada. Simplesmente definimos dois tipos extras de ações: condições booleanas sobre e para atribuições de elementos de conjunto Uma . Uma única tarefa pode mude o valor de apenas uma variável. As expressões são construídas a partir de variáveis e convalores constantes, com os operadores aritméticos e relacionais usuais.

No espírito da linguagem de validação PROMELA, podemos definir uma condição a ser executada cortável apenas se for avaliado como *verdadeiro* e permitir que uma atribuição seja sempre executável. Nota cuidadosamente que o modelo estendido de comunicar máquinas de estados finitos é um *finito* modelo de estado, e quase todos os resultados que se aplicam a máquinas de estado finito também se aplicam a este modelo.

E / S ESTENDIDA

As ações de entrada e saída também podem ser generalizadas. Vamos definir as ações de I / O como conjuntos de valores ordenados e finitos. Os valores podem ser expressões em variáveis de A , ou simplesmente constantes. Por definição, o primeiro valor de tal conjunto ordenado define o fila de destino para E / S, dentro do intervalo $1 \dots |\square M|$. Os valores restantes definem um estrutura de dados que é anexada ou recuperada da fila quando a ação de I / O é realizada. A semântica de executabilidade pode novamente ser definida como em PROMELA.

EXEMPLO

Considere o seguinte fragmento PROMELA, com base em um exemplo do Capítulo 5.

```
proctipo Euclides
{
pvar x, y;
Em? x, y;
```

Página 189

178
MÁQUINAS DE ESTADO FINITO
CAPÍTULO 8
Faz
:: (x > y) -> x = x - y
:: (x < y) -> y = y - x
:: (x == y) -> quebra
od;
Fora! X
}

O processo começa recebendo dois valores nas variáveis x e y , e completa retornando o maior divisor comum desses dois valores para sua fila de saída. o correspondência de máquina de estado finito estendida é mostrada na Tabela 8.11, onde combinamos todos condições, atribuições e operações de I / O em uma única coluna.

Tabela 8.11 - Máquina de estado finito estendida

Estado Atual	Ação	Próximo Estado
q_0		
Em? X, y		
q_1		
q_1		
$x > y$		
q_2		
q_1		
$x < y$		
q_3		
q_1		

Estado Atual	Ação	Próximo Estado
q_0		
Em? X, y		
q_1		
q_1		
$x > y$		
q_2		
q_1		
$x < y$		
q_3		
q_1		

```

x = y
q4
q2
x = xy
q1
q3
y = yx
q1
q4
Fora! X
q5
q5
-
-

```

O conjunto A possui dois elementos, x e y .

Agora temos um mapeamento simples de modelos PROMELA para estado finito estendido máquinas. Algoritmo 8.3, por exemplo, agora pode ser usado para expressar a combinação comportamento de dois processos PROMELA por um único processo. Observamos antes que este técnica pode ser especialmente útil em combinação com um método de *ocultação* que remove ações internas de uma máquina sem perturbar seu comportamento externo. Nós dê uma olhada mais de perto em tais métodos na próxima seção.

8.9 GENERALIZAÇÃO DE MÁQUINAS

Considere o seguinte modelo PROMELA .

```

1 proctype generalize_me (canais; byte p)
2 {chan interno [1] de {byte}; 
3
int r, q;
4
5
interno! cookie;
6
r = p / 2;
7
Faz
8
:: (r <= 0) -> pausa
9
:: (r> 0) ->
10
q = (r * r + p) / (2 * r);
11
E se
12
:: (q! = r) -> pular

```

```
SEÇÃO 8.9  
GENERALIZAÇÃO DE MÁQUINAS  
179  
13  
:: (q == r) -> pausa  
14  
fi;  
15  
r = q  
16  
od;  
17  
interno? cookie;  
18  
E se  
19  
:: (q <p / 3) -> ans! pequeno  
20  
:: (q> = p / 3) -> ans! ótimo  
21  
fi  
22}
```

Um processo deste tipo começará enviando um `cookie` de mensagem para uma mensagem local canal. Em seguida, ele executará alguns cálculos horríveis, usando apenas variáveis locais, ler de volta a mensagem do canal `interno`, e enviar um de dois possíveis mensagens através de um canal de mensagens externo `ans`.

Agora, para começar, nada detectável mudará no comportamento externo deste processo se removermos as linhas 2, 5 e 17. O canal de mensagem é estritamente local, e há nenhum comportamento possível para o qual qualquer uma das ações realizadas no canal pode ser inexecutável. As linhas 5 e 17 são, portanto, equivalentes a operações de `pular` e podem ser excluído do modelo. Reduções, ou *podas*, desse tipo produzem máquinas que têm um comportamento externo equivalente às máquinas não reduzidas. Isso não é verdade para o próximo tipo de redução que discutimos: generalização.

O horrível cálculo realizado pelo processo, entre as linhas 6 e 16, não envolvem quaisquer variáveis globais ou interações de mensagens. Uma vez que o valor inicial de variável `p` for escolhido, a mensagem resultante enviada para o canal `ans` é fixa. Se nós somos interessado apenas no comportamento externo de processos do tipo `generalize_me`, independentemente do valor preciso de `p`, o modelo pode ser reescrito como

```
proctipo generalizado (canais; byte p)  
{  
E se  
:: ans! pequeno  
:: ans! ótimo  
fi  
}
```

Esta especificação apenas diz que dentro de um tempo finito após um processo desse tipo é instanciado, ele envia uma mensagem do tipo `pequeno` ou uma mensagem do tipo `grande` e termina. Para justificar a redução, devemos, é claro, mostrar que o loop na origem

A especificação final sempre terminará. Se este não for o caso, ou não puder ser provado, a redução correta seria

```
180  
MÁQUINAS DE ESTADO FINITO  
CAPÍTULO 8  
proctipo generalizado (canais; byte p)  
{  
E se  
:: (0)  
:: ans! pequeno  
:: ans! ótimo  
fi  
}
```

onde a possibilidade de bloqueio é preservada explicitamente.

Chamamos uma redução deste tipo, onde o comportamento desinteressante, mas estritamente local, é removido, uma *generalização*. Um processo do tipo `generalizado` pode fazer tudo que um processo do tipo `generalize_me` pode fazer, mas pode fazer mais. O processo `generalizado`

pode, por exemplo, para qualquer dado parâmetro p , retornar qualquer uma das duas mensagens, enquanto os processos não generalizados escolherão apenas um. Os processos generalizados são *apenas* mais geral na forma como pode produzir saída, não na forma como pode aceitar entradas, ou em geral da mesma forma que outros processos podem restringir seu comportamento por meio de objetos globais.

A utilidade das generalizações na validação do protocolo pode ser explicada como segue. Considere dois módulos de protocolo A e B cujo comportamento combinado é muito complexo para ser analisado diretamente. Queremos validar um requisito de correção para os processos no módulo A. Podemos fazer isso simplificando o comportamento no módulo B, por exemplo, combinando, podando, generalizando e minimizando máquinas. Se o comportamento em módulo B é generalizado como discutido acima, o novo módulo B ainda será capaz de comportando-se precisamente como o módulo B não modificado, mas pode fazer mais. Se pudermos provar

a observância de um requisito de correção para o módulo A na presença do módulo B generalizado, o que pode ser mais fácil, o resultado será necessariamente válido também para o original, mais complexo, o módulo B, porque o comportamento original é um subconjunto do novo comportamento.

Duas coisas devem ser observadas. Primeiro, se estivermos interessados em provar uma propriedade de módulo A que simplificar seu ambiente, que neste caso é o módulo B. Nós não alterar o próprio módulo A. Em segundo lugar, é importante que o comportamento modificado do módulo

B não permite, em virtude das modificações, que o módulo A passe no teste. Isto é garantido pelo fato de que o módulo generalizado B continua a aderir a todos os restrições que podem ser impostas por A, por meio de objetos globais, como canais de mensagem e variáveis. A validação, então, nos dá o melhor dos dois mundos. Executa um teste mais forte, uma vez que valida propriedades para condições mais gerais do que as definidas em o protocolo original, mas é mais fácil de executar, uma vez que um processo generalizado pode ser menor que seu original.

Um método geral para a redução de uma definição arbitrária de protótipo PROMELA pode ser descrito a seguir.

Identifique as estruturas de seleção e repetição em que todos os guardas são condições em variáveis locais apenas, e em que a união de todos os guardas é verdadeira.

Substitua cada um dos guardas identificados na primeira etapa pela declaração PROMELA `pular`.

Página 192

SEÇÃO 8.10
MODELOS RESTRITOS
181

Substitua todas as atribuições às variáveis locais que não fazem mais parte de nenhuma condição, com `pular`.

Remova todas as declarações redundantes e minimize ou simplifique o novo protótipo corpo, por exemplo, combinando cláusulas iguais em estruturas de seleção e repetição e removendo instruções de salto redundantes.

Se aplicarmos este método ao tipo de processo `generalize_me`, após podar o interações no canal `interno`, podemos reduzi-lo a

```
proto generalizado_2 (canais; byte p)
{
Faz
:: pausa
:: pular
od;
E se
:: ans! pequeno
:: ans! ótimo
fi
}
```

que é semelhante e tem o mesmo comportamento externo que a (segunda) versão que derivado anteriormente com um pouco mais de ondulação à mão. Observe que o loop na versão acima não necessariamente termina.

Uma aplicação mais substancial desta técnica de generalização e o resultado a redução da complexidade é discutida no Capítulo 14.

8.10 MODELOS RESTRITOS

Para concluir este capítulo, examinamos duas outras variantes interessantes do modelo finito básico modelo de máquina de estado que foi aplicado ao estudo de problemas de protocolo. Muitas variações do modelo de máquina de estado finito básico têm sido usadas para a análise de sistemas de protocolo, restrições e extensões. As versões restritas têm o vantagem, pelo menos em princípio, de um ganho de poder analítico. As versões estendidas têm a vantagem de um ganho no poder de modelagem. As variantes mais populares do máquinas de estado finito são redes de tokens formalizadas, geralmente derivadas do modelo de rede de Petri.

Abaixo, revisamos brevemente o modelo da Rede de Petri e discutimos uma das variações, o FIFO Net.

PETRI NETS

Uma Rede de Petri é uma coleção de *lugares*, *transições* e *bordas* direcionadas. Cada borda contra conecta um lugar a uma transição ou vice-versa. Os lugares são representados graficamente por círculos, transições por barras e bordas por arcos direcionados. Informalmente, um lugar corresponde a uma condição e uma transição corresponde a um evento. Os *locais de entrada* da transição T são os lugares que estão diretamente conectados a T por uma ou mais arestas. A entrada lugares correspondem a condições que devem ser cumpridas antes do evento correspondente a T pode ocorrer. Os locais de saída de uma transição correspondem de forma semelhante ao efeito de o evento nas condições representadas pelos lugares.

Página 193

182

MÁQUINAS DE ESTADO FINITO
CAPÍTULO 8

Cada lugar que corresponde a uma condição cumprida é marcado com um ou mais *tokens* (às vezes chamada de *pedra*). A ocorrência de um evento é representada na Rede de Petri como o *disparo* de uma transição. Uma transição é habilitada se houver pelo menos um token em cada um de seus locais de entrada. O efeito de um disparo é que um token é adicionado à marca de todos os locais de saída da transição de disparo, e um token é removido do marcações de todos os seus locais de entrada.

Duas transições são consideradas *conflitantes* se compartilharem pelo menos um local de entrada. Se o compartilhado

local contém precisamente um token, ambas as transições podem ser ativadas para disparar, mas o o disparo de uma transição desabilita a outra. Por definição, o disparo de qualquer combinação de duas transições é sempre mutuamente exclusivo: apenas uma transição pode ser acionada por vez.

Ao atribuir zero ou mais tokens a cada lugar na rede, obtemos uma marca inicial ing. Cada disparo cria uma nova marcação. Uma série de disparos é chamada de *execução sequência*. Se para uma determinada marcação inicial, todas as sequências de execução possíveis podem ser feitas

infinitamente longo, a marcação inicial, e trivialmente todas as marcações subsequentes, são consideradas estar *vivo*. Se em uma determinada marcação nenhuma transição estiver habilitada para disparar, diz-se que a rede está *pendurada*.

Uma criação inicial é considerada *segura* se nenhuma sequência de execução subsequente pode produzir um

marcando onde qualquer lugar tem mais de um token.

Uma série de propriedades foi comprovada sobre as redes de Petri, mas principalmente sobre ainda outras versões simplificadas. Dois exemplos dessas variantes são:

Redes de Petri nas quais exatamente uma borda é direcionada de e para cada local. Tal as redes são chamadas de *gráficos marcados*. Em um gráfico marcado, não pode haver conflito de transposições.

Redes de Petri em que todas as transições têm no máximo um local de entrada e uma saída Lugar, colocar. Essas redes são chamadas de *diagramas de transição*.

A Figura 8.4 dá um exemplo de uma rede de Petri modelando um problema de deadlock. Inicialmente, as duas transições superiores t_1 e t_2 são habilitadas. Depois que t_1 dispara, a transição t_3 torna-se ativado. Se disparar, está tudo bem. Se nesta marcação, no entanto, a transição t_2 dispara, a rede vai *travar*.

Um token em uma rede de Petri simboliza mais do que o cumprimento de uma *condição*, como descrito acima. Ele também simboliza um *ponto de fluxo de controle* no programa e simboliza o *privilegio* de ir além de um certo ponto. Um token modela um compartilhado recurso que pode ser reivindicado por mais de uma transição. Todas essas abstrações sim-

bolize a aplicação de ordens parciais no conjunto de possíveis sequências de execução no sistema modelado. Especialmente relevante para o problema de modelagem de protocolo é que misturar essas abstrações pode tornar mais difícil do que o necessário distinguir computação a partir da comunicação em um modelo de rede de Petri.

A complexidade de uma representação da Rede de Petri aumenta rapidamente com o tamanho do problema

sendo modelado. É virtualmente impossível traçar uma rede clara para sistemas de protocolo que incluem mais de dois ou três processos. Isso torna os modelos da rede de Petri relativamente fraco no poder de modelagem em comparação com a comunicação de máquinas de estado finito, sem oferecendo um aumento no poder analítico. Não há, por exemplo, nenhum programa padrão cedências, além da análise de acessibilidade, para analisar uma rede de Petri para a presença de

Página 194

SEÇÃO 8.10
MODELOS RESTRITOS

183

□
□
t1
t3
t2
t4

Figura 8.4 - Rede de Petri com estado travado

estados pendurados. Nem existem procedimentos padrão para simplificar uma grande rede de Petri em um ou mais equivalentes menores.

Uma nota final sobre o poder de modelagem das Redes de Petri básicas. Nós observamos acima disso os locais em uma rede de Petri podem ser usados para modelar as condições. É bastante fácil modelar testes lógicos *e* e *ou* em locais usando arestas múltiplas, mas não há uma maneira geral de modelar uma *não* operação lógica (negação). Com uma *não* operação lógica , seria possível definir que uma transição pode ser disparada se um local não tiver tokens.

Claro, existem muitas boas aplicações da teoria da Rede de Petri. Eles têm sido aplicada com sucesso ao estudo de uma série de problemas teóricos em computação paralela tação. Pelas razões pragmáticas acima, no entanto, concluímos que as Redes de Petri não nos dá uma vantagem no estudo de projeto de protocolo e problemas de validação.

FIFO NETS

Redes FIFO são uma generalização interessante das Redes de Petri e um aditivo relativamente recente à gama de ferramentas propostas para estudar sistemas distribuídos (ver Notas).

Uma rede FIFO, como uma rede de Petri, tem lugares, bordas e transições, mas os lugares contêm símbolos em vez de tokens. Os símbolos são anexados e reclamados do lugares por disparos de transição. Eles são armazenados pelos locais nas filas FIFO. Ambos as bordas de entrada e saída das transições são rotuladas com nomes de símbolos. Um transi só pode disparar se a fila de cada um de seus locais de entrada pode entregar o símbolo que corresponde à borda conectando a transição para aquele lugar. Ao disparar as etiquetas nas bordas de saída, especifique quais símbolos devem ser acrescentados às filas do locais de saída correspondentes.

A generalização das Redes FIFO é forte o suficiente para torná-las equivalentes em computação poder racional para as máquinas de estados finitos que definimos anteriormente. Infelizmente, não há

Página 195

184
MÁQUINAS DE ESTADO FINITO
CAPÍTULO 8

melhores procedimentos para analisar redes FIFO para erros de protocolo interessantes, como impasse. Em alguns casos, existem procedimentos para versões restritas de Redes FIFO, mas novamente, as restrições geralmente reduzem o poder de modelagem muito severamente para torná-los de interesse como uma ferramenta geral para projetar ou analisar sistemas de protocolo.

8.11 RESUMO

O modelo formal de uma máquina de estado finito foi desenvolvido no início dos anos 1950 para o estudo de problemas em complexidade computacional e, de forma independente, para o estudo de problemas no projeto de circuitos combinatórios e sequenciais. Existem quase como

muitas variantes do modelo básico de uma máquina de estado finito, pois há aplicativos.

Para o estudo de problemas de projeto de protocolo, precisamos de um formalismo no qual possamos modelar as primitivas de interações de processo da forma mais sucinta possível. Com isso em mente, desenvolvemos um modelo de máquina de estado finito estendido que pode modelar diretamente passagem de mensagens e manipulação de variáveis. Sua semântica está intimamente ligada a a semântica da PROMELA .

Existem três critérios principais para avaliar a adequação das ferramentas de modelagem formal:

Poder de modelagem

Poder analítico

Clareza descritiva

O objetivo principal da modelagem é obter ganho de poder analítico. Deveria ser mais fácil analisar o modelo do que analisar o sistema original que está sendo modelado.

Escolhemos a máquina de estados finitos como nosso modelo básico. Existe um pequeno conjunto de propriedades úteis que podem ser facilmente estabelecidas com uma análise estática de estado finito modelos de máquinas. Mais importante, no entanto, a manipulação do estado finito as máquinas podem ser automatizadas e ferramentas de análise dinâmica mais sofisticadas podem ser desenvolvidas. Estudamos essas ferramentas na Parte IV deste livro. A clareza descritiva do máquinas de estado finito é discutível. Pode-se argumentar que eles comercializam descritivos clareza para poder analítico. Usando PROMELA como uma forma intermediária de um máquina de estado finito estendida, no entanto, podemos contornar esse problema.

O modelo da máquina de Turing falha em todos os três critérios listados acima quando aplicado para o estudo de problemas de protocolo. Em particular, a definição de "ambiente" é difícil de explorar na modelagem de comunicações. Talvez ainda mais importante entretanto, muitos problemas de interesse, como ausência de impasse, são intratáveis para Modelos de máquinas de Turing. O modelo é poderoso demais para o nosso propósito.

Redes de Petri têm sido usadas para o estudo de sistemas distribuídos desde o seu início em início dos anos 1960. A rede de Petri e a rede FIFO têm um sim-conceito atraente plausibilidade que se baseia principalmente na representação gráfica do mecanismo de pro-interação cessa. Esta vantagem na clareza descritiva, no entanto, é perdida quando o tamanho do problema excede um limite modesto. Além de cerca de cinquenta estados por processo, o as redes tornam-se inescrutáveis. Outro problema mais sutil é distinguir a sincronização de controle de fluxo em um modelo de rede de Petri. Ambos são modelados com a mesma ferramenta: o token. Pode-se argumentar que a clareza descritiva é negociada aqui

Página 196

CAPÍTULO 8
NOTAS BIBLIOGRÁFICAS

185

para simplicidade conceitual. Para a modelagem de sistemas de protocolo, isso acaba sendo uma troca infeliz. Protocolos de tamanho realista normalmente têm muitas vezes o número de estados além dos quais uma rede de Petri se torna inutilizável. As restrições do modelo implica uma perda de poder de modelagem que não é compensada por um ganho comparável em poder analítico.

EXERCÍCIOS

8-1. Explique a diferença entre o travessão introduzido como uma notação de conveniência na Seção 8.2 e o Σ introduzido na Seção 8.3.

8-2. Aplique o Algoritmo 8.1 à Tabela 8.1.

8-3. Defina as regras para a executabilidade de *um* no Algoritmo 8.1, assumindo um síncrono em vez de acoplamento assíncrono de máquinas.

8-4. Altere o Algoritmo 8.3 para combinar qualquer número de máquinas.

8-5. Implemente os algoritmos 8.1 a 8.3 em sua linguagem de programação favorita. Invente um *syn-imposto* para especificar um sistema de máquinas de estados finitos. Especifique as Tabelas 8.4 e 8.5 neste formalismo e usar seus programas para minimizar as máquinas correspondentes, para combinar em uma única máquina e simular a execução da descrição resultante.

8-6. Modele o comportamento das Tabelas 8.6 e 8.7 no PROMELA .

8-7. Os operadores *run* e *chan* no PROMELA tornam os sistemas modelados ilimitados?

8-8. Encontre um algoritmo que detecta quais filas de mensagens a partir da definição de uma comunicação máquinas de estado finito são usadas apenas internamente, para armazenar informações de estado, e que remove-os da especificação aumentando o número de estados.

8-9. (S. Purushothaman) São dois estados de máquina equivalentes se um dos dois estados contém uma transição não executável que falta ao outro estado (cf. a receber de um sempre vazio fila de mensagens)?

8-10. Derive uma descrição formal de máquina de estado finito para os processos de exemplo A e B em

página 172 e mostrar que eles não são equivalentes.

NOTAS BIBLIOGRÁFICAS

A teoria das máquinas de estado finito tem uma longa história e pelo menos partes dela podem ser encontrado em muitos livros de ciência da computação, por exemplo, Aho, Hopcroft e Ullman [1974], Aho, Sethi e Ullman [1986]. A ideia original da máquina de estados finitos é atribuído a McCulloch e Pitts [1943]. Mais intimamente ligado à teoria que foi subsequentemente desenvolvidos são os nomes de DA Huffman, GH Mealy, EF Moore e AM Turing. O artigo original sobre máquinas de Turing é Turing [1936]. Por mais discussão recente, ver, por exemplo, Kain [1972]. Os primeiros trabalhos de Huffman no conceito de máquinas de estado finito e equivalência de estado foi publicado em Huffman [1954] e reimpresso em Moore [1964]. O primeiro artigo de Edward Moore sobre máquinas de estado finito é Moore [1956]. No modelo de Moore, a saída de uma máquina de estados finitos depende apenas de seu estado atual, não na transição que o produziu. Os primeiros artigos de Moore são coletado em Moore [1964]. Artigo original de George Mealy, na máquina de estados finitos modelo pode ser encontrado em Mealy [1955]. O modelo de Mealy é um pouco mais geral do que

Página 197

186

MÁQUINAS DE ESTADO FINITO

CAPÍTULO 8

Moore's. Em seu modelo, a saída de uma máquina de estados finitos depende da última transição ção que foi executada, não necessariamente no estado atual.

Para uma introdução mais geral à teoria básica e sua aplicação ao circuito design, consulte, por exemplo, Harrison [1965], Hartmanis e Stearns [1966], Kain [1972], Shannon e McCarthy [1956]. O "problema do castor ocupado" foi introduzido em Rado [1962] e posteriormente estudado em Lin e Rado [1965].

O modelo formal de uma máquina de estado finito foi aplicado ao estudo da comunicação protocolos de cátions desde as primeiras publicações, por exemplo, Bartlett, Scantlebury e Wil-Kinson [1969]. Há muito tempo é o método de escolha em quase toda modelagem formal e técnicas de validação, cf. Bochmann e Sunshine [1980]. O modelo foi o primeiro aplicado a um problema de validação de protocolo em Zafiropulo [1978]. Uma introdução muito legível dução, a teoria da comunicação de máquinas de estados finitos pode ser encontrada em Brand e Zafiropulo [1983].

Uma excelente visão geral de vários métodos para derivar relações de equivalência para processos atuais, e a complexidade dos algoritmos correspondentes, podem ser encontrados em Kanellakis e Smolka [1990]. A generalização das máquinas está intimamente relacionada com o conceito de uma "projeção de protocolo" que foi introduzido em Lam e Shankar [1984].

O modelo de Petri foi descrito pela primeira vez em Petri [1962]. Veja também Agerwala [1975] para uma discussão do poder de modelagem da Rede de Petri e para algumas extensões. Uma discussão de As redes FIFO podem ser encontradas em Finkel e Rosier [1987]. Existem, é claro, muitos outros modelos analíticos interessantes para sistemas concorrentes. Uma visão geral e avaliação pode ser encontrado em, por exemplo, Holzmann [1979].

Página 198

TESTE DE CONFORMIDADE 9

187 Introdução 9.1

188 Teste Funcional 9.2

189 Teste Estrutural 9.3

195 Derivando Seqüências UIO 9.4

196 Tours de Transição Modificados 9.5

197 Um Método Alternativo 9.6

199 Resumo 9.7

200 exercícios

200 notas bibliográficas

9.1 INTRODUÇÃO

Os objetivos do teste de conformidade do protocolo e da validação do protocolo são facilmente confundidos.

Um *teste de conformidade* é usado para verificar se o comportamento externo de um determinado implemento

a mentação de um protocolo é equivalente à sua especificação formal.

Uma *validação* é usada para verificar se a própria especificação formal é logicamente compatível consistente.

Se uma especificação formal tem um erro de projeto, uma implementação fiel dessa especificação não deve passar em um teste de conformidade se e somente se contiver o mesmo erro. Um teste de força deve falhar somente se a implementação e a especificação forem diferentes. Um teste de validação de consistência do protocolo, entretanto, deve sempre revelar o erro de projeto. No neste capítulo, estudamos métodos de teste de conformidade. Os capítulos 11 e 13 são dedicados para validação de consistência.

Referência

Especificação

Testador

teste

sequência

Implementação

Sob teste

Figura 9.1 - Teste de conformidade

Recebemos uma especificação de referência conhecida, por exemplo, em uma máquina de estado finito para

tapete, e uma implementação desconhecida. Para todos os efeitos práticos, a implementação é uma caixa preta com um conjunto finito de entradas e saídas. O único tipo de experimento que nós pode fazer com a caixa preta é fornecer-lhe sequências de sinais de entrada (mensagens) e observe os sinais de saída resultantes. A implementação em teste, normalmente referido como o *IUT*, passa no teste apenas se todas as saídas observadas corresponderem àquelas prescrita pela especificação formal. Uma série de sequências de entrada que é usada para exercitar a implementação do protocolo dessa maneira é chamado de *suite de testes de conformidade*. O teste é derivado da especificação de referência, de preferência por um procedimento mecânico

187

Página 199

188

TESTE DE CONFORMIDADE

CAPÍTULO 9

(Figura 9.1).

Existem dois problemas principais a serem resolvidos.

Encontrar um procedimento geralmente aplicável e eficiente para gerar uma conformidade conjunto de testes para uma determinada implementação de protocolo.

Encontrar um método para aplicar o conjunto de testes a uma implementação em execução.

O segundo problema parece mais simples do que é. O IUT pode ser uma única camada em um hierarquia de funções de protocolo com duas interfaces para camadas circundantes, como ilustrado tratado na Figura 2.12 na página 31. Para testar esta camada, podemos precisar de um superior e um testador inferior e algum método sistemático para coordenar as sequências que eles geram erate. Outro fator complicador existe quando o IUT e o testador estão fisicamente separados uns dos outros, conforme ilustrado na Figura 9.2.

canal real

canal virtual

Interface

Interface

Testador

Rede

Rede

N

N – I

N + I

Inscrição

Implementação

Sob teste

Figura 9.2 - O problema geral de teste de conformidade

O testador só pode acessar o IUT por meio de uma conexão de rede remota e pode não ser capaz de fornecer entradas e recuperar saídas do IUT em um maneira confiável. Neste capítulo, discutimos apenas o primeiro problema: o problema de derivando sequências de teste de conformidade de alta qualidade.

9.2 TESTE FUNCIONAL

O teste de conformidade de protocolo tornou-se um problema quando os administradores do primeiro

redes públicas de dados tiveram que determinar a adequação dos equipamentos comerciais que era para ser usado em suas redes. O problema era verificar a conformidade do equipamento ao padrão de rede sem ter acesso ao, muitas vezes proprietário, detalhes internos do equipamento. No início dos anos 1980, as primeiras tentativas de construir conjuntos de teste de protocolo ativo, portanto, tinham dois objetivos principais. Para estabelecer que uma determinada implementação realiza todas as funções do original especificação, em toda a gama de valores de parâmetro. Para estabelecer que uma determinada implementação pode rejeitar adequadamente entradas erradas em um forma consistente com a especificação original.

Um exemplo de teste funcional do primeiro tipo pode ser um *teste* básico de *interconexão*, destinada a estabelecer que o IUT é minimamente capaz de configurar e derrubar um padrão

Página 200

SEÇÃO 9.3
TESTE ESTRUTURAL
189

conexão. Um exemplo de teste do segundo tipo poderia ser um *teste de formato*, usado para verifique se o IUT rejeita adequadamente as violações do formato de pacote exigido e violações da consistência do conteúdo do pacote (por exemplo, checksum ou erros de contagem de bytes). O problema encontrado nesses testes é um conflito entre complexidade e padronização. É virtualmente impossível testar exaustivamente todos os comportamentos possíveis de um implementação desconhecida simplesmente investigando-a e observando suas respostas. Há sim sempre uma possibilidade de que alguma sequência não experimentada de sondas revelaria um novo comportamento que é inaceitável. O conjunto de testes específico selecionado para um teste de conformidade deste tipo, portanto, é sempre uma pequena seleção do conjunto infinito de todos os testes possíveis suites. Para evitar que um fabricante manipule um dispositivo para passar uma determinada conformidade teste em vez de torná-lo equivalente à especificação adequada, os conjuntos de testes para o teste de conformidade funcional não pode ser padronizado ou publicado.

Há, no entanto, uma injustiça básica em exigir que um fabricante passe em um teste sem tornar público o que é o teste, ou sem padronizar o teste de tal forma que todos os fabricantes concorrentes devem submeter seus equipamentos ao mesmo teste. Em última análise, a complexidade da realização dos testes, a dificuldade de padronização eles, e a incerteza de seu valor levou a uma nova abordagem para conformidade testando. Essa nova abordagem tem o seguinte propósito.

Para estabelecer que a *estrutura* de controle da implementação está em conformidade com o estrutura da especificação. Implementação e especificação têm o mesmo estrutura se eles modelam conjuntos equivalentes de estados e permitem a mesma transição de estado ções.

Embora um conjunto de testes deste tipo possa ser facilmente padronizado, não há métodos disponíveis capaz ainda que funcione para protocolos de complexidade arbitrária, levando em conta, para instância, variáveis internas, valores de parâmetro de mensagem e configurações de temporizador. Boa métodos são conhecidos apenas por uma classe restrita de protocolos que podem ser especificados por máquinas de estado finito não estendidas. O restante deste capítulo é dedicado a uma discussão desses métodos.

9.3 TESTE ESTRUTURAL

Nenhum dado ou valor de parâmetro é considerado neste tipo de teste. Em vez disso, a ênfase está na estrutura de controle do protocolo. Novamente, sondando um dispositivo desconhecido para comparecer sua estrutura interna com a de uma especificação de referência é geralmente impossível.

Temos que fazer algumas suposições simplificadoras.

1. O IUT modela uma máquina de estado finito determinística com um máximo conhecido número de estados e com um vocabulário de entrada e saída conhecido.
2. O IUT produz uma resposta a um sinal de entrada dentro de uma quantidade finita conhecida de tempo.
3. Os estados e as transições do IUT formam um gráfico fortemente conectado: cada estado no gráfico pode ser alcançado a partir de todos os outros estados da máquina por meio de um ou mais transições de estado.

Um *estado* do IUT, para os fins desta discussão, é definido como uma condição estável em que o IUT está aguardando um novo sinal de entrada. Uma *transição* é definida como o

190

TESTE DE CONFORMIDADE
CAPÍTULO 9

consumo de um sinal de entrada, a possível geração de um sinal de saída e o possível mudança para um novo estado. Para um resultado de teste reproduzível, o movimento deve ser um fator determinante mímistico, o que significa que o modelo de uma máquina de estados finitos que podemos usar é um subconjunto do modelo discutido no Capítulo 8. No entanto, uma vez que estamos discutindo implementações de creta em vez de projetos abstratos, o determinismo provavelmente não será uma restrição.

As três propriedades listadas acima são requisitos. Sem eles, um teste de conformidade do tipo a ser discutido não é possível. No entanto, também assumimos que o IUT corresponde a uma máquina de estado finito completamente especificada.

4. Em cada estado, o IUT pode aceitar e responder a todos os símbolos de entrada do vocabulário completo do sistema. Uma resposta nula, ou seja, uma transição de volta para o mesmo estado, é uma resposta válida.

Essa *suposição de completude* nos permite generalizar os algoritmos abaixo por removendo um caso especial, mas não é um requisito. Em muitos casos, uma conformidade o teste pode até ser reduzido se nem todas as combinações de entrada possíveis precisarem ser testadas. O IUT também pode ter propriedades que podem simplificar a tarefa de teste de conformidade em si. Ao contrário dos três primeiros requisitos, as três propriedades a seguir são veniente, mas não essencial.

5. *Propriedade de status*. Quando uma mensagem de "status" é recebida, o IUT responde com uma mensagem de saída que identifica exclusivamente seu estado atual. O IUT não mudar de estado.

6. *Redefina a propriedade*. Quando uma mensagem de "reset" é recebida, o IUT responde por fazendo uma transição para um estado inicial conhecido, independente de seu estado atual. O IUT não precisa produzir uma saída.

7. *Defina a propriedade*. Quando uma mensagem de "set" é recebida no estado inicial do sistema, o IUT responde fazendo uma transição para o estado especificado em um parâmetro de essa mensagem. O IUT não precisa produzir uma saída.

Dada uma máquina com todas as sete propriedades listadas acima, um teste de conformidade pode ser realizado

formado da seguinte forma.

ALGORITMO 9.1 - TESTE DE CONFORMIDADE

1. Para todas as combinações possíveis de um estado i e um sinal de entrada j , execute o seguinte três etapas.

2. Use a mensagem de *reinicialização* para trazer o IUT ao estado inicial e, em seguida, use o *conjunto* mensagem para transferir o IUT para o estado i .

3. Aplique o sinal de entrada j . Verifique se qualquer saída recebida, incluindo a saída nula, corresponde à saída exigida pela especificação.

4. Use a mensagem de *status* para interrogar o IUT sobre seu estado final. Verifique se este o estado final corresponde ao exigido pela especificação.

O teste verifica se o IUT é capaz de realizar corretamente todas as transições de estado em a especificação formal. O conjunto de sinais de entrada testados deve, é claro, incluir o definir, redefinir e mensagens de *status*. Se o IUT passar nesses testes, ele é capaz de reproduzir duzir o comportamento da especificação formal, mas permanece desconhecido se o IUT é

SEÇÃO 9.3
TESTE ESTRUTURAL
191

capaz de qualquer outro comportamento. Especificamente, se o IUT estiver com defeito, ele pode violar o primeiro requisito para o teste de conformidade que listamos acima. A aceitação de um sinal de entrada que está fora do vocabulário oficial de entrada pode então causar uma transição de o IUT com defeito em um conjunto de estados que produz comportamento errôneo.

Dentro dessas restrições, o resultado do Algoritmo 9.1 é o melhor que podemos esperar alcançar com um teste de conformidade. Mas também é o melhor algoritmo possível? O custo do teste pode ser expresso como o comprimento do conjunto de testes, ou seja, como o número total de mensagens enviadas para o IUT. Suponha que a especificação formal contém

$S = |\square Q|$ estados e tem um vocabulário de entrada de V mensagens distintas, que inclui o definir, redefinir e mensagens de status. O comprimento do conjunto de testes para o Algoritmo 9.1 então é

$$4 SV$$

Após cada teste, o IUT é forçado de volta ao estado inicial. Podemos evitar isso se nós pode encontrar uma sequência de transições de estado que passa por cada estado e cada transição posição pelo menos uma vez. Essa sequência de transições é chamada de *roteiro de transição*. No melhor esse tour de transição começa com uma única mensagem de redefinição e exercita cada transição exatamente uma vez, a cada vez seguida por uma mensagem de status para verificar o estado de destino. Uma mensagem definida não é mais necessária. O comprimento do conjunto de testes agora é mínimo

$$1 + 2 SV$$

O problema agora é encontrar o tour de transição mínimo ou um que seja o mais próximo possível para isso. É um problema padrão da teoria dos grafos. Uma *turnê de Euler* em um dirigido gráfico é uma sequência de transições que começa e termina no mesmo estado e contém cada transição exatamente uma vez. Uma condição suficiente para a existência de uma viagem Euler é que o gráfico (a máquina de estados finitos) seja fortemente conectado e *simétrico*, ou seja, cada vértice (estado) deve ser o destino e a origem do mesmo número de bordas (transições).

Dado um gráfico simétrico e fortemente conectado, um passeio de Euler pode ser encontrado com um procedimento padrão que é resumido no Algoritmo 9.2. Na primeira etapa, o algoritmo rithm deriva uma árvore de abrangência simples do gráfico. A árvore de abrangência de um gráfico contém todos os seus vértices, mas apenas um subconjunto das arestas. Na árvore de abrangência que é construída no Algoritmo 9.2 qualquer vértice pode ter várias arestas de entrada, mas está restrito a apenas uma borda de saída, conforme ilustrado na Figura 9.3. Na teoria dos grafos, uma árvore geradora com esta propriedade é chamada de *arborescência generalizada*.

Página 203

192

TESTE DE CONFORMIDADE

CAPÍTULO 9

uma

b

c

d

e

f

g

Figura 9.3 - A Spanning Arborescence

Cada aresta dirigida representada na arborescência de abrangência deve estar presente no gráfico original. Na Figura 9.3, por exemplo, deve haver arestas no gráfico original do vértice b para a , de g para b e assim por diante. Na segunda etapa do Algoritmo 9.2, o árvore é usada para decidir em qual ordem as arestas devem ser adicionadas ao passeio de transição. As arestas na arborescência de abrangência são adicionadas por último.

Uma aresta i no gráfico que começa no vértice s é escrita (s, i) . Seu destino está escrito $dest(s, i)$. O conjunto de vértices que são representados na arborescência de abrangência é chamado T . Após a primeira etapa do Algoritmo 9.2 ser concluída, T deve ser igual ao conjunto de vértices (estados) no grafo original Q .

ALGORITMO 9.2 - DERIVANDO UM TOUR DE TRANSIÇÃO

1. Spanning Arborescence - Escolha um vértice arbitrário do gráfico original e adicione -lo para definir T . Este vértice se tornará a raiz da arborescência abrangente. Em seguida, selecione uma aresta (s, i) com $s \in T$ e $dest(s, i) \notin T$ adiciona o vértice s e a aresta (s, i) à extensão arborescência. Vértice s é adicionado à T . Continue a crescer a árvore até que não haja mais vértices pode ser adicionado.

2. Tour de transição - começando no vértice que foi escolhido como o nó raiz do abrangendo arborescência, selecione uma aresta de saída e move para o correspondente vértice de destino. A prioridade mais baixa na seleção de bordas de saída é dada a as arestas que fazem parte da arborescência de vâo. As outras arestas podem ser escolhidas em ordem arbitrária. Continue a aumentar o tour de transição até que mais arestas não possam ser adicionadas. Como exemplo, considere o gráfico simétrico na Figura 9.4. Para identificar as arestas, nós rotulou-os com letras. Vários rótulos representam várias arestas. Tem, por exemplo, três arestas direcionadas de q_0 a q_2 , denominadas d, e e f .

q_0

q_3

q1
q2
 d, e, f
 h
 a, b
 c
 g
 e, u, j

Figura 9.4 - Um gráfico simétrico

Uma arborescência generalizada com raiz q_3 pode conter, por exemplo, arestas i, d e b .

Página 204

SEÇÃO 9.3
TESTE ESTRUTURAL
193

Usando o Algoritmo 9.2, dois passeios de transição diferentes com base neste arborescência são então $c, e, j, g, a, f, h, b, d, i$ e $g, a, f, j, c, e, h, b, d, i$.

Se o gráfico que estamos considerando não for simétrico, devemos primeiro transformá-lo em um gráfico simétrico duplicando arestas. Cada borda que é duplicada então corresponde a uma transição que será executada mais de uma vez na transição final Tour. Na teoria dos grafos, a duplicação é chamada de *aumento* do gráfico original. O Algoritmo 9.3 é uma maneira simples de derivar tal aumento (cf. Exercício 9-12).

ALGORITMO 9.3 - AUMENTAÇÃO DE GRÁFICOS

1. Rotule cada vértice no gráfico com um número inteiro que representa a diferença entre o número de arestas de saída e de entrada para esse vértice. Este número pode ser positivo, zero ou negativo. Uma vez que, por definição, cada aresta tem uma origem e um destino, a soma de todos os valores do rótulo deve ser zero.
2. Selecione um vértice A com um rótulo negativo e um vértice B com um rótulo positivo. Encontre o caminho mais curto de A a B percorrendo o menor número de arestas no *original* gráfico. Duplicique as bordas ao longo deste caminho. Atualize os rótulos de A e B . Os rótulos nos vértices intermediários não mudam.
3. Se o gráfico aumentado for simétrico, o algoritmo termina. O custo do aumento é o número total de arestas que foram duplicadas. Se o gráfico não for simétrico, volte ao passo 2.

Verifique, por exemplo, que o gráfico na Figura 9.4 é um aumento simétrico do diagrama de transição de estado na Figura 8.1 (cf. Exercício 9-4).

A etapa 2 do algoritmo exige o cálculo da distância mais curta entre dois vértices. Uma série de algoritmos foi estudada para resolver este problema de forma eficiente. Consulte as notas bibliográficas no final deste capítulo para uma visão geral. O custo de mate do aumento depende de uma maneira sutil da escolha dos vértices que é feito na etapa 2 do Algoritmo 9.3. Felizmente, há apenas um número finito de maneiras de onde essas escolhas podem ser feitas cada vez que a etapa 2 é executada. Portanto, poderíamos tentar encontrar o aumento de custo mínimo por meio de pesquisa exaustiva. Existem, no entanto, melhores métodos. Um método é estudar o aumento do gráfico como um problema de fluxo de rede. O problema pode então ser representado como um problema de *fluxo de custo mínimo*, que pode ser resolvido em tempo polinomial (ver Notas Bibliográficas).

Depois de um aumento simétrico, o Algoritmo 9.2 pode ser usado para derivar um passeio de transição. Para um aumento de custo mínimo, o tour de transição produzido pelo Algoritmo 9.2 irá também ser o mais curto e, portanto, o conjunto de testes de menor custo para o IUT. O problema de encontrar um tour de transição em um gráfico não simétrico, onde cada transição é exercida pelo menos uma vez, e possivelmente mais de uma vez, é conhecido como o *Carteiro Chinês Problema*. Conforme indicado acima, o problema pode ser resolvido em tempo polinomial.

Para derivar a sequência de teste com Algoritmos 9.1, assumimos que o IUT tem três propriedades: uma mensagem de reconfiguração, uma mensagem definida e uma mensagem de status. A mensagem definida é uma raridade que provavelmente não estará presente em muitos IUTs, mas, felizmente, na construção de sequências de teste com base em um tour de transição com o Algoritmo 9.2, não

Página 205

194
TESTE DE CONFORMIDADE
CAPÍTULO 9

preciso mais.

A ausência de *reinicialização* e as mensagens de *status* são mais problemáticas. O reset

a mensagem pode ser substituída por uma sequência de transições, chamada *sequência de homing*, que é conhecido por trazer o sistema de volta ao estado inicial, qualquer que seja o seu estado atual estar. Em geral, uma sequência de homing é definida como um procedimento adaptativo, onde as respostas geradas pela máquina podem ser usadas para determinar qual será a próxima entrada mensagem deve ser. Pode-se mostrar que todo estado finito fortemente conectado máquina deve ter tal sequência de homing adaptativa. Melhor ainda, o homing a sequência pode ser derivada algorítmicamente. Pode-se mostrar que nunca é preciso mais do que $S(S-1)/2$ transições antes que a máquina alcance um estado conhecido (ver Bibliografia). Para alcançar o estado inicial do sistema após esse ponto ser alcançado, é necessário entre zero e $S - 1$ transições extras. Considere, por exemplo, a máquina na Tabela 9.1.

Tabela 9.1 - Exemplo

A máquina tem dois estados, então $S = 2$ e $S(S-1)/2 = 1$. Uma sequência de comprimento um pode nos dizer em qual estado a máquina está. Nunca leva mais do que $(S-1) + S(S-1)/2 = 2$ entradas para redefinir a máquina para o estado q_0 com certeza. Um desafio mais significativo é colocado pela omissão da mensagem de *status*. O também não é provável que a propriedade de status esteja presente em um IUT, se não for necessária para o mau funcionamento do protocolo. Para substituir a mensagem de status, podemos usar uma sequência de transições chamadas de *assinatura de estado* ou sequência de *entrada / saída única* (UIO). A UIO sequência pode determinar se o IUT está em um determinado estado quando o UIO começa. UMA A sequência UIO, então, tem o objetivo oposto de uma sequência homing: identifica o primeiro em vez do último estado da sequência. Para poder verificar cada estado no IUT, devemos ser capazes de derivar uma sequência UIO para cada estado separadamente. Nem todas as sequências UIO são necessariamente diferentes. Na verdade, pode ser possível derivar um sequência UIO única que pode ser usada para identificar qualquer estado em uma máquina de estado finito. Essa sequência é chamada de *sequência distintiva*. A máquina de estado finito que foi

discutido no Capítulo 8 (cf. Figura 8.1) ilustra, no entanto, que nem todos os estados finitos as máquinas têm uma sequência distinta. Também pode ser mostrado, em grande parte da mesma forma, nem todos os estados têm uma sequência UIO. Nas próximas duas seções, primeiro suponha que uma sequência UIO pode ser derivada para todos os estados na especificação. Nós mostrar como essas sequências UIO podem ser usadas para substituir as mensagens de status em uma transição tour de ção.

O método discutido abaixo é popular, mas deve ser lembrado que, com o

Página 206

SEÇÃO 9.4
DERIVANDO SEQUÊNCIAS DE UIO
195

substituição da mensagem de status por uma sequência UIO, o poder de detecção de falhas do o tour de transição é reduzido. As sequências UIO, afinal, pressupõem um implemento correto tação. Uma máquina defeituosa poderia, em princípio, enganar um observador fazendo-o acreditar que um

determinado estado foi alcançado gerando accidentalmente apenas as respostas certas para um sequência UIO pré-computada. Um método alternativo que não é baseado em UIO sequências é discutido na Seção 9.6.

9.4 SEQUÊNCIAS DE UIO DERIVADAS

Nesta seção, mostramos primeiro como as sequências UIO podem ser derivadas, assumindo que existir. Na próxima seção, mostramos como o tour de transição pode ser modificado para combater o efeito colateral da aplicação de sequências UIO.

Até agora, o método mais conhecido para encontrar sequências UIO é enumerar todas as E / S possíveis sequências e verificá-las quanto à propriedade UIO. O Algoritmo 9.4 faz isso construindo uma árvore em expansão exponencial de sequências de E / S. Os nós na árvore a distância N a partir da raiz correspondem às sequências de E / S de comprimento N . Cada nó associou a ele dois conjuntos de estados. O primeiro conjunto, P , contém uma partição do conjunto de S estados em classes, onde S , por definição, é igual a $|Q|$. Dois estados estão em a mesma classe do particionamento P se e somente se eles não podem ser distinguidos de um ao outro pela aplicação da sequência de I / O representada pelo nó: o especificação produz os mesmos resultados nesta sequência, não importa qual destes estados é escolhido como um estado inicial. Os membros do segundo conjunto (ordenado) T definem para cada estado em S , qual será o estado final se a sequência de I / O representada pelo nó foram aplicados com esse estado como um estado inicial.

Seja $dest(i, j)$ novamente o estado que o IUT deve atingir se a entrada j for recebida enquanto em estado i , e deixe a saída (i, j) definir o sinal de saída que é gerado durante a transição ção, se houver. Além disso, deixe $T[i]$ definir o i -ésimo elemento do conjunto T para o nó atual, e $T_j[i]$ o i -ésimo elemento do conjunto T para seu j -ésimo nó sucessor.

ALGORITMO 9.4 - DERIVAÇÃO DE UIO

1. Inicialmente, o particionamento P consiste em um único conjunto que inclui todos os estados S do especificação. O conjunto T tem S membros. O valor inicial para o i -ésimo membro em T , com

$1 \leq i \leq S$, é i . A árvore de sequências de I / O é inicializada em um único nó, chamado de raiz nó, que corresponde à sequência nula. Inicialmente, o nó raiz é a única folha

nó na árvore. (Um nó folha é um nó sem sucessores.)

2. Classifique os nós folha da árvore em uma lista e exclua as duplicatas. Para cada nó folha agora atribua V nós sucessores na árvore, um para cada sinal de entrada possível. Conjunto $T_j[i] = dest(T[i], j)$.

3. O particionamento P_j para o j -ésimo nó sucessor é derivado do atual particionando P da seguinte maneira. Vamos definir O definir os sinais de saída associados a cada um dos os estados S para a última transição. Considere cada classe em P separadamente. Faça uma lista de todos os sinais de saída distintos que são gerados pelos estados incluídos na classe considerado. Esta classe em P é agora dividida em subclasses de forma que todos os estados que geraram o mesmo sinal de saída são atribuídos à mesma subclasse. Se todos os estados dentro da classe considerada gerou o mesmo sinal de saída para o último símbolo de entrada

Página 207

196
TESTE DE CONFORMIDADE
CAPÍTULO 9

aplicadas, todas permanecem na mesma classe de particionamento.

4. Se houver uma classe no particionamento de P neste ponto que contém apenas um estado, o

O nó que contém esse particionamento definirá uma sequência de UIO para esse estado.

5. As etapas 2 a 4 são repetidas até que as sequências UIO para todos os estados tenham sido encontradas (ou até a memória disponível está esgotada).

Este algoritmo procura sequências UIO para todos os estados na especificação ao mesmo Tempo. Porque ele verifica exaustivamente todas as sequências de entrada possíveis, com o mais curto sequências sendo inspecionadas primeiro, ele encontra as sequências UIO mais curtas primeiro. O algoritmo requer uma quantidade crescente de espaço para prosseguir na busca por sequências além dos primeiros níveis. No pior caso, o número de nós na árvore para sequências de comprimento n é

i = 0

©

$$V_{eu}^n$$

Isso significa que, para um alfabeto de entrada de dez ou mais mensagens, torna-se impraticável para pesquisar sequências com mais de cinco ou seis etapas. O problema de determinar se um estado tem uma sequência UIO foi provado ser difícil de PSPACE e de forma semelhante o problema de determinar a sequência UIO mais curta possível, dado que pelo menos um tal sequência existe (ver Notas Bibliográficas). Na prática, no entanto, as sequências UIO muitas vezes podem ser encontrados dentro dos limites do algoritmo.

9.5 TOURS DE TRANSIÇÃO MODIFICADOS

Em seguida, deve ser mostrado como as sequências UIO podem ser incorporadas em uma transição tour para construir um teste de conformidade. Um problema é que uma sequência UIO irá, em geral, deixar o IUT em um estado diferente daquele que está sendo verificado e, portanto, interfere com o tour de transição.

Chame a sequência UIO que identifica o estado i UIO_i e chame seu estado final de $dest$ (UIO_i).

Para cada estado i , agora aumentamos o gráfico da especificação original com um

"pseudo-transição" para cada símbolo de entrada j no estado i :

$$(i, dest(UIO_{dest(i,j)}))$$

Esta pseudo-transição consiste na borda percorrida para o teste do sinal de entrada j a seguir baixado pela verificação do estado de destino, usando a sequência UIO para esse estado.

Com estados S e símbolos de entrada V , existem trivialmente pseudo-transições SV . o problema de modificar um tour de transição existente para a inclusão de sequências UIO, então, é realmente o problema de encontrar um novo passeio de transição através do pseudo-apenas transições. Chame o gráfico contendo apenas pseudo-transições de pseudo-gráfico.

apenas transições. Chamamos o gráfico contendo apenas pseudo-transições de pseudo-gráfico. Fazemos um aumento simétrico do pseudo-gráfico e, em seguida, calculamos um Euler passeio com o algoritmo 9.2. Para o aumento, podemos usar o Algoritmo 9.3, mas com uma exceção importante: o cálculo da distância mais curta na etapa 2 é baseado em o gráfico original sem as pseudo-transições.

Este problema de encontrar um passeio de transição através de um subconjunto das arestas de um gráfico (ou seja,

9.6 UM MÉTODO ALTERNATIVO

O método baseado em sequências UIO pode ser usado para produzir testes de conformidade de boa qualidade, mas tem algumas desvantagens. Em primeiro lugar, nem todos os estados em uma máquina de estados finitos

necessariamente tem uma sequência UIO, e mesmo se tiver, a sequência UIO pode ser muito longo para ser derivado algorítmicamente. O problema de derivar sequências UIO é PSPACE completo, o que significa que apenas sequências UIO muito curtas podem ser encontradas em prática. Em segundo lugar, as sequências UIO só podem identificar com segurança os estados em um IUT correto.

É desconhecido e incognoscível qual é o seu comportamento para IUTs defeituosos. Em particular, eles não podem garantir que qualquer tipo de falha em um IUT permaneça detectável com o passeios de transição modificados.

Se outras suposições podem ser feitas sobre os tipos de falhas no IUT, a construção

É possível obter um teste confiável, que pode ser feito com um algoritmo de tempo polinomial.

A suposição principal é que nenhuma falha pode aumentar o número de estados alcançáveis ou

o número de sinais de entrada do IUT. O método que discutimos abaixo é baseado no uso de *sequências de caracterização*.

CARACTERIZANDO SEQUÊNCIAS

Suponha que a especificação do protocolo original corresponda a um estado finito minimizado máquina. Para cada dois estados desta máquina existe uma sequência finita de entradas que acionam uma sequência diferente de saídas. Essa sequência é chamada de *sequência de caracterização*. Pode-se mostrar que toda sequência de caracterização tem um comprimento menor do que S etapas, o número de estados na máquina. Embora haja $S(S-1)/2$ pares distintos de estados em uma máquina, é fácil ver que não mais do que $(S-1)$ diferentes sequências de caracterização são necessárias para separar qualquer combinação de dois estados. As sequências de caracterização $S-1$ podem ser selecionadas a partir do conjunto máximo das sequências $S(S-1)/2$ como se segue.

ALGORITMO 9.5 - SELEÇÃO DE SEQUÊNCIAS DE CARACTERIZAÇÃO

1. Selecione dois estados arbitrários da máquina e encontre uma sequência que os separe. As diferentes sequências de saída em resposta a esta sequência podem ser usadas para particionar o S estados em pelo menos dois conjuntos diferentes. Os conjuntos são blocos em uma partição de estados.
 2. Selecione um desses blocos contendo mais de um estado. Selecione dois estados de esse bloco e encontre uma sequência da coleção original que possa separá-los.
 3. O número de conjuntos de estados (blocos em uma partição dos estados) é aumentado em pelo menos um conjunto extra para cada nova sequência de caracterização que encontrarmos. O procedimento, portanto, pode ser repetido no máximo $S-1$ vezes.
- Neste momento, cada bloco no particionamento contém apenas um único estado. Para quaisquer dois estados em dois blocos diferentes, agora selecionamos uma sequência que pode separá-los. O conjunto de sequências de caracterização $S-1$ selecionadas da coleção original pode ser usado para distinguir entre qualquer par de estados na máquina.
- Seja $CS(i, j)$ a sequência de caracterização que distingue o estado i do estado j . Deixe $P(i)$ ser uma sequência de entradas que leva a máquina do estado inicial ao estado i em

Página 209

198

TESTE DE CONFORMIDADE
CAPÍTULO 9

a especificação de referência e seja R a mensagem de reinicialização que retorna a máquina para o estado inicial. O teste de conformidade agora pode ser executado da seguinte maneira. O algoritmo começa numerando os estados da máquina de estados finitos na pesquisa em amplitude ordem. Os números são usados posteriormente para garantir que os estados que podem ser alcançados no menor número de transições do estado inicial é testado primeiro.

ALGORITMO 9.6 - TESTE DE CONFORMIDADE

1. Numere os estados da máquina em ordem de largura. Isso pode ser feito por construindo uma árvore abrangente dos estados. O estado inicial da máquina torna-se o nó raiz da árvore. Inicialmente, é a única folha da árvore (um nó sem sucessores) e obtém o número mais baixo na ordem de pesquisa em amplitude.
2. A cada folha da árvore, conectamos todos os estados que podem ser alcançados por um único transição na máquina de estados finitos. Nenhum estado, entretanto, pode ser adicionado à árvore mais que uma vez. As novas folhas são numeradas consecutivamente, em ordem arbitrária. Etapa 2 é repetido até que todos os estados da máquina sejam representados.
3. O k -ésimo estado na ordem de busca em largura, $1 \leq k \leq S$, é testado com a entrada sequência.

$$\square(i), i \square< k \square\square RP(i) CS(i, k) RP(k) CS(i, k)$$

A sequência de teste verifica se o k -ésimo estado pode ser distinguido de todos os estados com um menor número de pesquisa em amplitude, ou seja, de todos os estados verificados antes.

Passar no teste k -ésimo nesta série mostra que o IUT tem pelo menos k estados distintos e que faz a transição ao longo das bordas da árvore, do estado inicial para cada um desses estados, são implementados corretamente.

4. Em seguida, todas as transições restantes da máquina devem ser verificadas. Em geral, para cada estado i e transição j , devemos realizar o seguinte teste

$$RP(i)j$$

Podemos pular o teste de transições que correspondem às arestas na árvore de pesquisa em largura; eles já foram testados na etapa 3. Depois que a saída em resposta à entrada j foi verificado, o novo estado que foi alcançado deve ser verificado novamente, usando o método da etapa 3, comparando-o sistematicamente com todos os outros estados na especificação.

A ordem de busca em largura garante que os caminhos $P(i)$ sejam verificados em uma transição de uma vez. Se o estado i só pode ser alcançado a partir do estado inicial, depois de passar algum outro estado j , a ordem de pesquisa garante que o estado j seja verificado primeiro. O custo total de identificação dos estados S é $O(S^3)$. O custo de verificar um único transição é $O(S^2)$, e como há transições VS , o custo total é $O(VS^3)$. O custo,

finalmente, de derivar as sequências características (outra teoria de grafos padrão problema) é $O(VS^2)$.

Este custo do teste de conformidade, portanto, ainda é polinomial em S e V e, ao contrário o método baseado UIO, o custo da sua construção é também polinomial em S e V .

Além disso, ao contrário do método baseado em UIO, um teste de conformidade como este pode sempre ser

construída e tem a garantia de detectar qualquer falha no IUT além daquelas que aumentar o número de estados ou sinais de entrada. Algoritmo 9.6 assume a existência de uma mensagem de reinicialização confiável. Yannakakis e Lee mostraram que para uma máquina sem

Página 210

SEÇÃO 9.7

RESUMO

199

uma mensagem de redefinição ainda existe um teste de conformidade de comprimento polinomial com o mesmo

cobertura de falhas como Algoritmo 9.6 (ver Notas Bibliográficas). Nenhum algoritmo de tempo polinomial

Ritmo é conhecido, entretanto, por derivar tal sequência de teste para uma máquina sem reinicialização.

9.7 RESUMO

Um teste de conformidade é projetado para verificar se uma implementação desconhecida de um protocolo pode ser considerado equivalente a uma especificação conhecida. O teste pode nunca produzir uma resposta que seja totalmente confiável. Apenas a presença de desejáveis o comportamento pode ser testado, não a ausência de comportamento indesejável. É, portanto sempre possível que uma implementação seja capaz de respostas que não fazem parte da especificação.

Em princípio, existem duas abordagens para o problema de teste de conformidade. Um é um abordagem bastante *ad hoc* onde, por tentativa e erro, a disposição correta do principal pro As funções do tocol são verificadas para uma gama de valores de parâmetro tão ampla quanto possível. Para

exemplo, se o objetivo do protocolo é o gerenciamento de conexão, podemos testar sequências escolhidas internamente para configuração e desmontagem da conexão. Um segundo método é sondar sistematicamente a implementação com sequências de teste para estabelecer se é estrutura interna, vista como uma máquina de estado finito, conforma-se com a estrutura da especificação. Valores de parâmetro e dados não são considerados neste tipo de teste.

Em vez disso, o foco está na estrutura de controle do protocolo apropriado. A maioria dos progress no desenvolvimento de ferramentas de teste de conformidade de protocolo foi feito com o segundo tipo de teste.

Não surpreendentemente, um teste de conformidade eficaz pode ser muito facilitado se especificação e implementação foram desenvolvidas com a viabilidade de um teste em mente.

Um teste de conformidade sistemático só é possível se o IUT tiver pelo menos os três empates listados na Seção 9.3. Um algoritmo particularmente simples (Algoritmo 9.1) pode ser aplicado se, além disso, o IUT tiver um reset, status e transição definida em cada estado.

O trabalho árduo em testes de conformidade ocorre quando um ou mais dos desejáveis faltam pertences.

Sem a propriedade set, o melhor método é encontrar um tour de transição de todos os estados usando Algoritmos 9.2 e 9.3, testando a resposta do IUT a todos os sinais de entrada possíveis.

Uma mensagem de status pode ser substituída por uma sequência de teste, chamada de sequência UIO, que pode

revelar da mesma forma o estado da máquina. O Algoritmo 9.4 pode ser usado para derivar estes Sequências UIO. O principal problema a ser resolvido aqui é que as sequências UIO disturb o estado do IUT quando eles são aplicados. A seção 9.5 mostra como uma transição O tour pode ser construído de forma a evitar esse problema.

Um método alternativo, que também evita o uso de mensagens de status e conjunto, é discutido na Seção 9.6. Todos os métodos discutidos podem falhar quando um erro de implementação no IUT aumenta o número de estados alcançáveis ou o número de sinais de entrada. No ausência de tais erros, a sequência de teste de conformidade produzida pelo Algoritmo 9.6 pode garantir a detecção de todas as falhas. A duração da sequência de teste, no entanto, pode ser consideravelmente maior do que aquele baseado em sequências UIO.

200

TESTE DE CONFORMIDADE

CAPÍTULO 9

EXERCÍCIOS

9-1. Explique em detalhes por que é essencial que cada um dos três primeiros requisitos da estrutura de um IUT ser cumprida para que um teste de conformidade seja viável.

9-2. É necessário que a máquina de estados finitos modelada pelo IUT tenha sido minimizada?

9-3. Como os algoritmos neste capítulo terão que ser alterados para especificações incompletas máquinas de estado finito?

9-4. O aumento simétrico da Figura 8.1 mostrado na Figura 9.4 é único. Se não, é um aumento com o menor custo?

9-5. Um teste de conformidade pode detectar se o IUT tem mais estados do que o formal especificação? Menos estados?

9-6. A cobertura de falhas de um teste de conformidade pode chegar a 100%?

9-7. Explique a diferença entre uma sequência distinta, uma sequência homing, um personagem sequência de terização e uma sequência UIO.

9-8. Como a cobertura de falhas de um teste de conformidade seria afetada se a aplicação de um mensagem de status, ou seu equivalente, foram excluídos de um tour de transição?

9-9. Escreva um algoritmo para encontrar sequências de homing.

9-10. Escreva um algoritmo para encontrar sequências de caracterização.

9-11. Como o custo dos dois algoritmos acima (o número de operações a serem realizadas formada no pior caso) dependem do número de arestas e vértices no gráfico? pode você derivar limites superiores?

NOTAS BIBLIOGRAFICAS

Os métodos de teste de conformidade são de interesse para todos os usuários de protocolo que desejam avaliar

a qualidade das implementações de protocolo. Eles também são de interesse internacional organismos de normalização, que visam fornecer uma certificação de terceiros neutros de protocols e implementação de protocolo. Organizações como a ISO e o CCITT são no processo de desenvolvimento de padrões e diretrizes para a certificação de protocolos destinado a cumprir, por exemplo, o modelo de referência para Open Systems Intercon-secção discutida no Capítulo 2, Rayner [1987].

O problema geral de teste de conformidade, ou seja, o problema de implementação de teste ções de máquinas de estado finito estendidas arbitrárias, usando testadores remotos em um banco de dados

rede, é uma área de pesquisa ativa. O progresso é relatado no trabalho anual do IFIP Grupo 6.1 Simpósios sobre Especificação, Teste e Verificação de Protocolo, IFIP [1982- presente]. Uma excelente visão geral do problema geral pode ser encontrada em Rayner [1987]. O trabalho também está em andamento para padronizar uma notação para suítes de teste de conformidade, denominada *Tree and Tabular Combined Notation* ou TTCN, consulte ISO [1987]. Um dos primeiros esforços para desenvolver um centro independente para a avaliação e a certificação de implementações de protocolo foi iniciada pelo National Physical Laboratory (NPL) na Inglaterra, no início dos anos 1980, Rayner [1982, 1987]. Nos EUA, este trabalho foi realizado pelo Instituto Nacional de Ciência e Tecnologia (NIST, anteriormente denominado National Bureau of Standards ou NBS), Nightingale [1982]. Um over-a visão de outros centros de teste e certificação é fornecida em Wang e Hutchinson [1987].

CAPÍTULO 9

NOTAS BIBLIOGRÁFICAS

201

Duas questões complicam o trabalho dos centros de certificação. Primeiro, a certificação centros devem ser capazes de realizar testes remotos de implementações, através de um confiável rede de dados. Em segundo lugar, os testes de certificação às vezes têm que ser aplicados a um único camada de protocolo em uma hierarquia de camadas de outra forma confiáveis.

Os centros de certificação têm se concentrado principalmente no serviço, ou funcional, conforme teste de mance. O teste estrutural, conforme descrito aqui, é um desenvolvimento mais recente. O trabalho de teste de conformidade que descrevemos é baseado em ambos os estados finitos teoria da máquina e na teoria dos grafos. O conceito de uma sequência de teste foi estudado como no início de 1956 por EF Moore em um de seus primeiros artigos sobre máquinas de estado finito, Moore [1956]. Moore também foi o primeiro a definir sequências de homing e distinguir sequências. O conceito de sequência distinta foi desenvolvido em Gill

[1962] e em Hennie [1964]. Huffman estudou independentemente problemas semelhantes a aqueles em Moore [1956]. Seus resultados podem ser encontrados em Huffman [1964]. Uma completa discussão de sequências homing e sequências de caracterização pode ser encontrada em Kohavi [1978].

Naito [1981] e Sarikaya [1984] estiveram entre os primeiros a estudar o protocolo sistemático técnicas de geração de teste usando tours de transição. O conceito de uma sequência UIO foi introduzido em Hsieh [1971] e discutido em Friedman e Menon [1971].

De forma independente, também foi descoberto por Gobershtain [1974]. KK Sabnani e AT Dahbura [1985, 1988] redescobriu o princípio e aplicou-o à conformidade problema de teste. O termo sequência UIO foi cunhado por eles. Hsieh usou o termo sequência de *E / S simples*; Gobershtain usou a *palavra de verificação*. Yannakakis e Lee [1990] introduziu o termo *assinatura de estado*.

Um método para reduzir o comprimento de uma sequência de teste de conformidade computando vários As sequências UIO por estado são descritas em Shen e Lombardi [1989].

COMPLEXIDADE

O problema de encontrar o tour de transição de comprimento mínimo de uma máquina de estados finitos, descrito por exemplo em Klee [1980], pode ser resolvido em tempo polinomial. Algoritmo 9.2 vem de Edmonds e Johnson [1973] e foi aplicado à conformidade

problema de teste em Uyar e Dahbura [1986]. Um aumento simétrico de um gráfico também pode ser encontrado em tempo polinomial com algoritmos de fluxo de rede. Um passeio Euler pode ser encontrado em um tempo que é linear no número de transições no simétrico gráfico. O algoritmo correspondente pode ser encontrado em Edmonds e Johnson [1973].

O problema de encontrar um passeio de transição através de um subconjunto das transições em um gráfico,

por exemplo, as pseudo-transições correspondentes às sequências UIO em um tour de transição, pode ser mostrado como NP-completo. Se, no entanto, o gráfico consiste em pseudo-transições estão fracamente conectadas, o problema se reduz a um que pode ser resolvido em tempo polinomial, conforme mostrado em Aho, Dahbura, Lee e Uyar [1988]. Algo eficiente ritmos para a derivação de viagens de transição para classes restritas de estado finito as máquinas são estudadas em Edmonds e Johnson [1973]. Os algoritmos foram os primeiros aplicado a testes de conformidade de protocolo em Uyar e Dahbura [1986] e estendido em

Página 213

202

TESTE DE CONFORMIDADE

CAPÍTULO 9

Aho, Dahbura, Lee e Uyar [1988]. O problema do carteiro chinês foi o primeiro descrito pelo matemático chinês MK. Kuan [1962].

Em geral, o custo de atravessar uma transição na máquina de estado finito pode ser dado por um número real. Um algoritmo bem conhecido para encontrar a distância mais curta (ou seja, o caminho de menor custo) entre dois vértices em um gráfico, dadas essas restrições, é O algoritmo de caminho mais curto de Dijkstra, por exemplo, Dijkstra [1959], Aho [1974], Price [1971]. Existem, no entanto, também métodos mais rápidos, ver, por exemplo, Tarjan [1983]. Em conformidade teste de mance, muitas vezes é possível associar simplesmente um custo unitário com o percurso de todas as transições. Neste caso, o melhor algoritmo para encontrar caminhos mais curtos é amplitude-primeira pesquisa.

Soluções eficientes para problemas de custo mínimo - fluxo máximo também são discutidos em detalhes em Tarjan [1983] e Gibbons [1985]. Uma visão geral pode ser encontrada em Aho, Dahbura, Lee e Uyar [1988].

Algoritmo 9.6, e a análise de sua complexidade, foi descrito em Yannakakis e Lee [1990]. É semelhante ao método W de Chow [1978]. Uma prova do

A dureza PSPACE do problema de derivação UIO também pode ser encontrada em Yannakakis e Lee [1990].

Uma questão não discutida aqui é o problema de estimar a qualidade ou cobertura de falhas de um teste de conformidade. Observe que, apesar dos três primeiros requisitos da Seção 9.3, um IUT com defeito pode muito bem ter mais estados ou mais entradas do que nossa referência especificação. Em Vasilevskii [1973], foi mostrado que uma sequência de verificação torna-se inherentemente exponencial se as falhas aumentarem o número de estados. A máquina também pode têm respostas não determinísticas ou podem não estar fortemente conectadas, conforme necessário. Sempre há um número infinito de tais implementações que podem passar por um determinado con-

teste de desempenho sem ser equivalente à especificação de referência. Em teoria, há portanto, a cobertura de falhas de cada teste de conformidade do tipo que descrevemos deve abordagem zero. No entanto, a cobertura de falha relativa do teste de conformidade individual os métodos podem diferir. Métodos empíricos para medir tais diferenças são ilustrado em Dahbura e Sabnani [1988] e Sidhu e Leung [1989]. De forma aleatória modificar transições em uma descrição de máquina de estado finito do IUT pode ser medido ured qual porcentagem desta classe restrita de erros é detectada por um teste de conformidade método ing. Até agora, no entanto, esses testes só foram usados com sucesso para confirmar resultados que também podem ser comprovados teoricamente.

Não discutidos neste capítulo são dois métodos alternativos para testes de conformidade que foram explorados. O método W foi introduzido em Vasilevskii [1973] e elaborado em Chow [1978]. O método também é explicado em Shih e Sidhu [1986]. O segundo método é baseado no uso de gramáticas para gerar sequências de teste e foi explorado em Linn e McCoy [1983] e Probert e Ural [1983]. Um general uma visão geral dos métodos de teste é fornecida em Sidhu [1990]. Um estudo formal interessante do o problema de teste de conformidade é descrito em Brinksma et al. [1989].

Página 214

SÍNTESE DO PROTOCOLO 10

203 Introdução 10.1
203 Derivação do protocolo 10.2
208 Algoritmo de Derivação 10.3
210 Projeto Incremental 10.4
210 Place Synchronization 10.5
211 Resumo 10.6
212 exercícios
212 Notas Bibliográficas

10.1 INTRODUÇÃO

Um dos problemas abertos mais difíceis no projeto de protocolo é encontrar uma disciplina de que pode garantir *a priori* a derivação de um protocolo funcionalmente correto que está livre de erros dinâmicos, como deadlock. Uma disciplina de design adequada levará para um produto menor e mais eficaz, mais fácil de manter e modificar. Por enquanto, pouco progresso foi feito nesta área. Este capítulo é, portanto, necessariamente tentativa.

Discutimos brevemente três métodos para a construção interativa de especificações corretas de protocolo ções. Os dois primeiros desses métodos enfocam a funcionalidade de um projeto de protocolo; o terceiro enfatiza a estrutura.

Tenha em mente que um método de síntese de protocolo não pode sintetizar especificações de serviço ções. Nenhuma ferramenta automatizada pode determinar a finalidade de um novo protocolo. O design problema é encontrar um protocolo que (1) realize um *determinado* serviço, e (2) o faça em um maneira livre de erros. Todos os três métodos discutidos abaixo assumem que uma especificação de serviço

existe, seja de forma formalizada ou informalmente na mente do usuário do ferramenta de síntese.

Na próxima seção, ilustramos um método de derivação de protocolo que nos permite sincronizar thesise os componentes do protocolo de uma especificação formal. Fazemos isso por formaliz a especificação do serviço de tal forma que uma estrutura de esqueleto para o protocolo regras de procedimento de cada processo de comunicação podem ser extraídas dele. O syn-esses processos dimensionados podem ser ajustados manualmente.

10.2 DERIVAÇÃO DE PROTOCOLO

Na validação do protocolo, podemos verificar as afirmações que o usuário faz sobre o estrutura de possíveis diálogos entre processos. Um diálogo é uma sequência de mensagens trocas de sábios que podem ser observadas em uma determinada interface, por exemplo, um conjunto de canais.

Considere o problema de projetar um protocolo de gerenciamento de conexão.
203

Página 215

204
SÍNTESE DE PROTOCOLO
CAPÍTULO 10

uma

b

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Figura 10.1 - Interface

Existem dois processos, *um* e *b*, que compartilham o acesso a um link de dados full-duplex, indicados com duas setas na Figura 10.1. Processa *um* e *b* têm de coordenar o início e o término das transferências de dados pelo link.

Normalmente, o designer é solicitado a fornecer duas especificações de processo, uma para cada lado da conexão, em uma tentativa de restringir os possíveis diálogos a uma conjunto definido. Uma afirmação sobre essas restrições pode ser formalizada e verificada por um validador de protocolo automatizado. Na síntese do protocolo, podemos tentar resolver este problema ao redor começando com uma especificação do conjunto de diálogos permitidos e a partir deles, de modo que, por construção, esses processos sejam incapazes de exibir qualquer comportamento diferente do declarado.

Fornecemos especificações para dois processos. Qualquer um dos lados pode iniciar uma conexão; E se ambos os processos tentam fazer isso ao mesmo tempo que a tentativa falha. O comportamento pode ser especificado como uma máquina de seis estados, como segue, em uma notação informal semelhante PROMELA. A notação *a* → *b* codifica informalmente a direção na qual uma mensagem fluxos.

```
gerente de especificações
{
  ocioso:
    E se
      :: b-> a! connect -> goto bOpens
      :: a-> b! connect -> goto aOpens
    fi;
    aOpens:
    E se
      :: b-> a! accept -> goto conectado
      :: b-> a! connect -> goto idle
      /* conflito */
    fi;
    bOpens:
    E se
      :: a-> b! accept -> goto conectado
      :: a-> b! connect -> goto idle
      /* conflito */
    fi;
    conectado:
    E se
      :: b-> a! desconectar -> goto bCloses
      :: a-> b! desconectar -> goto aCloses
    fi;
```

Página 216

205

```
aCloses:
b-> a! desconectar -> goto idle;
bCloses:
a-> b! desconectar -> ir para inativo
}
```

Esta especificação descreve as trocas de mensagens que são visíveis na interface entre *uma* e *b*, isto é, na linha pontilhada na Figura 10.1. Pode haver outras mensagens que são tratados por *a* ou *b*, e pode haver muitos outros testes e manipulações de dados a ser executado. A especificação acima é, portanto, parcial.

Algumas das mensagens devem ser enviadas pelo processo *a* e outras recebidas por *a*.

Podemos derivar uma descrição do processo de esqueleto da especificação que descreve pré-especificamente as restrições para o processo *a*. Mecanicamente, podemos derivar o seguinte máquina de estado para o processo *a*. Podemos dizer que é o derivado do gerenciador de especificações com respeito a *a*.

```
proctype D_manager_D_a ()
```

```

{
R0:
E se
:: b! conectar -> goto R1
:: a? conectar -> goto R2
fi;
R1:
E se
:: a? conectar -> goto R0
:: a? aceitar -> goto R3
fi;
R2:
E se
:: b! conectar -> goto R0
:: b! aceitar -> goto R3
fi;
R3:
E se
:: b! desconectar -> goto R4
:: a? desconectar -> goto R5
fi;
R4:
a? desconectar -> goto R0;
R5:
b! desconectar -> goto R0
}

```

A máquina de estado para o processo *b* é semelhante, uma vez que a especificação do protocolo é simétrica. A derivação é trivial neste caso e pode ser facilmente feita à mão. Em geral, entretanto, a derivação é mais sutil.

Considere o seguinte exemplo que descreve o comportamento de um bit alternado simples protocolo. A interface é a mesma mostrada na Figura 10.1. A especificação para o mensagens que cruzam a interface na linha pontilhada, no entanto, agora são formalizadas como segue baixos.

```

spec abp
{
Faz
:: a-> b! msg0;

```

206

SÍNTESE DE PROTOCOLO

CAPÍTULO 10

```

Faz
:: b-> a! ack0; quebrar
:: b-> a! ack1; a-> b! msg0
od;
a-> b! msg1;
Faz
:: b-> a! ack0; a-> b! msg1
:: b-> a! ack1; quebrar
od
od
}
```

Esta única especificação descreve completamente o comportamento de duas máquinas de protocolo, o emissor *a* e o receptor *b*. As duas derivações produzem os seguintes resultados.

```

proctype D_abp_D_a ()
{
R0:
b! msg0 -> goto R1;
R1:
E se
:: a? ack0 -> goto R2
:: a? ack1 -> goto R0
fi;
R2:
b! msg1 -> goto R1
}
proctype D_abp_D_b ()
{
R0:
b? msg0 -> goto R1;
R1:
E se
:: a! ack0 -> goto R2
:: a! ack1 -> goto R0
fi;
R2:
```

```
b? msg1 -> goto R1
}
```

De acordo com esta especificação, o reconhecimento errado pode ser repetido pelo receptor e será ignorado pelo remetente. Como resultado, o esqueleto da máquina de estado para *b* inclui um comportamento permitido, mas não desejável. Para evitar isso, devemos reescrever o processo derivado, manualmente, da seguinte forma, dividindo o estado R1 em duas metades:

```
proctype D_abp_D_b ()
{
R0:
b? msg0 -> goto R11;
R11:
E se
:: a! ack0 -> goto R2
fi;
R12:
E se
:: a! ack1 -> goto R0
fi;
R2:
b? msg1 -> goto R12
}
```

que pode ser simplificado via

Página 218

```
207
proctype D_abp_D_b ()
{
R0:
b? msg0 -> goto R11;
R11:
a! ack0 -> goto R2;
R12:
a! ack1 -> goto R0;
R2:
b? msg1 -> goto R12
}

para sua forma final:
proctype D_abp_D_b ()
{
R0:
b? msg0 -> a! ack0;
b? msg1 -> a! ack1;
ir para R0
}
```

Agora vamos ver como a derivação é afetada se expandirmos a especificação com um mensagem para um terceiro processo *c* que registra todas as mensagens transmitidas e confirmadas corretamente

sábios com número de seqüência zero.

```
especificação abp2
{
Faz
:: a-> b! msg0;
Faz
:: b-> a! ack0; a-> c! log; quebrar
:: b-> a! ack1; a-> b! msg0
od;
a-> b! msg1;
Faz
:: b-> a! ack0; a-> b! msg1
:: b-> a! ack1; quebrar
od
od
}
```

A derivada da especificação para *c* é simplesmente

```
proctype D_abp2_D_c ()
{
R0:
c? log -> goto R0
}
```

A derivada de *b* permanece inalterada, mas a derivada de *a* torna - se

```
proctype D_abp2_D_a ()
{
R0:
b! msg0 -> goto R1;
```

```

R1:
E se
:: a? ack0 -> goto R2
:: a? ack1 -> goto R0
fi;
R2:
c! log -> goto R3;
R3:
b! msg1 -> goto R4;

```

Página 219

208
SÍNTSE DE PROTOCOLO
CAPÍTULO 10

```

R4:
E se
:: a? ack0 -> goto R3
:: a? ack1 -> goto R0
fi
}

```

10.3 ALGORITMO DE DERIVAÇÃO

O esqueleto da máquina pode ser derivado de uma especificação em duas etapas. Primeiro, se nós derivar uma máquina para o processo p , todas as mensagens na especificação que não são enviados ou recebidos por p são substituídos por `skip`. A seguir, todas as especificações do tipo

$q \rightarrow p!$ mensagem

são traduzidos para

$p?$ mensagem

e, da mesma forma, todas as especificações

$p \rightarrow q!$ mensagem

tornar-se

$q!$ mensagem

A última etapa é lidar com casos como esses

```

R0:
E se
:: p? message0 -> goto R1
:: pular -> ir para R2
fi

```

onde o salto foi inserido na primeira etapa. Neste caso, um evento fora do derivado processo pode fazer o sistema mudar de estado, provavelmente mudando o comportamento futuro do ambiente do processo derivado. O processo derivado não faz, e não pode, saber quando ou se essa transição invisível ocorre. Deve, no entanto, ser capaz de aceitar qualquer mensagem recebida que possa chegar após a transição invisível levar Lugar, colocar. Portanto, para o exemplo acima, o estado $R0$ do processo derivado *herda* todos receber operações do estado $R2$, junto com as transições correspondentes.

Se o estado $R2$ for especificado

```

R2:
p? mensagem1 -> goto R3
o novo estado R0 torna-se
R0:
E se
:: p? message0 -> goto R1
:: p? mensagem1 -> goto R3
fi

```

Se o estado $R2$ oferece uma escolha

Página 220

SEÇÃO 10.3
ALGORITMO DE DERIVAÇÃO

209

```

R2:
E se
:: p? mensagem1 -> goto R3
:: q! mensagem2 -> goto R0
:: p? mensagem3 -> goto R0
fi

```

nós herdamos apenas as operações de recebimento e gravamos

```

R0:
E se
:: p? message0 -> goto R1
:: p? mensagem1 -> goto R3
:: p? mensagem3 -> goto R0

```

```
fi  
A única possibilidade restante é se o estado R2 especifica apenas uma operação de envio:
```

```
R2:  
q! mensagem2 -> goto R0  
Neste caso, a transição para pular é omitida e escrevemos  
R0:  
E se  
:: p? message0 -> goto R1  
fi  
que simplifica para  
R0:  
p? mensagem0 -> goto R1
```

Este último caso pode ser sinalizado como uma possível inconsistência na especificação. O a especificação neste caso requer que um processo aguarde um evento que não pode observar. A última etapa de derivação acima é repetida até que todas as transições "ocultas" tenham sido removido. Observe que, se o estado alvo R2 tem suas próprias transições de salto, a última derivação etapa de operação pode exigir a inspeção de ainda mais estados.

```
R2:  
E se  
:: p? mensagem1 -> goto R3  
:: pular -> goto R0  
:: pular -> goto R4  
fi
```

O algoritmo de derivação pode produzir máquinas de estado esqueleto para os processos alvo que cumprem as restrições da especificação. Ilustra um dos propósitos de um procedimento de síntese de protocolo: oferecer assistência automatizada a um designer de protocolo. O designer pode se concentrar na definição de apenas um item central: o protocolo própria especificação.

Infelizmente, este procedimento de design não dá nenhuma garantia de que a interação do processos derivados não levarão a erros dinâmicos, como deadlocks. Concentrando nesse aspecto do problema de design leva a um tipo diferente de procedimento de design, que discutiremos a seguir.

Página 221

210
SÍNTSE DE PROTOCOLO
CAPÍTULO 10

10.4 PROJETO INCREMENTAL

O seguinte método de design, publicado originalmente em 1980, é frequentemente usado como um guia linha para tentativas de construir procedimentos de síntese de protocolo. O procedimento é interativo, e assume a existência de uma especificação de serviço independente que o designer seguirá durante o desenvolvimento dos processos de protocolo.

O usuário especifica apenas as transmissões de mensagens. O sistema deduz onde no protocolo as ações de recebimento correspondentes são necessárias. Inicialmente, todos os processos, ou seja, o

As "máquinas de estado esqueleto" do primeiro método são atribuídas a um estado inicial fictício. O designer agora pode selecionar um dos estados no sistema e estendê-lo com uma mensagem transmissão sábia. O designer deve especificar o nome da mensagem, seu parâmetro ters, e seu destino. Para o processo de transmissão da mensagem, o designer também deve especificar um estado sucessor para a ação de envio: um existente ou um novo estado do processo criado.

Para cada borda de transmissão adicionada a um dos processos desta forma, a síntese o software rastreia todos os estados possíveis do processo de destino em que a mensagem pode ser recebido e atualiza a máquina de estado para esse processo automaticamente. O usuário tem que nomear os estados sucessores para todos os eventos de recepção de mensagens que são incluídos. Após cada atualização, o procedimento de design incremental pode avisar o designer que *tuplas de estado estável* foram criadas. Uma tupla de estado estável é definida como um composto estado do sistema no qual nenhuma mensagem está em trânsito ou armazenada em buffers. Se tal compo- o estado do sistema for alcançável, o estado deve persistir até que um dos processos envie um mensagem. Se nenhum dos processos pode transmitir uma mensagem, o estado estável alcançável tupla corresponde a um impasse.

O designer neste método só pode especificar ações de envio. O lugar onde o ações de recebimento correspondentes são necessárias é deduzido pelo software de síntese. Isto evita recepções não especificadas e certos tipos de impasse, mas não pode garantir

a correção *funcional* do protocolo. Ou seja, o método de síntese não pode garantir que um protocolo assim sintetizado realizará um determinado serviço.

10.5 SINCRONIZAÇÃO DE LOCAIS

A terceira abordagem pode ser considerada um meio-termo entre os dois primeiros métodos discutido acima. Este método começa com uma especificação de serviço escrita como um expressão dos requisitos de sincronização. Os símbolos na expressão de serviço são os nomes dos primitivos de serviço. Os operadores da expressão determinam como a execução dessas primitivas deve ser sincronizada. Na expressão

$a_1; (b_2 \parallel c_3)$

(1)

os sobrescritos denotam pontos de acesso de serviço, os locais físicos onde o serviço primitivos são executados. O ponto e vírgula é usado para indicar uma execução sequencial: o execução da primitiva de serviço a_1 , no lugar representado por 1, deve ter sido completada antes que as primitivas b_2 ou c_3 possam ser executadas nos locais 2 ou 3, respectivamente. As barras paralelas são usadas para indicar que as duas subexpressões podem ser executadas

Página 222

SEÇÃO 10.6
RESUMO

211

simultaneamente. Os parênteses são usados para agrupamento. Uma única barra entre dois subexpressões implica alternância, qualquer uma das duas subexpressões pode ser executada, mas não ambos.

Para fazer cumprir os requisitos de sincronização formalizados na expressão de serviço, o algoritmo de síntese pode derivar um protocolo que controla a execução do serviço primitivos. A execução sequencial na expressão (1), por exemplo, pode ser aplicada tendo o primeiro primitivo a_1 completo, transmitindo uma mensagem única do lugar 1 para os lugares 2 e 3, e atrasando a execução das primitivas b_2 e c_3 até que mensagem chegou.

O método de síntese é atraente, mas também apresenta desvantagens. O método pode derivar protocolos para apenas uma classe limitada de requisitos de sincronização global. Nem todos protocolos para o tocol podem ser facilmente expressas nesses termos. Considere um leitor / escritor protocolo para uma base de dados compartilhada entre vários processos. Um método para proteger a integridade dos dados é permitir que vários leitores estejam ativos, mas permitir o acesso a a maioria de um processo de gravação por vez, e somente quando nenhum processo de leitura estiver ativo.

Se um processo leitor i , exigindo acesso ao item n , é representado pelo símbolo r_{n_i} ,

e

o processo de gravação correspondente é representado por w_{n_i} ,

o problema de projeto é agora

escrever uma expressão regular nesses símbolos, usando os operadores; \parallel e $|$. Para sustentar descrever bem a solução, devemos contar o número de processos ativos de cada tipo e expressar o requisito de sincronização como condições nessas contagens. Mas o a expressão regular não nos permite fazer isso. Se uma expressão de sincronização pode ser encontrado, pode não ser mais fácil encontrá-lo do que inventar o protocolo final diretamente.

10.6 RESUMO

Um método ideal para o projeto de protocolo seria construir um modelo do zero que pode ser comprovado como correto pela construção. Não existe tal método, embora muitos interessantes tentativas foram feitas. Neste capítulo, demos uma visão geral de três dessas tentativas. O primeiro método permite extrair máquinas de estado de esqueleto de um sindicato formalizada de um requisito de correção. O método tem desvantagens, os mais importantes são:

O método não auxilia na correta formalização do protocolo própria especificação.

Os processos derivados devem, em alguns casos, ser ajustados para remover o permitido, mas comportamento indesejável. O método não oferece ajuda aqui, nem pode nos ajudar a verificar que as alterações preservem a correção das derivações.

O segundo método orienta interativamente o designer de protocolo para um design completo e emite avisos sobre potenciais bloqueios. As desvantagens mais importantes disso abordagem são:

O método não garante que o protocolo construído realize um determinado serviço.

O método não garante a ausência de erros dinâmicos, como deadlocks. Isto só pode alertar para a possibilidade de um deadlock. Quando o número de possíveis

Página 223

212

SÍNTSE DE PROTOCOLO

CAPÍTULO 10

os estados de impasse aumentam, como acontece em um design de tamanho realista, torna-se rapidamente impossível para um designer humano verificar manualmente que todos os potenciais impasses estados são efetivamente inacessíveis.

O terceiro método deriva protocolos de expressões concisas de sincronização global requisitos de instalação. Sua principal desvantagem é:

Apenas uma classe restrita de problemas de projeto de protocolo pode ser expressa no sistema regular linguagem de expressão na qual o método se baseia.

Todos os três métodos compartilham uma outra desvantagem que talvez seja ainda mais importante tância: eles não parecem realmente facilitar o processo de design.

EXERCÍCIOS

10-1. Tente derivar o protocolo de Lynch (Capítulo 2) e partes do protocolo do servidor de arquivos (Capítulo 7) com um método de síntese de protocolo.

10-2. Alguns métodos de síntese de protocolo que foram descritos na literatura garantem "correção por construção" com a ajuda de um algoritmo de análise de alcançabilidade exaustiva ritmo que é executado sobre especificações parciais durante o projeto. Considere o possível desvantagens deste método.

10-3. Compare o método de sincronização de local com o método de derivação de protocolo. Ambos comece com uma "especificação de serviço" abstrata da qual um protocolo é derivado. Como os dois tipos de especificações de serviço são diferentes? Eles têm o mesmo expressivo poder?

10-4. Desenvolva um método de síntese de protocolo viável e envie a solução para o autor para o próxima edição deste livro.

NOTAS BIBLIOGRÁFICAS

Este capítulo deu apenas uma breve visão geral das metodologias de síntese, uma vez que, infelizmente, não existe nenhum que possa resolver adequadamente o problema de projeto do protocolo.

O método mais conhecido para a síntese de protocolo é o método incremental da Seção 10.4. Foi descrito pela primeira vez em Brand e Zafiropulo [1980]. O método tem muitas variações e até mesmo aplicado em algoritmos de validação de protocolo. o método de sincronização de local da Seção 10.5 é um método formal para derivar partes de um protocolo de nível inferior de uma especificação de serviço de nível superior. O método é totalmente desenvolvido em Gotzheim e Bochmann [1986]. Uma variante também pode ser encontrada em Chu e Liu [1988].

O método de derivação da Seção 10.2 pode ser visto como uma extensão de um trabalho anterior em métodos para derivar a descrição de uma entidade de protocolo de uma especificação de seu parceiro de comunicação, ver Zafiropulo et al. [1980], Gouda [1983], Merlin e Bochmann [1983].

Não estudado aqui é uma aplicação recente potencialmente interessante da teoria de controle para o problema de síntese de protocolo que foi relatado em Rudie e Wonham [1990]. Nisso abordagem, o sistema de protocolo original é descrito pela primeira vez como um processo não controlado em que todas as ações viáveis, como transferência de mensagens, acontecem caoticamente. Um alto-a especificação de nível de serviço detalha as restrições do sistema. Supondo que o

Página 224

CAPÍTULO 10

NOTAS BIBLIOGRÁFICAS

213

processo contém um número suficiente de pontos de controle, um protocolo pode então ser derivado como uma restrição mínima do comportamento do processo caótico que satisfaz o sistema tensões.

Vários métodos também foram estudados para particionar um programa sequencial em um programa distribuído, preservando a funcionalidade e correção, por exemplo, Moitra [1985], Prinnoth [1982]. Esses algoritmos requerem uma solução inicial para o problema, através da derivação de um programa sequencial, antes que o próprio método de síntese possa ser aplicado.

Uma visão geral dos métodos de síntese de protocolo pode ser encontrada em Chu [1989].

VALIDAÇÃO DE PROTOCOLO 11

- 214 Introdução 11.1
- 214 Um Método de Prova Manual 11.2
- 218 Métodos de validação automatizada 11.3
- 226 O Algoritmo Supertrace 11.4
- 231 Detecção de Ciclos de Não Progresso 11.5
- 234 Detectando Ciclos de Aceitação 11.6
- 235 Verificando Reivindicações Temporais 11.7
- 235 Gestão da Complexidade 11.8
- 237 Limite dos Modelos PROMELA 11.9
- 238 Resumo 11.10
- 239 exercícios
- 240 Notas Bibliográficas

11.1 INTRODUÇÃO

No Capítulo 9, estudamos o problema de verificar se a implementação de um protocolo está em conformidade com uma especificação formal. Agora discutimos o problema de verificar a consistência lógica da própria especificação formal, independente de uma implementação. Para consistência, assumimos que a especificação é formalizada como uma validação de modelo no PROMELA, embora isso não seja essencial para muitos dos algoritmos que distribuímos.

Primeiro, descrevemos um método de prova manual baseado na noção de invariantes de estado só.

Em seguida, mostramos como o mesmo princípio pode ser usado para construir uma validação automatizada

sistema de dação. Finalmente, estendemos os algoritmos para apoiar também a verificação de outros requisitos de correção que podem ser expressos na PROMELA (consulte o Capítulo 6).

A maioria dos sistemas de validação automatizados são baseados em análises exaustivas de

acessibilidade. Para

estabelecer a observância de invariantes de estado, então, basta verificar a sua correção com um teste booleano simples para cada estado que é alcançável a partir de um determinado sistema inicial

Estado. O principal problema que deve ser abordado no projeto de tal sistema é o "problema de explosão de espaço de estado." Para protocolos de tamanho realístico, o número de estados alcançáveis do sistema geralmente são muito grandes para análises puramente exaustivas. Nós dis-

discutir a natureza deste problema e algumas das contra-estratégias que têm sido desenvolvido.

11.2 UM MÉTODO DE PROVA MANUAL

Considere um sistema de transmissão simples, com um emissor S e um receptor R . Processo S envia mensagens para o processo R em um meio de transmissão não confiável que pode perder, mas não inserir, reordenar ou distorcer mensagens. Cada mensagem transmitida carrega uma sequência número. Inicialmente, esse número é zero e é incrementado em um a cada nova mensagem transmitida. Ele pode crescer arbitrariamente. O receptor reconhece o

214

215

recebimento de mensagens ecoando os números de sequência em um dispositivo igualmente não confiável

canal de retorno. O receptor armazena o maior número de sequência que recebeu em um variável local B . O remetente tenta rastrear esse número mantendo uma contagem em uma variável local A . O valor de A é igual ao maior número de sequência que o remetente pode ter certeza de que R recebeu. Inicialmente, temos

$$A = \square B = 0$$

A seguir, assumimos que o emissor e o receptor simplesmente trocam a sequência números e nenhum outro dado. O protocolo é então definido por quatro operações atômicas, dois em cada processo. Eles podem ser formalizados na PROMELA da seguinte forma, onde para o por enquanto, vamos fingir que os dados do tipo `int` têm um alcance ilimitado. `w` é um constante positiva arbitrária.

```
mtype = {mesg, ack}
proctype S ()
```

```

{
int A;
Faz
:: R! Mesg (A + rand ()% W)
/* S1 */
:: S? Ack (A)
/* S2 */
od
}
proctipo R ()
{
int B, b;
Faz
:: S! Ack (B)
/* R1 */
:: atomic {
/* R2 */
R? Mesg (b);
B = fct (b, B);
}
od
}

```

A transição R_2 consiste em duas instruções que são, pelo menos conceitualmente, executadas em um passo indivisível. Na primeira etapa, uma nova mensagem é recebida. Na segunda etapa a novo valor para B é obtido por meio de uma função $fct ()$. A função grava a recepção de uma mensagem numerada b e retorna um valor $X \in \square B$ para o qual pode garantir que todos mensagens com números menores que X foram gravadas por $fct ()$ antes. Poderia conseguir isso, por exemplo, definindo

```

E se
:: (b == B + 1) -> B = b
:: (b! = B + 1) -> pular
fi

```

forçando as mensagens a serem recebidas em sequência, mas também poderia ser mais liberal (ver Capítulo 4).

Supondo que haja r mensagens na fila R e confirmações s em S , com $r \in 0 \text{ es } \infty$

Página 227

%
VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

a seguinte condição é invariável para as confirmações que são armazenadas em buffer em S :

$A \delta \square S[1] \delta \square S[2] \delta \dots \delta \square S[s] \delta \square B$

(1)

A exatidão deste invariante do sistema é comprovada por indução. Primeiro observe que em no estado inicial os canais estão vazios e o invariante se reduz a $A \delta \square B$, que se mantém trivialmente, uma vez que $A = \square B = 0$. Em seguida, observe que se o invariante se mantém em um sistema arbitrário

tem estado que deve manter em todos os seus estados sucessores, uma vez que não pode ser invalidado pelo

quatro operações atômicas:

$S1$ não altera nenhuma das variáveis em (1). $S2$ transforma (1) em

$A = \square S[1] \delta \square S[2] \delta \dots \delta \square S[s] \delta \square B$

que deve ser mantida se (1) for mantida. $R1$, supondo que o reconhecimento não seja perdido, transforma (1) em

$A \delta \square S[1] \delta \square S[2] \delta \dots \delta \square S[s] \delta \square S[s+1] = \square B$

que também deve ser mantida se (1) for realizada antes de $R1$ ser executado. $R2$ pode aumentar, mas nunca diminui o valor de B e, portanto, também não pode invalidar o invariante.

Juntos, isso prova a validade do invariante (1). O próximo invariante se aplica ao r mensagens esperando na fila R :

$R[I] < \square R[J] + \square W$, para $0 \leq i \leq r$ e $i < j \leq r + 1$

(2)

onde, por conveniência, definimos

$R[0] = \square B$ e $R[r+1] = \square A$

No estado inicial, com $r = 0$, a fila está vazia e o invariante torna-se $B < \square A + \square W$

que trivialmente vale para todos $W > 0$, uma vez que $A = \square B = 0$. Devemos verificar novamente que a exatidão do invariante não é afetada pelas quatro operações atômicas.

S_1 pode adicionar um elemento $r \square + \square 1$ à fila R (se a mensagem não for perdida):

$A \delta \square R [r + 1] < \square A + \square W$

(2a)

e então incrementar r . Existem apenas dois casos a considerar onde o invariante agora pode ser violado: $i = \square r$ e $j = \square r$. Para $i = \square r$, estados invariantes (2)

$R [r] < \square R [r + 1] + \square W$

Por definição, isso significa

$R [r] < \square A + \square W$

que (2a) claramente não pode violar. Para $j = \square r$, estados invariantes (2)

$R [i] < \square R [r] + \square W$, para $0 \leq i < \square r$

Uma vez que (2a) garante que $R [r] \in \square A$, após a conclusão de S_1 , isso se reduz a

$R [i] < \square A + \square W$, para $0 \leq i < \square r$

Página 228

%

que deve ser mantida se for mantida antes de S_1 ser executado. S_2 só pode aumentar o valor de A , como resultado direto de (1). R_1 não muda nenhuma das variáveis em (2). R_2 exclui uma mensagem da fila, removendo assim uma das condições do invariante. Ou não tem efeito ou define $R [0] = \square R [1]$, que também não pode perturbar a correção de (2).

Isso completa a prova de invariante (2).

O INVARIANTE DO PROTOCOLO DE JANELA

As invariantes (1) e (2) podem ser usadas para provar uma propriedade mais geral da janela protocolo.

$B - \square W \delta \square R [i] < \square B + \square W$ para $1 \leq i \leq \square r$

(3)

Para provar isso, primeiro observe que pelo invariante (2) temos

$R [i] < \square A + \square W$ para $1 \leq i \leq \square r$

Visto que pela invariante (1) também temos $A \delta \square B$, o lado direito de (3) é facilmente comprovado.

Em segundo lugar, por invariante (2) temos

$B < \square A + \square W$ ou $B - \square W < \square A$

Visto que pela invariante (1) também temos $A \delta \square R [i]$, o lado esquerdo de (3) também é provado.

Invariante (3) implica que o receptor pode deduzir o verdadeiro valor de uma mensagem (ou seja, seu número de sequência) mesmo se apenas parte do valor for transmitido, por exemplo, o valor o módulo 2 W . É uma demonstração elegante de que o protocolo ARQ de repetição seletiva, discutido no Capítulo 4, precisa de um intervalo de números de sequência que é o dobro da janela tamanho W .

DISCUSSÃO DAS PROVAS MANUAIS

A técnica de prova que discutimos foi descrita pela primeira vez por Stein Krogdahl e posteriormente refinado por Donald Knuth. É baseado na noção de invariantes de estado. Ao contrário os métodos usados na maioria dos sistemas de validação automatizados, este método não é baseado em a inspeção de estados de sistema alcançáveis, mas na inspeção de transições de estado.

Normalmente, há muito menos transições de estado do que estados de sistema alcançáveis. O exemplo ilustra isso muito bem: uma vez que os números de sequência são ilimitados, o número de estados do sistema alcançável é infinito, mas o número de transições de estado é restrito a quatro. O esforço necessário para verificar se uma transição não pode invalidar um invariante de sistema alternativo, entretanto, pode ser substancial.

Em um trabalho independente, Mohamed Gouda (ver Notas Bibliográficas) argumentou que todos as provas manuais podem ser construídas em apenas duas noções básicas:

Invariantes do sistema, e

Fórmulas bem fundamentadas

Uma fórmula bem fundamentada pode ser usada, por exemplo, para provar a rescisão ou para construir provas de indução. Para construir tal prova, devemos encontrar uma quantidade que é inevitavelmente diminuída durante a vida útil do programa e que fornece um resultado desejável de

Página 229

o programa quando atinge um mínimo. Para encontrar os invariantes certos e bem fórmulas fundamentadas podem ser difíceis. Em geral, as provas manuais devem ser cuidados estruturadas

totalmente, exigindo que o usuário encontre e prove uma série de invariantes intermediários antes a correção de uma propriedade mais geral pode ser demonstrada. A vantagem de esta abordagem é que força o usuário a compreender completamente os problemas de design lem e a solução sugerida.

Esta vantagem, no entanto, pode se tornar uma desvantagem quando o método é aplicado a problemas maiores. As provas manuais podem ser entediantes e são inevitavelmente suscetíveis suscetível de erro humano, bem como o projeto do protocolo que é o assunto da prova. Para cada invariante que deve ser comprovado, o método pode exigir uma inspeção manual de todos transições de estado atômico dentro do sistema. As técnicas manuais se dividem em casos em que a validação é mais necessária, ou seja, para protocolos maiores. Aceitamos aqui, portanto, que há uma necessidade de ferramentas automáticas para nos ajudar tanto na construção provas, ou em encontrar contra-exemplos para reivindicações de correção (um eufemismo para "depuração"). Afinal, mesmo uma prova não é uma prova, a menos que sua validade possa ser verificado. Para citar Lamport [1977]:

"Uma prova formal é aquela que é suficientemente detalhada, e realizada de forma suficientemente sistema formal preciso, para que possa ser verificado por um computador."

Embora não haja um algoritmo simples que possa automatizar os métodos de prova manual discutimos, há, pelo menos para sistemas de estados finitos, uma alternativa. O alternativo torna-se possível se basearmos nosso método de prova diretamente no sistema alcançável estados, em vez de indiretamente nas transições que os conectam. Métodos deste tipo pode ser usado para validar propriedades de estados e propriedades de sequências de estados, conforme discutido no Capítulo 6. O restante deste capítulo é dedicado a um debate avaliação desses métodos.

11.3 MÉTODOS DE VALIDAÇÃO AUTOMATIZADOS

Vejamos a estrutura geral dos sistemas de validação automatizados com base no alcance análise de bilidade. Inicialmente, consideraremos apenas a validação de propriedades de estado, como como violações de asserção e rescisões impróprias. Discutimos com alguns detalhes o limite itações dos métodos de análise de acessibilidade e as estratégias que foram desenvolvido para explorá-los. Nas seções posteriores, mostramos como o método pode ser estendido para validar propriedades de sequências de estados, como condições de não progresso e reivindicações temporais, conforme discutido no Capítulo 6.

ALGORITMOS DE ANÁLISE DE REACABILIDADE

Um algoritmo de análise de acessibilidade tenta gerar e inspecionar todos os estados de uma distribuição sistema buted que são alcançáveis a partir de um determinado estado inicial. Implicitamente, ele construirá

todas as sequências de execução possíveis, embora, dependendo do tipo de algoritmo usado, nem todas as informações sobre sequências de estados estão necessariamente disponíveis para análise. Lá são três tipos principais de algoritmos de análise de acessibilidade. Na ordem em que eles estão listados aqui, eles podem ser aplicados a sistemas de complexidade crescente:

Pesquisa completa (sistemas até 10⁵ estados)

Página 230

SEÇÃO 11.3

MÉTODOS DE VALIDAÇÃO AUTOMATIZADOS

219

Pesquisa parcial controlada (sistemas de até 10⁸ estados)

Simulação aleatória (sistemas maiores)

A pesquisa completa é o algoritmo mais simples. Ele realiza a análise mais completa de os três tipos de algoritmo, mas só pode analisar a menor classe de protocolos.

Quantificamos as limitações posteriormente neste capítulo. Se o método de pesquisa completo exceder seu limites, é efetivamente reduzido a um método de pesquisa parcial não controlado, e a qualidade da análise se deteriora rapidamente.

A pesquisa parcial controlada tenta otimizar a qualidade da análise de acessibilidade especificamente para os casos em que uma pesquisa completa é inviável. Ele tenta fazer isso selecionando uma fração ótima de todo o espaço de estado que pode ser pesquisado dentro de uma determinada con

limitações de memória e tempo.

As técnicas de simulação aleatória são especificamente destinadas à validação de sistemas de uma complexidade que derrota até mesmo a busca parcial controlada. O espaço de estado do sistema pois esses sistemas podem ser estimados como tão grandes que nenhuma técnica de pesquisa parcial pode fazer uma seleção sensata. A melhor pesquisa possível nesses casos é aleatória, ou aleatório tendencioso, caminhada do espaço de estados.

Existem duas medidas diferentes para expressar as capacidades de uma acessibilidade ferramenta de análise: *cobertura* e *qualidade*. A cobertura da pesquisa é facilmente quantificada como o número de estados do sistema testados dividido pelo número de estados no espaço de estado completo. Uma medida talvez mais apropriada, mas menos facilmente quantificável, é a capacidade de pesquisa para encontrar erros: o número de erros distintos encontrados dividido pelo número total de erros presentes. Na comparação dos três métodos básicos de pesquisa abaixo, usamos ambas as medidas. No Capítulo 13, desenvolvemos um sistema de validação de protocolo automatizado para modelos PROMELA que podem realizar análise de acessibilidade em todos os três modos básicos: simulações aleatórias e pesquisas de espaço de estado totalmente exaustivas ou parciais.

11.3.1 PESQUISA DE ESPAÇO DE ESTADO COMPLETO

O algoritmo de pesquisa padrão completo ou exaustivo explora todos os compostos alcançáveis estados do sistema de um conjunto de máquinas de estados finitos em interação. Com que precisão a interação

ção entre as máquinas é definida é amplamente irrelevante para o projeto da pesquisa algoritmo. O modelo de máquina de estado básico pode ser estendido com mensagem finita filas ou variáveis locais e globais. Conforme discutido no Capítulo 8, essas adições não estender o poder do modelo de máquina de estado finito, desde que sejam definidos sobre um domínio finito.

Uma máquina de estado, neste modelo, é definida por um número finito de estados e transições de estado ções. Cada transição de estado tem duas partes: uma pré-condição e um efeito. O pré-condição é normalmente uma condição booleana no estado da máquina, as filas e as variáveis. A transição está habilitada e pode ser executada apenas se a pré-condição detém. O efeito de uma execução pode alterar o estado do sistema, por exemplo, o estados da máquina local, as filas e as variáveis, e talvez até mesmo o estado de outras máquinas (por exemplo, em um sistema de encontro multi-partidário).

O sistema como um todo é definido pelo composto de todas as máquinas individuais, variáveis,

Página 231

220

VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

e estados de fila, e a combinação de todas as transições de estado locais habilitadas simultaneamente ções. Daqui em diante, o termo *estado* é usado como uma abreviatura para *estado do sistema composto*.

Onde isso pode causar confusão, usamos os termos *estado da máquina* ou *estado do sistema*.

Dado um estado inicial para cada máquina no sistema, os conjuntos de estados da máquina e estados do sistema podem ser divididos em duas classes disjuntas: estados alcançáveis e estados inacessíveis. Normalmente, é necessário que o sistema não contenha nenhum estados de *máquina* capazes : eles corresponderiam ao código não executável em uma implementação ção. Normalmente, também, o conjunto de estados inalcançáveis do *sistema* é de várias ordens de magnitude maior do que o conjunto de estados do sistema alcançáveis. O conjunto de sistema inacessível os estados devem incluir todos os estados de erro.

Uma análise exaustiva de alcançabilidade tenta determinar quais estados são alcançáveis e que não são. Cada estado alcançável e cada sequência de estados alcançáveis podem ser verificado para um determinado conjunto de critérios de correção. Esses critérios podem ser condições gerais

ções que devem ser mantidas para qualquer protocolo, como a ausência de deadlocks ou buffer over- é executado ou podem ser requisitos específicos de protocolo, como uma reivindicação temporal sobre o funcionamento adequado de uma disciplina de retransmissão de mensagens. Em muitos casos, protocolo requisitos específicos podem ser formalizados como invariantes de estado, a exatidão dos quais pode ser verificado com um teste booleano simples em cada estado de sistema alcançável.

No algoritmo abaixo, a análise de alcançabilidade começa com uma pequena rotina chamada `start()` que inicializa dois conjuntos: um conjunto de trabalho de estados do sistema a serem analisados, chamado

`w`, e um conjunto de estados que foram analisados, chamado `um`.
`começar()`

```

{
W = {estado_inicial}; /* conjunto de trabalho: a ser analisado */
A = {};
/* estados analisados anteriormente */
analisar();
}

```

O conjunto A também é conhecido como *espaço de estado do sistema*. Quando o algoritmo termina, deve incluir todos os estados do sistema alcançáveis. A estrutura básica da alcançabilidade do algoritmo de análise é o seguinte.

```

analisar()
/* pesquisa exaustiva ou completa */
{
if (W está vazio) return;
q = último elemento de W;
adicone q a A;
if (q == error_state)
Reportar erro();
outro
{
para cada estado sucessor s de q
if (s não está em A ou W)
{
adicone s a W;
analisar();
}
}
exclua q de W;
}

```

Página 232

SEÇÃO 11.3
MÉTODOS DE VALIDAÇÃO AUTOMATIZADOS
221

A ordem em que os estados são recuperados do conjunto de trabalho W parece irrelevante no início, mas acaba sendo um importante ponto de controle. Se os estados são armazenados no conjunto W no *primeiro a entrar*

ordem de saída (ou seja, pilha), o algoritmo realiza uma pesquisa em profundidade do estado árvore do espaço. Se os estados forem armazenados e removidos na ordem do *primeiro a entrar, primeiro a sair*, isso muda para

uma pesquisa abrangente (o elemento q deve ser excluído após a recuperação do conjunto W neste tipo de algoritmo). Uma pesquisa abrangente tem a vantagem de encontrar o erro mais curto sequências primeiro. Uma pesquisa em profundidade, no entanto, tem a vantagem de exigir um menor conjunto de trabalho W. Uma explicação intuitiva para isso é que o tamanho de W em uma profundidade

pesquisa é uma função da profundidade da árvore de pesquisa, mas uma função de sua largura em um pesquisa em amplitude. A profundidade da árvore de pesquisa depende do comprimento máximo de uma sequência de execução única. A largura da árvore, no entanto, é determinada pelo número máximo de sequências de execução distintas, que geralmente é muito maior número.

Como exemplo, considere um protocolo em que cada estado tem dois sucessores. O Estado o espaço é então equivalente a uma árvore binária em expansão. Após n transições, a amplitude de a árvore de pesquisa tem 2^n estados. A profundidade da árvore, no entanto, é de apenas n estados. Há uma outra vantagem importante na disciplina de pesquisa em profundidade. Quando um erro for descoberto, gostaríamos que o algoritmo fosse capaz de produzir uma execução sequência que leva ao erro por meio de uma sequência válida de transições de estado, a partir do estado inicial do sistema. Com um método de pesquisa em largura, o caminho do

estado do sistema deve ser reconstruído a partir de informações armazenadas no espaço set estado A. Com uma busca em profundidade, no entanto, tal caminho não precisa ser reconstruído: uma sequência é implicitamente definida pela ordem de empilhamento de conjunto W.

DISCUSSÃO DO MÉTODO DE PESQUISA COMPLETA

O principal problema com a estratégia de busca completa é sua aplicabilidade restrita. Isto é importante notar que a cobertura de toda a técnica de pesquisa de espaço de estado não é necessariamente 100%: depende do tamanho do espaço de estado e da quantidade de memória que está disponível para pesquisa. Se o tamanho do espaço de estado é R e o máximo número de estados que podem ser armazenados na memória durante a pesquisa é A, tanto a capacidade e a qualidade da pesquisa só pode chegar a 100% quando $R \leq A$. Quando $R > A$ a cobertura reduz para A/R , mas a *qualidade* da pesquisa provavelmente será pior.

Para protocolos grandes, o algoritmo de busca exaustivo se deteriora em um baixo pesquisa parcial de qualidade.

Considere um protocolo para dois processos, cada um com 100 estados, uma fila de mensagens, e cada um acessando cinco variáveis locais. As duas filas de mensagens são restritas a cinco slots cada, e o intervalo efetivo das variáveis locais é assumido como limitado a dez valores. O número de mensagens distintas trocadas é dez. Neste relativamente pequeno sistema de exemplo, existem 10^5 estados possíveis das variáveis de protocolo. Cada processo pode estar em um de 10^2 estados diferentes, então dois processos podem estar no máximo em 10^4 diferentes estados do sistema composto. Finalmente, cada fila pode conter entre zero e cinco mensagens, onde cada mensagem pode ser uma em dez mensagens possíveis. O total número de estados do sistema no pior caso, então é

Página 233

222
VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

$10^{10} \cdot 10^4 \cdot$

$\begin{array}{l} \rangle \\ | \\ \{ \square_i = \square_0 \end{array}$

(C)

$\begin{array}{l} ^5 \\ 10_i \\ \} \\ | \\ J \\ ^2 \end{array}$

ou na ordem de 10^{24} estados diferentes. Se assumirmos, de forma bastante irreal, que cada estado pode ser codificado em 1 byte de memória e pode ser analisado em 10 μ s segundos de CPU tempo, ainda precisaríamos de uma máquina com pelo menos 10^{15} vezes mais memória do que atualmente disponível, e precisaria de aproximadamente 10^{11} anos de tempo de CPU para realizar um análise exaustiva.

Felizmente, o número de estados efetivamente alcançáveis é geralmente muito menor do que o número total de estados do sistema calculado acima. Afinal, é o propósito de um protocolo para restringir o comportamento dos processos de protocolo e, portanto, o número de estados efetivamente alcançáveis, a fim de realizar a funcionalidade desejada. Mesmo assim sistemas de protocolo relativamente pequenos podem gerar facilmente de 10^5 a 10^9 alcance estados do sistema capazes. O número de estados cresce dramaticamente se, por exemplo, o tamanho de uma fila de mensagens é aumentada, ou se as suposições sobre o comportamento do "ambiente" em que o protocolo é executado (por exemplo, as características do canal) estão relaxados.

O método de pesquisa exaustiva inevitavelmente falha quando o espaço de estado cresce além de aproximadamente 10^5 estados. Um cálculo rápido do "verso do envelope" pode ilustrar isso.

Se um estado do sistema pode ser armazenado em S bytes de memória, e temos uma máquina com M bytes disponíveis, podemos gerar e analisar não mais do que estados M/S . M é um constante dependente da máquina que normalmente está na faixa de 10^6 a 10^7 . Valores para S estão normalmente no intervalo de 10^1 a 10^2 bytes, com valores maiores correspondendo ao maior número de estados alcançáveis. Isso leva a uma estimativa do estado máximo tamanho do espaço de cerca de 10^5 estados. Este valor também pode ser encontrado experimentalmente executando o algoritmo de pesquisa completo até esgotar a memória disponível.

Isso significa que, em muitos casos, o método de pesquisa completa só é viável se pudermos reduzir a complexidade de nossos modelos de validação ao máximo que uma determinada máquina pode analisar. A complexidade dos modelos de protocolo pode ser reduzida substancialmente pela estrutura técnicas de aplicação e estratificação, mas em alguns casos, mesmo após tais reduções, a sondagem itens a serem analisados permanecem inherentemente complexos e não podem ser mais reduzidos sem perdendo recursos essenciais.

Como um exemplo, considere o protocolo de janela descrito no Capítulo 7. É um simples protocolo, sem nenhuma simplificação adicional óbvia. Em sua forma básica, este protocolo é bom dentro do intervalo do método de pesquisa completo. Conforme ilustrado no Capítulo 14, no entanto, o

a complexidade da pesquisa aumenta dramaticamente se as suposições sobre o canal o comportamento é relaxado e pode impossibilitar uma busca completa.

11.3.2 PESQUISA PARCIAL CONTROLADA

Se o espaço de estado for maior do que a memória disponível pode acomodar, o esgotamento estratégico de busca ativa discutida acima efetivamente se reduz a uma busca parcial, sem

Página 234

SEÇÃO 11.3
MÉTODOS DE VALIDAÇÃO AUTOMATIZADOS
223

garantindo que as partes mais importantes do protocolo sejam inspecionadas. Este observador vação levou ao desenvolvimento de uma nova classe de algoritmos que tentam especificamente explorar os benefícios de uma pesquisa parcial. Eles são baseados na premissa de que na maioria casos de interesse prático, o número máximo de estados que podem ser analisados, A , é apenas uma fração do número total de estados alcançável R . Uma pesquisa parcial controlada, então, tem os seguintes objetivos:

Para analisar precisamente os estados A , com $A = \square M / S$

Para selecionar esses estados A do conjunto completo de estados R alcançáveis em tal maneira que todas as principais funções do protocolo são testadas

Selecionar os estados A de modo que a *qualidade* da pesquisa, ou seja, a probabilidade de encontrar qualquer erro, é melhor do que a *cobertura* A / R

Um algoritmo para a busca parcial se parece exatamente com o algoritmo anterior para um busca exaustiva, com apenas uma diferença: nem todos os estados sucessores são analisados.

```
analisar()
/* pesquisa parcial */
{
    if (W está vazio) return;
    q = último elemento de W;
    adicione q a A;
    if (q == error_state)
        Reportar erro();
    outro
    {
        para alguns estados sucessores de q
        if (s não está em A ou W)
        {
            adicione s a W;
            analisar();
            /* recursivo */
        }
    }
    exclua q de W;
}
```

É interessante notar que mesmo uma seleção aleatória de estados sucessores é superior a uma busca parcial não controlada, uma vez que garante que o espaço de estado completo é sampled, em vez do fragmento desconhecido que por acaso é gerado primeiro em uma pesquisa. A seleção também pode ser baseada em uma heurística que favorece as execuções que são provavelmente revelarão erros de design rapidamente. Muitas maneiras diferentes de organizar um ambiente controlado

pesquisa parcial foram estudados. Eles incluem métodos baseados em:

Limits de profundidade

Pesquisas dispersas

Pesquisas guiadas

Pesquisas probabilísticas

Pedidos parciais

Seleções aleatórias

Discutimos os primeiros cinco métodos brevemente a seguir. O último método, baseado em seleções, é desenvolvido no restante deste capítulo. Referências a mais detalhadas as descrições de todas as técnicas estão incluídas nas notas bibliográficas.

Página 235

224
VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

DEPTH-BOUNDS

Uma técnica de pesquisa parcial bastante padrão e simples é a colocação de um limite

o comprimento das sequências de execução que são analisadas. Limita a pesquisa a um útil subconjunto de comportamentos, excluindo, por exemplo, casos degenerados de múltiplas sobreposições execuções. No algoritmo de pesquisa completo, por exemplo, permite-nos restringir o máximo imo tamanho de conjunto de trabalho W .

PESQUISA DE DISPERSÃO

Em uma pesquisa dispersa, as execuções são selecionadas que levam mais perto de estados de conflito em potencial.

Um dos requisitos de um estado de deadlock, por exemplo, é que não há mensagens pendente (todos os canais estão vazios). O algoritmo, portanto, favorece as operações de recebimento sobre as operações de envio. O objetivo do método é aumentar a probabilidade de encontrar erros rápido.

PESQUISA GUIADA

Em uma pesquisa guiada, o critério de seleção de estado é uma função de custo que é dinamicamente avaliada para cada estado sucessor. A função de custo pode ser fixa, como em uma dispersão pesquisa ou pode ser alterado dinamicamente durante a pesquisa. Não se sabe muito sobre os tipos de funções de custo, ou "expressões de orientação", que podem ser úteis.

PESQUISA PROBABILÍSTICA

Em uma pesquisa probabilística, os estados sucessores são explorados em ordem decrescente de seus capacidade de ocorrência. Todas as transições no sistema são rotuladas, no mínimo com uma etiqueta que dá a eles uma probabilidade "alta" ou "baixa" de ocorrência, e essas marcas são utilizado como critério de seleção.

ORDENS PARCIAIS

O principal fator responsável pelo problema da explosão do espaço de estados é o grande número de intercalações possíveis de eventos simultâneos. Conforme mostrado no Capítulo 5 (página 96), nem todas as intercalações são necessariamente relevantes na busca de estados de erro.

Existem várias maneiras de explorar pedidos parciais. Um primeiro método é baseado na definição de uma heurística para qualquer

Exploração de estado de progresso razoável ou

Exploração de estado de progresso máximo

Ambas as heurísticas funcionam atribuindo uma prioridade de pesquisa aos processos de protocolo. o número de transições que são inspecionadas durante a pesquisa é limitado, com preferência dado às transições que pertencem a processos de alta prioridade. Transições na parte inferior os processos de prioridade são considerados apenas se todos os processos de prioridade mais alta forem bloqueados. No

uma técnica de exploração de progresso *justo*, a prioridade relativa de um processo é *diminuída* quando uma de suas transições é executada durante a busca; no progresso *máximo* técnica de exploração a prioridade relativa é *aumentada*.

Um segundo método mais recente para explorar pedidos parciais é baseado em definições formais de relações de equivalência no comportamento do sistema. O objetivo é podar essa parte de uma pesquisa que pode ser *comprovada* a ser irrelevante. (Referências a estes e outros

técnicas são coletadas nas notas bibliográficas.)

SELEÇÕES ALEATÓRIAS

Em uma pesquisa parcial controlada com base em seleções aleatórias de estados sucessores, nenhum esforço

é feito para prever onde os erros prováveis no espaço de estado serão encontrados. Nós vamos argumentar abaixo que esta não é apenas a técnica mais simples de implementar, mas também é provavelmente produzirá a pesquisa da mais alta qualidade. É a única técnica que pode satisfazer todos os três requisitos para uma pesquisa parcial controlada que foram listados no início deste seção.

DISCUSSÃO DE MÉTODOS DE PESQUISA PARCIAL CONTROLADA

As primeiras quatro técnicas para controlar a pesquisa parcial que discutimos acima têm um problema principal em comum. Todos os quatro métodos tentam prever onde estão os erros em um protocolo pode ser encontrado. Esta é uma abordagem inherentemente arriscada. Como um corolário de

Lei de Murphy, os erros tendem a se esconder onde um designer ou validador decidiu não olhar. Ao lado da seleção aleatória de estados sucessores, as técnicas baseadas em

pedidos parciais podem, em princípio, evitar esse problema. As dependências entre processos, no entanto, podem ser sutis. Considere, por exemplo, um sistema de três processos A , B , e C , onde A e B interagem com C , mas não com o outro. Seria tentador concluir que, uma vez que A e B são disjuntos, todas as intercalações possíveis de seus os comportamentos são necessariamente equivalentes. Mas, infelizmente, essa suposição é inválida. Observe que o comportamento do processo A pode depender de B ‘s comportamento indiretamente através de sua mútua interação com C . Cada intercalação distinta das ações de A e B pode ser significativo na determinação do resultado.

Para determinar mecanicamente, portanto, quais intercalações particulares podem ser seguramente ignorado em pesquisas de espaço de estado pode não ser trivial. Uma avaliação precisa pode muito bem ser mais caro do que uma busca exaustiva completa e, portanto, ser autodestrutivo como um técnica de otimização.

Um problema final com os primeiros cinco métodos é que, embora possam reduzir o tamanho do o espaço de estado, nenhum desses métodos fornece uma ferramenta para combinar o tamanho do espaço de estado para o tamanho da memória disponível. Para todos esses métodos, o tamanho da fratura ção do espaço de estado que é efetivamente pesquisado só pode ser determinada experimentalmente contagem e é dependente do protocolo. Isso significa que podemos ter que realizar muitas validações, com diferentes critérios de seleção, antes de podermos encontrar aquele que analisa precisamente os estados M / S . Na Seção 11.4, desenvolveremos a ideia da aleatoriedade seleção de estados sucessores e mostrar que pode ser usado para resolver efetivamente também este problema.

Antes de fazer isso, discutimos um método de exploração do espaço de estado final, também com base no seleção aleatória de estados sucessores, mas desta vez sem qualquer tentativa de construir um estado espaço

Página 237

226
VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

11.3.3 SIMULAÇÃO RANDOM

Os métodos de busca parcial controlada têm ampla aplicabilidade. Existem aplicativos, no entanto, onde também uma busca parcial controlada se torna inviável. Pode-se tentar, por exemplo, para aplicar um algoritmo de validação de protocolo diretamente a altamente detalhado, código empilhado que é executado em uma máquina. Nesses casos, o parâmetro S , medindo o tamanho de um único estado de sistema composto em bytes, pode variar de 10^3 a 10^5 bytes de memória.

Mesmo em uma máquina maior, o maior número de estados que podem ser mantidos por um a pesquisa parcial cai para algumas centenas de estados do sistema, na melhor das hipóteses, em um espaço de estado que é muitas ordens de magnitude maior. Em casos como esses, a única abordagem sensata é descartar os conjuntos A e W do algoritmo de busca e explorar o espaço de estados com um simulação aleatória ou 'passeio aleatório'. O algoritmo é o seguinte.

```
analisar()
/* simulação aleatória */
{
    q = estado inicial;
    enquanto (1)
    /* para sempre */
    {
        if (q é error_state)
        {
            Reportar erro();
            q = estado inicial;
        } outro
        q = um estado sucessor de q;
    }
}
```

Esta técnica funciona amplamente independente do tamanho e complexidade do sistema sendo modelado; até mesmo sistemas de “tamanho infinito” podem ser explorados dessa maneira. o cobertura do método, é claro, não pode ser medida, embora, em princípio, um cobertura exaustiva de espaços de estados finitos é garantida, desde uma quantidade suficiente de Tempo. Na prática, esta não é uma diretriz muito útil, uma vez que uma quantidade "suficiente" de

o tempo pode facilmente significar um século de tempo de computação ou pior. Em experimentos, como-nunca, Colin West foi capaz de mostrar que, mesmo para uma *capa* de pesquisa incomensuravelmente pequena, *idade, qualidade* ou capacidade de detecção de erros da pesquisa pode ser adequada.

O restante deste capítulo é dedicado ao desenvolvimento e motivação de uma técnica de pesquisa parcial controlada que foi denominada *supertrace*. Uma implementação em C de um algoritmo de busca exaustivo para PROMELA, com uma opção de supertrace para grandes problemas, é discutido nos Capítulos 12 a 13.

11.4 O ALGORITMO SUPERTRACE

Dados M bytes de memória, como podemos organizar uma pesquisa de espaço de estado para usar precisamente

M bytes, nem mais nem menos, e realizar a maior pesquisa possível dentro desse arena? Para responder a essa pergunta, olhamos com um pouco mais de detalhes no armazenamento de memória métodos que são tradicionalmente usados. A maneira padrão de manter o conjunto de espaço de estado *Tanto* em um algoritmo de pesquisa completo quanto parcial é usar uma técnica de armazenamento chamada

hashing. Hashing nos permite determinar rapidamente se ou não um novo estado s é já é membro do conjunto A e pode ser descartado ou ainda não está em A e precisa ser inserido. O método é usar o conteúdo de s para calcular um valor hash $h(s)$, que

Página 238

SEÇÃO 11.4
O ALGORITMO SUPERTRACE
227

é usado como um índice em uma tabela de consulta de estados. A tabela é organizada conforme mostrado em

Figura 11.1.

Lista Ligada

Tabela de pesquisa

estado: s

$h(s)$

$H - 1$

0

Figura 11.1 - Consulta de tabela de hash

Suponha que temos slots H na tabela de hash. A função hash $h(s)$ é definida como que ele retorna um valor arbitrário no intervalo $0 \dots (H - 1)$. Para o mesmo estado $s \in A$, $h(s)$ sempre retorna o mesmo valor. Mas também existe a possibilidade de que dois estados diferentes produz o mesmo valor de hash. No caso que estamos estudando, a tabela hash terá que acomodar um grande número de estados, o que significa *um > H*. A função hash irá em seguida, produza o mesmo valor hash para uma média de diferentes estados A / H . Todos os Estados que hash para o mesmo valor são armazenados em uma lista vinculada que é acessível através da pesquisa tabela sob o índice calculado (o valor hash). Em média, então, quando a mesa está cheio, cada novo estado deve ser comparado a outros estados A / H antes de ser inserido na lista vinculada ou descartado como redundante. Quando A cresce além do primeiro H estados, o número de comparações necessárias cresce continuamente, e a eficiência da pesquisa degrada: há uma penalidade de tempo para a análise de sistemas com mais de H estados.

Um valor típico para H é 10^4 slots. A própria mesa ocupa $H \cdot B$ bytes de memória, mais B bytes para cada estado inserido, onde B é o tamanho de um ponteiro de endereço.

Na maioria das máquinas $B = 4$, o que significa que uma mesa com, digamos, 256.000 slots requer mais de 1 Mbyte de sobrecarga que não pode mais ser usado para armazenar estados. Para acompanhar modifique o maior espaço de estado possível, portanto, um pequeno valor para H é necessário. Como mostrado acima, no entanto, um pequeno valor para H significa uma baixa eficiência de pesquisa. Se pudéssemos de alguma forma conseguir usar um valor muito grande para H , o número de hash conflitos podem ser minimizados e, portanto, a velocidade do algoritmo de pesquisa pode ser otimizado. Vamos supor que podemos usar o algoritmo de pesquisa completo com um valor para H em a ordem de 10^8 slots. Em um espaço de estado de até 10^5 estados, podemos esperar ter menos

Página 239

de $10^5 / 10^8$ conflitos de hash, ou menos de um conflito em mil estados. Isso significa que raramente haverá mais de um único estado na lista vinculada que está conectado a cada slot na tabela hash. Mas isso significa que não temos que armazenar o estado completo descrições na tabela hash: em todos, exceto alguns casos, o índice da tabela hash (o valor) identifica exclusivamente um estado. A única contabilidade necessária é lembrar se um slot na tabela hash está preenchido ou não. Um único bit de armazenamento por estado será suficiente. E se

temos M bytes de memória disponível, temos $8M$ bits para o espaço de estado (assumindo 8 bits por byte). Uma máquina de 10 Mbyte pode nos dar um espaço de estado grande o suficiente para detém 80 milhões de estados. A função hash $h(s)$ é usada para calcular a posição de um bit na arena memória disponível M . Um valor de bit de 1 agora indicará que o estado correspondente a este valor hash foi analisado anteriormente. O próprio estado não é armazenado.

Como nenhum estado é armazenado, não há estados para comparar um novo estado: o bit posição identifica exclusivamente o estado. O método pode funcionar bem se o o espaço de estado é esparsa e de fato H é muito grande. Um grande valor de H cria hash conflitos raros para todos os casos em que $A < H$. Mais importante, porém, quando $A > H$ o hashing define automaticamente um método de pesquisa parcial aleatório que corresponde ao cobertura da pesquisa para a memória disponível. O método, portanto, se aproxima uma busca exaustiva por protocolos menores e lentamente muda para um controle parcial método de pesquisa para protocolos maiores. Para protocolos menores, no entanto, não precisamos de um método de pesquisa parcial: podemos usar uma técnica tradicional de pesquisa exaustiva.

Supertrace é uma técnica de pesquisa parcial controlada que se destina apenas ao validação de sistemas de protocolo que não podem ser analisados exaustivamente.

Como uma técnica de busca exaustiva, o algoritmo supertrace compararia habilmente com quase qualquer outro método padrão de pesquisa em profundidade, simplesmente porque pode não garantir 100% de cobertura devido à possibilidade de conflitos hash não resolvidos (cf. Tabelas 13.1 e 13.2 no Capítulo 13). Mostraremos, no entanto, que como uma busca parcial técnica, o novo algoritmo é superior a outros métodos.

CONFLITOS DE HASH

A sobrecarga da tabela de pesquisa com um algoritmo de supertrace reduz de

$HB + (S + \square B)A$

bytes para

$H/8$

bytes. No entanto, uma vez que os estados não são mais armazenados, não podemos mais compensar para conflitos de hash. Notavelmente, este defeito tem um efeito positivo na sobrecarga comportamento do algoritmo durante pesquisas parciais. É assim que funciona.

Se um novo estado s é gerado e é verificado que o sinalizador está definido no índice $h(s)$, devemos concluir que o estado s foi analisada antes e deve ser ignorado. Quando um hash conflito ocorre, a conclusão acima está errada e a pesquisa irá ignorar um estado que

deveria ter sido analisado: a pesquisa é truncada. Como $A/H \square 0$, o número de hash conflitos que serão encontrados se aproximam de zero, e o método se aproxima (mas nunca pode garantir ser) uma pesquisa totalmente exaustiva. Na verdade, portanto, é melhor escolha H o maior possível.

O valor máximo para H que pode escolher para um dado tamanho de memória M é $H = 8M$. Vamos ver como esse algoritmo se compara a uma pesquisa parcial tradicional.

Os requisitos de memória são os mesmos. O limite para a cobertura do tradicional procurar, no entanto, é $um = \square M/S$. Armazenar os mesmos estados M/S na tabela hash do algoritmo modificado, com $H = 8M$, dá uma razão $A/H = \square M/(8MS) = 1/(8S)$.

Para um valor típico de $S \sim 100$, a probabilidade de um conflito de hash, em seguida, se aproxima de 10

Mas o novo algoritmo não está restrito a um máximo de estados M/S . Ele pode analisar um máximo de H estados distintos. Os conflitos de hash, que aumentam conforme o espaço de estado preenche, agora trabalhe para espalhar os estados que são selecionados para análise em todo o conjunto de estados alcançáveis de maneira aproximadamente aleatória.

Existem dois casos a serem considerados. Para $R < M/S$, a cobertura do algoritmo tradicional ritmo será o mesmo ou ligeiramente melhor do que o novo algoritmo, uma vez que evita o efeito dos conflitos de hash. No entanto, quando $R < M/S$ não precisamos de uma pesquisa parcial algoritmo, uma vez que ainda podemos realizar uma pesquisa exaustiva (tradicional) em memória. O algoritmo supertrace não deve ser usado nesses casos.

Para problemas com $R > M/S$, a cobertura do novo algoritmo, ou seja, o número total de estados efetivamente analisados em comparação com o número total de estados alcançáveis, é substancialmente maior do que a cobertura do algoritmo tradicional. Para $R > M$ it aproxima-se de $8M/R$, em comparação com $M/(SR)$ para o algoritmo tradicional (ver também a Figura 11.2).

Se a descrição de estado S se tornar maior, o algoritmo tradicional pode analisar menos e menos estados, mas o desempenho do novo algoritmo permanece o mesmo. Se, para M fixo em 10⁷ bytes de memória, S cresce de 100 a 1000 bytes por estado, a cobertura de um o algoritmo de busca parcial tradicional cai de 10⁵ para 10⁴ estados analisáveis. O coverage do novo algoritmo, no entanto, permanece constante em um máximo de $H = 8,10^7$ estados analisáveis.

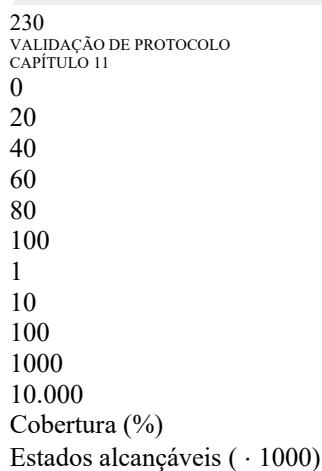
O efeito é ilustrado, para um tamanho fixo S , na Figura 11.2. Aumentar S é equivalente a movendo a linha pontilhada e a linha tracejada para a esquerda: o comportamento do algoritmo tradicional

o ritmo muda, mas o comportamento do algoritmo do supertrace permanece constante.

Para espaços de estado maiores do que um algoritmo de pesquisa exaustivo pode acomodar, o método tradicional quebra muito rapidamente, sua cobertura caindo por um fator de dez para cada aumento de dez vezes no número de estados alcançáveis. A cobertura do novo algoritmo é substancialmente melhor.

Quando $A \ll R$, A é da mesma ordem de magnitude que H , o que significa que uma grande fração

Página 241



.....
Supertrace

Parcial

Pesquisa Completa

100% de cobertura

Figura 11.2 - Comparação de dois algoritmos

do espaço de estado ainda pode ser analisado, os conflitos de hash atuando como uma poda aleatória que dispersa a pesquisa sobre o espaço de estado superdimensionado. Para protocolos ainda maiores com $Uma > \square H$ a cobertura da pesquisa aproxima H/A , ou $8H/R$.

MÚLTIPLOS HASHINGS

As funções hash nos ajudam a fazer uma seleção rápida e aleatória de estados a partir de um grande espaço de estado, e assim implementa uma busca parcial controlada eficiente. Suponha que hipotéticos 10 Mbytes de memória disponíveis para a pesquisa e um espaço de estado de 800 milhões de estados de 100 bytes cada. A cobertura de todos os métodos de pesquisa tradicionais, excepto supertrace e simulação aleatória, é limitado para a análise de $10^7/100$ em $8,10^8$ estados ou 0,0125%. Uma única execução do supertrace daria uma cobertura máxima

idade de $8,10^7/8,10^8$ ou 10%. A questão é: Será que algum dia conseguiremos uma cobertura ainda melhor

idade com as mesmas restrições do sistema? Surpreendentemente, a resposta para o supertrace algoritmo é: Sim.

A função hash pode ser usada como parâmetro em pesquisas repetidas. Suponha que o primeiro pesquisa com função hash H_1 selecionados 80 milhões de estados são aleatórios dos 800 milhões leões alcançáveis. Uma segunda pesquisa com uma função hash diferente H_2 também selecione 80 milhões de estados, mas fará uma seleção diferente. Podemos esperar que haverá uma sobreposição de 10% entre os dois conjuntos de estados, mas a cobertura combinada de as duas pesquisas agora subiram para $80 + 72$ milhões de estados dos 800 milhões de didatos, ou 19%. Continuando este processo, podemos, em teoria, chegar arbitrariamente perto de um cobertura de 100% do espaço estadual, desde que um número suficiente de independentes funções hash podem ser encontradas.

O validador desenvolvido no Capítulo 13 usa este princípio para aumentar a cobertura de pesquisas. Ele usa duas funções hash em cada execução.

Página 242

SEÇÃO 11.5
DETECÇÃO DE CICLOS DE NÃO PROGRESSO
231

11.5 DETECÇÃO DE CICLOS DE NÃO PROGRESSO

Até agora, discutimos apenas a validação de propriedades de estado, usando um método simples algoritmo de análise de acessibilidade de ala. A complexidade do algoritmo, mesmo para este caso simples, está em PSPACE. Portanto, fizemos um esforço deliberado para encontrar o implementação mais rápida e econômica, de modo que a gama de problemas aos quais podemos aplicá-la seja

tão grande quanto possível. Mas não terminamos. Existem outras propriedades que podemos ser interessado em provar, especificamente para modelos de validação PROMELA. Se, como nós temos argumentado acima, a eficiência de uma análise de acessibilidade direta é uma preocupação, o a eficiência dos tipos mais sutis de validação é crucial.

Uma verificação direta de condições de não progresso pode ser baseada na construção ção e inspeção de todos os componentes fortemente conectados no gráfico de acessibilidade que é implicitamente definido pelo espaço de estado do sistema que está sendo analisado. Isto abordagem, embora comumente usada, falha quando o espaço de estado é muito grande para ser armazenado

completamente. Aqui, exploramos uma opção diferente que tem um custo modesto e, a maioria importante, que pode ser usado em combinação com um algoritmo de supertrace para fazer validações de sistemas muito grandes.

Nosso primeiro problema é detectar ciclos no gráfico de alcançabilidade que não passam quaisquer estados marcados como estados de progresso. O algoritmo que desenvolvemos é apenas para identificar

ciclos de não progresso. Não vamos tentar combiná-lo com uma busca simultânea por violações de afirmações e rescisões indevidas. Uma primeira tentativa de encontrar o não ciclos de progresso é realizar uma análise padrão de alcançabilidade em profundidade, onde todos

as sequências são truncadas quando um estado de progresso é alcançado. Ou seja, os estados de progresso são

tratados como se não tivessem sucessores. Todos os ciclos que podem ser construídos em busca de este tipo, devem ser ciclos sem progresso. O tamanho do espaço de estado que é criado em esta pesquisa é no máximo igual ao tamanho de uma pesquisa direta em profundidade. É provável que seja

menor devido aos truncamentos nos estados de progresso.

Para ver como isso pode ser implementado, consulte o algoritmo para o espaço de estado completo pesquisa fornecida na Seção 11.3.1. Um ciclo é detectado se a pesquisa em profundidade atinge um estado que já está no conjunto de trabalho W , assumindo que os estados são extraídos do conjunto W no último na ordem de primeiro a sair.

A falha deste método é que ele não nos permite detectar ciclos que não passam através do estado inicial do sistema. Pode muito bem haver uma sequência de execução cíclica (como definido no Capítulo 6) que primeiro passa por um número finito de estados, alguns dos quais podem ser marcados como estados de progresso, antes de entrar em um ciclo estritamente sem progresso estados. Esta observação, no entanto, leva imediatamente a um novo algoritmo que faz trabalhos.

Um ciclo sem progresso pode começar em qualquer estado do sistema alcançável. Portanto, devemos inspecionar

dois espaços de estado distintos: um criado pela pesquisa em profundidade original e outro que é criado quando as transições dos estados de progresso são desabilitadas. A tarefa da nossa pesquisa algoritmo é inspecionar todos os prefixos possíveis de uma sequência cíclica no estado original espaço e veja se ele pode continuar em um ciclo no segundo espaço de estado. O implemento a mentação é simples. Podemos adicionar um demônio de dois estados ao nosso modelo de validação que

Página 243

232

VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

define em qual modo a pesquisa irá operar, como segue

```
proctype demon () {bit magic = 0; mágica = 1}
```

O estado inicial do processo demoníaco é logo antes da atribuição, com magia variável igual a zero. O segundo e último estado do demônio é imediatamente após o atribuição, com magia igual a um. O processo demon pode mudar do inicial estado para o estado final não deterministicamente, e uma vez que mudou, não pode voltar.

O valor da variável `magic` define em qual modo a pesquisa é realizada. Quando `mágica` é zero, uma pesquisa normal em profundidade é executada, sem qualquer verificação de erro.

Quando a `magia` é uma, todas as transições que se originam nos estados de progresso são desativadas. Todos

as sequências de execução subsequentes devem terminar. Se houver algum ciclo de estados que são alcançáveis enquanto a `magia` é uma, deve ser um ciclo sem progresso.

O valor da `magia` só pode mudar uma vez em qualquer sequência de execução, e pode apenas mudar de zero para um. Vamos supor que, depois que a `magia` mudou de valor, um ciclo de estados é detectado que não é um ciclo de não progresso. Por definição, esse ciclo contém pelo menos uma transição originada em um estado de progresso. Esta transição só pode ocorrer quando a `magia` é igual a zero. Isso significa que o valor da variável `mágica` muda de zero para um e volta pelo menos uma vez a cada vez durante o ciclo. Isto contradiz a observação anterior de que a `magia` só muda de valor uma vez.

O algoritmo que construímos posteriormente tem a propriedade de que, se houver nenhum progresso ciclo existe, pelo menos um será detectado. Para provar isso, vamos supor que existe um componente fortemente conectado alcançável que contém apenas não progresso estados. (Um componente fortemente conectado é qualquer conjunto de estados em que cada membro pode alcançar todos os outros membros do conjunto por meio de uma ou mais transições.)

O algoritmo gera duas cópias de cada estado alcançável; há uma cópia na qual `magia` é igual a zero, e uma cópia com `magia` igual a um. Há uma transição da primeira cópia para a segunda que corresponde à única transição que o demônio processo pode fazer. Considere o caso em que nenhum estado do fortemente conectado componente foi gerado com `magia` igual a um. Considere o primeiro tal estado que é gerado. Uma vez que o componente fortemente conectado é considerado acessível, e a transição do processo demon é sempre executável, isso deve acontecer em algum ponto na pesquisa. Chame esse estado de estado de *semente*. A árvore de pesquisa em profundidade que é enraizado na semente tem todos os estados do componente fortemente conectado como sucessores, incluindo a si mesmo (pela definição de um componente fortemente conectado). Desde a semente estado também é alcançável a partir de si mesmo (pela mesma definição) por meio de estados de não progresso apenas

(por nossa suposição original), ele deve ser revisitado. No momento em que a semente é revisitada, um ciclo é detectado. Pela nossa prova anterior, esse ciclo deve ser um ciclo sem progresso.

Pode haver, é claro, muitos caminhos diferentes por meio de uma comunicação fortemente conectada ponente, cada um dos quais pode representar um tipo diferente de ciclo de não progresso. o algoritmo acima não garante que todas as variantes sejam detectáveis em um único execução da pesquisa. Isso garante que pelo menos uma variante seja detectada. Se não ciclos de progresso são detectados, portanto, podemos ter certeza de que nenhum existe.

A Figura 11.3 ilustra como um caso difícil de um ciclo de não progresso é detectado. o os círculos representam os estados do sistema e as setas representam as transições. Os estados são

Página 244

SECÃO 11.5 DETECÇÃO DE CICLOS DE NÃO PROGRESSO

233

1

2

3

P

4

5....

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.....

6

7....

(uma)

1

2

3

P

6

7....

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

4

5

(b)

Figura 11.3 - Detecção de um Ciclo de Não Progresso

numerados na ordem em que são visitados durante uma pesquisa. O estado marcado com P é um estado de progresso. O estado 1 é o estado inicial do sistema. A linha tracejada do estado 1 para o estado 2 é uma sequência de execução arbitrária. As linhas pontilhadas indicam correspondências de estado.

Lembre-se de que, após a criação de cada estado, nosso algoritmo de pesquisa em profundidade verifica se o estado foi criado antes. Se uma correspondência for encontrada, a pesquisa é truncated. Se a correspondência ocorrer no conjunto de trabalho W , ou seja, na pilha, um ciclo é detectado. Na Figura 11.3a, um fragmento do espaço de estado é mostrado como seria criado em um pesquisa em profundidade inicial. Na Figura 11.3b os mesmos estados são mostrados após a transição do processo demoníaco para o estado em que as transições começando nos estados de progresso são Desativado. Os números indicam a ordem em que os estados são visitados.

Antes da transição do processo demoníaco, na Figura 11.3a, apenas um ciclo é detectado pelo método normal de pesquisa em profundidade. É detectado quando o quinto estado visitado é encontrado para corresponder ao segundo estado, que está na pilha. O loop é benigno, uma vez que contém o estado de progresso. A pesquisa continua, após a remoção dos estados 4, 3 e 2 da pilha, com o novo estado 6. O sétimo estado visitado corresponde ao quarto, e a pesquisa é concluída. A última partida não produz um ciclo, porque estado 4 não está mais na pilha.

O loop de não progresso através dos estados marcados 2, 6, 7, 4 e 5 permanece não detectado na pesquisa de profundidade padrão. Após a transição do demônio progresso, todas as transições do estado marcado com P são desativadas. Isso significa que os estados agora são visitados na ordem indicada na Figura 11.3b. O primeiro ciclo, inofensivo, pode agora não pode mais ser construído, mas o segundo ciclo pode, e é detectado corretamente. Com a adição de um processo demônio simples de dois estados, o algoritmo é trivial para implemento. Uma implementação em C é dada no Apêndice E. Seu custo é duplo compreensão dos requisitos de tempo e espaço. Talvez a vantagem mais importante de

Página 245

234

VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

o algoritmo, no entanto, é que pode ser usado em combinação com qualquer par controlado método de pesquisa inicial. Especificamente, pode ser usado com uma técnica de espaço de estado de bits, como

usado no algoritmo supertrace.

11.6 DETECÇÃO DOS CICLOS DE ACEITAÇÃO

A detecção de ciclos de aceitação (ver Capítulo 6, página 118) é substancialmente mais difícil do que a detecção de ciclos de não progresso, discutida na última seção. Desta vez, todos os ciclos de execução que passam por pelo menos um estado de aceitação devem ser detectados. Nós estamos interessados em encontrar um algoritmo que continue a trabalhar com supertrace, para que sua aplicação a problemas muito grandes não está excluída.

O seguinte algoritmo é devido a Mihalis Yannakakis (veja as Notas Bibliográficas).

O custo do algoritmo é, na pior das hipóteses, uma duplicação do tempo e espaço necessários mentos da pesquisa básica. Conduzimos uma pesquisa em profundidade com dois espaços de estado em vez de uma (ou seja, duas cópias do conjunto A). Chamar o segundo espaço de estado C . Quando não

estado de aceitação é encontrado, o conjunto C permanece sem uso e a pesquisa é precisamente o o mesmo de antes. Para cada estado de aceitação que é removido do conjunto de trabalho W e adicionado para definir A (ou seja, após todos os seus estados sucessores terem sido visitados), o algoritmo muda define A e C e começa uma nova pesquisa. Chame o estado de aceitação de *semente* desse pesquisa. Se a qualquer momento durante esta pesquisa o estado da semente puder ser revisitado, uma aceitação

ciclo é encontrado, e um erro pode ser declarado (ou seja, a reivindicação temporal é satisfeita, o que significa que um comportamento indesejável é possível). Quando nenhum erro for encontrado, o segunda pesquisa termina quando todos os sucessores da semente ter sido adicionado ao conjunto C . Em neste ponto, os conjuntos A e C são trocados novamente, e a pesquisa em profundidade continua enquanto antes.

Nenhum estado será visitado mais de duas vezes esta pesquisa, uma vez no conjunto A e uma vez no conjunto C .

Não é difícil nos convencermos de que qualquer ciclo encontrado por este algoritmo é necessário necessariamente um ciclo de aceitação. É mais difícil mostrar, no entanto, que na ausência de hash colisões, qualquer ciclo de aceitação existente também é encontrado.

Suponha que existam estados de aceitação que pertencem a um ou mais fortemente conectados componentes no gráfico de alcançabilidade. Se todos os estados no gráfico de alcançabilidade forem

numerados na ordem em que são adicionados ao conjunto A , considere o estado de aceitação com o número mais baixo. Chame esse estado de *semente*. Porque a semente pertence a um componente fortemente conectado é alcançável a partir dele mesmo. O ciclo de aceitação é detectado se e somente se nenhum dos estados intermediários ao longo desse caminho foram adicionados para definir C antes da semente. Se houver tal estado, no entanto, ele necessariamente tem um menor número de pesquisa do que a semente. Todos os estados ao longo do caminho em que estamos interessados pertencem, por definição, ao mesmo componente fortemente conectado que a semente. Se algum desses estados tem um número de pesquisa menor, e foi adicionado ao conjunto C antes, seu conjunto completo de sucessores devem ter sido analisados também, *antes de chegarmos à semente*. Isso significa que todos esses estados sucessores têm um número de pesquisa menor do que a semente. O conjunto de sucessores, no entanto, incluem a semente, porque todos eles pertencem fortemente ao mesmo componente conectado. Isso significa que esta não é a primeira visita à semente, que contradiz nossa suposição.

Página 246

SEÇÃO 11.8
GESTÃO DE COMPLEXIDADE
235

11.7 VERIFICAÇÃO DE REIVINDICAÇÕES TEMPORAIS

Para verificar as reivindicações temporais, como foram definidas no Capítulo 6, cada transição de estado em o gráfico de alcançabilidade para o sistema original, sem a reivindicação temporal, deve ser combinado com uma transição de estado na máquina de estado finito que representa o afirmação.

Felizmente, esse requisito é relativamente fácil de atender e compatível com super vestígio. Após a geração de um estado sucessor durante a pesquisa padrão, incluímos um teste extra, uma transição forçada do processo de reivindicação temporal para um novo estado. Se tal uma transição não pode ser feita, a pesquisa pode ser truncada como se uma correspondência de estado fosse encontrada.

Isso significa que o comportamento indesejável que é expresso na reivindicação não pode ser realizado após a última transição no sistema. Os detalhes de uma implementação em C são fornecidos no Apêndice E. No melhor caso possível, se não houver transição do estado do sistema pode ser correspondido por uma transição na reivindicação, o tempo e espaço exigimentos do novo algoritmo são reduzidos a quase zero. No pior caso possível, como sempre, o tamanho do espaço de estado é multiplicado pelo número de estados alcançáveis do afirmação.

A despesa de pior caso da validação de reivindicações temporais aumenta linearmente com o tamanho da reivindicação, medido como o número de estados do estado finito estendido máquina que define a reivindicação. Com a discussão das duas últimas seções, podemos compare a complexidade da validação de diferentes tipos de correção PROMELA requisitos. A despesa mínima é incorrida para a validação de propriedades de estados, como afirmações e rescisões impróprias. Pode ser duas vezes mais difícil de verificar para propriedades de não progresso e $2N$ vezes mais difícil de verificar uma reivindicação temporal de N estados.

Se invertermos esse argumento, podemos dizer que, com a mesma qualidade de pesquisa, para o validação de propriedades de estado, o sistema pode ser $2N$ vezes maior do que para a validação de reivindicações temporais. Portanto, é importante que um sistema de validação, como PROMELA, nos permite validar cada tipo de imóvel separadamente, i para que o mais simples requisitos não incorrem à custa dos mais complicados. O tamanho do sistema determina em todos os casos precisamente quais tipos de validação de uma determinada qualidade podem ser

realizada. Se a melhor qualidade de pesquisa que pode ser obtida para um determinado sistema é insuficiente, podemos fazer duas coisas:

Expresse o requisito de correção de forma diferente para que possa ser verificado mais eficientemente

Expresse o comportamento do sistema de forma diferente em um esforço para reduzir seu tamanho final Discutimos o segundo método com mais detalhes abaixo.

1. As reivindicações temporais podem ser usadas para expressar propriedades de estado, e até mesmo condições de não progresso, e podem portanto, pode ser usado como um mecanismo padrão único para especificar os requisitos de correção.

Página 247

11.8 GESTÃO DE COMPLEXIDADE

A validação de sistemas de protocolo que geram até algumas centenas de milhares de estados está bem ao alcance dos sistemas de validação automatizados que descrevemos. A validação de sistemas maiores, no entanto, pode ser um desafio substancial na gestão de complexidade. Pode-se muito bem alegar que a própria gestão da complexidade é a questão mais importante na concepção de uma estratégia de validação. Nesta seção nós revejamos algumas das questões principais.

A discussão das técnicas de pesquisa parcial (página 224) foi a principal motivação para a técnica de gerenciamento de complexidade que escolhemos como base do super algoritmo de rastreamento. Duas outras questões importantes ainda precisam ser discutidas:

Métodos de redução

Composição incremental

Ambos os métodos são aplicados antes que uma pesquisa de espaço de estado seja iniciada, em vez de ter efeito durante uma pesquisa como nas pesquisas parciais. Portanto, eles se aplicam a todos os métodos de pesquisa, de pesquisas totalmente exaustivas a passeios aleatórios. Nós os discutimos em mais detalhes abaixo.

MÉTODOS DE REDUÇÃO

O design de um modelo de validação determina trivialmente a complexidade da validação que deve ser executado. Se as técnicas de estratificação e estruturação de protocolo forem aplicadas, muitas vezes é possível separar, sem perda de generalidade, a validação de múltiplos funções de protocolo ortogonal. Um exemplo disso é dado no Capítulo 14, onde a validação do protocolo de controle de fluxo do Capítulo 7 é separada da validação do protocolo de controle de sessão.

No Capítulo 8, Seção 8.9, discutimos uma técnica para reduzir ainda mais a complexidade de um modelo de validação por generalizações sistemáticas que não afetam o escopo de um validação. Idéias semelhantes foram baseadas na noção de "projeções de protocolo", como descrito pela primeira vez por Lam e Shankar.

Em alguns casos, no entanto, ainda pode ser difícil ou impossível encontrar o comportamento ideal preservação da redução. Nesses casos, temos mais um gerenciamento de complexidade opção. Existem muitos parâmetros de modelagem que controlam a gama de possíveis comportamentos definidos por um modelo. Os fatores determinantes para a complexidade de Modelos PROMELA, por exemplo, são o número de processos, filas de mensagens e variáveis e o tamanho das filas de mensagens. Diminuindo o número de slots na mensagem filas de sage podem reduzir a quantidade máxima de assincronia em um sistema simultâneo e diminuir drasticamente o número de estados do sistema composto acessíveis, sem necessariamente diminuindo seu escopo. Um modelo de validação muitas vezes pode ser analisado exaustivamente

restringindo alguns desses parâmetros. O modelo, é claro, se torna um par-toque um quando as configurações dos parâmetros forem diminuídas. Isso significa que muitas vezes temos um escolha entre realizar uma busca exaustiva por um modelo parcial ou um modelo parcial procure um modelo completo. Qual abordagem é a mais apropriada depende naturalmente de o problema que está sendo estudado.

COMPOSIÇÃO INCREMENTAL

Nos algoritmos de análise de acessibilidade que discutimos até este ponto, temos assumiu que todos os processos assíncronos que contribuem para o comportamento global do protocolo são combinados em uma única etapa na geração do estado do sistema global espaço. Em alguns casos, um método de *composição incremental* pode ser usado para reduzir o tamanho do espaço de estado que está sendo construído. (Veja o Algoritmo 8.3, e veja também Capítulo 8, Seção 8.7.) Com este método, primeiro geramos o conjunto de todos os estados do sistema composto de dois ou mais dos processos de protocolo. Este estado parcial

o espaço é então reduzido pela minimização da máquina de estado padrão e então composto com os processos restantes, novamente de forma incremental.

Em uma aplicação típica deste método, em cada etapa duas máquinas de estado separadas são substituído por uma máquina de estado, que é reduzida em tamanho antes de ser combinada com o outras máquinas. Para funcionar, este método obviamente requer que o modelo de validação consistem em *mais* de duas máquinas de estado (processos assíncronos). Além disso, depende crucialmente na capacidade do usuário de encontrar precisamente essas combinações de máquinas de estado

que pode produzir as maiores reduções. A redução destina-se a remover comportamento que é interno às máquinas que são combinadas. Reduz a máquina combinada ao comportamento *externo* das máquinas que entraram em colapso.

Isso significa que o método funciona melhor se for aplicado a máquinas que estão fortemente acopladaspled (ou seja, eles trocam muitas mensagens) e que são relativamente independentes de o resto do sistema. Se o usuário, por engano, combina duas máquinas que estão dis-conjunta, o problema da explosão do espaço de estado é agravado: efetivamente as duas máquinas seria substituído por uma máquina de estado composto irredutível que define o comp. produto cartesiano completo de todos os estados nas duas máquinas de estados individuais. Uma grande fratura

ção desses estados pode ser reconhecida como inalcançável apenas quando a composição restante passos de instalação são executados.

Vários pesquisadores implementaram o método de composição incremental e aplicou-o a modelos de validação que usam apenas comunicações de *rendezvous*. A vantagem A vantagem aqui é que os pontos de encontro podem desaparecer nas etapas de redução. Não é claro se o método ainda pode ser eficaz quando aplicado a sistemas como PROMELA que permite trocas assíncronas de mensagens em buffer. Nestes casos, o buffers internos podem complicar o processo de minimização.

11.9 LIMITAÇÃO DE MODELOS DE PROMELA

Não é imediatamente óbvio que qualquer modelo PROMELA pode ser reduzido a um sistema de estado finito e validado com os algoritmos que discutimos neste capítulo. Um modelo de validação PROMELA permite um número arbitrário de instantes de processos tiações e um número arbitrário de filas de mensagens a serem criadas. O seguinte programa, por exemplo, é válido na PROMELA .

Página 249

238

VALIDAÇÃO DE PROTOCOLO

CAPÍTULO 11

proctipo A ()

{

chan Ain = [1024] de {int, int};

Faz

:: executar A ()

od

}

init {executar A ()}

Simular a execução deste modelo exigiria uma quantidade infinita de memória e uma extensão infinita de tempo. Qualquer execução real do modelo, no entanto, só pode ocorrem em uma máquina finita. A maioria dos modelos são, portanto, finitos por design e podem até ser argumentado que a possibilidade de crescimento infinito é um erro de projeto.

PROMELA restringe o número máximo de processos e filas de mensagens que podem ser criado. O limite preciso não está definido. Em algum ponto durante a execução do programa de exemplo acima da instrução de `execução` se tornará inoperante e bloqueará o último processo que foi criado. Cada modelo PROMELA é, portanto, por definição, um modelo finito sistema de estado e pode ser analisado com um algoritmo de análise de alcançabilidade padrão. Cada processo tem um número fixo de estados, cada fila de mensagens tem um número fixo de slots, e o intervalo de todas as variáveis usadas no sistema é fixo. Quando o modelo é executado, ele só pode atingir um número finito de estados possíveis. Em algum ponto do execução do processo A () acima, por exemplo, a instrução `run` torna-se não executada capaz e proíbe um maior crescimento.

Nos Capítulos 12 e 13, discutimos a implementação de um programa que converte Especificações PROMELA nos modelos de estado finito necessários. O programa implementa todos os três modos de pesquisa básicos que discutimos: simulação aleatória, estado de bit

pesquisa espacial e a pesquisa completa no espaço de estados.

11,10 RESUMO

Dado um protocolo novo e cuidadosamente projetado, como podemos ter confiança de que não falhar de alguma forma inesperada? Por exemplo, podemos querer provar que o protocolo é robusta sob comportamento de canal adverso, ou podemos querer mostrar que certos indesejáveis eventos capazes, como deadlocks do sistema, não podem ocorrer. Os métodos que descrevemos neste capítulo são baseados na verificação dos requisitos de correção que podem ser expressa como invariantes do sistema: propriedades que permanecem invariavelmente verdadeiras para todos os possíveis execuções do sistema.

O método de prova manual que fornecemos é baseado em uma inspeção exaustiva de transposição de estado

instalações, e a variante automatizada é baseada em uma inspeção exaustiva do sistema estados. O procedimento de validação manual pode funcionar para sistemas de até dez ou vinte transições de estado, mas é amplamente independente do número de estados. A credibilidade dessas provas manuais, no entanto, é na melhor das hipóteses inversamente proporcional ao seu comprimento.

O procedimento automatizado não tem essa desvantagem, mas sua aplicabilidade depende crucialmente no número de estados do sistema alcançáveis. Para sistemas relativamente pequenos, até

Página 250

CAPÍTULO 11
EXERCÍCIOS
239

a aproximadamente 10^5 estados de sistema alcançáveis, podemos aplicar um teste totalmente exaustivo busca espacial. O objetivo da busca exaustiva é mostrar a *ausência* de erros.

Se puder ser concluído sem relatar quaisquer erros, é certo que o protocolo pode não violar nenhum dos critérios de correção.

Para sistemas maiores, até aproximadamente 10^8 estados de sistema alcançáveis, a melhor validaçãoção que pode ser realizada é uma pesquisa parcial controlada. O propósito de um parcial a pesquisa é mostrar a *presença* de erros, não a ausência. A busca parcial é projetado de tal forma que, se for aplicado a um protocolo que contém um erro, otimiza nossas chances de expô-lo dentro das restrições da máquina na qual o algoritmo de validação é executado. Discutimos três maneiras diferentes de alcançar este objetivo:

Usando heurísticas de pesquisa para restringir a pesquisa parcial aos estados do sistema que são prováveis para conter os erros.

Usando uma técnica de hashing que aumenta drasticamente o número de estados do sistema que pode ser manipulado.

Usando métodos de redução para simplificar os modelos de validação antes de serem submetidos para uma pesquisa.

O primeiro método tem a desvantagem de tentar prever onde os erros são prováveis ser, uma estratégia inherentemente perigosa. A segunda estratégia não tem esse problema e acaba sendo o único que nos permite corresponder ao escopo da análise às restrições do sistema em que o algoritmo de validação é executado, o que sempre que eles podem ser. A candidatura ao PROMELA é elaborada nos próximos dois capítulos. Para problemas de validação excepcionalmente grandes, finalmente, a única validação viável método é uma simulação aleatória que tenta explorar tantos estados do sistema quanto possível, tentando localizar os estados que podem violar as invariantes do sistema.

EXERCÍCIOS

11-1. Use a técnica de prova manual para mostrar que o protocolo de bit alternado preserva o exatidão do protocolo de janela invariante para um tamanho de janela de um.

11-2. Modifique o algoritmo de pesquisa parcial para incluir uma exploração espacial de estado máximo ou razoável heurística.

11-3. A seguinte solução para o problema de exclusão mútua de Dijkstra (ver Capítulo 2, e Dijkstra [1965]) apareceu nas *Communications of the ACM*, Hyman [1966]. É reproduzido aqui como foi publicado (em pseudo Algol).

1 array booleano $b(0; I)$ inteiro k, i, j ;

2 processo de comentário i , com $i \in 0 \cup I$;

3 $C0: b(i) := \text{falso}$;

4 $C1: \text{se } k! = I, \text{ então comece}$

5 $C2: \text{se } \neg b(j) \text{ então vá para } C2;$

6
senão $k := i$; vá para o final de $C1$;
7
outra seção crítica;
8
 $b(i) := \text{verdadeiro}$;
9
restante do programa;
10
vá para $C0$;

Página 251

240
VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11
11
fim

Mostre que a solução está incorreta modelando a solução em PROMELA e realizando uma validação automatizada com um dos algoritmos de análise de acessibilidade discutidos neste capítulo.

11-4. Os algoritmos de análise de acessibilidade que consideramos verificam a observância do sistema invariantes do *estado*. Considere possíveis extensões para o algoritmo básico de pesquisa completa para verificar para propriedades de *sequências de estado* do sistema : caminhos através do espaço de estado global. o que extensões são necessárias, por exemplo, para poder provar ou refutar para a alternância protocolo de bits de que não há sequência infinita de transições em que a sequência de 1 bit o número permanece inalterado?

11-5. Existem algoritmos que podem encontrar todos os componentes fortemente conectados em um cílico direcionado gráfico, por exemplo, Aho, Hopcroft & Ullman [1974, p. 192]. Considere como tal algoritmo poderia ser usado para estender as capacidades dos analisadores de alcançabilidade, qual o custo em adicionar complexidade de tempo e espaço seria, e como essas extensões seriam afetadas por pesquisa parcial.

NOTAS BIBLIOGRÁFICAS

A técnica de prova manual baseada em invariantes do sistema é devida a Krogdahl [1978] e Knuth [1981]. A prova da janela invariante discutida aqui também foi dada primeiro em Knuth [1981]. O método também foi usado mais recentemente em Brown, Gouda e Miller [1989]. O método de validação manual de Gouda baseado em invariantes de estado e as fórmulas bem fundamentadas são inspiradas no artigo seminal Floyd [1967].

Várias outras tentativas foram feitas para desenvolver ferramentas automatizadas de validação de protocolo

que não são baseadas na análise de acessibilidade. Experiência inicial com algumas versões dessas ferramentas foram relatadas em Schwabe [1981] e Sunshine e Smallberg [1982]. Uma nova teoria promissora de prova manual é baseada na especificação Oxford linguagem z . Ver, por exemplo, Duke, Hayes, King and Rose [1988], Duke, Hayes e Rose [1988] e Hayes, Mowbray e Rose [1989].

O trabalho em métodos automatizados de validação de protocolo foi iniciado por Brand e Joyner [1978], Hajek [1978], West e Zafiropulo [1978], West [1978], Zafiropulo [1978], e Razouk e Estrin [1980]. O trabalho de Colin West e Pitro Zafiropulo [1978] forneceu uma primeira demonstração de que, com ferramentas automatizadas, mesmo protocolos que têm resistiu ao escrutínio de anos de desenvolvimento em uma padronização internacional a organização pode, dentro de alguns segundos do tempo do computador, mostrar falhas. No neste caso, o protocolo foi a Recomendação CCITT X.21, e a ferramenta de validação foi uma implementação direta da teoria de validação desenvolvida em Zafiropulo [1978]. Um trabalho subsequente importante foi relatado em Zafiropulo et al. [1980], Rubin e West [1982]. Excelentes pesquisas do trabalho de validação de protocolo pode ser encontrado nos anais de conferências do IFIP, como IFIP [1983], ou em abril de 1980 edição especial sobre " Arquiteturas e protocolos de rede de computadores " do IEEE Transactions on Communications , que contém a referência padrão Bochmann e Sunshine [1980].

Existem muitos resultados sobre a complexidade computacional da tarefa de validação de um

Página 252

CAPÍTULO 11
NOTAS BIBLIOGRÁFICAS
241

comunicando modelo de máquina de estado finito, ver por exemplo Cunha e Maibaum [1981], Brand e Zafiropulo [1983], Apt e Kozen [1986], Reif e Smolka [1988].

Em geral, o problema de encontrar deadlocks em um sistema de comunicação de estado finito máquinas é PSPACE completo na melhor das hipóteses, e torna-se formalmente indecidível quando os canais de mensagens são ilimitados.

Este resultado, é claro, não significa que qualquer análise posterior da máquina de estados finitos modelos é inútil. Isso significa que a complexidade de um algoritmo de validação de protocolo ritmo é a principal preocupação. Esses algoritmos não podem carregar mais sobrecarga do que estritamente

necessário para resolver o problema. Embora possa ser tentador estender um algoritmo de pesquisa ritmo para capturar características mais sutis, geralmente é desaconselhável fazê-lo se o método é sobreviver à aplicação a problemas de tamanho realista.

A necessidade de técnicas de busca parcial foi descrita pela primeira vez em West [1986b] e em Holzmann [1985, 1987a]. Uma visão geral de uma gama de heurísticas de pesquisa que desde então foi inventado para buscas parciais pode ser encontrado em Lin, Chu e Liu [1987]. O correu método de exploração espacial do estado dom, estudado pela primeira vez por Colin West [1986b, 1989]. As técnicas de pesquisa parcial probabilística foram descritas por Maxemchuck e Sabnani [1987]. Uma técnica de pesquisa dispersa com expressões de orientação foi introduzida no Pageot e Jard [1988]. Uma heurística para ordens parciais foi sugerida pela primeira vez em Holzmann [1985]. Várias abordagens mais formais foram investigadas nos últimos anos, por exemplo, Probst [1990], Valmari [1990] e Godefroid [1990]. O estado de progresso justo a heurística de exploração foi sugerida pela primeira vez em Rubin e West [1982], e posteriormente explorado em Gouda e Han [1985]. A exploração do estado de progresso máximo era descrito em Gouda e Yu [1984]. O conceito de "projeções de protocolo" foi introduzido produzido em Lam e Shankar [1984].

A técnica de espaço de estado de bits foi descrita pela primeira vez em Holzmann [1987b] e elaborada em Holzmann [1988]. A técnica de hashing é baseada em uma técnica muito mais antiga denominado "armazenamento de dispersão", descrito em Morris [1968] e aplicado em McIlroy [1982]. A técnica de pesquisa de espaço de estado de bits pode ser facilmente aplicada a todos os modelos baseados em FSM,
eg, Rafiq e Ansart [1983], Estelle, eg, Richier et al. [1987], o modelo S / R, Aggarwal, Kurshan e Sharma [1983], e modelos da Rede de Petri, por exemplo, Bourguet [1986], para citar apenas alguns.

A extensão do algoritmo de pesquisa exaustivo com recursos de comprovação de asserção foi descrito em Holzmann [1987a]. Uma comparação de algoritmos de pesquisa com base em a análise de acessibilidade apareceu em Holzmann [1990].

O algoritmo para a detecção de ciclos de não progresso não foi publicado antes.

O algoritmo para a detecção de ciclos de aceitação, por exemplo, no contexto de um reclamação temporal, é devido a Mihalis Yannakakis da AT&T Bell Laboratories. Foi primeiro descrito em Courcoubetis, Vardi, Wolper e Yannakakis [1990]. Um algoritmo padrão ritmo para detectar componentes fortemente conectados em um gráfico pode ser encontrado em Aho, Hopcroft e Ullman [1974].

A aplicação de modelos de estado finito puro ao problema de validação de protocolo pode ser

Página 253

242

VALIDAÇÃO DE PROTOCOLO
CAPÍTULO 11

encontrado em, por exemplo, Brand e Zafiropulo [1983], Bochmann [1983] ou Knudsen [1983].

Muitas abordagens interessantes para o problema de validação de protocolo não poderiam ser descartadas xingado aqui. Em particular, isso vale para o trabalho no modelo S / R e ômega regular línguas, Aggarwal, Kurshan e Sharma [1983], Har'El e Kurshan [1990], e sistemas de verificação de modelo para verificação de circuito, por exemplo, Clarke [1982], Browne, Clarke, Dill e Mishra [1986].

Página 254

UM SIMULADOR DE PROTOCOLO 12

244 SPIN - Visão geral 12.2
245 Expressões 12.3
255 Variáveis 12.4
265 Declarações 12.5
275 Controle de Fluxo 12.6
282 Processos e Tipos de Mensagem 12.7
292 Macro Expansão 12.8
293 SPIN - Opções 12.9
294 Resumo 12.10
295 exercícios
296 Notas Bibliográficas

12.1 INTRODUÇÃO

Sem as ferramentas adequadas, pode ser possível projetar um protocolo correto, mas na maioria dos casos, será impossível estabelecer formalmente a sua correção com qualquer medida de confiabilidade. Até agora, temos nos ocupado principalmente com o desenvolvimento de um disciplina de design baseada no uso de modelos de validação formais. Neste capítulo nós estender esta disciplina com uma ferramenta de software para simular o comportamento de validação de modelos escritos em PROMELA. A ferramenta é chamada de SPIN, que é a abreviação de: simples PROMELA intérprete¹. SPIN pode simular a execução de um modelo de validação por interpretando PROMELA declarações na mosca. Ele pode ser usado em parcial ou completo projetos de protocolo, em qualquer nível de abstração. Pode nos dizer rapidamente se nós ou não estamos no caminho certo com um design e, como tal, pode ser uma ferramenta de design valiosa. O programa que desenvolvemos neste capítulo não tentará validar a correção dos requisitos. Essa é uma tarefa para um validador (Capítulo 13). Uma pequena exceção a isso é feita para declarações de assert PROMELA, desde os requisitos correspondentes são validados como um mero efeito colateral de sua execução. A validação de não progresso, ciclos, estados finais inválidos e reivindicações temporais, no entanto, estão fora do escopo de um simulador. O programa completo do simulador contém cerca de 2.000 linhas de texto. Inclui um analisador léxico, um analisador e um planejador de processo. Requer uma feira quantidade de explicação e alguma familiaridade com C e UNIX para resolver isso. No capítulo. Mas, fique tranquilo, não é necessário entender os detalhes do implemento para poder usar o simulador. Abaixo, discutimos primeiro a estrutura geral e o tipo de saída que o simulador pode gerar. Em seguida, discutimos uma pequena versão do programa, apenas para avaliar expressões PROMELA, e mostrar como funciona.

1. Os termos *simulador*, *interpretador* e *avaliador* são usados como sinônimos neste capítulo.
243

Página 255

244
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

Finalmente, estendemos este programa em um intérprete completo, adicionando o que falta: peças uma por uma. No próximo capítulo o SPIN é estendido com uma opção para realizar validações de protocolo automatizadas rápidas. Uma lista de fontes para a versão final do SPIN pode ser encontrado nos Apêndices D e E.

12.2 SPIN - VISÃO GERAL

Para construir o interpretador, contamos com as ferramentas de programação UNIX yacc, lex e make. Uma familiaridade casual com essas ferramentas é, portanto, assumida. Nos concentraremos aqui em o que as ferramentas podem fazer por nós, em vez de explicar como funcionam internamente. No caso de emergência consulte um manual do UNIX ou consulte as Notas Bibliográficas no final do capítulo. Apresenta indicações para outras literaturas que podem ser úteis.

UMA EXECUÇÃO DE SIMULAÇÃO DE AMOSTRA

Considere o programa de exemplo do Capítulo 5 para calcular o fatorial de um número inteiro inteiro.

```
fato de proctype (int n; chan p)
{
    resultado interno;
    E se
        :: (n <= 1) -> p! 1
        :: (n >= 2) ->
            filho chan = [1] de {int};
            executar fato (n-1, filho);
            filho? resultado;
            p! n * resultado
        fi
    }
    iniciar
{
```

```

resultado interno;
filho chan = [1] de {int};
executar fato (12, criança);
filho? resultado;
printf ("resultado:% d \ n", resultado)
}

```

Executar o analisador neste programa produz a seguinte saída:

```

$ spin factorial
resultado: 479001600
13 processos criados

```

onde \$ é um prompt do sistema UNIX . E felizmente,

```
12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 479001600
```

Executar o simulador no modo detalhado nos dá um pouco mais de informações sobre o executado, por exemplo, imprimindo todas as transmissões de mensagens, com o número do processo de realizá-los, o conteúdo da mensagem que está sendo enviada e o nome do canal de destino.

Página 256

SEÇÃO 12.3 EXPRESSÕES

245

```

$ spin -s factorial
proc 12 (fato) linha 5, enviar 1
-> fila 12 (p)
proc 11 (fato) linha 10, Enviar 2
-> fila 11 (p)
proc 10 (fato) linha 10, Enviar 6
-> fila 10 (p)
proc 9 (fato) linha 10, Enviar 24
-> fila 9 (p)
proc 8 (fato) linha 10, enviar 120
-> fila 8 (p)
proc 7 (fato) linha 10, Enviar 720
-> fila 7 (p)
proc 6 (fato) linha 10, enviar 5040
-> fila 6 (p)
proc 5 (fato) linha 10, enviar 40320
-> fila 5 (p)
proc 4 (fato) linha 10, Enviar 362880 -> fila 4 (p)
proc 3 (fato) linha 10, Enviar 3628800 -> fila 3 (p)
proc 2 (fato) linha 10, Enviar 39916800 -> fila 2 (p)
proc 1 (fato) linha 10, Enviar 479001600 -> fila 1 (p)
resultado: 479001600
13 processos criados

```

A coluna com o valor da mensagem agora nos dá implicitamente uma contagem contínua do factorial sendo computado. Se ainda mais informações forem necessárias, também podemos executar o simulador com sinalizadores adicionais para imprimir, por exemplo, recepções de mensagens ou os valores

de variáveis. Mas o exemplo acima é suficiente por enquanto.

12.3 EXPRESSÕES

Uma função específica que o simulador deve realizar é a avaliação de expressões.

Em uma declaração como

```
dados crunch! (3 * 12 + 4/2)
```

o simulador deve avaliar três expressões:

O valor da crise de destino

O valor dos dados do tipo de mensagem e

O valor do argumento $3 * 12 + 4/2$

A avaliação de expressões pode parecer insignificante no início, mas como PROMELA é

fundada no conceito de executabilidade, a avaliação das declarações em geral é realmente no centro do simulador. Para manter as coisas simples, vamos começar com um pequeno programa que não pode fazer mais do que avaliar expressões PROMELA .

Temos que dizer ao nosso programa como as expressões válidas se parecem e como deveriam ser avaliado. O primeiro problema exige uma especificação gramatical. Se ignorarmos a variável nomes por um tempo, a forma mais simples de expressão é um número, digamos, um inteiro stand. Escrevemos uma constante como uma série de um ou mais dígitos, onde um dígito é qualquer símbolo

bol no intervalo de '0' a '9' . Se formalizarmos isso, podemos escrever

dígito:

```
'0' | '1' | '2' | '3' | '4'  
|
```

'5' | '6' | '7' | '8' | '9'

O termo que estamos definindo está no lado esquerdo dos dois pontos, e os símbolos definidores são no lado direito onde a barra vertical '||' é usado para separar alternativas. Recorrente
Dessa forma, podemos definir uma constante como uma série de um ou mais dígitos, como segue:

246

UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
const:
dígipto
|
dígipto const

E, da mesma forma, podemos especificar que uma expressão simples é apenas um número, escrevendo
expr
:
const

Agora é fácil marcar tipos de expressões mais interessantes. De quaisquer dois válidos
expressões que podemos tornar válidas adicionando, subtraindo, multiplicando ou
dividindo-os. Recursivamente novamente, definimos

expr
:
const
|
expr '+' expr
|
expr '-' expr
|
expr '*' expr
|
expr '/' expr
|
'(' expr ')'

A última linha é para boa forma: se $2 + 5$ é uma expressão válida, então é $(2 + 5)$. Isso também
nos permite forçar a ordem de avaliação das subexpressões, mas mais sobre isso mais tarde.

Formalmente, o que definimos aqui são três *regras de produção*. No lado esquerdo de
a regra (antes dos dois pontos) escrevemos a frase que queremos definir. Do lado direito
escrevemos uma série de maneiras alternativas nas quais a frase pode ser construída,
separados por barras verticais. Os caracteres citados são chamados de *literais*. Nomes escritos em
as capitais são chamadas de *terminais* ou *tokens*. Todo o resto é chamado de *não terminal* e
deve ser definido, ou seja, deve ocorrer em algum lugar do lado esquerdo de uma regra de produção.
Um analisador léxico é usado para reconhecer os terminais e literais e passá-los para um
analisador. O analisador pode então restringir-se a verificar a gramática e construir um
"árvore de análise." Esta estrutura geral é ilustrada na Figura 12.1.

PROMELA
Especificação
Lexical
Analizador
Parser
Agendador
Simulação
Resultado

Figura 12.1 - Estrutura Geral do Simulador

O planejador avalia o programa caminhando pela árvore de análise, avaliando seu
nós de acordo com a semântica da linguagem. Portanto, é importante que o
parser entrega uma estrutura em árvore ao escalonador que está conectado de tal forma que
pode ser avaliado em tempo real. Mas, mais sobre o analisador e o planejador mais tarde; Deixe-nos
primeiro olhe para o analisador léxico.

Uma primeira especificação do analisador léxico é bastante simples de definir com o
Lex da ferramenta UNIX. Na verdade, com lex, é fácil considerar os números uma classe especial de token
nomeado CONST (maiúsculo, porque agora é um terminal), com seu valor numérico
anexado como um atributo. Desta forma, a especificação da gramática pode assumir o
existência dos tokens de número, ao invés de ter que analisar e verificar se há sin-
imposto. Essa é a aparência da especificação do analisador léxico. Chamamos isso primeiro
versão do programa spine (avaliador da expressão SPIN). Como sempre, a linha

os números não fazem parte do texto do programa.

```
1 / ***** coluna: lex.l ***** /
2
3% {
4 #incluir "spin.h"
5 #incluir "y.tab.h"
6
7 int lineno = 1;
8%
9
10 %%
11 [ \t]
{ / * ignorar o espaço em branco * /
12 [0-9] +
{yyval.val = atoi (yytext); return CONST; }
13 \n
{lineno ++; }
14
{return yytext [0]; }
```

O programa salta sobre o espaço em branco (linha 11), conta novas linhas (linha 13), calcula o valor das sequências de dígitos (linha 12) e o retorna como parte de um token do tipo CONST . Todo o resto é transmitido como literal (linha 14), ou seja, como um único caractere.

Contamos com o yacc para produzir a definição do nome CONST . Está contido no arquivo de cabeçalho y.tab.h que está incluído na linha 5. Mas antes de discutirmos a entrada para yacc , temos que voltar brevemente à definição da gramática.

Ainda temos que definir o que significa uma expressão como $2 + 5 * 3$. Tanto quanto nosso programa, pode igualmente significar $(2 + 5) * 3$ ou $2 + (5 * 3)$. O convenção é que a multiplicação e a divisão têm uma precedência maior do que a adição e subtração, o que significa que a segunda interpretação é a correta. Nós vamos veja abaixo como essas regras de precedência podem ser declaradas formalmente em uma gramática especificação.

Todas as regras e definições fornecidas até agora podem ser expressas com o gerador de analisador UNIX yacc . Uma especificação completa do yacc para a gramática definida até agora é a seguinte:

Página 259

```
1 / ***** coluna: spin.y ***** /
2
3% {
4 #incluir "spin.h"
/ * define quais nós são etc * /
5%
6
7% sindicato {
/ * define o tipo de dados de * /
8
int val;
/ * a pilha de análise interna * /
9
Nó * nó;
10}
11
12% token
<val> CONST / * token CONST tem um valor
* /
13% tipo
<node> expr
/ * expressões produzem nós * /
14
15% restantes
'+' '-'
/ * operadores associativos à esquerda * /
16% restantes
'*' '/'
/ * idem, precedência mais alta * /
17 %%
18
19 / ** Regras gramaticais ** /
20
21 programa: expr
```

```

{printf ("=% d \ n", eval ($ 1)); }
22
;
23 expr
: '(' expr ')'
{$$ = $ 2; }
24
| expr '+' expr
{$$ = nn (0, '+', $ 1, $ 3); }
25
| expr '-' expr
{$$ = nn (0, '-', $ 1, $ 3); }
26
| expr '*' expr
{$$ = nn (0, '*', $ 1, $ 3); }
27
| expr '/' expr
{$$ = nn (0, '/', $ 1, $ 3); }
28
| CONST
{$$ = nn ($ 1, CONST, 0, 0); }
29
;
30 %%

```

Partes desta especificação parecerão familiares. As linhas 23 a 29 definem a estrutura de expressões e as linhas 15 e 16 definem as regras de precedência. A linha 12 define CONST para ser um token com um atributo do tipo val . A linha 21 define que, nesta versão, nós espera que um programa PROMELA consista em uma única expressão.

Existem também algumas frases novas. A linha 13, por exemplo, contém a definição para o representação interna de uma expressão: uma estrutura de dados denominada nó . Usamos yacc para construir uma árvore de análise de um programa PROMELA ou, neste caso, uma árvore de análise de uma expressão.

Por exemplo, analisar a expressão $2 + 5 * 3$ produz a árvore mostrada na Figura 12.2.

A árvore de análise define a estrutura do programa e nos ajudará mais tarde a determinar como deve ser interpretado.

O arquivo yacc é usado para gerar o que é chamado de *analizador LALR (1)* que pode construir o árvore na Figura 12.2. O analisador verifica sua entrada em uma passagem de l EFT para a direita. Constrói

um r derivação ightmost em sentido inverso, utilizando, no máximo, um look- um token de cabeça. Cada nó em

a árvore de análise é representada por uma estrutura de dados que é definida no arquivo de inclusão spin.h , nas linhas 3 a 7. A estrutura é referida em spin.y nas linhas 9 e 13.

```

SEÇÃO 12.3
EXPRESSÕES
249
+
CONST
2
*
CONST
5
CONST
3
Nó
Nó
Nó
Atributo
Figura 12.2 - Árvore de Análise para 2 + 5 * 3
1 / ***** coluna: spin.h ***** /
2
3 typedef struct Node {
4
int
nval;
/* atributo de valor
*/
5
short ntyp;

```

```

/ * tipo de nó
*
6
struct Node * lft, * rgt; / * árvore de análise de filhos * /
7} Nó;
8
9 Nó externo * nn ();
/* aloca nós
*/
10 extern char * emalloc ();
/* aloca memória
*/
11 saída externa vazia ();

```

Nós usamos apenas operadores aritméticos binários, então cada nó na árvore precisa apontar a no máximo dois descendentes. Em `spin.h`, eles são chamados de `lft` e `rgt`, dois indicadores para estruturas do tipo `Nó`. A definição de um `Nó` também contém um campo para armazenar o tipo de nó, o qual pode ser um operador, como `'+'`, `'*'`, ou pode ser um terminal não do tipo `CONST`. Os nós terminais, é claro, não têm descendentes (cf. Figura 12.2), mas eles têm um atributo do tipo `nval` que é usado para armazenar os dados numéricos valor da constante calculado pelo analisador léxico e passado para o analisador em o campo `yyval.val` do token (linha 12 de `lex.l`).

Agora, de volta ao analisador. Sempre que uma subexpressão é reconhecida, ela é lembrada em uma estrutura do tipo `Node`. Em nosso exemplo, usamos a função `nn ()` para preparar tal estrutura. Seu tipo é declarado em `spin.h` na linha 9. A definição do procedimento `em si` é o seguinte:

Página 261

250
UM SIMULADOR DE PROTOCOLO

CAPÍTULO 12

```

Nó *
nn (v, t, l, r)
Nó * l, * r;
{
Nó * n = (Nó *) emalloc (sizeof (Nó));
n-> nval = v;
n-> ntyp = t;
n-> lft = l;
n-> rgt = r;
return n;
}

```

A rotina aloca memória para um novo nó na árvore de análise, contando com `emalloc ()` para verificar as condições de erro e retorna um ponteiro para a estrutura inicializada. Para exemplo, quando a expressão `5 * 3` é analisada, a linha 28 em `spin.y` é chamada duas vezes, uma para `5`, uma vez para `3`. Cada chamada produz uma subexpressão do tipo `CONST` que é passada para a linha 26. Os nós do tipo `CONST` não têm descendentes, mas os campos de atributos são definidos como o valor da constante. O valor produzido por `lex.l` está disponível em um pré-definido parâmetro denominado `$ 1`, onde `1` se refere ao primeiro campo na regra de produção on-line 28. Nesse caso, há apenas um campo do lado direito dessa regra, então `$ 1` também é o apenas parâmetro válido. O tipo do campo é um número inteiro neste caso, conforme definido em linhas 8 e 12.

Na linha 26, um novo nó é construído do tipo `'*'` com as duas subexpressões de digite `CONST` como descendentes. Operadores aritméticos, como `'*'`, são passados como literais do analisador léxico para o analisador. As estruturas de dados que representam os dois sub-expressões para a multiplicação são novamente passadas por `yacc` em dois parâmetros, chamados `$ 1` e `$ 3`. Eles apontam para o primeiro e o terceiro campo da regra de produção.

Quando a árvore de análise completa for construída, ela será passada para a regra de produção online 21 e pode ser interpretado. Aqui está o código do intérprete.

```

eval (agora)
Nó * agora;
{
if (now! = (Node *) 0)
switch (agora-> ntyp) {
case CONST: retorna agora-> nval;
case '/': return (eval (now-> lft) / eval (now-> rgt));
case '*': return (eval (now-> lft) * eval (now-> rgt));
case '-': return (eval (now-> lft) - eval (now-> rgt));
case '+': return (eval (now-> lft) + eval (now-> rgt));
padrão: printf ("spin: tipo de nó inválido% d \ n", agora-> ntyp);

```

```

    saída (1);
}
return 0;
}

```

A cláusula padrão protege contra tipos de nós desconhecidos. Os tipos conhecidos de nós são avaliados recursivamente até que a resposta final seja produzida.

Se juntarmos todas essas peças, teremos o avaliador de expressão PROMELA . Aqui está

Página 262

SEÇÃO 12.3
EXPRESSÕES
251

uma lista completa, junto com as rotinas restantes que escaparam da menção acima.

```

1 / ***** coluna: spin.h *****
2
3 typedef struct Node {
4
5     int
6     nval;
7     /* atributo de valor
8     */
9     short ntyp;
10    /* tipo de nó
11   */
12    struct Node * lft, * rgt; /* árvore de análise de filhos */
13} Nó;
14
15% {
16 #incluir "spin.h"
17 #incluir "y.tab.h"
18
19 int lineno = 1;
20%
21
22 %%
23 [\t]
24 [0-9] +
{yyval.val = atoi (yytext); return CONST; }
25 \n
{lineno ++; }
26
{return yytext [0]; }
27
28 / ***** coluna: spin.y *****
29
30% {
31 #include "spin.h"
32    /* define quais nós são etc */
33
34% sindicato {
35    /* define o tipo de dados de */
36    int val;
37    /* a pilha de análise interna */
38    Nó * nó;
39
39% token
<val> CONST /* token CONST tem um valor
*/
40% tipo
<node> expr
/* expressões produzem nós */
41

```

```

42% sairam
'+' '-'
/* operadores associativos à esquerda */
43% sairam
'*' '/'
/* idem, precedência mais alta */
44 %%
45
46 / ** Regras gramaticais ** /
47
48 programa: expr
{printf ("%= d \ n", eval ($ 1)); }
49
;
50 expr
: '(' expr ')'
{$$ = $ 2; }
51
| expr '+' expr
{$$ = nn (0, '+', $ 1, $ 3); }


```

Página 263

252
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

```

52
| expr '-' expr
{$$ = nn (0, '-', $ 1, $ 3); }
53
| expr '*' expr
{$$ = nn (0, '*', $ 1, $ 3); }
54
| expr '/' expr
{$$ = nn (0, '/', $ 1, $ 3); }
55
| CONST
{$$ = nn ($ 1, CONST, 0, 0); }
56
;
57 %%
58
59 / ***** coluna: run.c *****
60
61 #incluir "spin.h"
62 #include "y.tab.h"
63
64 eval (agora)
65
Nó * agora;
66 {
67
if (now! = (Node *) 0)
68
switch (agora-> ntyp) {
69
case CONST: retorna agora-> nval;
70
case '/': return (eval (now-> lft) / eval (now-> rgt));
71
case '*': return (eval (now-> lft) * eval (now-> rgt));
72
case '-': return (eval (now-> lft) - eval (now-> rgt));
73
case '+': return (eval (now-> lft) + eval (now-> rgt));
74
padrão: printf ("spin: tipo de nó inválido% d \ n", agora-> ntyp);
75
saída (1);
76
}
77
return 0;
78}
79
80 / ***** coluna: main.c *****
81
82 #include "spin.h"
83 #incluir "y.tab.h"
84
85 principal ()


```

```

86 {
87
88 yyparse ();
89
90
91 yywrap ()
/* uma rotina ficticia */
92 {
93
94 return 1;
95
96 yyerror (s1, s2)
/* chamado por yacc em erros de sintaxe */
97
98 char * s1, * s2;
99
100 extern int lineno;
101 char buf [128];
102 sprintf (buf, s1, s2);
103 printf ("spine: linha% d:% s \ n", lineno, buf);
104
105 char *

```

Página 264

SEÇÃO 12.3
EXPRESSÕES

253

```

106 emalloc (n)
107 {extern char * malloc (); /* funções de biblioteca */
108
109 extern char * memset ();
110
111 char * tmp = malloc (n);
112 if (! tmp)
113 {
114 printf ("spine: memória insuficiente \ n");
115 saída (1);
116 }
117
118 memset (tmp, 0, n);
/* limpar memória */
119 Nó *
120 nn (v, t, l, r)
121
Nó * l, * r;
122
123
Nó * n = (Nó *) emalloc (sizeof (Nó));
124
n-> nval = v;
125
n-> ntyp = t;
126
n-> lft = l;
127
n-> rgt = r;
128
return n;
129}

```

Para compilar este conjunto de programas, o seguinte *makefile* pode ser usado.

```

# ***** spine: makefile *****
CC = cc
# Compilador ANSI C

```

```

CFLAGS =
# ainda sem sinalizadores
YFLAGS = -v -d -D # verboso, depuração
OFILES = spin.o lex.o main.o run.o
coluna: $ (OFILES)
$ (CC) $ (CFLAGS) -o spine $ (OFILES)
% .o:
% .c spin.h
# todos os arquivos dependem de spin.h
$ (CC) $ (CFLAGS) -c $%. C

```

O *makefile* define quais sinalizadores devem ser passados para *yacc* e *cc* e registra o dependências entre os arquivos de origem. Afirma, por exemplo, que quando *spin.h* alterações, todos os arquivos de objeto devem ser recriados. O *makefile* é lido por outro utilitário UNIX chamado *make* para produzir a espinha do programa executável . Aqui está o diálogo que é impresso no meu sistema se este programa for compilado e invocado com um exemplo de expressão sion.

```

$ make
yacc -v -d -D spin.y
cc -c y.tab.c
rm y.tab.c
mv y.tab.o spin.o

```

Página 265

254

UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
lex lex.l
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc -c main.c
cc -c run.c
cc -o spine spin.o lex.o main.o run.o
\$ echo "2 + 5 * 3" | coluna
= 17
\$

O programa completo deve, é claro, reconhecer alguns outros recursos da linguagem antes de simular programas PROMELA . Eles podem ser agrupados em cinco classes:

Variáveis

Afirmações

Construções de fluxo de controle

Processos

Expansão macro

VARIÁVEIS (Seção 12.4, página 255)

Temos que considerar declarações de variáveis, atribuições a variáveis e referências variáveis, geralmente em expressões construídas a partir de toda a gama de aritmética e lógica operadores, mais os operadores especiais *len* e *run* . Nós consideramos variáveis dos cinco tipos básicos de dados *bit* , *booleano* , *byte* , *curto* , e *int* , além do canal de declaração identificadores de tipo com a palavra-chave *chan* .

DECLARAÇÕES (Seção 12.5, página 265)

Existem dois tipos de declarações incondicionais: atribuições a variáveis e o declaração de impressão que adicionamos ao simulador. Existem também cinco tipos básicos de condições

instruções tradicionais: condições booleanas, *tempo limite* , enviar e receber instruções e afirmar . O salto de "pseudo-instrução" pode ser implementado como o equivalente do condição (1) .

FLUXO DE CONTROLE (Seção 12.6, página 275)

Temos que considerar as especificações de fluxo de controle sequencial: *goto* , *break* , *select* ção, repetição e declarações atômicas .

PROCESSOS E TIPOS DE MENSAGEM (Seção 12.7, página 282)

Mais importante, temos que implementar o código para análise e interpretação global declarações para tipos de processos e tipos de mensagens.

EXPANSÃO MACRO (Seção 12.8, página 292)

Uma das partes mais fáceis do código. A expansão da macro é obtida roteando a entrada para SPIN por meio do pré-processador C padrão antes que a análise comece.

PARA O BRAVO

Cada uma das próximas quatro seções concentra-se em uma dessas quatro extensões do pequeno pro-

espinha grama , que foi discutido acima. Essa discussão, em última análise, nos leva ao completo

Página 266

SEÇÃO 12.4
VARIÁVEIS
255

fonte do simulador de SPIN, conforme listado no Apêndice D. A discussão abaixo é principalmente destinado ao benefício daqueles que desejam expandir ou modificar o código, ou a linguagem que ele analisa. Sinta-se à vontade para pular seções ou ir diretamente para um mais seguro

parte do capítulo, como a Seção 12.9 que explica o uso geral do programa.

O corajoso que se aprofunda no código pode ocasionalmente querer se referir a este ver ou para as visualizações no início de cada seção. Para ver o código no contexto eles também podem consultar o apêndice. O Apêndice D inclui um índice de todos os código, junto com referências aos números de página que o explicam.

12.4 VARIÁVEIS

A maior parte do código necessário para a manipulação de variáveis está contido em dois arquivos. O primeiro arquivo, `sym.c` , contém a definição de um manipulador geral de tabela de símbolos.

O segundo arquivo, `vars.c` , contém o código para armazenar e manipular variáveis de os tipos de dados básicos. Não nos preocupamos muito com as variáveis do tipo `chan` ainda.

A maior parte disso virá na próxima extensão quando a passagem de mensagens for implementada. Por enquanto, vamos apenas analisar as declarações. Para implementar totalmente o outro tipos de variáveis, precisamos de alguns novos ganchos no analisador léxico, o analisador e no a rotina de avaliação da expressão. As próximas quatro subseções, então, enfocam esses extensões:

Extensões para o analisador léxico (seção 12.4.1, página 255)

Novas rotinas de tabela de símbolos (seção 12.4.2, página 257)

Extensões para o analisador (seção 12.4.3, página 260)

Novo código para o avaliador (seção 12.4.4, página 263)

Começamos dando uma olhada nas mudanças que devem ser feitas no analisador léxico.

12.4.1 ANALISADOR LÉXICO

Há uma série de novos tokens que devem ser adicionados para reconhecer nomes de variáveis e declarações na entrada para o simulador. Existem também vários operadores PROMELA que consistem em mais de um caractere e são mais bem reconhecidos no analisador léxico e convertidos em tokens. Vejamos primeiro as declarações de variáveis. Reconhecendo o cinco tipos de dados básicos e a palavra-chave `chan` produz as seguintes regras extras em `lex.l` .

```
"int" {yyval.val = INT; Token TYPE; }
"curto" {yyval.val = SHORT; Token TYPE; }
"byte" {yyval.val = BYTE; Token TYPE; }
"bool" {yyval.val = BIT; Token TYPE; }
"bit" {yyval.val = BIT; Token TYPE; }
"chan" {yyval.val = CHAN; Token TYPE; }
```

Cada declarador é passado como um token do tipo `TYPE` . O atributo é uma constante, definida em `spin.h` , que especifica a largura de cada tipo de dados em bits (exceto para `CHAN`).

Página 267

256
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
#define BIT
1
/* tipos de dados
*/
#define BYTE
8
/* largura em bits */
#define SHORT 16
#define INT
32
#define CHAN
64

O termo `token` também é definido em uma macro

```
#define Token if (! in_comment) return
```

que em combinação com as seguintes duas linhas

```
"/ *"
{in_comment = 1; }
/* /"
{in_comment = 0; }
```

garante que nenhum tokens léxico seja passado para o analisador dentro dos comentários PROMELA . Os operadores de dois caracteres são reconhecidos pelas seguintes novas regras *lex* .

```
"<<"
{Token LSHIFT; /* shift bits left */}
">>"
{Token RSHIFT; /* shift bits right */}
"<="
{ Símbolo
LE; /* menos que ou igual a */ }
"> ="
{ Símbolo
GE; /* Melhor que ou igual a */ }
"=="
{ Símbolo
EQ; /* igual a */ }
"! ="
{ Símbolo
NE; /* diferente de */ }
"&&"
{ Símbolo
E; /* lógico e */ }
"||"
{ Símbolo
OU; /* lógico ou */ }
```

A maioria dos operadores de caractere único, como '<' ou '>' ainda são passados pela última linha de a seção de regras:

```
. {Token yytext [0]; }
que basicamente não foi modificado em relação à versão usada na lombada . Uma exceção é o operador de atribuição '=' , que é convertido em um token real, denominado ASGN . Isso é também dado um atributo de valor que é definido como o número da linha na qual a atribuição foi encontrado.
```

```
"="
{yyval.val = lineno; Token ASGN; }
```

Para facilitar a depuração, tantos tokens quanto possível são marcados com um número de linha que refere-se ao arquivo de origem PROMELA . A regra *lex* acima mostra como isso funciona para a declaração de atribuição. O tratamento de tokens para nomes alfanuméricos é implementado de forma ligeiramente diferente. Para simplificar o código que *lex* precisa produzir um pouco, procuramos nomes alfanuméricos predefinidos em uma tabela estática com procedimento

```
check_name () . O procedimento é chamado de dentro da regra lex
[a-zA-Z _] [a-zA-Z_0-9] * {Token check_name (yytext); }
```

Um nome arbitrário começa com uma letra maiúscula ou minúscula e é seguido por zero ou mais letras ou dígitos. Sublinhados são permitidos em nomes. Para reconhecer as palavras-chave len , run , e de (usado em chan initializers), por exemplo, podemos escrever:

```
SEÇÃO 12.4
VARIÁVEIS
257
estrutura estática {
char * s;
tok int;
} Nomes [] = {
"len",
LEN,
"corre",
CORRE,
"do",
DO,
0,
0,
};
check_name (s)
char * s;
{
registrar int i;
para (i = 0; nomes [i] .s; i++)
if (strcmp (s, nomes [i] .s) == 0)
{
```

```

yyval.val = lineno;
retornar nomes [i] .tok;
}
yyval.sym = pesquisa (s); /* tabela de símbolos */
retornar NOME;
}

```

Nomes não reconhecidos passam para as rotinas da tabela de símbolos, todos os outros voltam imediatamente com um número de linha anexado.

Todos os novos nomes de token devem ser definidos adequadamente na especificação da gramática yacc , mas

chegaremos a isso mais tarde. Vejamos primeiro a maneira como os nomes são armazenados em a tabela de símbolos. Até agora não temos processos separados, então todos os nomes são necessariamente

global. A rotina lookup () é definida em um novo arquivo sym.c com uma tabela de símbolos manipulador.

12.4.2 MANIPULADOR DA TABELA DE SÍMBOLOS

Cada novo nome é armazenado em uma estrutura de dados do tipo Symbol que é definido no novo versão do spin.h .

```

typedef struct Symbol {
Caracteres
*nome;
tipo curto;
/* variável ou tipo de canal
*/
int
nel;
/* 1 se escalar,> 1 se matriz */
int
* val;
/* valor (es) de tempo de execução, initl 0 */
Nó de estrutura
* ini; /* valor inicial ou chan-def */
struct Symbol * next; /* lista vinculada */
} Símbolo;

```

A estrutura Symbol contém o nome completo do símbolo sendo armazenado como um ponteiro a uma sequência de caracteres. Ele também contém seu tipo e o número de elementos que estão acessando sível se o nome for definido como uma matriz. O ponteiro ini aponta para um fragmento de árvore de análise

que pode ser avaliado para inicializar um novo identificador: um inicializador de canal para mensagem canais sábios ou uma expressão inteira para variáveis dos cinco tipos de dados básicos. o ponteiro inteiro val aponta para um local onde os valores de tempo de execução de variáveis globais são armazenados.

O último elemento da estrutura, a seguir , aponta para outro símbolo. Em sua forma mais simples então, a tabela de símbolos pode ser implementada como uma única lista vinculada apontada por Símbolo * syntab .

Inicialmente, a lista está vazia:

```
syntab = (Símbolo *) 0.
```

Inserir um novo símbolo sp na lista da frente leva apenas duas atribuições em C:

```
sp-> próximo = syntab;
```

```
syntab = sp;
```

As listas vinculadas voltarão mais algumas vezes nas outras extensões que criamos. Nós usá-los, por exemplo, para implementar canais de mensagem e para armazenar referência de processo ocorrências no agendador. A Figura 12.3 ilustra como a lista vinculada é usada para o símbolo rotinas de tabela de bol.

```

syntab *
Símbolo
Próximo *
Símbolo
Próximo *
NULL
nome, val,
..etc
nome, val,
..etc

```

Figura 12.3 - Lista vinculada para a tabela de símbolos

A rotina de pesquisa completa, usando a lista vinculada, pode ser escrita da seguinte forma:

```
Símbolo *
pesquisas)
char * s;
{
Símbolo * sp;
/* verifique se o símbolo já está na lista */
para (sp = symtab; sp; sp = sp-> próximo)
if (strcmp (sp-> nome, s) == 0)
return sp;
/* Sim, ele é */
/* se não, crie um novo símbolo */
sp = (Símbolo *) emalloc (sizeof (Símbolo));
sp-> nome = (char *) emalloc (strlen (s) + 1);
strcpy (sp-> nome, s);
sp-> nel = 1;
/* escalar */
```

Página 270

SEÇÃO 12.4

VARIÁVEIS

259

```
/ * insira-o na lista */
sp-> próximo = symtab;
symtab = sp;
return sp;
}
```

A rotina `emalloc()` aloca e limpa a memória para nós, então por padrão o tipo e o valor inicial de uma nova variável é zero. A rotina de pesquisa é chamada por o analisador léxico que, conforme definido, não tem contexto suficiente para determinar o tipo de um nome da variável ou mesmo se é usado como escalar ou como vetor. Uma rotina separada `settype()` é chamado pelo analisador para inicializar o campo de tipo de variáveis quando informações foram coletadas. Na mesma verificação, também podemos ter certeza de que um array não recebeu accidentalmente uma dimensão negativa. O primeiro argumento para `settype()` é uma lista vinculada de nomes, com um ponteiro na tabela de símbolos para cada nome.

```
settype (n, t)
Nó * n;
{
enquanto (n)
{
if (n-> nsym-> tipo)
yyerror ("redeclaração de '% s'", n-> nsym-> nome);
n-> nsym-> tipo = t;
if (n-> nsym-> nel <= 0)
yyerror ("tamanho de array inválido para '% s'", n-> nsym-> nome);
n = n-> rgt;
}
}
```

O código mantém a mesma contabilidade para todas as variáveis, escalares e matrizes semelhantes. Um escalar é simplesmente uma matriz de tamanho um.

Usar apenas uma única lista vinculada para armazenar todos os nomes, no entanto, torna o analisador léxico

gastam uma quantidade desproporcional de tempo procurando nomes de variáveis na inicial para loop de rotina `lookup()`. Para cada novo nome, a rotina seria forçada a olhar para todos os nomes inseridos anteriormente, antes que possa finalmente decidir que um novo símbolo deve ser

criado. Quanto mais longa a lista, mais severa se torna a penalidade de tempo. Uma norma

A solução para esse problema é usar um esquema de consulta de *tabela hash*. Usamos o nome de o símbolo para calcular um índice único em uma matriz de tabelas de símbolos (a tabela hash), e armazene o símbolo lá. O tempo médio de pesquisa diminui linearmente com o tamanho da tabela de hash. Isso leva à seguinte versão, com `Nhash` uma constante definida em `spin.h`.

O valor de `Nhash` deve ser do tipo 2^{n-1} , com n arbitrário.

```
Símbolo * symtab [Nhash + 1];
```

Página 271

260

UM SIMULADOR DE PROTOCOLO

```

CAPÍTULO 12
hash (s)
char * s;
{
int h = 0;
enquanto (* s)
{
h += * s++;
h <<= 1;
if (h & (Nhash + 1))
h |= 1;
}
return h & Nhash;
}
Símbolo *
pesquisas)
char * s;
{
Símbolo * sp;
int h = hash (s);
para (sp = symtab [h]; sp; sp = sp-> próximo)
return sp;
/* encontrado */
para (sp = symtab [h]; sp; sp = sp-> próximo)
return sp;
/* global */
sp = (símbolo *) emalloc (sizeof (símbolo));
/* adicionar
*/
sp-> nome = (char *) emalloc (strlen (s) + 1);
strcpy (sp-> nome, s);
sp-> nel = 1;
sp-> próximo = symtab [h];
symtab [h] = sp;
return sp;
}

```

12.4.3 PARSER

Agora, vamos dar uma olhada nas extensões que devemos fazer no analisador para reconhecer variáveis capazes, declarações e os novos operadores. Primeiro, lembre-se de que um token do tipo NAME tem um atributo do tipo Symbol . Devemos, portanto, estender a definição do pilha do analisador.

```
%União{
int val;
Nó * nó;
Símbolo * sym;
}
```

Existem também alguns novos nomes de token e algumas novas regras de precedência. Aqui está o que adicionamos.

```
% token <val> LEN OF
% token <val> CONST TYPE ASGN
% token <sym> NOME
% type <sym> var ivar
% type <node> expr var_list
% type <node> args arg typ_list
%direito
ASGN
%esquerda
OU
%esquerda
E
%esquerda
' | '
%esquerda
' & '
%esquerda
EQ NE
%esquerda
'>' '<' GE LE
%esquerda
LSHIFT RSHIFT
%esquerda
'+' '-'
%esquerda
```

```
'*' '/' '%'
%esquerda
'~' UMIN NEG
```

O operador de atribuição `ASGN` é associativo à direita e obtém a menor precedência de todos os operadores. O novo token do tipo `NAME` tem um atributo de símbolo apontando para um slot em a tabela de símbolos.

Uma sequência de zero ou mais declarações de variáveis é analisada da seguinte forma:

```
any_decl: /* vazio */
{$$ = (Nó *) 0; }
| one_decl ';' any_decl {$$ = nn (0, 0, ',', $ 1, $ 3); }
;
one_decl: TYPE var_list
{settype ($ 2, $ 1); $$ = $ 2; }
```

Assim que uma declaração de variável completa é reconhecida, na última produção acima, a rotina `settype ()` é chamada para armazenar as informações extras de tipo no símbolo tabela. Antes de olharmos para a definição de uma `var_list`, observe que um nome de variável pode ser seguido por um índice de matriz. As variáveis são, portanto, definidas na gramática como um `var` não terminal, do tipo `sym`.

```
var
: NOME
{$ 1-> nel = 1; $$ = $ 1; }
| NOME '[' CONST ']'
{$ 1-> nel = $ 3; $$ = $ 1; }
```

Por enquanto, apenas lembramos o tamanho do array especificado e verificamos seu valor mais tarde, quando um procedimento `settype ()` for chamado. Uma variável também pode ter uma inicialização de zação. Uma variável inicializada pode ser definida como um `ivar` não terminal, como segue baixos:

Página 273

262

UM SIMULADOR DE PROTOCOLO

CAPÍTULO 12

```
ivar
: var
{$$ = $ 1; }
| var ASGN expr
{$ 1-> ini = $ 3; $$ = $ 1; }
| var ASGN ch_init
{$ 1-> ini = $ 3; $$ = $ 1; }
;
ch_init: '[' CONST ']' OF '{' typ_list '}'
{if ($ 2) u_async++; else u_sync++;
cnt_mpars ($ 6);
$$ = nn (0, $ 2, CHAN, 0, $ 6);
}
;
```

O inicializador pode ser uma expressão que retorna um valor ou uma especificação de canal. As regras de produção acima permitem ambos. Algumas estatísticas são reunidas sobre o número de parâmetros de mensagem usados e o número de canais síncronos e assíncronos nels que são declarados. A lista de tipos de dados em um inicializador de canal também é rápida definiram.

```
typ_list: TYPE
{$$ = nn (0, 0, $ 1, 0, 0); }
| TYPE ',' typ_list
{$$ = nn (0, 0, $ 1, 0, $ 3); }
;
```

A verificação de que, por exemplo, um canal não foi inicializado com uma expressão pode ser colocado no código que executa as inicializações reais. O conjunto de regras de produção que lida com declarações de variáveis pode ser concluído definindo

```
var_list: ivar
{$$ = nn ($ 1, 0, TIPO, 0, 0); }
| ivar ',' var_list
{$$ = nn ($ 1, 0, TIPO, 0, $ 3); }
;
```

A rotina de alocação de nó `nn ()` deve agora também lidar com a nova referência de tabela de símbolos cias. É estendido da seguinte forma:

```
Nó *
nn (s, v, t, l, r)
Símbolo * s;
```

```

Nó * l, * r;
{
Nó * n = (Nó *) emalloc (sizeof (Nó));
n-> nval = v;
n-> ntyp = t;
n-> nsym = s;
n-> fname = Fname;
n-> lft = l;
n-> rgt = r;
return n;
}

```

Em seguida, temos que nos preparar para analisar os novos operadores que adicionamos. A maior parte é direto. A série de regras de produção de expressões apenas cresce um pouco.

Página 274

```

SEÇÃO 12.4
VARIÁVEIS
263
expr
: '(' expr ')'
{ $$ = $ 2; }
| expr '+' expr
{ $$ = nn (0, 0, '+', $ 1, $ 3); }
| expr '-' expr
{ $$ = nn (0, 0, '-', $ 1, $ 3); }
...
| expr AND expr
{ $$ = nn (0, 0, AND, $ 1, $ 3); }
| expr OU expr
{ $$ = nn (0, 0, OU, $ 1, $ 3); }
...
| expr LSHIFT expr
{ $$ = nn (0, 0, LSHIFT, $ 1, $ 3); }
| expr RSHIFT expr
{ $$ = nn (0, 0, RSHIFT, $ 1, $ 3); }
| '~' expr
{ $$ = nn (0, 0, '~', $ 2, 0); }
| '-' expr% prec UMIN { $$ = nn (0, 0, UMIN, $ 2, 0); }
| '!' expr% prec NEG
{ $$ = nn (0, 0, '!', $ 2, 0); }
| LEN '(' varref ')'
{ $$ = nn ($ 3-> nsym, $ 1, LEN, $ 3, 0); }
| varref
{ $$ = $ 1; }
| CONST
{ $$ = nn (0, $ 1, CONST, 0, 0); }
;
```

Talvez as adições mais interessantes sejam os novos tipos de expressões para o bit operações unárias sábias, aritméticas e lógicas: complemento bit a bit '`~`', menos unário e negação booleana. Uma vez que o operador menos também pode ser usado como um operador binário temos que definir sua precedência explicitamente com a palavra-chave `yacc % prec`. E claro, o operador de negação '`!`' irá dobrar em uma das próximas extensões como um envio binário operador, então sua precedência nesta regra gramatical também é definida explicitamente. Na versão final sessão, substituímos o token de caractere '`!`' com um token `SND`, que é novamente rotulado com um número de linha para melhorar o relatório de erros.

As referências de variáveis, usadas nas duas últimas regras de produção, são definidas da seguinte forma:

```

varref: NAME
{ $$ = nn ($ 1, 0, NOME, 0, 0); }
| NOME '[' expr ']'
{ $$ = nn ($ 1, 0, NOME, $ 3, 0); }
;
```

Existem apenas dois tipos de referências de variáveis, uma para escalares e outra para matrizes. Ambos retornam um nó do tipo `NOME` com o primeiro campo referindo-se ao símbolo que define o nome da variável. O ponteiro esquerdo especifica o índice opcional da matriz. Um nulo ponteiro no lugar de um índice avalia trivialmente como zero. Deve produzir um erro se tentamos determinar o "comprimento" de qualquer coisa que não seja uma variável de canal. o analisador, no entanto, não verifica isso.

12.4.4 AVALIADOR

A última rotina que devemos olhar para a adição de variáveis é o código do avaliador em `run.c`. Felizmente, a extensão é quase trivial. Existem apenas alguns casos novos em o interruptor. Por exemplo:

```

264
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
switch (agora-> ntyp) {
...
caso
NE: return (eval (now-> lft) != Eval (now-> rgt));
caso
EQ: return (eval (now-> lft) == eval (now-> rgt));
caso
OU: return (eval (now-> lft) || eval (now-> rgt));
caso
AND: return (eval (now-> lft) && eval (now-> rgt));
case LSHIFT: return (eval (now-> lft) << eval (now-> rgt));
case RSHIFT: return (eval (now-> lft) >> eval (now-> rgt));
...
case ASGN: return setval (now-> lft, eval (now-> rgt));
case NAME: return getval (now-> nsym, eval (now-> lft));
...

```

Os únicos novos casos interessantes são as referências de variáveis nas duas últimas linhas acima. Eles são implementados com duas chamadas de procedimento: `getval ()` e `setval ()`. Para processar uma atribuição, primeiro o valor a ser atribuído é determinado avaliando a expressão apontada via `now-> rgt`. Em seguida, a variável de destino `v`, possivelmente, seu índice, deve ser encontrado. Todas as informações estão disponíveis por meio do ponteiro esquerdo, que é passado para `setval ()`. Um nó do tipo `NAME` contém um ponteiro para a tabela de símbolos, com o nome da variável alfanumérica e contém um índice para referências de matriz ou um ponteiro nulo para escalares. As rotinas `getval ()` e `setval ()` são definidas em `vars.c`. Para variáveis globais, o único tipo de variável que temos até agora, a rotina `setval ()` passa o trabalho para a rotina `setglobal ()`, que verifica o índice e se necessário aloca memória para as variáveis.

```

setglobal (v, m)
Nó * v;
{
int n = eval (v-> lft);
if (checkvar (v-> nsym, n))
v-> nsym-> val [n] = m;
return 1;
}
com checkvar () definido da seguinte forma:
checkvar (s, n)
Símbolo * s;
{
int i;
if (n>= s-> nel || n <0)
{
yyerror ("erro de indexação de array, '%s'", s-> nome);
return 0;
}
if (s-> type == 0)
{
yyerror ("undecl var '%s' (assumindo int)", s-> nome);
s-> tipo = INT;
}

```

SEÇÃO 12.5
AFIRMAÇÕES

```

265
if (s-> val == (int *) 0)
/* não inicializado */
{
s-> val = (int *) emalloc (s-> nel * sizeof (int));
para (i = 0; i <s-> nel; i++)
{
if (s-> digite! = CHAN)
s-> val [i] = eval (s-> ini);
outro
s-> val [i] = qmake (s);
}
}
return 1;
}

```

Uma variável simples é inicializada avaliando a expressão no campo de inicialização de

o símbolo. Um ponteiro nulo neste campo, novamente, avalia trivialmente para a inicial padrão valor zero. Uma variável de canal é inicializada passando o ponteiro de inicialização para um rotina `qmake ()` que é discutida na próxima seção. Um conflito de tipo entre o inicializador e a variável aciona um erro no avaliador ou na rotina de construção do canal.

A rotina `getval ()` transfere para `getglobal ()`.

```
getglobal (s, n)
Símbolo * s;
{
if (checkvar (s, n))
retornar cast_val (s-> tipo, s-> val [n]);
return 0;
}
```

Um resultado zero é devolvido se o `checkvar` rotina falhar, isto é, se um erro de indexação foi detectou. A rotina `cast_val ()` interpreta o tipo da variável e as máscaras ou projeta valores de variáveis de acordo.

```
cast_val (t, v)
{
int i = 0; s curto = 0; caractere sem sinal u = 0;
if (t == INT || t == CHAN) i = v;
else if (t == SHORT) s = (short) v;
else if (t == BYTE) u = (unsigned char) v;
senão if (t == BIT) u = (unsigned char) (v & 1);
if (v! = i + s + u)
yyerror ("valor% d truncado na atribuição", v);
return (int) (i + s + u);
}
```

É considerado um erro se um valor mudar devido à conversão de tipo. Variáveis são apenas lançadas com o valor correto quando são lidos. Internamente, todos os valores são armazenados em 32 bits inteiros.

12.5 DECLARAÇÕES

Adicionar declarações é relativamente fácil neste ponto. Discutimos quatro conjuntos de extensões:

Extensões para o analisador léxico (seção 12.5.1, página 266)

Extensões para o analisador (seção 12.5.2, página 266)

Extensões para as rotinas de avaliação (seção 12.5.3, página 267)

Página 277

266

UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

A implementação da passagem de mensagens (seção 12.5.4, página 268)

Para implementar declarações passagem de mensagens que adicionar um novo arquivo `mesg.c`. Começamos por

adicionar cinco tipos de instruções: condições booleanas, tempo limite s, atribuições, `printf`, e declarações `assert`, mais a pseudo-declaração `skip`.

12.5.1 ANALISADOR LÉXICO

Três das declarações, `assert`, `printf`, e `tempo limite`, produzir novas entradas no tabela de pesquisa estática de nomes alfanuméricos no analisador léxico.

```
estrutura estática {
char * s;
tok int;
} Nomes [] = {
"afirmar",
AFIRMAR,
"printf",
IMPRESSÃO,
"tempo esgotado",
TEMPO ESGOTADO,
...
};
```

O número da linha anexado aos tokens pela rotina `checkname ()` nos permite pro-dar o feedback certo para um usuário quando uma declaração `assert` falhar durante uma simulação corre.

A pseudo instrução `skip` é traduzida na constante equivalente com a regra `lex`

```
"pular"
{yyval.val = 1; return CONST; }
```

Para implementar a instrução `printf` corretamente, devemos definir strings. O único lugar onde argumentos de string são usados está em `printf`, então podemos tratá-lo como um token especial com um atributo de símbolo. O ponteiro do símbolo mantém a string, ao invés do

```

nome variável. Isso produz mais uma regra lex .
\ ". * \
{yyval.sym = lookup (yytext); retornar STRING; }
Finalmente, adicionamos uma regra para traduzir setas em ponto-e-vírgulas.
"->
{ Retorna ';' ; / * separador de instrução * / }

```

12.5.2 PARSER

Para atualizar o analisador, devemos adicionar algumas novas regras de produção novamente para analisar o estado-mentos. Até agora, uma declaração pode ser uma atribuição, uma declaração impressa, uma declaração, um

salto, ou uma expressão (uma condição). O tempo limite é implementado como um especial variável predefinida que pode fazer parte de uma condição. É adicionado no código para análise expressões.

```

stmtnt: varref ASGN expr
{$$ = nn ($ 1-> nsym, $ 2, ASGN, $ 1, $ 3); }
| PRINT '(' STRING prargs ')' {$$ = nn ($ 3, $ 1, PRINT, $ 4, 0); }
| ASSERT expr
{$$ = nn (0, $ 1, ASSERT, $ 2, 0); }
| GOTO NAME
{$$ = nn ($ 2, $ 1, GOTO, 0, 0); }
| expr
{$$ = nn (0, lineno, 'c', $ 1, 0); }
...
expr
: TEMPO ESGOTADO
{$$ = nn (0, $ 1, TEMPO LIMITE, 0, 0); }
...

```

SEÇÃO 12.5
AFIRMAÇÕES
267

Fizemos uma regra de produção separada para a análise de referências de variáveis, chamada varref . Ele pode ser usado em mais alguns casos posteriormente.

Uma sequência de declarações é definida da seguinte forma:

```

sequência: passo
{add_seq ($ 1); }
| seqüência ';' degrau
{add_seq ($ 3); }
;
degrau
: any_decl stmtnt
{$$ = $ 2; }
;

```

Não há necessidade de agrupar as declarações no início de um corpo do programa na PROMELA , então nas regras acima, permitimos que cada declaração seja precedida por um ou mais declarações. A rotina add_seq () é definida em flow.c para marcar as instruções em um lista encadeada, mas veremos isso com mais detalhes na Seção 12.6. Por enquanto, um programa corpo é apenas uma sequência de instruções, que é analisada da seguinte forma:

```

corpo
: '{'
{open_seq (1); }
seqüência
{add_seq (parar); }
'}'
{$$ = close_seq (); }
;
```

A primeira chave aberta inicia uma nova lista vinculada para armazenar as instruções por meio de uma chamada na rotina

open_seq () . Quando uma sequência completa é reconhecida, um nó de parada especial é marcado no final e a sequência é fechada e passada para a árvore de análise. Nós olhamos para as rotinas para manipular as sequências em flow.c com mais detalhes posteriormente, quando discutir declarações compostas. Vamos primeiro agora considerar as adições que temos que make no código do interpretador para avaliar as instruções adicionadas até agora.

12.5.3 AVALIADOR

As adições ainda são modestas. O intérprete só precisa lidar com esses novos nós tipos.

```

case TIMEOUT: return Tval;
caso
```

```
'c': avaliação de retorno (agora-> lft); /* doença */
case PRINT: return interpret (agora);
case ASSERT: if (eval (now-> lft)) return 1;
yyerror ("assertion violated", (char *) 0);
embrulhar(); saída (1);
```

Timeouts são um recurso de modelagem do PROMELA ; eles são implementados como um teste em um variável predefinida aqui. No simulador final, o planejador pode ativar explicitamente ou desative os eventos de tempo limite para testar se o protocolo pode se recuperar de condições de exceção. As condições, identificadas pelo tipo de nó interno 'c' , são avaliadas recursivamente e retornar um status zero ou diferente de zero. A declaração assert é interpretada diretamente e causa uma saída de erro se falhar, imprimindo o número da linha que foi devidamente transportada como um atributo token. Mudamos os detalhes sujos da implementação da impressão instruções em um procedimento separado interpret () . Pode ser definido da seguinte forma:

Página 279

```
268
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
interpret (n)
Nó * n;
{
Nó * tmp = n-> lft;
char c, * s = n-> nsym-> nome;
int i, j;
para (i = 0; i < strlen (s); i++)
switch (s [i]) {
padrão: putchar (s [i]); quebrar;
case '\\': break; /* ignorar */
case '\\\\':
switch (s [++ i]) {
case 't': putchar ('\ t'); quebrar;
case 'n': putchar ('\ n'); quebrar;
padrão: putchar (s [i]); quebrar;
}
quebrar;
caso '%':
if ((c = s [++ i]) == '%')
{
putchar ('%'); /* literal */
quebrar;
}
if (! tmp)
{
yyerror ("muito poucos argumentos de impressão% s", s);
quebrar;
}
j = eval (tmp-> lft);
tmp = tmp-> rgt;
switch (c) {
case 'c': printf ("% c", j); quebrar;
case 'd': printf ("% d", j); quebrar;
case 'o': printf ("% o", j); quebrar;
case 'u': printf ("% u", j); quebrar;
case 'x': printf ("% x", j); quebrar;
padrão: yyerror ("unrecognized print cmd %% '% c'", c);
quebrar;
}
quebrar;
}
fflush (stdout);
return 1;
}
```

que reconhece um número modesto de conversões da função da biblioteca UNIX
printf () .

12.5.4 IMPLEMENTAÇÃO DE PASSAGEM DE MENSAGEM

As primitivas de passagem de mensagem síncrona e assíncrona são boas para outra duas a trezentas linhas de texto de origem. A parte mais fácil é a extensão do parser para passar as novas instruções ao interpretador.

O envio e o recebimento têm uma prioridade de avaliação relativamente baixa, equivalente a tarefa. Três novos tipos de declarações são adicionados.

SEÇÃO 12.5
AFIRMAÇÕES
269

```
stmtt: ...
| varref RCV margs
{$$ = nn ($ 1-> nsym, $ 2, 'r', $ 1, $ 3); }
| varref SND margs
{$$ = nn ($ 1-> nsym, $ 2, 's', $ 1, $ 3); }
```

com argumentos de mensagem definidos da seguinte maneira:

```
margs: arg
{$$ = $ 1; }
| expr '(' arg ')'
{$$ = nn (0, 0, ',', $ 1, $ 3); }
;
arg
: expr
{$$ = nn (0, 0, ',', $ 1, 0); }
| expr ',' arg
{$$ = nn (0, 0, ',', $ 1, $ 3); }
;
```

Os argumentos de uma operação de recepção só podem ser constantes ou nomes, mas é mais fácil para verificar essa parte da sintaxe posteriormente. Também existe um novo tipo de expressão

```
expr
: ...
| varref RCV '[' margs ']' {$$ = nn ($ 1-> nsym, $ 2, 'R', $ 1, $ 4); }
```

que corresponde a um teste livre de efeitos colaterais da executabilidade de uma instrução de recebimento (veja o Capítulo 5).

O interpretador agora tem três cláusulas extras para os novos tipos de nós internos 'r' , 's' , e 'R' . Isto corresponde ao seguinte código no run.c .

```
case LEN:
return qlen (agora);
case 's':
return qsend (agora);
case 'r':
return qrecv (agora, 1); /* full-receive */
case 'R':
return qrecv (agora, 0); /* teste apenas
*/
```

Os três procedimentos chamados aqui, juntamente com o procedimento para a inicialização do novos canais qmake () mencionou de passagem antes, são expandidos no arquivo mesg.c .

Vejamos primeiro a implementação de qmake () .

As descrições dos canais de mensagens são armazenadas em uma lista vinculada de estruturas de tipo de fila , com a seguinte definição de spin.h .

```
typedef struct Queue {
qid curto;
/* índice q de tempo de execução
*/
glen curto;
/* nr mensagens armazenadas */
nslots curtos, nflds; /* capacidade, flds / slot */
short * fld_width;
/* tipo de cada campo */
int
*conteúdo;
/* o buffer real
*/
struct Queue
* nxt; /* lista vinculada */
} Fila;
```

Em mesg.c , o topo da lista é definido assim.

```
Fila * qtab = (Fila *) 0;
/* lista vinculada */
nqs int = 0;
/* número de filas */
```

O resto é fácil. Damos aos canais de comprimento zero (encontro) um espaço para temporariamente segure uma mensagem conforme ela é passada de um remetente para um receptor.

```

{
Nó * m;
Fila * q;
int i; análise interna externa;
if (! s-> ini)
return 0;
if (s-> ini-> ntyp! = CHAN)
fatal ("inicializador de canal inválido para% s \ n", s-> nome);
if (nqs> = MAXQ)
fatal ("muitas filas (% s)", s-> nome);
q = (Fila *) emalloc (sizeof (Fila));
q-> qid = ++ nqs;
q-> nslots = s-> ini-> nval;
para (m = s-> ini-> rgt; m; m = m-> rgt)
q-> nflds++;
i = max (1, q-> nslots); / * 0-slot qs obtém 1 slot mínimo * /
q-> conteúdo = (int *) emalloc (q-> nflds * i * sizeof (int));
q-> fld_width = (short *) emalloc (q-> nflds * sizeof (short));
para (m = s-> ini-> rgt, i = 0; m; m = m-> rgt)
q-> fld_width [i ++] = m-> ntyp;
q-> nxt = qtab;
qtab = q;
ltab [q-> qid-1] = q;
return q-> qid;
}

```

Claro, um canal só pode ser criado se um inicializador for fornecido. É um erro fatal se o inicializador tiver o tipo errado ou se já existirem muitos canais. Para eficiência apenas, um índice para todos os canais ativos também é mantido em uma lista linear chamado ltab . Implementar qlen () agora é simples.

```

qlen (n)
Nó * n;
{
int qualq = eval (n-> lft) -1;
if (qualq <MAXQ && qualq> = 0 && ltab [qualq])
return ltab [whichq] -> qlen;
return 0;
}

```

Verificamos se o índice calculado está dentro da faixa correta, que o correspondente a fila existe e retorna o campo de comprimento da estrutura de dados correspondente. o a parte difícil permanece: a implementação de versões síncronas e assíncronas de qsend () e qrecv () . Usamos uma interface simples para decidir qual rotina usar.

SEÇÃO 12.5
AFIRMAÇÕES
271

```

qsend (n)
Nó * n;
{
int qualq = eval (n-> lft) -1;
if (whichq == -1)
{
printf ("Erro: envio para um canal não inicializado \ n");
qualq = 0;
}
if (qualq <MAXQ && qualq> = 0 && ltab [qualq])
{
if (ltab [whichq] -> nslots> 0)
retornar a_snd (ltab [whichq], n);
outro
retornar s_snd (ltab [qualq], n);
}
return 0;
}
qrecv (n, completo)
Nó * n;
{
int qualq = eval (n-> lft) -1;
if (whichq == -1)
{
printf ("Erro: recebendo de um canal não inicializado \ n");
qualq = 0;
}
if (qualq <MAXQ && qualq> = 0 && ltab [qualq])
retornar a_rcv (ltab [whichq], n, completo);
return 0;
}

```

Um encontro é acionado pelo remetente de uma mensagem. Se, naquele momento, pelo menos um processo for bloqueado na operação de recepção correspondente, um encontro pode ocorrer. A operação de recepção rendezvous, portanto, pode ser a mesma tanto para síncrona como operações assíncronas. Primeiro, consideramos o caso assíncrono. Sua forma básica parece o seguinte (a versão no Apêndice D contém mais alguns recursos que não são relevantes aqui):

```
a_snd (q, n)
Fila * q;
Nó * n;
{
Nó * m;
int i = q-> qlen * q-> nflds;
/* q offset */
int j = 0;
/* q campo # */
if (q-> nslots > 0 && q-> qlen >= q-> nslots)
return 0;
/* q está cheio */
para (m = n-> rgt; m && j <q-> nflds; m = m-> rgt, j++)
q-> conteúdo [i + j] = cast_val (q-> fld_width [j], eval (m-> lft));
```

Página 283

272

UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

```
q-> qlen++;
return 1;
}
```

Se o canal estiver cheio, um status zero é retornado, o que significa que a instrução é atualmente não executável. Se houver pelo menos um slot livre na fila, os campos são copiado e mascarado com as larguras de campo predefinidas. A operação de recebimento é um pouco mais envolvida.

```
a_rcv (q, n, completo)
Fila * q;
Nó * n;
{
Nó * m;
int j, k;
if (q-> qlen == 0) retorna 0;
/* q está vazio */
para (m = n-> rgt, j = 0; m && j <q-> nflds; m = m-> rgt, j++)
{
if (m-> lft-> ntyp == CONST
&& q-> conteúdo [j] != m-> lft-> nval)
return 0;
/* sem correspondência */
}
para (m = n-> rgt, j = 0; j <q-> nflds; m = (m)? m-> rgt: m, j++)
{
if (m && m-> lft-> ntyp == NOME)
setval (m-> lft, q-> conteúdo [j]);
para (k = 0; completo && k <q-> qlen-1; k++)
q-> conteúdo [k * q-> nflds + j] =
q-> conteúdo [(k + 1) * q-> nflds + j];
}
if (completo) q-> qlen--;
return 1;
}
```

O argumento `full` é zero quando a operação de recepção é usada como uma condição, como em `qname? [ack]`, e diferente de zero caso contrário. No primeiro caso, o procedimento não tem lado efeitos e apenas retorna o status de executabilidade do recebimento. Existem dois loops no procedimento. O primeiro verifica se todos os parâmetros de mensagem que são declaradas como constantes são correspondidas corretamente e verifica se os outros parâmetros são nomes de variáveis. O segundo ciclo copia os dados da fila para as variáveis especificado e, conforme necessário, desloca os campos de mensagem um slot.

Apenas a versão síncrona da operação de envio permanece para ser expandida.

```
s_snd (q, n)
Fila * q;
Nó * n;
{
Nó * m;
int i, j = 0; /* q campo # */
```

SEÇÃO 12.5
AFIRMAÇÕES
273

```
para (m = n-> rgt; m && j <q-> nflds; m = m-> rgt, j++)
q-> conteúdo [j] = cast_val (q-> fld_width [j], eval (m-> lft));
q-> qlen = 1;
if (complete_rendez ())
return 1;
q-> qlen = 0;
return 0;
}
```

O remetente primeiro anexa a mensagem e, em seguida, verifica se há um destinatário que pode executar o recebimento correspondente. Não há necessidade de verificar o comprimento da fila aqui: quando

a operação de envio é executável, todos os canais de encontro têm a garantia de estar vazios. Se nenhum recebimento correspondente for encontrado, a operação de envio falha, cancela a mensagem no fila e retorna um zero. Se houver uma operação de recebimento correspondente, ele será executado antes que a rotina `complete_rendez ()` retorne, e tanto o remetente quanto o receptor prossiga para a próxima instrução, deixando novamente o canal vazio. A rotina `complete_rendez ()` é na verdade uma pequena parte do agendador e está listado em `sched.c`. Ele bloqueia todas as instruções, exceto um recebimento síncrono e o código que é necessário para avaliar expressões.

```
complete_rendez ()
{
RunList * orun = X;
Elemento * e;
int res = 0;
Rvous = 1;
para (X = executar; X; X = X-> nxt)
if (X! = orun && (e = eval_sub (X-> pc)))
{
X-> pc = e;
res = 1;
quebrar;
}
Rvous = 0;
X = orun;
return res;
}
```

A rotina primeiro define um sinalizador global `Rvous` para garantir que apenas as operações de recebimento sejam ativado. A variável `x` é então apontada para cada processo executável para verificar se ele pode concluir o handshake de encontro. A rotina de avaliação que analisa cada

a opção de um composto para uma possível combinação é chamada `eval_sub ()`.

```
Elemento *
eval_sub (e)
Elemento * e;
{
Elemento * f, * g;
SeqList * z;
int i, j, k;
```

274
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

```
...
if (e-> sub)
{
para (z = e-> sub, j = 0; z; z = z-> nxt)
j++;
k = rand ()% j; /* não determinismo */
para (i = 0, z = e-> sub; i <j + k; i++)
{
if (i>= k && f = eval_sub (z-> this-> primeiro))
return f;
z = (z-> nxt)? z-> nxt: e-> sub;
}
} outro
{
if (e-> n-> ntyp == ATOMIC)
```

```

{
...
} else if (Rvous)
{
if (eval_sync (e-> n))
return e-> nxt;
} outro
return (eval (e-> n)) ? e-> nxt: (Element *) 0;
}
return (Element *) 0;
}

```

A rotina de avaliação pesquisa recursivamente nas opções de compostos. E se há mais de uma opção, o agendador escolhe uma aleatoriamente, usando a biblioteca rand de rotina () . Quando, no nível mais baixo da recursão, ele encontra uma instrução de um composto para avaliar, ele verifica o valor de Rvous e chama essa rotina quando está definido.

```

eval_sync (agora)
Nó * agora;
{
/* permitir apenas recebimentos sincronos
/* e tipos de nós relacionados
*/
se (agora)
switch (agora-> ntyp) {
case TIMEOUT: case PRINT:
case ASSERT:
case RUN:
case LEN:
case 's':
case 'c':
caso ASGN:
case BREAK:
case IF:
case DO:
case '.':
return 0;
case 'R':
case 'r':
if (! q_is_sync (agora))
return 0;
}
return eval (agora);
}

```

Voltaremos aos detalhes mais tarde, quando olharmos mais de perto o programador o próprio código.

Página 286

SEÇÃO 12.6
CONTROLE DE FLUXO
275

12.6 FLUXO DE CONTROLE

Nosso próximo trabalho é trazer alguma estrutura para a linguagem, implementando o selecionamento de repetição, repetição, pausa , goto e atômicas . Temos que construir um programa como um conjunto coerente de declarações, com uma disciplina de fluxo de controle que define qual estado

mento desse conjunto deve ser avaliado pelo agendador em cada etapa de execução. All routines que tratam explicitamente com o fluxo de controle são colocados no arquivo flow.c . Este seção discute:

Código para manipular sequências (seção 12.6.1, página 275)

Acompanhar os rótulos (seção 12.6.2, página 278)

Código para análise de declarações compostas (seção 12.6.3, página 279)

12.6.1 SEQUÊNCIAS

Lembre-se da definição do não-terminal programa em spin.y , que discutimos antes.

```

corpo:
'{'
{open_seq (1); }
seqüência
{add_seq (parar); }
'}'
{$$ = close_seq (); }
seqüência:
degrau

```

```

{add_seq ($ 1); }
| seqüência ';' degrau
{add_seq ($ 3); }

```

Um corpo de programa é uma sequência de instruções terminadas por um nó de parada especial . Mas desta vez, temos que carregar algumas informações extras para instruções compostas.

Uma instrução composta é basicamente uma bifurcação na sequência de execução, onde um sequência se divide em várias sequências de opções. Nós armazenamos cada indivíduo sequência de uma estrutura de tipo de sequência definida em spin.h .

```

typedef struct Sequence {
    Elemento * primeiro;
    Elemento * último;
} Seqüência;

```

Um conjunto de sequências é armazenado como uma lista vinculada, da seguinte maneira:

```

typedef struct SeqList {
    Seqüência
    *esta; /* uma sequência */
    struct SeqList * nxt; /* lista vinculada */
} SeqList;

```

E, claro, os nós da árvore de análise terão que acomodar as novas informações , então a estrutura de dados de um Node é expandida um pouco mais.

Página 287

```

276
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
typedef struct Node {
    int
    nval;
    /* atributo de valor
     */
    short ntyp;
    /* tipo de nó
     */
    Símbolo * nsym;
    /* novo atributo
     */
    Símbolo * fname;
    /* nome do arquivo de src
     */
    struct SeqList * seql; /* lista de sequências
     */
    Nó de estrutura
    * lft, * rgt; /* filhos na árvore de análise */
} Nó;

```

As instruções são adicionadas uma a uma a uma sequência com add_seq () . As declarações são mais convenientemente armazenado em estruturas do tipo Elemento . A definição em spin.h parece o seguinte:

```

Elemento typedef struct {
    Nó
    * n;
    /* define o tipo e conteúdo */
    int
    seqno;
    /* identifica exclusivamente este el */
    status de char unsigned; /* usado pelo gerador do analisador */
    struct SeqList * sub; /* subsequências, para compostos */
    Elemento struct * nxt; /* lista vinculada */
} Elemento;

```

Cada elemento em uma sequência é rotulado com um número de sequência (ou estado) único, que será útil na construção do validador no próximo capítulo. Os números são

entregue pela rotina new_el () .

```

Elemento *
new_el (n)
Nó * n;
{
    Elemento * m;
    if (n && (n-> ntyp == IF || n-> ntyp == DO))
        retornar if_seq (n-> seql, n-> ntyp, n-> nval);
    m = (Elemento *) emalloc (sizeof (Elemento));
    m-> n = n;
    m-> seqno = Elcnt++;
    return m;
}

```

O subcampo de um elemento aponta para as opções de um composto. Está definido, apenas para

esses tipos de instruções, na rotina `if_seq ()`, que é examinada em detalhes na Seção 12.6.3.

Os números de sequência são mantidos em um contador global `Elcnt` que é redefinido para um no início de cada novo processo. No código abaixo, isso acontece quando `open_seq ()` é chamado com um argumento diferente de zero.

A chave aberta de um `corpo` inicializa uma nova sequência por um procedimento de chamada `open_seq ()`. A chave de fechamento fecha a sequência e a passa por meio de `close_seq ()`. Inicializar uma nova sequência de elementos ou retornar uma completa é bastante simples frente. Em sua forma mais simples, é assim.

Página 288

```
SEÇÃO 12.6
CONTROLE DE FLUXO
277
vazio
open_seq (topo)
{
SeqList * t;
Sequência * s = (Sequência *) emalloc (sizeof (Sequência));
t = seqlist (s, cur_s);
cur_s = t;
if (topo) Elcnt = 1;
}
Seqüência *
close_seq ()
{
Sequência * s = cur_s-> this;
cur_s = cur_s-> nxt;
return s;
}
```

A listagem no Apêndice D realiza algumas verificações extras que são relevantes apenas para o validador.

A rotina `seqlist ()` anexa uma nova sequência a uma lista vinculada de sequências.

```
SeqList *
seqlist (s, r)
Sequência * s;
SeqList * r;
{
SeqList * t = (SeqList *) emalloc (sizeof (SeqList));
t-> this = s;
t-> nxt = r;
return t;
}
```

Adicionar um elemento à sequência atual acontece da seguinte maneira:

```
add_seq (n)
Nó * n;
{
Elemento * e;
if (! n) return;
mais interno = n;
e = dois pontos (n);
if (mais interno-> ntyp! = IF && mais interno-> ntyp! = DO)
add_el (e, cur_s-> this);
}
```

A rotina que aloca memória para um novo elemento `new_el ()` filtra o `comp` libere declarações e faz todo o trabalho duro para elas, para que não precisem ser repetidas acima. A rotina `add_el ()` que é usada aqui não é muito interessante.

Página 289

```
278
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
add_el (e, s)
Elemento * e;
Sequência * s;
{
if (! s-> primeiro)
s-> primeiro = e;
outro
s-> último-> nxt = e;
s-> último = e;
}
```

12.6.2 SALTOS E ETIQUETAS

A rotina `add_seq()` também captura rótulos, identificados por um tipo de nó `'.'`, e lembra-os em outra lista vinculada.

```
typedef struct Label {
    Símbolo * s;
    Símbolo * c;
    Elemento * e;
    etiqueta de estrutura
    * nxt;
} Rótulo;

O código é novamente direto.

set_lab (s, e)
Símbolo * s;
Elemento * e;
{
    Rótulo * l; contexto externo do símbolo *
    if (!s) return;
    l = (Label *) emalloc (sizeof (Label));
    l-> s = s;
    l-> c = contexto;
    l-> e = e;
    l-> nxt = labtab;
    labtab = l;
}
```

Quando o planejador tem que determinar o destino de um salto `goto`, ele consulta que lista e recupera um ponteiro para o elemento que carregou o rótulo.

```
Elemento *
get_lab (s)
Símbolo * s;
{
    Rótulo * l;
    para (l = labtab; l; l = l-> nxt)
    if (s == l-> s)
        retorno (l-> e);
    fatal ("rótulo indefinido% s", s-> nome);
    return 0;
/* não chega aqui */
}
```

A rotina é chamada no início de cada invocação da rotina de avaliação genérica

Página 290

SEÇÃO 12.6
CONTROLE DE FLUXO
279

`eval_sub()`, da seguinte maneira:

```
if (e-> n-> ntyp == GOTO)
    return get_lab (e-> n-> nsym);
```

Claro, para obter as instruções e rótulos `goto` nos lugares certos na árvore de análise, devemos adicionar mais algumas regras aos arquivos `lex` e `yacc`. A tabela de pesquisa em `lex.l` é expandido com

"vamos para",

VAMOS PARA,

enquanto a adição de um símbolo `GOTO` para `spin.y`. As duas regras de produção que reconhecem saltos e os rótulos são:

```
| GOTO NAME
{$$ = nn ($ 2, $ 1, GOTO, 0, 0); }
| NOME ':' stmnt
{$$ = nn ($ 1, $ 3-> nval, ':', $ 3, 0); }
```

O número da linha para uma instrução rotulada é extraído do nó que é passado através do terceiro parâmetro `$ 3`. O token `GOTO` carrega seu próprio número de linha que é copiado de `$ 1`.

12.6.3 DECLARAÇÕES DE COMPOSTO

O verdadeiro desafio é processar as declarações compostas. Existem alguns novos tokens para ser manuseado, como `se`, `fi`, `fazer`, `od`, `:::`, `pausa`, e `atômica`. O léxico o analisador é novamente estendido com uma linha para cada. Três novos tipos de instrução e dois novas regras de produção são adicionados às regras de produção em `spin.y`.

```
stmnt: ...
| Opções IF FI {$$ = nn (0, $ 1, IF, 0, 0);
$$ -> seq1 = $ 2;
}
| FAZ
{pushbreak (); }
opções OD
```

```

{ $$ = nn (0, $ 1, DO, 0, 0);
$$ -> seq1 = $ 3;
}
| QUEBRAR
{ $$ = nn (break_dest (), $ 1, GOTO, 0, 0); }
| ATOMIC
'{
{open_seq (0); }
sequência
'}
{ $$ = nn (0, $ 1, ATÔMICO, 0, 0);
$$ -> seq1 = seqlist (close_seq (), 0);
make_atomic ($$ -> seq1-> this);
}
;
opções: opção
{ $$ = lista_seq ($ 1, 0); }
| opções de opções { $$ = lista_seq ($ 1, $ 2); }
;
opção: SEP
{open_seq (0); }
sequência
{ $$ = close_seq (); }
;

```

Cada opção em uma declaração composta pode ser uma sequência de declarações e é novamente capturado em uma estrutura de dados do tipo `Sequência`. Múltiplas opções são novamente agrupadas em uma lista vinculada de sequências do tipo `SeqList` que foi definido antes.

Página 291

280

UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

As instruções de repetição podem ser encerradas com instruções de `interrupção`. Para acompanhar o destinos adequados, usamos `pushbreak ()` para colocar um rótulo interno em uma pilha para cada nova estrutura de repetição que é inserida, e uma rotina `breakdest ()` para recuperar o destino atual de uma instrução `break`, tanto quanto com os rótulos e `goto jumps` discutimos anteriormente.

```

typedef struct Lbreak {
Símbolo * l;
struct Lbreak * nxt;
} Lbreak;
pushbreak ()
{
Lbreak * r = (Lbreak *) emalloc (sizeof (Lbreak));
Símbolo * l;
char buf [32];
sprintf (buf, ": b% d", break_id++);
l = pesquisa (buf);
r-> l = l;
r-> nxt = breakstack;
breakstack = r;
}
Símbolo *
break_dest ()
{
if (! breakstack)
fatal ("declaração de interrupção perdida", (char *) 0);
return breakstack-> l;
}

```

Uma instrução `break`, se ocorrer, é traduzida com uma chamada em `break_dest ()` em um pule para a última instrução `break` que foi colocada na pilha.

Página 292

SEÇÃO 12.6
CONTROLE DE FLUXO
281

```

.....
.
.
.
.
.
.
.....
.
.
```

```

.
.
.
nxt
Elemento
e
sub
SeqList
nxt
SeqList
nxt
NULO
esta
Seqüência
esta
Seqüência
primeiro
último
nxt
Elemento
t
nxt
Opção 1
opção 2
E SE
FI

```

Figura 12.4 - Árvore de análise para uma estrutura de seleção

Agora é hora de recorrer a um dos procedimentos mais difíceis para analisar o composto declarações: `if_seq ()`. Uma estrutura de seleção ou repetição deve empurrar vários elementos na sequência de instruções. A Figura 12.4 ilustra a estrutura do nó que é construída por o procedimento `if_seq ()` para incluir instruções de seleção em uma sequência. Apenas com declarações de libra (seleções e repetições) anexam quaisquer nós ao `subcampo` de um `Element`. Esse campo de sub-sequências inicia uma lista vinculada (uma `SeqList`) de opções, com uma estrutura de `Sequência` completa por opção na instrução composta. Figura 12.4 mostra uma declaração de seleção com duas opções.

O `último` ponteiro da subseqüência que define uma opção é conectado ao elemento `ment` que segue imediatamente aquele que contém o `subcampo` original. No figura este é o elemento rotulado `t` (para o alvo). Ele formaliza que uma seleção a seqüência é encerrada sempre que uma opção é encerrada.

A estrutura construída para uma instrução de repetição é quase a mesma, com apenas uma exceção: o `último` campo de cada subseqüência é agora apontado para um `Elemento` que é colocado imediatamente antes do composto, no local da caixa pontilhada na Figura 12.4. isto formaliza que uma estrutura de repetição é repetida quando uma opção termina. o `A` instrução `break` na estrutura de repetição ainda apontará para o `Elemento t` de destino. O código que faz tudo isso acontecer é o seguinte:

Página 293

282
 UM SIMULADOR DE PROTOCOLO
 CAPÍTULO 12
`Elemento *`
`if_seq (s, tok, lnno)`
`SeqList * s;`
`{`
 `Elemento * e = new_el ((Nó *) 0);`
 `Elemento * t = new_el (nn ((Símbolo *) 0, lnno, '.',`
 `(Nó *) 0, (Nó *) 0)); /* alvo */`
 `SeqList * z;`
 `e-> n = nn ((Símbolo *) 0, lnno, tok, (Nó *) 0, (Nó *) 0);`
 `e-> sub = s;`
 `para (z = s; z; z = z-> nxt)`
 `add_el (t, z-> this);`
 `if (tok == DO)`
 `{`
 `add_el (t, cur_s-> this);`
 `t = new_el (nn ((Símbolo *) 0, lnno, BREAK, (Nó *) 0, (Nó *) 0));`
 `set_lab (break_dest (), t);`
 `breakstack = breakstack-> nxt; /* pop stack */`
 `}`
 `add_el (e, cur_s-> this);`
 `add_el (t, cur_s-> this);`

```

return e;
/* nó de destino para rótulo */
}

```

12.7 PROCESSOS E TIPOS DE MENSAGEM

A principal coisa que falta em nosso simulador neste ponto é o conceito de um processo.

Com essa extensão, podemos dar alguns retoques finais no software, também codando o escalonador, adicionando a distinção entre variáveis locais e globais. O código do planejador está confinado a um arquivo denominado `sched.c`. A extensão do léxico analisador de cal é mínimo neste ponto: a mera adição do `proctype` de palavras-chave,

`Init`, e `mtype`. As outras extensões são mais substanciais:

Extensões para o analisador (seção 12.7.1, página 282)

O agendador de processos (seção 12.7.2, página 285)

Interpretando variáveis locais (seção 12.7.3, página 289)

12.7.1 PARSER

Um programa PROMELA completo é construído a partir de uma série de unidades de programa definidas do seguinte modo:

```

programa: unidades
{sched (); }
;
unidades: unidade | unidade de unidades
;
unidade
: proc
| iniciar
| afirmação
| one_decl
| mtype
;

```

Página 294

SEÇÃO 12.7
PROCESSOS E TIPOS DE MENSAGEM

283

```

proc
: NOME DO PROCTYPE
{contexto = $ 2; }
'(' decl ')'
corpo
{pronto ($ 2, $ 5, $ 7);
contexto = (simbolo *) 0;
}
;
mtype: MTYPE ASGN '(' args ')' {setmtype ($ 4); }
| ';' /* opcional; como separador de unidades */
;
...
decl
: /* vazio */
{$$ = (Nó *) 0; }
| decl_lst
{$$ = $ 1; }
;
decl_lst: one_decl
{$$ = nn (0, 0, ',', $ 1, 0); }
| one_decl ';' decl_lst {$$ = nn (0, 0, ',', $ 1, $ 3); }
;
```

Uma unidade é uma declaração de processo, uma lista de tipos de mensagem, uma reivindicação temporal ou um

especificação `init`, cada uma das quais pode ser precedida por uma ou mais variáveis globais declarações. O módulo `init` e as reivindicações temporais são definidos como tipo especial de

processos:

```

iniciar
: INICIAR
{contexto = $ 1; }
corpo
{executável ($ 3, $ 1);
contexto = (simbolo *) 0;
}
;
reclamação: CLAIM
{contexto = $ 1;
if (Claimproc)
yerror ("reivindicação% s redefinida",
Claimproc);
}
```

```

Claimproc = $ 1-> nome;
}
corpo
{pronto ($ 1, (Nó *) 0, $ 3);
contexto = (símbolo *) 0;
}
;

```

Pode haver apenas um `init` e uma reivindicação temporal por especificação. O primeiro é obrigatório, o segundo opcional. A presença de uma reclamação é sinalizada no ponteiro global `Claimproc`.

As declarações de processo são colocadas em uma fila pronta de corpos de processo. Para nos permitir lembre-se das declarações de tipo de parâmetros formais, as declarações agora devem retornar um nó que contém a lista de parâmetros. Antes que o `corpo` de uma declaração de processo seja analisada, porém, uma variável de contexto é definida para identificar quaisquer nomes de variáveis e parâmetros

para ser reconhecido como local para a declaração do processo. Inicialmente, apenas o processo `init` é rotulado como executável. O procedimento `setmtype ()` pertence logicamente a `sym.c` e pode ser implementado da seguinte forma:

```
Nó * Tipo M = (Nó *) 0;
```

Página 295

```

284
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
vazio
setmtype (m)
Nó * m;
{
Nó * n = m;
if (Mtype)
yyerror ("mtype redeclared", (char *) 0);
Mtype = n;
enquanto (n)
/* verificação de sintaxe */
{
if (! n-> lft ||! n-> lft-> nsym
|| (n-> lft-> ntyp != NOME)
|| n-> lft-> lft)
/* variável indexada */
fatal ("definição de mtype ruim", (char *) 0);
n = n-> rgt;
}
}
Node * Symnode = 0;
vazio
syms (m)
Nó * m;
{
if (Symnode)
yyerror ("Definição de simetria duplicada", (char *) 0);
Symnode = m;
}

```

O procedimento apenas verifica a sintaxe e armazena os argumentos para processamento posterior. As definições de tipo de mensagem podem ser completamente escondidas do resto do programa se deixarmos o analisador léxico verificar a lista sempre que vir um `NOME` e mapear todas as mensagens nomes sábios encontrados lá em constantes. Podemos fazer isso na procedure `check_name ()`.

```

check_name (s)
char * s;
{
registrar int i;
para (i = 0; nomes [i] .s; i++)
if (strcmp (s, nomes [i] .s) == 0)
{
yylval.val = lineno;
retornar nomes [i] .tok;
}
if (yylval.val = ismtype (s))
return CONST;
yylval.sym = pesquisa (s); /* tabela de simbolos */
retornar NOME;
}

```

A rotina `ismtype ()` procura nomes na lista de tipos de mensagens.

SEÇÃO 12.7
PROCESSOS E TIPOS DE MENSAGEM

285

```
ismstype (str)
char * str;
{
Nó * n;
int cnt = 1;
para (n = tipo M; n; n = n-> rgt)
{
if (strcmp (str, n-> lft-> nsym-> nome) == 0)
return cnt;
cnt++;
}
return 0;
}
```

12.7.2 AGENDADOR

Os procedimentos `ready ()` e `runnable ()` requerem pouca imaginação; eles precisam simplesmente armazene seus argumentos em listas vinculadas onde o planejador pode encontrá-los. A lista de processos executáveis é definida da seguinte forma:

```
typedef struct ProcList {
Símbolo * n;
/* nome
 */
Nó
* p;
/* parâmetros */
Sequência * s;
/* corpo
 */
struct ProcList * nxt; /* lista vinculada */
} ProcList;
```

E a rotina que preenche a lista é

```
executável (s, n)
Sequência * s;
/* body */
Símbolo * n;
/* nome */
{
RunList * r = (RunList *) emalloc (sizeof (RunList));
r-> n = n;
r-> pid = nproc++;
r-> pc = s-> primeiro;
r-> maxseq = s-> último-> seqno;
r-> nxt = executar;
run = r;
}
```

A lista de execução real dos processos em execução tem um contador de programa `pc` e um ponteiro para valores atuais de variáveis locais, que chamamos de `symtab` novamente, uma vez que é basicamente outra lista da tabela de símbolos.

```
typedef struct RunList {
Símbolo * n;
/* nome
 */
int
pid;
/* id do processo
 */
int
maxseq;
/* usado pelo gerador do analisador */
Elemento * pc;
/* stmnt atual */
Símbolo * symtab;
/* variáveis locais */
struct RunList * nxt; /* lista vinculada */
} RunList;
```

286

UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

Inserir um processo na lista é fácil.

```
pronto (n, p, s)
Símbolo * n;
```

```

/* nome do processo */
Nó * p;
/* parâmetros formais */
Sequência * s;
/* corpo do processo */
{
ProcList * r = (ProcList *) emalloc (sizeof (ProcList));
r-> n = n;
r-> p = p;
r-> s = s;
r-> nxt = rdy;
rdy = r;
}

```

Mover um processo da lista de processos para a lista de execução é um pouco mais complicado, uma vez que

também os campos de parâmetro devem ser inicializados.

```

habilitar (s, n)
Símbolo * s;
/* nome do processo */
Nó * n;
/* parâmetros reais */
{
ProcList * p;
para (p = rdy; p; p = p-> nxt)
if (strcmp (s-> nome, p-> n-> nome) == 0)
{
executável (p-> s, p-> n);
setparams (executar, p, n);
retorno (nproc-nstop-1); /* pid */
}
return 0; /* processo não encontrado */
}

```

Onde

```

setparams (r, p, q)
RunList * r;
ProcList * p;
Nó * q;
{
Nó * f, * a;
/* pars formal e real */
Nó * t;
/* lista de pars de 1 tipo */
para (f = p-> p, a = q; f; f = f-> rgt) /* um tipo de cada vez */
para (t = f-> lft; t; t = t-> rgt, a = (a)? a-> rgt: a)
{
int k;
if (! a) fatal ("parâmetros reais ausentes: '% s'", p-> n-> nome);
k = eval (a-> lft);
/* deve ser inicializado */
if (typck (a, t-> nsym-> tipo, p-> n-> nome))
{
if (t-> nsym-> type == CHAN)
naddsymbol (r, t-> nsym, k); /* cópia de */
outro
{
t-> nsym-> ini = a-> lft;
adiciona símbolo (r, t-> nsym);
}
}

```

SEÇÃO 12.7
PROCESSOS E TIPOS DE MENSAGEM
287

```

}
}

com
naddsymbol (r, s, k)
RunList * r;
Símbolo * s;
{
Símbolo * t = (Símbolo *) emalloc (sizeof (Símbolo));
int i;
t-> nome = s-> nome;
t-> tipo = s-> tipo;
t-> nel = s-> nel;
t-> ini = s-> ini;
t-> val = (int *) emalloc (s-> nel * sizeof (int));

```

```

if (s-> nel! = 1)
fatal ("array na lista de parâmetros formal,% s", s-> nome);
para (i = 0; i <s-> nel; i++)
t-> val [i] = k;
t-> próximo = r-> symtab;
r-> symtab = t;
}
e
adiciona símbolo (r, s)
RunList * r;
Símbolo * s;
{
Símbolo * t = (Símbolo *) emalloc (sizeof (Símbolo));
int i;
t-> nome = s-> nome;
t-> tipo = s-> tipo;
t-> nel = s-> nel;
t-> ini = s-> ini;
if (s-> val)
/* se inicializado, copie-o */
{
t-> val = (int *) emalloc (s-> nel * sizeof (int));
para (i = 0; i <s-> nel; i++)
t-> val [i] = s-> val [i];
} outro
checkvar (t, 0); /* inicializa-o */
t-> próximo = r-> symtab;
/* adicionar */
r-> symtab = t;
}

```

Para ser capaz de criar novas instanciações de processo durante uma simulação, nós expanda a instrução run em run.c da seguinte maneira:

case RUN: return enable (now-> nsym, now-> lft);

Os processos só podem ser excluídos da lista de execução na ordem inversa da criação: um processo só pode desaparecer se todos os seus filhos tiverem desaparecido primeiro (Capítulo 5). Os pid's podem portanto, ser reciclado em ordem de pilha. No valor retornado por enable (), ,

Página 299

288

UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12

(nproc-nstop-1) , a contagem nproc é igual ao número total de processos criados, e nstop , o número de processos que foram excluídos.

A rotina mais interessante a ser discutida é a própria rotina de agendamento: sched () .

Sua parte relevante é a seguinte:

```

sched ()
{
Elemento * e, * eval_sub ();
RunList * Y;
/* processo anterior na fila de execução */
int i = 0;
...
para (Tval = 0; Tval <2; Tval++)
{
enquanto (i <nproc-nstop)
para (X = executar, Y = 0, i = 0; X; X = X-> nxt)
{
lineno = X-> pc-> n-> nval;
Fname = X-> pc-> n-> fname;
if (e = eval_sub (X-> pc))
{
X-> pc = e; Tval = 0;
} else /* processo encerrado? */
{
if (X-> pc-> n-> ntyp == '@'
&& X-> pid == (nproc-nstop-1))
{
if (Y)
Y-> nxt = X-> nxt;
outro
executar = X-> nxt;
nstop++; Tval = 0;
} outro
i++;
}
Y = X;
}

```

```

    }
}
embrulhar();
}

O planejador executa uma instrução em cada processo executável em round-robin
moda. Ele chama a rotina eval_sub () , que vimos anteriormente, para avaliar recursivamente
ate declarações compostas e sequências atômicas. A avaliação de um atômico
sequência só é bem-sucedida se toda a sequência puder ser concluída. O código é parte de
eval_sub () .
if (e-> n-> ntyp == ATOMIC)
{
f = e-> n-> seql-> this-> primeiro;
g = e-> n-> seql-> this-> last;
g-> nxt = e-> nxt;
if (! (g = eval_sub (f)))
return (Element *) 0;
Rvous = 0;
while (g && (g-> status & (ATOM | L_ATOM))
&& ! (f-> status & L_ATOM))
{
f = g;
g = eval_sub (f);
}

```

Página 300

SEÇÃO 12.7
PROCESSOS E TIPOS DE MENSAGEM

289
if (! g)
{
embrulhar();
lineno = f-> n-> nval;
fatal ("blocos de sequência atômica", (cha *) 0);
}
return g;
} else if (Rvous)
...

É um erro fatal se uma sequência atômica for bloqueada. Se um processo atinge um estado não
executável-

o planejador verifica se ele está em um estado de parada. Nesse caso, o processo é removido
da fila de execução e a contagem de processos encerrados nstop é incrementada. UMA
a variável global x aponta para o processo atualmente em execução. É usado principalmente pelo
novas rotinas para manipular variáveis locais, getlocal () e setlocal () , para
determinar em qual estrutura de processo as variáveis estão localizadas. O programador também
mantém um valor Tval que é usado pelo interpretador em run.c para determinar o ex-
capacidade de corte da instrução de tempo limite .

case TIMEOUT: return Tval;

Durante a execução normal, quando o sistema não está bloqueado, Tval é zero e condi-
ções que incluem um tempo limite não podem ser executadas. Para se recuperar de um impasse
potencial,

o planejador pode habilitar as instruções de tempo limite incrementando Tval . Se o sistema
não se recuperar, o planejador declara um verdadeiro estado de travamento e desiste.

12.7.3 VARIÁVEIS LOCAIS

Uma vez que as variáveis locais são criadas instantaneamente, na instanciação de novos processos,
o lugar lógico para o código que os manipula é no agendador. Se uma variável
name é local, duas variantes especiais de getvar () e setvar () são usadas.

```

getlocal (s, n)
Símbolo * s;
{
Símbolo * r;
r = findloc (s, n);
if (r) retorna cast_val (r-> tipo, r-> val [n]);
return 0;
}
setlocal (p, m)
Nó * p;
{
int n = eval (p-> lft);
Símbolo * r = findloc (p-> nsym, n);
se (r) r-> val [n] = m;
return 1;
}
```

A rotina `findloc ()` localiza o nome na tabela de símbolos do atualmente em execução processo de processamento, apontado por `X-> symtab`.

Página 301

```
290
UM SIMULADOR DE PROTOCOLO
CAPÍTULO 12
Símbolo *
findloc (s, n)
Símbolo * s;
{
Símbolo * r = (Símbolo *) 0;
if (n >= s-> nel || n < 0)
{
yyerror ("erro de indexação de array% s", s-> nome);
retorno (símbolo *) 0;
}
if (! X)
{
se (analisar)
fatal ("erro, não é possível avaliar a variável '% s'", s-> nome);
outro
yyerror ("erro, não é possível avaliar a variável '% s'", s-> nome);
retorno (símbolo *) 0;
}
para (r = X-> symtab; r; r = r-> próximo)
if (strcmp (r-> nome, s-> nome) == 0)
quebrar;
if (! r && ! Noglobal)
{
adiciona símbolo (X, s);
r = X-> symtab;
}
return r;
}
```

As variáveis locais e os estados do processo de qualquer processo em execução podem ser referidos em asserções e reivindicações temporais. Os ganchos no analisador que permitem a referência remota ing são simples. Referências a variáveis remotas e estados de processo requerem os dois últimos regras de produção:

```
| NOME '[' expr ']' '.' varref {$$ = rem_var ($ 1, $ 3, $ 6); }
| NOME '[' expr ']' ':' NOME
{$$ = rem_lab ($ 1, $ 3, $ 6); }
com
Nó *
rem_var (a, b, c)
Símbolo * a;
Nó * b, * c;
{
Nó * tmp;
if (! context || strcmp (context-> name, ": never:") != 0)
yyerror ("aviso: uso ilegal de '.' (fora nunca reclamar)", (char *) 0);
tmp = nn (a, 0, '?', b, (Nó *) 0);
retornar nn (c-> nsym, 0, 'p', tmp, c-> lft);
}
e
```

Página 302

```
SEÇÃO 12.7
PROCESSOS E TIPOS DE MENSAGEM
291
Nó *
rem_lab (a, b, c)
Símbolo * a, * c;
Nó * b;
{
if (! context || strcmp (context-> name, ": never:") != 0)
yyerror ("aviso: uso ilegal de ':' (fora nunca reclamar)", (char *) 0);
retornar nn ((símbolo *) 0, 0, EQ,
nn (lookup ("_ p"), 0, 'p', nn (a, 0, '?', b, (Nó *) 0), (Nó *) 0),
nn (c, 0, 'q', nn (a, 0, NOME, (Nó *) 0, (Nó *) 0), (Nó *) 0));
}
```

A referência é implementada como uma condição no estado do fluxo de controle de um processo, representado pela variável interna `_p`, e o valor do estado de um nome de rótulo. O valor que da variável especial `_p` é determinada, assim como as outras variáveis remotas, usando um

nó do tipo 'p'. O nome do rótulo é determinado com um novo nó do tipo 'q'. o tipo de nó '?' é usado apenas como um marcador temporário.

No avaliador, dois novos tipos de nó acionam chamadas nessas duas rotinas:

```
case 'p': return remotevar (agora);
case 'q': retorna remotelab (agora);
```

A primeira rotina, para referenciar o valor atual de uma variável local em um programa remoto cesso, é implementado com uma mudança de contexto no agendador, da seguinte forma:

```
remotevar (n)
Nó * n;
{
int pno, i, j;
RunList * Y, * oX = X;
pno = eval (n-> lft-> lft);
/* pid */
i = nproc - nstop;
para (Y = corrida; Y; Y = Y-> nxt)
if (--i == pno)
{
if (strcmp (Y-> n-> nome, n-> lft-> nsym-> nome))
yyerror ("proctype% s errado", Y-> n-> nome);
X = Y; j = getval (n-> nsym, eval (n-> rgt)); X = oX;
return j;
}
yyerror ("ref remoto: proc% s não encontrado", n-> nsym-> nome);
return 0;
}
```

A segunda rotina, para determinar o estado do fluxo de controle em um processo remoto que corresponde a um determinado nome de rótulo, é implementado com uma pesquisa na lista de rótulos, do seguinte modo:

Página 303

292

UM SIMULADOR DE PROTOCOLO
CAPÍTULO I2
remotelab (n)
Nó * n;
{
int i;
if (n-> nsym-> tipo)
fatal ("não é um nome de rótulo: '% s'", n-> nsym-> nome);
if ((i = find_lab (n-> nsym, n-> lft-> nsym)) == 0)
fatal ("labelname desconhecido:% s", n-> nsym-> nome);
return i;
}

12.8 EXPANSÃO MACRO

A versão final do SPIN no Apêndice D tem uma versão expandida do `main ()` que interpreta adequadamente sinalizadores de opção, aceita um argumento de arquivo e roteia sua entrada por meio

o pré-processador C em `/ lib / cpp` para expansão de macro. A saída do pré-processador é despejado em um arquivo temporário que é imediatamente desvinculado para se certificar de que desaparece do sistema de arquivos, mesmo se a execução do simulador for interrompida. o analisador, entretanto, mantém um link para o arquivo no ponteiro de arquivo predefinido `yyin`.

```
if (argc> 1)
{
arquivo de saída char [17], cmd [64];
strcpy (nome do arquivo, argv [1]);
mktemp (strcpy (outfile, "/tmp/spin.XXXXXXX"));
sprintf (cmd, "/ lib / cpp% s>% s", argv [1], arquivo de saída);
if (sistema (cmd))
{
desvincular (arquivo de saída);
saída (1);
} else if (! (yyin = fopen (outfile, "r")))
{
printf ("não é possível abrir% s \ n", arquivo de saída);
saída (1);
}
desvincular (arquivo de saída);
} outro
strcpy (nome do arquivo, "<stdin>");
```

O pré-processador coloca linhas no arquivo que parecem

```
# 1 "spin.examples / lynch"
```

O analisador léxico pode pegá-los e interpretá-los com uma regra extra que é

definido como segue:

```
\ # \ [0-9] + \ \"[^ \"]* \ "
/* Directiva do pré-processador */
int i = 1;
while (yytext [i] == '') i++;
lineno = atoi (& yytext [i]) - 1;
while (yytext [i] != '') i++;
Fname = lookup (& yytext [i + 1]);
}
```

O número da linha é lembrado na variável `lineno`. O nome do arquivo é armazenado no tabela de símbolos e um ponteiro global para ela são mantidos em `Fname`. Mas a maioria destes restantes

Página 304

SEÇÃO 12.9
SPIN - OPÇÕES
293

os recursos são cosméticos e podem ser alterados ou ignorados sem riscos indevidos.

12.9 SPIN - OPÇÕES

O simulador reconhece oito opções de linha de comando que podem ser usadas em qualquer computador binação. Dois (sinalizadores `a` e `t`) são específicos para o código de análise que ainda precisamos desenvolver no Capítulo 13. Os outros seis são discutidos abaixo.

rotaciona

Imprime uma linha no visor para cada mensagem enviada. Exemplo:

```
$ spin -s factorial
proc 12 (fato) linha 5, enviar 1
-> fila 12 (p)
proc 11 (fato) linha 10, Enviar 2
-> fila 11 (p)
proc 10 (fato) linha 10, Enviar 6
-> fila 10 (p)
proc 9 (fato) linha 10, Enviar 24
-> fila 9 (p)
proc 8 (fato) linha 10, enviar 120
-> fila 8 (p)
...

```

spin -r

Imprime uma linha no visor para cada mensagem recebida. Imprime o nome e `pid` do processo em execução e um número de linha de origem para seu estado atual. Exemplo:

```
$ spin -s -r factorial
proc 12 (fato) linha 5, enviar 1
-> fila 12 (p)
proc 11 (fato) linha 9, Recv 1
<- fila 12 (filho)
proc 11 (fato) linha 10, Enviar 2
-> fila 11 (p)
proc 10 (fato) linha 9, Recv 2
<- fila 11 (filho)
proc 10 (fato) linha 10, Enviar 6
-> fila 10 (p)
proc 9 (fato) linha 9, Recv 6
<- fila 10 (filho)
...

```

spin -p

Imprime uma linha no display para cada instrução executada. Exemplo:

```
$ spin -p factorial
proc 0 (_init) linha 18 (estado 2)
proc 1 (fato) linha 8 (estado 4)
proc 1 (fato) linha 9 (estado 5)
proc 2 (fato) linha 8 (estado 4)
proc 2 (fato) linha 9 (estado 5)
proc 3 (fato) linha 8 (estado 4)
...
proc 3 (fato) termina

```

spin -l

Adiciona o valor de todas as variáveis locais à saída. Esta opção, como a próxima, é mais útil em combinação com `-p`. Exemplo:

```
$ spin -p -l factorial
...
proc 12 (fato) linha 12 (estado 9)
fila 12 (p):
n = 1

```

294
 UM SIMULADOR DE PROTOCOLO
 CAPÍTULO 12
 proc 11 (fato) linha 4 (estado 8)
 resultado = 1
 fila 12 (filho):
 fila 11 (p): [2]
 n = 2
 ...
 proc 12 (fato) termina
 ...
spin -g

Adiciona os valores atuais de todas as variáveis globais às listagens.

spin -t12345

Inicializa o gerador de número aleatório com a semente 12345 especificada pelo usuário para assegure uma execução de simulação que pode ser reproduzida exatamente.

12,10 RESUMO

A última versão do SPIN contém cerca de 2.000 linhas de código-fonte, mais de dez vezes o tamanho do pequeno avaliador de expressão com o qual começamos este capítulo. Para dar um indicação do desempenho do SPIN , executamos o seguinte programa para calcular Números de Fibonacci. Ele cria e executa um total de 1000 processos.

```
/ ***** Sequência de Fibonacci ***** /
proctipo fib (curto n)
{
  curto a = 0;
  b curto = 1;
  c curto;
  atômico
  {
    Faz
    :: (b <n) ->
    c = b;
    b = a + b;
    a = c
    :: (b> = n) ->
    quebrar
    od
  }
  iniciar
  {
    int i = 1;
    atômico
    {
      Faz
      :: (i <1000) -> i = i + 1; executar fib (1000)
      :: (i> 999) -> pausa
      od
    }
  }
}
```

Em um computador DEC-VAX / 8550, uma simulação leva cerca de 7,6 segundos do usuário Tempo. Um programa otimizado para cálculo pode executar um programa semelhante duas a dez vezes mais rápido, mas é claro que não tem os recursos de sincronização e multi-processo de

CAPÍTULO 12
 EXERCÍCIOS
 295

PROMELA . As chamadas de função mais caras de uma simulação podem ser encontradas com o utilitário UNIX prof . Para o teste de Fibonacci, 80% do tempo de execução é gasto no seguinte seguites rotinas:

```
%
Tempo
#Calls
Nome
23 2.633 433411
_eval
13 1.533 166433
_findloc
11 1,233 116950
_eval_sub
9 1.050
0
```

```
strcmp  
8 0,917  
0  
mcount  
6 0,717 117482  
_getlocal  
5 0,617 117482  
_cast_val  
5 0,617 117482  
_getval
```

Se em algum ponto a eficiência do SPIN precisar ser melhorada, um bom alvo para otimizações seria o procedimento `eval ()`. (Veja também os exercícios.) Para simulações de protocolo, como-nunca, o programa é suficientemente rápido.

No próximo capítulo, veremos como podemos estender os recursos da SPIN com um gerador para validações de protocolo exaustivas. Conforme discutido no Capítulo 11, o desempenho potencial gargalos de mance em validadores de protocolo requerem atenção cuidadosa se quisermos duce uma ferramenta de valor prático. Portanto, mudamos a maioria das considerações de eficiência para essa parte do software SPIN .

EXERCÍCIOS

12-1. Altere a semântica da instrução de `tempo limite`, permitindo que uma contagem de tempo limite seja Especificadas. Adicione uma variável predefinida de `tempo` que é incrementada uma vez para cada ciclo através da lista de processos em execução pelo planejador. Aplicar e testar os novos recursos com um protocolo de amostra.

12-2. O simulador e a linguagem PROMELA usam uma quantidade assinada de 32 bits como a maior número. Isso impõe restrições ao uso do SPIN como calculadora. Qual é o maior factorial que pode ser calculado com o programa factorial da Seção 12.2?

12-3. Execute o teste Fibonacci em seu sistema e meça o tempo de execução. Faça o atômico sequências não atômicas e repita o teste. Explique o resultado.

12-4. Modele e simule um programa PROMELA de exemplo arbitrário deste livro.

12-5. Adicione mais recursos ao PROMELA . Por exemplo,

Permitir fragmentos de código C inline

Permitir outra palavra-chave em declarações compostas

Adicionar estruturas de dados semelhantes a C

(Rob Pike) Adicionar canais de *dispositivos*

Um canal de dispositivo é uma fila de mensagens predefinida que conecta um programa PROMELA ao mundo externo (por exemplo, adicione uma tela de terminal e um canal de teclado).

12-6. Adicione um otimizador de execução de pré-simulação que reescreve partes da árvore de análise instantaneamente. Boa

candidatos para otimização, por exemplo, são expressões que envolvem apenas referências constantes cias, como `(5 * 3 + 2)` .

Página 307

296

UM SIMULADOR DE PROTOCOLO

CAPÍTULO 12

12-7. Use o simulador para implementar uma estratégia de validação de passeio aleatório. Considere a viabilidade capacidade de validar cada um dos critérios de correção discutidos no Capítulo 6.

NOTAS BIBLIOGRÁFICAS

A melhor referência para a linguagem de programação C ainda é Kernighan e Ritchie [1978, 1988]. A segunda edição deste livro, publicada em 1988, é um excelente referência à nova versão do padrão ANSI da linguagem C. Outro bom disco A definição do padrão ANSI pode ser encontrada em Harbison e Steele [1987].

Muito mais sobre o design de analisadores e analisadores lexicais pode ser encontrado no livros famosos sobre *dragões* de Al Aho e outros. Veja por exemplo Aho e Ullman [1977] e Aho, Sethi e Ullman [1986]. Um guia muito útil para o uso do

As ferramentas UNIX *yacc* e *lex* também podem ser encontradas em Schreiner e Friedman [1985].

Um excelente tutorial sobre o desenvolvimento de programas C pode ser encontrado em Kernighan e Pike [1984]. O capítulo 8 desse livro é especialmente recomendado.

Canais de dispositivo (Exercício 12-5) também foram definidos na linguagem Squeak, e seu sucessor Newsqueak. Veja, por exemplo, Cardelli e Pike [1985].

Página 308

UM VALIDADOR DE PROTOCOLO 13

297 Introdução 13.1

298 Estrutura do Validador 13.2
299 O Kernel de Validação 13.3
302 A Matriz de Transição 13.4
303 Código Gerador-Validador 13.5
306 Visão Geral do Código 13.6
308 Simulação Guiada 13.7
310 Alguns aplicativos 13.8
315 Cobertura no Modo Supertrace 13.9
316 Resumo 13.10
316 exercícios
317 Notas Bibliográficas

13.1 INTRODUÇÃO

Para estender o simulador de protocolo do Capítulo 12 com um gerador validador, todos nós temos que fazer é ativar duas opções de linha de comando da lista de fontes no Appendix D:

- a , Para gerar um analisador SPIN específico de protocolo
- t , para seguir uma trilha de erro produzida por esse analisador

Para fazer isso, temos que substituir as duas rotinas fictícias `gensrc ()` e `match_trail ()` no Apêndice D com código real. Para ver como o analisador é usado consulte a Seção 13.8 ou Capítulo 14.

O analisador descrito aqui é baseado na discussão no Capítulo 11. Para manter o código razoavelmente simples, não discutiremos uma implementação completa do estado modelo vetorial. Mesmo sem isso, é preciso uma boa quantidade de código para produzir um validador de bom desempenho. Mas uma vez que o trabalho é feito corretamente, um validador eficiente pode ser produzido em questão de segundos e pode ser aplicada a problemas de complexidade arbitrária.

Os validadores produzidos pela SPIN desta forma estão entre os programas mais rápidos para pesquisas exaustivas conhecidas até o momento. Uma implementação completa do vetor de estado modelo pode garantir um desempenho ainda melhor, mas isso está muito além do escopo deste livro.

Os validadores podem ser usados em dois modos diferentes. Para modelos de pequeno a médio porte os validadores podem ser usados com um espaço de estado exaustivo. O resultado de todas as validações realizadas neste modo é equivalente a uma prova exaustiva de correção, para o requisitos de correção que foram especificados (por padrão, ausência de conflito). Para sistemas que são maiores, os validadores também podem ser usados no *modo supertrace*, com a técnica de espaço de estado de bits, conforme discutido no Capítulo 11. Nesses casos, as validações podem

ser executadas em quantidades muito menores de memória e ainda reter uma cobertura excelente

297

Página 309

298
UM VALIDADOR DE PROTOCOLO
CAPÍTULO 13

do espaço de estado. Os resultados de todas as validações realizadas no modo supertrace são superior a qualquer outro tipo de validação realizada dentro da mesma condição física limitações da máquina host (por exemplo, tamanho da memória e velocidade).

Para produzir um analisador, a árvore de análise que é construída pelo simulador SPIN é traduzido para um programa C e estendido com módulos de pesquisa de espaço de estado. O programa gerado é então compilado de forma independente. Quando é executado, performa a validação necessária. Se um erro for descoberto, o programa escreve uma simulação trilha de erro em um arquivo e para. A trilha de simulação pode ser lida pela simulação original, que pode então reproduzir a sequência de erro e permitir que o usuário investigue a causa do erro em detalhes.

Abaixo, discutimos primeiro a estrutura geral dos analisadores de protocolo que são gerados. Em seguida, fornecemos uma visão geral das rotinas que extraem o protocolo específico informações da árvore de análise SPIN. A seguir, discutimos as extensões do simulador para fornecer simulações guiadas. Concluímos com alguns exemplos de uso de a nova ferramenta.

13.2 ESTRUTURA DO VALIDADOR

A Figura 13.1 mostra os principais componentes dos analisadores que podem ser gerados.

corre()
novo_estado()
Transição
Matriz
cerqueirinha()
Tamanho Variável

*Espaço Estadual
uerror ()*

Figura 13.1 - Estrutura de SPIN validadores

Um procedimento chamado `run()` aloca memória e prepara todas as estruturas de dados que o validador usará durante a pesquisa. Ele chama um único procedimento `new_state()` para performar a pesquisa real. As duas principais estruturas de dados usadas por este procedimento `new_state()` são o espaço de estado e uma grande matriz de transição que codifica o completo PROMELA modelo de validação. Cada declaração no modelo produz uma entrada em esta matriz, definindo com precisão o predicado de executabilidade e o efeito da execução.

Cada definição de `proctype` contribui com entradas para esta matriz. O estado atual do sistema é mantido em um vetor de valores que pode crescer e encolher dinamicamente: um vetor de estado de borracha. Instruções de execução PROMELA anexam novo processos para o vetor de estado. O vetor de estado da borracha, portanto, cumpre um papel que é

Página 310

SEÇÃO 13.3
O KERNEL DE VALIDAÇÃO
299

semelhante à fila de execução no programador do simulador. Procedimento `new_state()` executa uma primeira pesquisa em profundidade de todas as instruções executáveis no modelo. Ao invés de

selecionando apenas uma instrução executável da lista de processos executáveis, como o simulador fez, o trabalho do validador é testar os efeitos de *todas as* instruções executáveis, em todas as intercalações possíveis. Antes de iniciar a análise para um novo estado `new_state()` consulta o espaço de estado, por meio da função `hash()` e decide se o estado atual foi analisado antes e pode ser ignorado.

Se um erro for encontrado, o procedimento `uerror()` é chamado para produzir uma trilha de erro para o simulador e, salvo indicação em contrário, a análise pára. Um erro pode ser qualquer violação dos requisitos de correção formal, por exemplo, uma falha de afirmação local ou um estado global do sistema em que todos os processos são bloqueados permanentemente.

Encontrando uma inconsistência no modelo SPIN e auxiliando o usuário na determinação de sua causas, é feito com duas ferramentas diferentes: validador e simulador. A justificativa por trás esta abordagem é a disciplina padrão do UNIX: cada ferramenta que desenvolvemos deve fazer um coisa, e fazê-lo bem.

O simulador foi projetado como uma ferramenta interativa. Tem um tempo de inicialização curto e pode dar uma visão detalhada do funcionamento do protocolo.

O validador é uma ferramenta não interativa. Tem um tempo de inicialização maior, pois requer a compilação de um programa intermediário, mas é otimizado para pesquisas exaustivas.

13.3 O KERNEL DE VALIDAÇÃO

O procedimento `new_state()` é o núcleo do analisador. Ele controla todas as execuções, monitora o progresso e executa as verificações de exatidão. Mais da metade do tempo de execução é gasto nesta rotina, com a maior parte do restante sendo usada no cálculo

çao de valores hash para acessar o espaço de estado.

O procedimento é definido estaticamente em um arquivo de cabeçalho denominado `pangen1.h`. Opções do código são habilitadas ou desabilitadas dependendo da presença ou ausência de rendezvous comunicações, reivindicações temporais, estados de aceitação ou estados de progresso e dependendo do tipo de armazenamento de espaço de estado selecionado. Ignorando, por enquanto sendo, todas essas opções, o algoritmo de pesquisa de estado exaustivo simples se parece com o seguinte baixos:

```
1 new_state ()  
2 {registrar Trans * t, * ta;  
3  
char n, m, ot, lst;  
4  
short II, tt;  
5  
curto De = agora.nr_pr-1;  
6  
curto para = 0;  
7 para baixo:  
8  
if (profundidade > profundidade máxima)  
9
```

```
{  
truncs ++;  
10  
goto Up;  
11  
}  
}
```

```
300  
UM VALIDADOR DE PROTOCOLO  
CAPÍTULO 13  
12  
if (To == 0)  
13  
{  
if (hstore ((char *) & now, vsize))  
14  
{  
truncs ++;  
15  
goto Up;  
16  
}  
17  
nstates ++;  
18  
}  
19  
if (profundidade > mreached)  
20  
mreached = profundidade;  
21  
n = tempo limite = 0;  
22  
23 Novamente:  
24  
para (II = De; II >= Para; II -= 1)  
25  
{  
este = pptr (II);  
26  
tt = (curto) ((P0 *) este) -> _p;  
27  
ot = (unsigned char) ((P0 *) this) -> _t;  
28  
para (t = trans [ot] [tt]; t; t = t-> nxt)  
29  
{  
30 #include "pan.m"  
31 P999:  
/* pula aqui quando o movimento é bem-sucedido */  
32  
if (m > n || (n > 3 && m != 0)) n = m;  
33  
profundidade++; trpt++;  
34  
trpt-> pr = II;  
35  
trpt-> st = tt;  
36  
if (t-> st)  
37  
{  
((P0 *) isto) -> _p = t-> st;  
38  
atingiu [ot] [t-> st] = 1;  
39  
}  
40  
trpt-> o_t = t; trpt-> o_n = n;  
41  
trpt-> o_ot = ot; trpt-> o_tt = tt;  
42  
trpt-> o_To = To;  
43  
if (t-> átomo & 2)  
44  
{  
De = Para = II; nlinks ++;  
45
```

```

    } outro
46
{
De = agora.nr_pr-1; Para = 0;
47
}
48
goto Down;
/* pseudo-recursão */
49 Up:
50
t = trpt->o_t; n = trpt->o_n;
51
ot = trpt->o_ot; II = trpt->pr;
52
tt = trpt->o_tt; este = pptr (II);
53
To = trpt->o_To;
54 #include "pan.b"
55 R999:
/* pula aqui quando terminar */
56
profundidade--; trpt--;
57
((P0 *) isto) ->_p = tt;
58
} /* todas as opções */
59
} /* todos os processos */
60

```

Página 312

SEÇÃO 13.3
O KERNEL DE VALIDAÇÃO

```

301
61
if (n == 0)
62
{
if (!endstate () && now.nr_pr)
63
{
if (!tempo limite)
64
{
tempo limite = 1;
65
goto novamente;
66
}
67
uerror ("impasse");
68
}
69
}
70
se (profundidade> 0) vá para cima;
71}

```

O procedimento é chamado uma vez e não retorna até que seja realizada uma pesquisa completa. formado ou um erro encontrado. Nesta versão, sem todos os enfeites de um implemento completo mentação, o único tipo de erro verificado é um estado final inválido. O trabalho principal é feito em dois loops `for`. O primeiro, na linha 24, faz um loop em todos os atualmente em execução processos. O segundo, na linha 28, verifica exaustivamente todas as instruções executáveis em cada processo. O proctipo do processo atual é armazenado em uma variável local `ot`, e o estado do processo é mantido em uma variável local `tt`. Essas duas variáveis juntas são usado para indexar a *matriz de transição*, `trans [ot] [tt]`, na linha 28. Um ponteiro `T` aponta à definição da própria transição: a condição, o efeito e o próximo estado.

A execução das próprias transições está oculta em um arquivo `pan.m` que está incluído na linha 30. É uma troca de caso simples que registra todas as transições que são definidas no sistema. Se uma transição for executável, ela leva ao rótulo P999. Se não for executável um continue é executado, o que nos traz de volta ao loop interno na linha 28.

Uma transição bem-sucedida produz um novo estado que deve ser analisado precisamente no da mesma forma que o atual. É aqui que normalmente uma etapa de recursão é executada.

O tempo e espaço necessários para as chamadas de procedimento recursivas, no entanto, podem ser facilmente

evitado se a recursão for substituída por iteração. Vejamos como isso é implementado.

As linhas 32 a 42 realizam algumas tarefas domésticas para preparar o validador para a análise de um estado recém-gerado. A contagem de profundidade é aumentada, um ponteiro é incrementado para o `stptr` da pilha no nível do usuário, que mantém, entre outros, a trilha de execução. Se a transição foi rotulada como `atômica`, a linha 43 garante que o processo atual irá continuar executando na próxima etapa, precedendo opções para execuções em outras processos. O caso padrão é invocado na linha 46, definindo que todos os atualmente em execução processos devem ser considerados. A etapa de recursão é substituída na linha 48 com um salto para o rótulo `Down`.

No retorno da pseudo recursão, com um salto para o rótulo `Up` na linha 48, todos as variáveis locais relevantes são recuperadas por meio do ponteiro da pilha `stptr`, que, se tudo estiver bem,

aponta exatamente para o local onde foram salvos antes do salto correspondente para Baixa. Em seguida, o vetor de estado é restaurado ao seu valor original executando uma inversão operação que desfaz o efeito da última transição para a frente que foi explorada. o código que faz isso está oculto em um arquivo separado `pan.b` incluído na linha 48.

Arquivo também contém uma chave de caso, que depende do ponteiro para a matriz de transição `t` para

Página 313

302

UM VALIDADOR DE PROTOCOLO

CAPÍTULO 13

aponte o programa para a operação correta a ser executada.

O próprio estado, armazenando informações de estado sobre todos os processos em execução e todas as filas e variáveis atualmente acessíveis são mantidas em uma variável global `agora`, embora isso não seja visível no corpo deste procedimento.

O rótulo Novamente, na linha 23, com o salto correspondente na linha 65, é usado para implementar o mecanismo de recuperação de `tempo limite`. Se a pesquisa travar, isso será notado pela primeira vez online

61 pelo valor zero da variável `n`. Uma verificação rápida é realizada para ver se o deadlock não é de fato um estado final válido. Se este teste falhar, os tempos limite são ativados e segunda tentativa é feita para realizar uma transição com um retorno ao rótulo `novamente`. Se este também falha, a rotina de erro `uerror()` é chamada, o que pode acionar a escrita de um trilha de erro e, opcionalmente, aborta a pesquisa.

As extensões que são necessárias para implementar toda a gama de verificações de exatidão discutido no Capítulo 6, triplicar o tamanho do algoritmo, embora de uma forma não muito excitante. Para verificar reivindicações temporais, por exemplo, a pesquisa executa alternadamente o estado atômico mentos no modelo e na reivindicação. O bit de alternância, que determina para onde olhar para que a próxima instrução seja executada, é adicionado à variável `tau`. Implementar passagem de mensagem de rendezvous, uma variável global `boq` (abreviação de " bloqueado na fila ") é definido após cada operação de envio de encontro. A variável bloqueia todas as outras operações do que uma operação de recepção correspondente. Efetivamente, então, o handshake enviar-receber torna-se uma etapa indivisível, embora o validador a execute como dois transições. As outras extensões tornam o algoritmo um pouco mais difícil de ler, mas não mude de uma forma fundamental. Na discussão, portanto, restringimos nossa auto principalmente para a versão básica. Uma lista do algoritmo completo para exaustiva a validação é fornecida no Apêndice E.

13.4 A MATRIZ DE TRANSIÇÃO

A matriz de transição mostrada na Figura 13.1 desempenha um papel central na pesquisa.

Algumas precauções são tomadas para garantir que não contenha movimentos espúrios isso pode retardar a busca. É construído a partir de elementos do seguinte tipo:

```
typedef struct Trans {
    átomo curto;
    /* esta é uma transição atômica */
    int st;
    /* o próximo estado / instrução
     */
    int forw;
```

```

/* índice para transição direta */
int back;
/* índice para transição de retorno */
struct Trans * nxt; ",
} Trans;

```

Cada proctype na especificação SPIN original define um conjunto de entradas no matriz: uma para cada estado de fluxo de controle. A seguinte matriz é usada para acompanhar eles.

```
Trans * trans [NPROCS] [NSTATES]
```

As transições de um processo do tipo 19 para o estado de fluxo de controle 90 podem ser encontradas através

Página 314

SEÇÃO 13.5
O CÓDIGO VALIDADOR-GERADOR
303

```

o ponteiro
trans [19] [90]
átomo: 0
st: 51
verso: 41
forw: 14
trans [19] [51]
átomo
st
costas
para a frente
pan.m: switch (t-> forw) {
caso 1: ...
caso 2: ...
caso 14: ...
pan.b: switch (t-> back) {
caso 1: ...
caso 2: ...
caso 41: ...

```

Figura 13.2 - Elementos da matriz de transição

O primeiro campo de uma estrutura `Trans` é usado para rotular transições que são atômicas. Um zero entrada significa que a transição é normal e assíncrona. O campo `st` define o estado de sucessor que é alcançado se uma transição `forw` executada com sucesso. O próximo campo `forw` é um índice que identifica a operação correta a ser executada. Ele indexa um switch de transições que é gerado no arquivo de inclusão `pan.m`. Da mesma forma, `back` é um índice em a chave de `pan.b` que identifica a operação que pode desfazer o efeito do `forw` transição de ala e restaurar o vetor de estado ao seu estado original.

Sempre que houver uma escolha não determinística de transições a serem feitas, por exemplo, para SPIN estruturas de seleção e repetição, as opções são colocadas em uma lista vinculada que é conectado por meio do ponteiro `nxt` do elemento de transição (não mostrado na Figura 13.2). Para examinar

em todas as opções executáveis, o loop interno de `new_state ()` simplesmente percorre o lista vinculada.

13.5 O CÓDIGO VALIDADOR-GERADOR

Parece que tudo o que falta fazer para fazer o analisador funcionar é gerar o matriz de transição. Bem, não exatamente. Uma vez que estamos gerando código de qualquer maneira, também podemos gerar rotinas específicas de protocolo para manipular as filas, instanciando o processo instâncias e semelhantes. É um pedaço de código bastante substancial que gera tudo isso informação, mas vale a pena.

Página 315

304
UM VALIDADOR DE PROTOCOLO
CAPÍTULO 13
CANAIS DE MENSAGEM

Antes de discutir o código do gerador do validador em si, vamos dar uma olhada rápida no código que tenta produzir. Das 25 rotinas relevantes que fazem parte de cada analisador gerado, 7 lidar com os canais de mensagem:

```

addqueue ()
delq ()
q_restor ()
qsend ()
qrecv ()
cancelar o envio ()
unrecv ()

```

A primeira rotina, por exemplo, implementa a instrução PROMELA `chan`, e o segunda rotina remove um canal quando ele sai do escopo, ou seja, quando a criação o processo foi encerrado. Para cada tipo diferente de canal de mensagem, uma fila separada o modelo é gerado. Por exemplo,

```

typedef struct Q1 {
unsigned char Qlen;
/* q_size */
unsigned char _t;
/* q_type */
struct {
unsigned fld0: 32;
} conteudo [1];
} Q1;

```

define uma fila com um slot e um campo de mensagem. Dois campos de fila são predefinidos: um especifica o tipo de canal e o outro o número atual de mensagens que a fila correspondente armazena. A rotina `q_restor ()` é usada para trás move para restaurar uma fila excluída ao seu último estado conhecido, pouco antes de uma exclusão com `qdel ()`. As mensagens são anexadas a uma fila com o procedimento `qsend (em, fld0, ...)`. O parâmetro `fld0` indica o valor do primeiro parâmetro da mensagem. O procedimento `qrecv (from, slot, fld, done)` recupera um único campo de mensagem `fld` do slot `slot` em uma fila de . O parâmetro `feito` é definido como um após todos os campos terem sido extraídos e a mensagem pode ser removida da fila. Observe que um único A operação de recepção pode ter vários efeitos colaterais ao definir as variáveis. A definição de um procedimento que lê um parâmetro de mensagem por vez é uma solução geral simples para esse problema.

Cada ação tem um desfazer. As contrapartes de envio e recebimento são nomeadas `unsend ()` e `unrecv ()`, que, respectivamente, removem uma mensagem completa da cauda de um fila ou colocar um de volta em sua cabeça.

PROCESSOS

Três rotinas lidam com processos.

```

addproc ()
delproc ()
p_restor ()

```

Cada tipo de processo é novamente definido em um modelo diferente. Por exemplo, para o

SEÇÃO 13.5
O CÓDIGO VALIDADOR-GERADOR
305

programa factorial que encontramos

```

typedef struct P1 {
/* factorial */
não assinado _t: 2;
/* proctype */
_p sem sinal: 4;
/* Estado
*/
resultado interno;
criança sem sinal;
int n;
unsigned char p;
} P1;

```

O tipo de processo e o estado atual do processo são uma parte padrão do modelo. A largura necessária dos campos de bits é calculada pelo gerador. Esses dois primeiros campos são usado para indexar a matriz de transição. As entradas restantes reservam slots para variedades locais capazes. O procedimento `addproc ()` anexa um desses modelos ao vetor de estado, e `delproc ()` o remove. O procedimento `p_restor ()` é usado em retrocessos para restaura um processo excluído ao seu último estado conhecido.

MANUTENÇÃO DO ESPAÇO DO ESTADO

O espaço de estado pode ser acessado de duas maneiras diferentes, selecionáveis por um pré-processador

diretiva chamada `BITSTATE`. A menos que este nome tenha sido definido, um espaço de estado completo é

construído com um método de pesquisa exaustiva tradicional. Para acessar o espaço de estado, o a rotina `hstore ()` é usada. Visto do procedimento `new_state ()`, tem a seguinte aparência:

```
if (hstore ((char *) & now, vsize))
{
truncs++;
goto Up;
}
nstates++;
```

A variável `vsize` fornece o tamanho atual do vetor de estado em bytes. O global `agora` aponta para ele. Se `hstore ()` retornar o valor booleano `falso`, o estado é novo e deve ser analisado. Como efeito colateral de `hstore ()`, o estado também é armazenado por completo no estado

espaço. Na próxima vez que este mesmo estado for encontrado, a rotina retornará o valor booleano `true`, o que significa que o estado pode ser ignorado. Internamente `hstore ()` usa uma função hash rápida chamada `s_hash ()`.

Se o nome `BITSTATE` for definido explicitamente durante a compilação, mais memória supertrace eficiente, rotinas de memória de espaço de estado de bits são usadas. Eles invocam um duplo função de hash de valor chamada `d_hash ()`. A rotina usa o vetor de estado para calcular dois valores de hash diferentes (consulte o capítulo 11, página 239). Uma verificação é então feita no duas posições de bits no espaço de estado, e se uma correspondência dupla for encontrada, o estado é assumido como tendo sido analisado antes. É assim que funciona:

Página 317

306

UM VALIDADOR DE PROTOCOLO

CAPÍTULO 13

```
d_hash ((unsigned char *) & now, vsize);
j3 = (1 << (J1 & 7)); j1 = J1 >> 3;
j4 = (1 << (J2 & 7)); j2 = J2 >> 3;
if ((SS [j2] & j3) && (SS [j1] & j4))
{
truncs++;
goto Up;
}
SS [j2] |= j3; SS [j1] |= j4;
/* armazenamento */
nstates++;
```

Primeiro `d_hash ()` é chamado para produzir dois valores de hash para os primeiros bytes `vsize` do vetor de estado, armazenado em `agora`. Os valores são escritos nas variáveis globais inteiras `J1`, e `J2`. As próximas operações usam os 3 bits de ordem inferior de `J1` e `J2`, usando o

bit mask 7 e atribui-a a `j3` e `j4`. Os bits restantes são deslocados para baixo por três posições de bits e atribuídos a `j1` e `j2`. O teste

```
if ((SS [j2] & j3) && (SS [j1] & j4))
```

seleciona as posições de bit calculadas e somente se ambos os bits estiverem em uma correspondência é assumido.

A última linha define os dois bits com uma operação binária OU, para garantir uma correspondência futura em o mesmo estado. É a única operação de armazenamento realizada: uma economia na memória de $(8 \cdot vsize - 2)$ bits por estado, assumindo 8 bits por byte.

AS ROTINAS RESTANTES

O restante das rotinas de validação é bastante simples. Existe uma rotina `endstate ()` para determinar se a combinação atual de estados do processo é um fim válido estado, comparando-os com os estados de parada conhecidos em cada processo. Uma rotina `assert ()` verifica as asserções definidas pelo usuário e produz uma trilha de erro se alguma for violada.

Existe uma rotina `r_ck ()` para cada tipo de processo no sistema que executa a verificação de acessibilidade após a pesquisa em profundidade, verificando se todos os controles relevantes estado de fluxo na especificação foi realmente alcançado por pelo menos um dos processos. Duas rotinas principais lidam com a matriz de transição, `configurável ()` e `retrans ()`. O primeiro define a tabela (matriz) com seu conteúdo padrão, conforme produzido pelo gerador, usando a estrutura de árvore de análise. O segundo passa rapidamente pela estrutura

ture otimizá-lo um pouco para a tarefa de validação. As escolhas aninhadas, por exemplo, são reescrito em escolhas únicas, sem, é claro, violar a semântica da PROMELA .

13.6 VISÃO GERAL DO CÓDIGO

O código para o gerador está incluído em quatro arquivos C. No momento em que escrevo, uma contagem dessas rotinas produzidas:

```
$ wc pangen [1-4] .c
424
1341
8812 pangen1.c
501
1784 13595 pangen2.c
102
303
1659 pangen3.c
170
583
4024 pangen4.c
```

Três arquivos de cabeçalho contêm código fixo que está incluído em cada programa gerado:

Página 318

SEÇÃO 13.6
VISÃO GERAL DO CÓDIGO

307

```
$ wc pangen [1-2] .h
910
3363 22276 pangen1.h
127
528
3389 pangen2.h
108
348
2151 pangen3.h
```

E, finalmente, mais um arquivo é usado para implementar a opção de simulação guiada.

```
$ wc pangen5.c
158
489
3292 pangen5.c
```

O código completo para a opção de validação exaustiva pode ser encontrado no Apêndice E.

Aqui destacamos apenas as partes principais.

A rotina que é realmente chamada pelo simulador, se a opção de linha de comando -a for dado, é chamado `gensrc ()`. Ele está incluído no `pangen2.c`. Ele começa criando os cinco os arquivos de destino `pan`. [chtmb] e copiando algum código de `pangen2.h` para eles. isto em seguida, chama o procedimento `putproc ()` uma vez para cada `proctype` básico que foi analisado: uma vez para a descrição do processo de `inicialização`, armazenado na fila de `execução` do simulador , e

uma vez para cada processo na fila de espera. Essas chamadas geram todo o código para o matriz de transição e para o caso muda com as instruções de transição. O restante procedimentos para fazer a execução do analisador estão incluídos principalmente no arquivo `pangen1.c` e são invocados em chamadas no fechamento de `gensrc ()`.

O trabalho real de traduzir partes da árvore de análise em código C acontece apenas dois procedimentos: `putstmt ()` e `undostmnt ()`. Não há mágica aqui, apenas o geração de código, com algum cuidado para reduzir os requisitos de tempo de execução de validações. Os números de estado são fornecidos pelo campo `seqno` nos elementos da árvore de análise: todos

declaração básica é atribuída a um número de sequência único pelas rotinas de análise, como explicado no Capítulo 12. Um conjunto de transições é atribuído a cada estado para indexar o interruptores de caso. As transições são numeradas separadamente (observe que é provável que ser mais transições do que estados se as estruturas de seleção forem usadas), e eles são armazenados em a matriz de transição.

Cada sequência em um corpo de processo resulta em uma chamada de procedimento `putseq ()`. o seqüência é traduzida uma declaração por vez em uma ordem amplamente arbitrária. Cada elemento na sequência que foi traduzida é rotulada como `CONCLUÍDO` no campo de status. Para as transições, os ponteiros entre os elementos são seguidos, pulando tantos intermedie as etapas possíveis, usando a rotina `huntini ()`. O código real que reproduz duz que o efeito de uma transição direta é gerado por `putstmt ()` , listado em

pangen2.c . O código que pode desfazer o efeito, quando a primeira pesquisa de profundidade se desenrola, é gerado por undostmnt () , listado em pangen4.c . Em vez de dar uma detalhada expor todo o código que está sendo gerado, vamos considerar a tradução de um código específico tipo de declaração: uma atribuição. A rotina putstmtnt () contém código que, após substituições de macro, chega a Os seguintes:

Página 319

```
308
UM VALIDADOR DE PROTOCOLO
CAPÍTULO 13
caso ASGN:
fprintf (fd, "(trpt + 1) -> oval =");
putstmtnt (fd, agora-> lft, m, pid);
fprintf (fd, "; \n \t \t");
putstmtnt (fd, agora-> lft, m, pid);
fprintf (fd, "=");
putstmtnt (fd, now-> rgt, m, pid);
quebrar;
Dada a árvore de análise para a atribuição SPIN
nips = 12 + 3 * crunch;
isso é traduzido na sequência
caso 34:
(trpt + 1) -> oval = now.nips;
now.nips = (12 + (3 * ((P1 * this) -> crunch)));
m = 3; goto P333;
assumindo que 34 é o número na matriz de transição atribuída ao atual
transição, nips é uma variável global, uma parte permanente do vetor de estado agora , e
crunch é uma variável local que está acessível através do ponteiro predefinido para o modelo
do processo atual no vetor de estado isso . A primeira linha é um backup do antigo
valor dos nips globais em um campo especial da pilha que é usado para organizar a pesquisa
no procedimento new_state () . Há uma compensação de 1 para explicar o fato de que
oficialmente, não sabemos ainda se a transição será executável ou não. Somente se
a execução é executável se o ponteiro da pilha for aumentado e o valor do backup será
no lugar certo para a operação de desfazer.
```

O código para a geração da operação de desfazer correspondente é o seguinte:

```
caso ASGN:
putstmtnt (tb, agora-> lft, m, pid);
fprintf (tb, "= trpt-> oval");
checkchan (agora-> rgt, m, pid);
quebrar;
```

que para a mesma instrução produz este código

```
caso 28:
now.nips = trpt-> oval;
goto R333;
assumindo novamente que 28 é o índice atribuído à operação desfazer atual na transição
matriz de posição. A chamada adicional em checkchan () no código de desfazer acima é para
certifique-se de que nenhum canal foi criado como efeito colateral da atribuição. Se então,
esses canais devem ser excluídos novamente na transição reversa.
```

13.7 SIMULAÇÃO GUIADA

A última extensão para o código-fonte do simulador a ser discutida é a implementação do procedimento match_trail () . O código pode ser encontrado no arquivo pangen5.c . Procura por a trilha de simulação no arquivo pan.trail , onde o validador o coloca. Em sua forma básica, a trilha tem o seguinte formato:

Página 320

```
SEÇÃO 13.7
SIMULAÇÃO GUIADA
309
0: 0: 1
1: 0: 2
2: 0: 3
3: 0: 4
4: 0: 5
5: 0: 6
6: 0: 8
7: 3: 15
```

Cada linha especifica uma transição em três campos inteiros, separados por dois pontos. O primeiro campo é um número de etapa, contando de zero a qualquer que seja o comprimento da trilha de erro talvez. O segundo campo é o número do processo, com 0 para o processo de `inicialização`, 1 para o primeiro processo que foi iniciado em uma instrução `run` e assim por diante. O último número em cada linha da trilha de erro identifica o estado para o qual o processo se move. o trabalho do simulador é seguir a trilha e tocar em todos os estados listados. Se, por enquanto, nós omitir a recuperação de erros, processos de reivindicação temporal e o tratamento de estados de parada, o código tem a seguinte aparência:

```
1 match_trail ()
2 {ARQUIVO * fd;
3
4 int i, pno, nst;
5
6 if (! (fd = fopen ("pan.trail", "r")))
7 {
8 printf ("spin -t: não foi possível encontrar 'pan.trail' \n");
9 saída (1);
10 }
11 Tval = 1; /* timeouts podem fazer parte da trilha */
12 while (fscanf (fd, "% d:% d:% d \n", & profundidade, & pno, & nst) == 3)
13 {
14 i = nproc - nstop;
15 /* número de procs em execução */
16 para (X = executar; X; X = X-> nxt)
17 if (--i == pno) /* localizar processo pno */
18 quebrar;
19 lineno = X-> pc-> n-> nval;
20 Faz
21 /* trazê-lo para o estado nst */
22 {
23 X-> pc = d_eval_sub (X-> pc, pno, nst);
24 } enquanto (X && X-> pc && X-> pc-> seqno! = nst);
25 }
26 printf ("rotação: a trilha termina após% d etapas \n", profundidade);
27 embrulhar();
28 }
```

Depois de abrir o arquivo de trilha (linhas 5-8), uma diretiva por vez é lida a partir da trilha (linhas 10). O processo certo está localizado (linhas 12-14), e é executado até a direita estado é alcançado (linhas 16-18).

Um estado de parada é identificado pelo novo estado 0, um estado não existente. É executado como um remoção do processo que foi identificado. O código completo para `match_trail()` em O Apêndice E tem verificações extras para prepará-lo para os casos em que o modelo de validação é incapaz de seguir a trilha, por exemplo, se o modelo foi alterado desde que a trilha foi escrita. Nestes casos, o simulador reportará, por exemplo,

```

:: (m <N-1) -> m = m - 1
:: (m <N-1) -> m = m - 1; n = n + 1
od

```

Os dois caminhos de execução quase iguais podem, com a implementação atual do simulador, leva a uma trilha ambígua. O problema pode ser evitado de forma direta, removendo a ambigüidade:

```

Faz
:: (m> = N-1) -> pausa
:: (m <N-1) -> m = m - 1
E se
:: pular
:: n = n + 1
fi
od

```

13.8 ALGUMAS APLICAÇÕES

A opção de análise é chamada a partir do simulador original com o sinalizador `-a`, para instância, como segue:

```
$ spin -a factorial
```

Neste ponto, normalmente dentro de um segundo, geramos um programa que consiste em cinco arquivos C separados: um arquivo de cabeçalho, os dois casos alternam com para frente e para trás transições, um arquivo principal com as rotinas C principais e um arquivo com a matriz de transição e algumas rotinas relacionadas.

```

$ wc pan.?
55
197
1161 pan.b # movimentos para trás
731
2159 14822 rotinas pan.c # c
108
409
2526 pan.h # header
120
482
2925 pan.m # move para frente
129
377
Matriz de transição 2580 pan.t #
1143
3624 24014 no total

```

O programa pode ser compilado de duas maneiras diferentes. O padrão

```
$ cc -o pan pan.c
```

gera um analisador que constrói um espaço de estado completo, descartando qualquer chance de incompletude. Ele fornece 100% de cobertura, a menos que fique sem memória. Opção-aliado, um validador de supertrace mais econômico pode ser gerado com o comando

Página 322

SEÇÃO 13.8
ALGUMAS APLICAÇÕES

311

```
$ cc -DBITSTATE -o pan pan.c
```

Em ambos os casos, a validação é iniciada digitando

```
$ pan
pan: impasse
pan: escreveu pan.trail
... etc
```

Se o validador encontrar um erro, ele grava a trilha de simulação. A trilha é usada com o simulador em qualquer um dos modos discutidos no Capítulo 12, por exemplo, com o `-s` opção:

```
$ spin -t -s factorial
.... etc.
```

onde o novo sinalizador `-t` dirá ao simulador para seguir a trilha em `pan.trail` em vez de realizar uma simulação aleatória.

Considere a seguinte versão da PROMELA do protocolo de Lynch, discutida nos capítulos 2 e 5.

Página 323

312
UM VALIDADOR DE PROTOCOLO
CAPÍTULO 13
1 #define MIN 9

```

/ * primeira mensagem de dados a enviar * /
2 # define MAX 12
/ * última mensagem de dados a enviar * /
3 # define FILL
99
/ * mensagem de preenchimento * /
4
5 mtype = {ack, nak, err}
6
Transferência de 7 proctipo (chan chin, chout)
8 {byte o, i, last_i = MIN;
9
10
o = MIN + 1;
11
Faz
12
:: queixo? nak (i) ->
13
afirmar (i == last_i + 1);
14
chout! ack (o)
15
:: queixo? ack (i) ->
16
E se
17
:: (o <MAX) -> o = o + 1
/* Próximo */
18
:: (o> = MAX) -> o = FILL
/* feito */
19
fi;
20
chout! ack (o)
21
:: queixo? err (i) ->
22
chout! nak (o)
23
od
24}
25
Canal de 26 proctype (canal de entrada, saída)
27 {byte md, mt;
28
Faz
29
:: in? mt, md ->
30
E se
31
:: out! mt, md
32
:: fora! err, 0
33
fi
34
od
35}
36
37 init
38 {chan AtoB = [1] de {mtype, byte};
39
chan BtoC = [1] de {mtype, byte};
40
chan CtoA = [1] de {mtype, byte};
41
atômica {
42
executar transferência (AtoB, BtoC);
43
canal de execução (BtoC, CtoA);
44
executar transferência (CtoA, AtoB)
45
};
46
AtoB! Err, 0
/* start */
47}

```

Algumas mensagens de dados inteiros são inseridas no sistema para nos permitir olhar pelo menos algumas trocas de mensagens. Também adicionamos um tipo de processo para modelar o esperado comportamento do canal de comunicação: mensagens de distorção aleatória. Podemos simular o comportamento do sistema com o antigo código do simulador, por exemplo

Página 324

SEÇÃO 13.8
ALGUMAS APLICAÇÕES

313

```
$ spin -s lynch
proc 0 (_init)
linha 46, Send err, 0
-> fila 1 (AtoB)
proc 1 (transferência)
linha 22, Send nak, 10 -> queue 2 (chout)
proc 2 (canal)
linha 31, enviar nak, 10 -> fila 3 (saída)
proc 3 (transferência)
linha 14, Send ack, 10 -> queue 1 (chout)
... etc.
```

Isso pode ou não atingir a violação de asserção, dependendo de como o não determinante ismo é resolvido em cada etapa.

Para uma validação da mesma especificação, geramos e compilamos o programa de validação grama, vamos supor no modo supertrace, da seguinte maneira:

```
$ spin -a lynch
$ cc -o pan pan.c
```

Agora temos um programa executável chamado pan . Para ver quais opções ele aceita realizar a pesquisa, podemos tentar

```
$ pan -?
opção desconhecida
-cN parar no enésimo erro (padrão = 1)
-Eu encontro loops de não progresso
-mN profundidade máxima N (padrão = 10k)
-wN tabela de hash de  $2^N$  entradas (padrão = 18)
```

Podemos, por exemplo, definir a profundidade máxima de pesquisa (o tamanho da pilha de backtrace) para outro valor diferente do padrão de 10.000 passos, ou podemos alterar o tamanho do hash tabela.

Com o armazenamento de espaço de estado completo, o tamanho da tabela hash deve ser maior que ou igual ao número total de estados alcançáveis que é esperado, para evitar um séria penalidade de tempo para a resolução das colisões de hash (consulte o Capítulo 11).

No modo supertrace, o tamanho da tabela hash é igual ao número de bits no espaço de estado, então o sinalizador -w realmente seleciona o tamanho real do espaço de estado que é usado para a pesquisa. Por padrão, este espaço de estado é definido como $2^{22} = 4.194.304$ bits = 524.288 bytes. O tamanho do espaço de estado determina o número máximo de afirmações que podem ser analisados. Para o caso padrão, é cerca de $4.194.304 / 2 = 2.097.152$ estados, independente do tamanho do vetor de estado. (Nesta implementação ção de dois bits são usados para cada estado armazenado.)

A cobertura da pesquisa será menor à medida que nos aproximamos desse limite. Nós discutimos um indicador dessa cobertura, o fator hash, com alguns outros exemplos mais tarde. Nós primeiro experimente o validador no modo exaustivo.

Página 325

314
UM VALIDADOR DE PROTOCOLO
CAPÍTULO 13

```
$ pan
afirmação violada (i == (last_i + 1))
pan: abortado (na profundidade 53)
pan: escreveu pan.trail
pesquisa de espaço de estado completo para:
violações de asserção e estados finais inválidos
pesquisa não foi completada
vetor 56 byte, profundidade alcançada 53, erros: 1
58 estados, armazenados
2 estados, vinculados
1 estado, correspondido
total:
61
conflitos de hash: 0 (resolvido)
```

(tamanho 2^18 estados, stackframes: 0/16)

No final de cada execução, o validador imprime o número de estados armazenados, vinculados e coincide. Os estados armazenados são estados que foram adicionados ao espaço de estado, seja por completo

ou na forma compactada como dois bits, dependendo de como o programa foi compilado.

Estados vinculados são estados que foram encontrados em uma sequência atômica, nenhum estado verificações de espaço são executadas neles. Os estados correspondentes são os que foram analisados e depois revisitado.

O validador encontrou um erro documentado no arquivo `pan.trail`. Nós podemos agora avalie essa trilha de erro para o simulador para ver precisamente o que está acontecendo. Para instância:

```
$ spin -t -s -r lynch
proc 0 (_init)
linha 46, Send err, 0
-> fila 1 (AtoB)
proc 1 (transferência)
linha 21, Erro de recebimento, 0
<- fila 1 (queixo)
proc 1 (transferência)
linha 22, Send nak, 10 -> queue 2 (chout)
proc 2 (canal)
linha 29, Recv nak, 10 <- fila 2 (entrada)
proc 2 (canal)
linha 32, Send err, 0
-> fila 3 (saída)
proc 3 (transferência)
linha 21, Erro de recebimento, 0
<- fila 3 (queixo)
proc 3 (transferência)
linha 22, Send nak, 10 -> queue 1 (chout)
proc 1 (transferência)
linha 12, Recv nak, 10 <- fila 1 (queixo)
proc 1 (transferência)
linha 14, Send ack, 10 -> queue 2 (chout)
....
proc 3 (transferência)
linha 21, Erro de recebimento, 0
<- fila 3 (queixo)
proc 3 (transferência)
linha 22, Send nak, 99 -> queue 1 (chout)
proc 1 (transferência)
linha 12, Recv nak, 99 <- fila 1 (queixo)
spin: "lynch" linha 13: afirmação violada
#processes: 4
_p = 3
proc 3 (transferência)
linha 11 (estado 15)
proc 2 (canal)
linha 28 (estado 6)
proc 1 (transferência)
linha 13 (estado 3)
proc 0 (_init) linha 47 (estado 6)
4 processos criados
```

A execução do simulador pode ser repetida com diferentes sinalizadores, por exemplo, imprimindo valores de variáveis

e estados de processo, até que a causa do erro seja determinada e possa ser reparada.

13.9 COBERTURA NO MODO SUPERTRACE

A cobertura da pesquisa no modo supertrace é determinada pelo número de hash colisões que ocorrem. Esse número, é claro, geralmente é desconhecido. Pode ser determinante extraído comparando uma execução com armazenamento de estado total a uma execução com um espaço de estado de bit, mas

isso nem sempre é viável. O número de colisões de hash, no entanto, depende dos críticos na proporção do tamanho da tabela de hash, ou seja, o número de bits no estado espaço e o número de estados armazenados. Chamamos esse fator de *fator hash*. Isto é calculado pelo validador após cada execução como o tamanho da tabela hash dividido pelo número de estados armazenados. Um número alto (mais de 100) se correlaciona com uma boa cobertura era. Números baixos (perto de 1) indicam cobertura ruim.

Uma vez que armazenamos dois bits por estado no modo supertrace, o fator de hash pode estar em qualquer lugar de 2^N até e incluindo 0,5, onde N pode ser definido pelo usuário para pegar o máximo quantidade de memória disponível na máquina de destino. (Para espaço de estado completo armazenamento, o limite inferior do fator de hash é zero.) Por meio de testes empíricos com e o espaço de estado de bits é executado, pode ser confirmado que um fator de hash de 100 ou mais virtualmente garante uma cobertura de 99% a 100% de todos os estados alcançáveis. Por exemplo, Tabela 13.1 fornece os resultados dos testes com um modelo de protocolo que tem 334.151 estados alcançáveis. A execução original neste caso foi a versão completa do espaço de estado, usando 45,6 Mbyte de memória. Ele armazenou todos os estados e resolveu um total de 66.455 conflitos de hash no caminho. A execução foi repetida, primeiro com uma validação de supertrace usando o sinalizador `-w25`, dando um fator de hash de 100,9 e uma cobertura de 99,45%. Em virtude da função hash de bit duplo ção, o número de conflitos de hash é substancialmente menor do que na primeira execução. O pré-O número exato pode ser encontrado subtraindo o número de estados alcançados e armazenados em a primeira execução a partir do número de estados alcançados (mas não armazenados) na segunda execução. No cada uma das próximas três validações de supertrace, reduzimos pela metade o fator de hash usando o bandeiras `-w24`, `-w23`, e `-w22`. A última execução não usa mais do que 6,3 Mbyte de memória, de quais 6 Mbyte, tanto na versão de armazenamento de espaço de estado completo quanto na versão de supertrace, é usado para armazenar a trilha de backup que tinha 300.000 passos para todas as execuções deste protocolo de teste. (Isso também explica por que a quantidade de memória necessária não apenas metade de cada vez que o argumento para o sinalizador `-w` é diminuído.)

Tabela 13.1 - Correlação entre o fator de hash e a cobertura

Pesquisa de estados de fator de hash armazenados colisões de hash cobertura usada

exhaustivo

-

334.151

66.455

45,6 Mb

100%

supertrace

100,9

332.316

1.835

9,9 Mb

99,45%

supertrace

50,9

329.570

4.581

7,9 Mb

98,62%

supertrace

25,7

326.310

7.841

6,9 Mb

97,65%

supertrace

13,0

322.491

11.660

6,3 Mb

96,51%

Para comparação, uma execução do método de armazenamento de espaço de estado completo que é restrito a 6,3

Mbyte de memória para armazenar seu espaço de estado, previsivelmente, obtém menos cobertura. o a busca exaustiva efetivamente se degrada em uma busca parcial não controlada, como ilustrado

tratado na Tabela 13.2.

Página 327

316

UM VALIDADOR DE PROTOCOLO
CAPÍTULO 13

Tabela 13.2 - Cobertura de pesquisas parciais

Pesquisa de estados de fator de hash armazenados colisões de hash cobertura usada

exhaustivo

-

83.961

389.671

6,3 Mb

25,12%

supertrace

13,0

322.491

11.660

6,3 Mb

96,51%

O espaço de estado do bit é claramente o método de escolha aqui.

13,10 RESUMO

Muitas ferramentas de validação automatizadas exigem um esforço considerável do usuário para traduzir um modelo de validação no código de baixo nível que é usado para executar o validador. O a interpretação dos relatórios de erro produzidos por essas ferramentas pode exigir a consideração hábil engenhosidade humana. Com o gerador simulador e validador SPIN , e o vali- linguagem de tradução PROMELA , tentamos fornecer um ambiente de design de alto nível em que tudo, desde protocolos simples, até projetos completos para distribuição sistemas de passagem de mensagens podem ser completamente testados e depurados antes de serem implementado. Essas ferramentas podem nos ajudar a lidar de forma eficaz com as notoriamente difíceis problemas de assincronia e simultaneidade. As ferramentas são portáteis, poderosas e eficiente.

EXERCÍCIOS

13-1. Considere como o código deve ser alterado para substituir a ordem de pesquisa em profundidade por largura primeiro. Quais são os requisitos de memória?

13-2. Modifique o código para otimizar a implementação de rotinas de envio e recebimento, e ure seu efeito.

13-3. Adicione uma opção ao SPIN para restringir as execuções de validação para execuções " justas ". Isto opção é baseada na suposição de um " planejador de processo justo ". Isso significa que qualquer processo que pode executar uma instrução é considerado habilitado para fazê-lo dentro de um tempo finito. Todas as execuções infinitas (ciclos) que violam essa suposição de justiça podem ser ignoradas. Todos loops de não progresso ou ciclos de aceitação que violam esta suposição devem ser ignorado. Dica: execute uma verificação extra antes de relatar qualquer erro nas sequências cíclicas.

13-4. (EA Emerson - P. van Eijk) Implementar um método que possa dar uma melhor previsão do cobertura de validações parciais de supertrace. Faça isso iniciando uma validação de supertrace por selecionar 1000 estados aleatoriamente no espaço de estados. (Como?) Armazene esses estados completos em um separar a tabela de pesquisa e verificar durante a validação do supertrace quantos desses 1000 estados são alcançados. A fração de estados atingidos é um indicativo da cobertura.

Quão confiável é a estimativa? Quão caro?

13-5. Modifique o gerador do validador para permitir a geração automática de implementos de protocolo atividades mentais de PROMELA código. Observe que o código C já foi gerado para todas as transições e ações. Substitua o procedimento de pesquisa `new_state()` por um planejador, conforme usado no código do simulador (Capítulo 12), e permite que certos canais sejam identificados como especiais canais de dispositivo (por exemplo, arquivos) que podem ser vinculados a rotinas de biblioteca C que acessam os canais de E / S brutos. Sua solução não precisa conter mais de duas páginas de código.

Página 328

CAPÍTULO 13
NOTAS BIBLIOGRÁFICAS

317

NOTAS BIBLIOGRÁFICAS

O validador descrito possui vários predecessores de escopo e desempenho variáveis.

Para os interessados, os artigos de Holzmann [1984a, 1985, 1988] documentam mais mudanças significativas. O último desses documentos contém uma explicação detalhada da modelo de vetor de estado e o método de espaço de estado de bits. O método descrito em Holzmann [1988] é a única versão desta sequência que atinge um melhor desempenho mance, em termos de tempo de execução e uso de memória, do que o método descrito aqui. isto requer substancialmente mais código para implementar.

Página 329

USANDO O VALIDADOR 14

318 Introdução 14.1
318 Um Protocolo de Telégrafo Ótico 14.2
320 Algoritmo de Dekker 14.3
322 Uma Validação Maior 14.4
325 Validação de Controle de Fluxo 14.5
Validação de camada de sessão 334 14.6
347 Resumo 14.7
347 exercícios
347 Notas Bibliográficas

14.1 INTRODUÇÃO

É hora de colocar em uso as ferramentas que desenvolvemos nos últimos três capítulos. Primeiro a molhar os pés, vejamos dois exemplos simples. O primeiro é a reconstrução de um protocolo usado nos telégrafos ópticos em 1794 (consulte o Capítulo 1). O segundo é um pequeno, mas muito importante, exemplo do Capítulo 5: o algoritmo de Dekker para fornecer dois processos concorrentes acessam mutuamente exclusivos a uma seção crítica de seu código.

14.2 UM PROTOCOLO DE TELEGRAFIA ÓTICA

Os detalhes dos protocolos de comunicação usados nos telégrafos ópticos construídos no finais do século 18 são difíceis de encontrar. A melhor fonte é um livreto publicado pela O inventor sueco de um telégrafo de venezianas Edelcrantz [1796], que vem completo com tabelas de codificação e descrições informais e elaboradas da codificação e assinatura exigidas métodos de naling. Todas as estações ao longo de uma linha, exceto a primeira e a última, tiveram que monitorar duas estações vizinhas quanto ao tráfego de entrada. Dois operadores de telégrafo eram portanto, geralmente em serviço. No modelo de validação que construímos para o telégrafo óptico vamos, portanto, também usar dois processos assíncronos, um para modelar as ações de cada operador.

Para transferir uma mensagem, o operador de envio teve que definir o telégrafo em sua estação para um sinal de `inicio` especial , que teve que ser confirmado com um sinal de `atenção` do estação de recepção. O sinal de início pode então ser removido, e a primeira mensagem transferido. Cada mensagem teve que ser reproduzida fielmente pelo receptor antes do o remetente pode removê-lo do telégrafo. (O sistema Edelcrantz também permitiu o uso de um sinal de `erro` especial , mas não o modelaremos aqui.) Fim de uma mensagem foi sinalizado com um sinal de `parada` especial . Depois que o sinal de `parada` foi transferido, o o telégrafo foi liberado para outro tráfego, por exemplo, para o tráfego que flui no sentido contrário direção.

Claramente, um operador não poderia usar o telégrafo em sua estação para receber ou
318

Página 330

319
tráfego de saída se seu colega já o estiver usando. Modelamos o estado do tele-
gráficos com um array booleano `ocupado [N]` , onde N é o número de estações telegráficas.
O modelo de validação abaixo coloca três estações em um anel (é improvável que fossem
já usado dessa forma), com dois operadores por estação, isso dá um total de seis processos.

```
1 #define true
1
2 #define false
0
3
4 bool ocupado [3];
5
6 canais para cima [3] = [1] de {byte};
7 canais para baixo [3] = [1] de {byte};
8
9 mtype = {iniciar, atenção, dados, parar}
```

```

10
Estação de 11 proctype (byte id; chan in, out)
12 {fazer
13
:: in? start ->
14
atômico {! ocupado [id] -> ocupado [id] = verdadeiro};
15
fora! atenção;
16
Faz
17
:: in? data -> out! data
18
:: em? parar -> quebrar
19
od;
20
fora! pare;
21
ocupado [id] = falso
22
:: atômico {! ocupado [id] -> ocupado [id] = verdadeiro};
23
fora! começar;
24
em? atenção;
25
Faz
26
:: out! data -> in? data
27
:: out! stop -> break
28
od;
29
em? parar;
30
ocupado [id] = falso
31
od
32}
33
34 init {
35
atômica {
36
estação de execução (0, up [2], down [2]);
37
estação de execução (1, up [0], down [0]);
38
estação de execução (2, up [1], down [1]);
39
40
estação de execução (0, down [0], up [0]);
41
estação de execução (1, para baixo [1], para cima [1]);
42
estação de execução (2, para baixo [2], para cima [2])
43
}
44}

```

Se rodarmos uma simulação aleatória neste protocolo, rapidamente encontraremos um problema.

```

320
USANDO O VALIDADOR
CAPÍTULO 14
$ spin -r -s óptico
proc 6 (estação)
linha 23, Send start
-> fila 3 (saída)
proc 5 (estação)
linha 23, Send start
-> fila 2 (saída)
proc 4 (estação)
linha 23, Send start
-> fila 1 (saída)
proc 3 (estação)
linha 13, inicio de recebimento

```

```

<- fila 2 (entrada)
proc 2 (estação)
linha 13, inicio de recebimento
<- fila 1 (entrada)
proc 1 (estação)
linha 13, inicio de recebimento
<- fila 3 (entrada)
#processes: 7
proc 6 (estação)
linha 24 (estado 19)
proc 5 (estação)
linha 24 (estado 19)
proc 4 (estação)
linha 24 (estado 19)
proc 3 (estação)
linha 14 (estado 4)
proc 2 (estação)
linha 14 (estado 4)
proc 1 (estação)
linha 14 (estado 4)
proc 0 (_init) linha 44 (estado 8)
7 processos criados

```

A simulação fica paralisada depois que todas as três estações enviam simultaneamente o `início` mensagem. As três mensagens são recebidas, mas a armadilha de deadlock é fechada. Três os operadores estão à espera de uma confirmação das suas mensagens de `início`, os outros três estão esperando que o telegrafo seja divulgado por seus colegas antes que eles possam enviar o sinal de atenção necessário. No estado de deadlock, três processos estão na linha 14 e os outros três na linha 24 na estação de origem do proctipo.

O problema do impasse é uma variante curiosa da famosa *filosofia gastronômica* de Dijkstra. problema de *phers*.

14.3 ALGORITMO DE DEKKER

Para construir um modelo de validação útil, estendemos o algoritmo de Dekker com dois booleanos variáveis, `ain` e `bin`, como segue:

```

1 #define true
1
2 #define false
0
3 # define Aturn
falso
4 #define Bturn
verdadeiro
5
6 bool x, y, t;
7 bool ain, bin;
8
9 proctipo A ()
10 {x = verdadeiro;
11
t = Bturn;
12
(y == falso || t == Aturn);
13
ain = verdadeiro;
14
assert (bin == false); /* seção Crítica */
15
ain = falso;
16
x = falso
17}
18

```

```

321
19 proctipo B ()
20 {y = verdadeiro;
21
t = Aturn;
22
(x == falso || t == Bturn);
23
bin = verdadeiro;
24
assert (ain == false); /* seção Crítica */

```

```

25
bin = false;
26
y = falso
27}
28
29 init
30 {corrida A (); corrida B ()
31}

```

As variáveis `ain` e `bin` são definidas como verdadeiras apenas quando o processo `A ()` ou `B ()`, respec-

efetivamente, entra em sua seção crítica. Uma simples declaração `assert ()` pode ser usada para verificar

que ambos os processos não podem estar em suas seções críticas ao mesmo tempo.

Primeiro, vamos fazer uma simulação aleatória. O modelo de validação acima é armazenado em um arquivo

chamado "dekker." Tentamos

```

$ spin dekker
3 processos criados

```

Nenhuma violação de asserção é relatada, mas a execução não é muito informativa. Nós tentamos novamente, desta vez imprimindo todas as declarações.

```

$ spin -p dekker
proc 0 (_init) linha 31 (estado 2)
proc 1 (A)
linha 11 (estado 2)
proc 0 (_init) linha 31 (estado 3)
proc 2 (B)
linha 21 (estado 2)
proc 1 (A)
linha 12 (estado 3)
proc 2 (B)
linha 22 (estado 3)
proc 1 (A)
linha 13 (estado 4)
proc 1 (A)
linha 14 (estado 5)
proc 1 (A)
linha 15 (estado 6)
proc 1 (A)
linha 16 (estado 7)
proc 1 (A)
linha 17 (estado 8)
proc 2 (B)
linha 23 (estado 4)
proc 2 (B)
linha 24 (estado 5)
proc 2 (B)
linha 25 (estado 6)
proc 2 (B)
linha 26 (estado 7)
proc 2 (B)
linha 27 (estado 8)
proc 2 (B)
termina
proc 1 (A)
termina
proc 0 (_init) termina
3 processos criados

```

Podemos repetir isso algumas vezes para ganhar confiança de que de fato o algoritmo parece executar conforme anunciado. Mas isso não é prova. Podemos facilmente fazer uma pesquisa exaustiva para

estabelecer de uma vez por todas que o algoritmo está correto. Primeiro, geramos e compilamos o analisador.

```

322
USANDO O VALIDADOR
CAPÍTULO 14
$ spin -a dekker
$ cc -o pan pan.c

```

Isso é tudo que há para fazer; exceto para a validação exaustiva, é claro.

```

$ pan
pesquisa de espaço de estado completo para:
violações de asserção e estados finais inválidos

```

```

vetor 16 byte, profundidade alcançada 19, erros: 0
81 estados, armazenados
0 estados, vinculado
36 estados, combinados
total:
117
conflitos de hash: 0 (resolvido)
(tamanho máximo 2^18 estados, stackframes: 3/0)
não alcançado em proctype _init:
atingiu todos os 3 estados
não alcançado no proctipo B:
atingiu todos os 8 estados
não alcançado no proctipo A:
atingiu todos os 8 estados

```

As primeiras duas linhas nos informam que tipo de validação está sendo realizada. Uma vez que não há reivindicações porais ou estados de progresso foram definidos, uma pesquisa básica para violações de asserção

e estados finais inválidos são executados. A próxima linha diz que o vetor de estado para este modelo de validação ocupou 16 bytes de memória, a sequência de execução única mais longa tinha 19 passos de comprimento e, infelizmente, não foram encontrados erros. Um total de 81 sistemas alcançáveis

dez estados foi registrado. 36 vezes as execuções simbólicas realizadas pelo validador retornou o sistema a um estado alcançável que foi analisado antes. Não havia hash conflitos. Se houvesse algum, uma vez que esta é uma busca completa de espaço de estado, eles teriam sido resolvidos com uma lista vinculada na tabela de hash. Todos os estados em todos os processos, finalmente, foram encontrados para ser alcançáveis e, implicitamente, provamos que nenhuma sequência de execução pode violar as afirmações de correção: o validador tentou todas elas. Sem dúvida, o algoritmo impõe exclusão mútua.

14.4 A VALIDAÇÃO MAIOR

A validação do projeto do protocolo de transferência de arquivos do Capítulo 7 é um trabalho maior. O projeto completo exigia que abordássemos um grande número de pequenos problemas, todos eles que poderia ser resolvido com algum grau de confiança. Mas tendo resolvido estes sub-problemas nosso trabalho não está feito. A consistência lógica do design completo é difícil para avaliar. Todas as pequenas soluções juntas definem o comportamento de um composto maior máquina que pode interagir com seu ambiente de várias maneiras.

Depois de concluir o projeto, a máquina composta responderá de uma maneira ou outro a todas as possíveis sequências de eventos que o ambiente pode oferecer: o aqueles que tínhamos em mente quando fizemos o design inicial, e todos aqueles que nunca pensei em. Um designer de protocolo aprende rapidamente que a segunda classe de sequências é geralmente maior que o primeiro. Nossa trabalho aqui é descobrir se, apesar disso, o projeto crie os critérios para o protocolo são atendidos.

SEÇÃO 14.4
UMA VALIDAÇÃO MAIOR
323

Uma lista completa do modelo de protocolo, conforme validado aqui, é fornecida no Apêndice F. Se todos vai bem, podemos provar ou contestar, por exemplo, que este protocolo está livre de deadlocks, pode se recuperar facilmente de abortos do usuário e transmite dados de forma confiável no presença de erros de transmissão.

O protocolo completo contém 12 processos assíncronos e 20 canais de mensagens. o modelo é de uma complexidade realista e fornece um bom caso de teste para a aplicabilidade de nossas ferramentas. É tentador começar tentando realizar uma validação exaustiva do modelo completo. Uma validação exaustiva direta do modelo, no entanto, funciona unicamente nas armadilhas discutidas no Capítulo 11; nunca pode haver memória suficiente ou tempo suficiente para concluí-lo. Um limite de memória arbitrariamente colocado de 16 Mbytes, para por exemplo, se esgota rapidamente e produz o seguinte resultado. O máximo a profundidade da pesquisa foi adivinhada.

```

$ spin -a pftp
# o modelo completo, conforme listado no aplicativo. F
$ cc -DMEMCNT = 24 -o pan pan.c # definir limite de memória em 2^24 bytes
$ pan -m15000

```

```

# profundidade máxima de pesquisa 15.000 passos
pan: sem memória
pesquisa completa de espaço de estado para:
violações de asserção e estados finais inválidos
pesquisa não foi completada
vetor 256 bytes, profundidade atingiu 7047, erros: 0
57316 estados, armazenados
44880 estados, vinculados
76300 estados, combinados
total: 178496
conflitos de hash: 10319 (resolvido)
(tamanho máximo 2^18 estados, stackframes: 0/1009)
memória usada: 16777241

```

A busca exaustiva deteriorou-se em uma busca parcial não controlada quando esgotou os 16 Mbytes de memória disponível. Como argumentado no Capítulo 11, um estado de bit técnica de espaço pode alcançar uma cobertura melhor nesses casos, mesmo dentro de limites de memória. Por exemplo, com uma arena de memória 8 vezes menor do que antes, um pouco

a análise do espaço de estados atinge aproximadamente 40 vezes mais estados:

```

$ cc -DMEMCNT = 21 -DBITSTATE -o pan pan.c # 8 vezes menos memória
$ pan -w22 -m15000
# 2^22 = 4 Mbit = 0,5 Mbyte de espaço de estado
bit state space search por:
violações de asserção e estados finais inválidos
vetor 256 bytes, profundidade atingiu 14.999, erros: 0
2136023 estados, armazenados
1987936 estados, vinculados
3499761 estados, correspondidos
total: 7623720
fator de hash: 1,963603 (melhor cobertura se > 100)
(tamanho máximo 2^22 estados, stackframes: 0/2365)
memória usada: 1507425
# espaço de estado + 15.000 pilha de slots
não alcançado em proctype _init:
atingiu todos os 13 estados

```

Página 335

324

USANDO O VALIDADOR

CAPÍTULO 14

```

não alcançado em proctype data_link:
linha 20 (estado 14)
alcançado: 13 de 14 estados
não alcançado no proctype fc:
...
alcançado: 61 de 73 estados
não alcançado no proctype fserver:
linha 29 (estado 30)
alcançado: 29 de 30 estados
não alcançado na sessão proctype:
...
alcançado: 96 de 99 estados
não alcançado no proctipo presente:
...
alcançado: 32 de 34 estados
não alcançado em proctype userprc:
atingiu todos os 17 estados

```

O analisador inspecionou 7,6 milhões de estados do sistema composto, dos quais mais de 2 milhões eram distintos. As descrições de estado tinham 256 bytes de comprimento. Existem, como sempre, uma série de indicações de que a análise estava incompleta.

O fator de hash é muito baixo. O fator de hash deve ser superior a cem, antes que possamos esteja confiante de que há cobertura suficiente (Capítulo 13).

O limite de profundidade de 15.000 passos era muito pequeno (observe a profundidade alcançada de 14.999

passos). A pesquisa teria que ser repetida com um limite de profundidade maior para evitar truncamento.

A lista de códigos não alcançados, abreviada acima, mostra que nem todas as partes do modelo foram exercitados.

Podemos aumentar um pouco a cobertura escolhendo uma área de memória maior, mas os resultados não são encorajadores:

```

$ cc -DMEMCNT = 23 -DBITSTATE -o pan pan.c
# usa mais memória
$ pan -w25 -m45000
# permite até 32 milhões de estados
bit state space search por:

```

```

violações de asserção e estados finais inválidos
vetor de 256 bytes, profundidade alcançou 36569, erros: 0
18302437 estados, armazenados
19482180 estados, vinculados
33989843 estados, correspondentes
total: 71774460
fator de hash: 1,833331 (melhor cobertura se > 100)
(tamanho máximo 2^25 estados, stackframes: 0/6167)
memória usada: 6857209
...

```

Desta vez, em menos da metade da arena da memória do primeiro, a "busca completa" analisamos mais de 300 vezes mais estados usando o algoritmo supertrace. Ainda assim, o indicações são que a cobertura é ruim. Se quisermos fazer melhor, temos que ter um diferente aproximação. Em vez de realizar um único teste monolítico de todas as camadas ao mesmo com o tempo, podemos dividir o problema de validação em partes menores e mais gerenciáveis. (Veja também a discussão de técnicas de gerenciamento de complexidade, como redução e

Página 336

SEÇÃO 14.5
VALIDAÇÃO DE CONTROLE DE FLUXO
325

generalização nos capítulos 8 e 11.) Na fase de projeto já fizemos um esforço para separar problemas ortogonais, como controle de erros, controle de fluxo e controle de sessão. Este esforço pode valer a pena agora. A exatidão da camada de controle de fluxo, por exemplo, é completamente independente da correção da camada de controle de sessão. Podemos lá- para reduzir a complexidade da validação substancialmente, validando o protocolo módulos separadamente.

Design por refinamento passo a passo e validação por abstração passo a passo são técnicas complementares.

Cada validação separada pode alcançar uma cobertura muito melhor do que uma valida monolítica ção de todas as camadas juntas.

Vejamos as camadas uma por uma. A exatidão do controle de erro depende de a precisão do método de soma de verificação, que foi discutido no Capítulo 3. Validação de um algoritmo de soma de verificação por análise exaustiva de alcançabilidade seria inadequada apropriado; é um mero cálculo. Nós olhamos para a validação do protocolo principal camadas: controle de fluxo, controle de sessão e apresentação. Baseamos a validação no suposições feitas anteriormente sobre o comportamento dos três ambientes processos: o usuário, o servidor de arquivos e o link de dados.

14.5 VALIDAÇÃO DO CONTROLE DE FLUXO

O principal requisito de correção para a camada de controle de fluxo é que ela não pode perder ou reordene as mensagens, apesar de o módulo de protocolo inferior perder mensagens.

No Capítulo 7, expressamos uma correção da camada de controle de fluxo, usando uma rotulagem de mensagens com três cores, vermelho , branco , e azul . Para realizar a validação, usamos o processo emissor e receptor de teste descrito no Capítulo 7, estendido com alguns código. Antes de qualquer dado ser transferido, o remetente de teste deve sincronizar os dois fluxos processos da camada de controle. O código é emprestado da camada de sessão original (ver Capítulo 7 e Apêndice F).

```

proctype test_sender (bit n)
{
    byte par, alternar;
    ses_to_flow [n]! sincronizar, alternar;
    Faz
    :: flow_to_ses [n]? sync_ack, par ->
    E se
    :: (par! = alternar)
    :: (par == alternar) -> pausa
    fi
    :: tempo limite ->
    ses_to_flow [n]! sincronizar, alternar
    od;
    toggle = 1 - alternar;
    Faz
    :: ses_to_flow [n]! branco
    :: ses_to_flow [n]! vermelho -> pausa
    od;
}

```

Página 337

```
Faz
:: ses_to_flow [n]! branco
:: ses_to_flow [n]! blue -> break
od;
Faz
:: ses_to_flow [n]! branco
:: pausa
od
}
proctype test_receiver (bit n)
{
Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? vermelho -> quebra
:: flow_to_ses [n]? blue -> assert (0)
od;
Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? vermelho -> assert (0)
:: flow_to_ses [n]? blue -> break
od;
fim:
Faz
:: flow_to_ses [n]? branco
:: flow_to_ses [n]? vermelho -> assert (0)
:: flow_to_ses [n]? blue -> assert (0)
od
}
```

O último ciclo no receptor foi rotulado como um estado final. É onde esperaríamos o processo receptor esteja em todos os estados finais válidos do sistema. Não é aconselhável confiar em o sistema atinge um estado de deadlock quando uma mensagem incorreta é recebida. o processo receptor bloqueia em recepções não especificadas, mas os outros processos podem con continuar, por exemplo, com retransmissões. Por esse motivo, uma violação de asserção explícita é forçado no modelo de validação acima.

Este emissor e receptor de teste modelam a camada de protocolo superior para a camada de controle de fluxo

processo. A camada de protocolo inferior é o enlace de dados. Foi modelado da seguinte forma:

```
proctype data_link ()
{tipo de byte, seq;
fim:
Faz
:: flow_to_dll [0]? tipo, seq ->
E se
:: dll_to_flow [1]! tipo, seq
:: pular 7 * perder *
fi
:: flow_to_dll [1]? tipo, seq ->
E se
:: dll_to_flow [0]! tipo, seq
:: pular 7 * perder *
fi
```

Página 338

A única função do modelo de enlace de dados é simular a perda de mensagens. Há sim, no entanto, uma maneira equivalente e mais simples de modelar o mesmo comportamento. Podemos nos conectar os dois processos de controle de fluxo diretamente e modificá-los para descartar aleatoriamente qualquer mensagens que chegam. Essa redução nos permite remover dois processos e duas mensagens canais sábios do modelo pela adição de apenas uma cláusula à parte do receptor o processo da camada de controle de fluxo (consulte o Apêndice F).

```
#if LOSS
:: err_to_flow [N]? type, m / * perde qualquer mensagem * /
#fim se
```

Usamos uma diretiva de pré-processador `LOSS` para habilitar ou desabilitar a possibilidade de perda de mensagem nas validações. (A mensagem é recebida, mas não respondida.) No modelo de validação de camada de controle de fluxo listado no Apêndice F, há um outro pré-processo

diretiva sor, denominada `DUPS`. Pode ser usado para modelar a possibilidade de mensagens duplicadas sábios desencadeando retransmissões prematuras, ou seja, a retransmissão de mensagens que não estão realmente perdidos. Outra etapa em nosso esforço para reduzir a complexidade do validação pode ser agrupar o código em declarações atômicas sempre que pudermos fazer isso com segurança, e para combinar o emissor e o receptor de teste em um único testador de nível superior. (Veja a composição mental, discutida nos Capítulos 8 e 11.) O código completo para o testador superior tem a seguinte aparência:

```

1 proctipo superior ()
2 {byte s_state, r_state;
3
4 tipo de byte, alternar;
5
6 ses_to_flow [0]! sincronizar, alternar;
7
Faz
8
:: flow_to_ses [0]? sync_ack, digite ->
9
E se
10
:: (tipo! = alternar)
11
:: (tipo == alternar) -> pausa
12
fi
13
:: tempo limite ->
14
ses_to_flow [0]! sincronizar, alternar
15
od;
16
toggle = 1 - alternar;
17
Faz
18
/* remetente */
19
:: ses_to_flow [0]! branco, 0
20
:: atomic {
21
(s_state == 0 && len (ses_to_flow [0]) <QSZ) ->
22
ses_to_flow [0]! vermelho, 0 ->
23
s_state = 1
24
}

```

Página 339

328
USANDO O VALIDADOR
CAPÍTULO 14
25
26
:: atomic {
27
(s_state == 1 && len (ses_to_flow [0]) <QSZ) ->
28
ses_to_flow [0]! azul, 0 ->
29
s_state = 2
30
}
31
/* receptor */
32
:: flow_to_ses [1]? branco, 0
33
:: atomic {
34
(r_state == 0 && flow_to_ses [1]? [red]) ->
flow_to_ses [1]? vermelho, 0 ->

```

35
r_state = 1
36
}
37
:: atomic {
38
(r_state == 0 && flow_to_ses [1]? [blue]) ->
39
afirmar (0)
40
}
41
:: atomic {
42
(r_state == 1 && flow_to_ses [1]? [blue]) ->
43
flow_to_ses [1]? azul, 0;
44
quebrar
45
}
46
:: atomic {
47
(r_state == 1 && flow_to_ses [1]? [red]) ->
48
afirmar (0)
49
}
50
od;
Final 51:
52
Faz
53
:: flow_to_ses [1]? branco, 0
54
:: flow_to_ses [1]? vermelho, 0 -> assert (0)
55
:: flow_to_ses [1]? blue, 0 -> assert (0)
56
od
57}

```

A estrutura do sistema de teste que descrevemos é mostrada na Figura 14.1.

Superior
Testador
Controle de fluxo
Controle de fluxo

Figura 14.1 - Validação da camada de controle de fluxo

O círculo representa o modelo de nível superior que foi adicionado especificamente para esta validação. As duas caixas são os processos da camada de controle de fluxo sendo validados. Pelo construções do testador superior, sabemos que se houver algum erro no controle de fluxo camada, o módulo de testador superior irá disparar em uma falsa afirmação.

Página 340

SEÇÃO 14.5
VALIDAÇÃO DE CONTROLE DE FLUXO
329

CANAIS IDEAIS

Em uma primeira execução de validação, verificamos se, na ausência de erros, os dados são transferidos corretamente e a reivindicação temporal não pode ser violada. O script de inicialização se parece com o seguinte

baixos:

```

1 /*
2 * Modelo de Validação PROMELA
3 * VALIDAÇÃO DA CAMADA DE CONTROLE DE FLUXO
4 */
5
6 # definir PERDA
0
/ * perda de mensagem * /
7 # define DUPS
0
/ * msgs duplicadas * /
8 # define QSZ

```

```

2
/* tamanho da fila
 */
9
10 mtype = {
11
vermelho branco azul,
12
abortar, aceitar, ack, sync_ack, fechar, conectar,
13
criar, dados, eof, abrir, rejeitar, sincronizar, transferir,
14
FATAL, NON_FATAL, COMPLETE
15
}
16
17 canais_para_fluxo [2] = [QSZ] de {byte, byte};
18 canais_flow_to_ses [2] = [QSZ] de {byte, byte};
19 canais_dll_to_flow [2] = [QSZ] de {byte, byte};
20 canais_flow_to_dll [2];
21
22 #include "flow_cl"
23 #include "upper_tester"
24
25 init
26 {
27
atômica {
28
flow_to_dll [0] = dll_to_flow [1];
29
flow_to_dll [1] = dll_to_flow [0];
30
execute fc (0); execute fc (1);
31
executar superior ()
32
}
33}

```

Os arquivos de inclusão contêm as definições de modelo que acabamos de discutir. O fluxo com os processos da camada de trol estão diretamente ligados às duas primeiras atribuições no programa inicial
cess e são iniciados nas duas instruções de execução subsequentes . O seguinte num-
A listagem da camada de controle de fluxo, conforme testado, é útil para referência cruzada
código inacessível.

```

330
USANDO O VALIDADOR
CAPÍTULO 14
1 /*
2 * Modelo de validação de camada de controle de fluxo
3 */
4
5 #define true
1
6 #define false
0
7
8 # define M 4
/* números de sequência de intervalo */
9 # define W 2
/* tamanho da janela: M / 2
*/
10
11 proctype fc (bit n)
12 {bool
ocupado [M];
/* mensagens pendentes
*/
13
byte
q;
/* seq # msg mais antiga não confirmada */
14
byte
m;
/* seq # última mensagem recebida */

```

```

15
byte
s;
/* seq # próxima mensagem a enviar */
16
byte
janela;
/* nº de mensagens pendentes */
17
byte
tipo;
/* tipo de mensagem
*/
18
mordeu
recebeu [M];
/* manutenção do receptor */
19
mordeu
x;
/* variável de rascunho
*/
20
byte
p;
/* seq # da última mensagem confirmada */
21
byte
I_buf [M], O_buf [M];
/* buffers de mensagem */
22
23
/* parte do remetente */
24 final:
Faz
25
:: atomic {
26
(janela <W && len (ses_to_flow [n])> 0
27
&& len (flow_to_dll [n]) <QSZ) ->
28
ses_to_flow [n]? tipo, x;
29
janela = janela + 1;
30
ocupado [s] = verdadeiro;
31
O_buf [s] = tipo;
32
flow_to_dll [n]! tipo, s;
33
E se
34
:: (digite! = sincronizar) ->
35
s = (s + 1)% M
36
:: (tipo == sync) ->
37
janela = 0;
38
s = M;
39
Faz
40
:: (s> 0) ->
41
s = s-1;
42
ocupado [s] = falso
43
:: (s == 0) ->
44
quebrar
45
od
46
fi
47
}
48

```

```

:: atomic {
49
(janela > 0 && ocupado [q] == false) ->
50
janela = janela - 1;
51
q = (q + 1)% M
52
}
53 #if DUPS
54
:: atomic {

```

```

SEÇÃO 14.5
VALIDAÇÃO DE CONTROLE DE FLUXO
331
55
(len (flow_to_dll [n]) <QSZ
56
&& window > 0 && busy [q] == true) ->
57
flow_to_dll [n]! O_buf [q], q
58
}
59 #endif
60
:: atomic {
61
(tempo limite && len (flow_to_dll [n]) <QSZ
62
&& window > 0 && busy [q] == true) ->
63
flow_to_dll [n]! O_buf [q], q
64
}
65
66
/* parte do receptor */
67 #if PERDA
68
:: dll_to_flow [n]? type, m /* perde qualquer mensagem */
69 #endif
70
:: dll_to_flow [n]? digite, m ->
71
E se
72
:: atomic {
73
(tipo == ack) ->
74
ocupado [m] = falso
75
}
76
:: atomic {
77
(tipo == sincronização) ->
78
flow_to_dll [n]! sync_ack, m;
79
m = 0;
80
Faz
81
:: (m <M) ->
82
recebido [m] = 0;
83
m = m + 1
84
:: (m == M) ->
85
quebrar
86
od
87
}
88

```

```

:: (tipo == sync_ack) ->
89
flow_to_ses [n]! sync_ack, m
90
:: (digite! = ack && type! = sync && type! = sync_ack) ->
91
E se
92
:: atomic {
93
(recebido [m] == verdadeiro) ->
94
x = ((0 <pm && pm <= W)
95
|| (0 <p-m + M && p-m + M <= W));
96
E se
97
:: (x) -> flow_to_dll [n]! ack, m
98
:: (! x) /* else ignorar */
99
fi
100
fi
101
:: atomic {
102
(recebido [m] == falso) ->
103
I_buf [m] = tipo;
104
recebido [m] = verdadeiro;
105
recebido [(m-W + M)% M] = falso
106
}
107
fi
108
:: (recebido [p] == verdadeiro && len (flow_to_ses [n]) <QSZ
109
&& len (flow_to_dll [n]) <QSZ) ->

```

Página 343

```

332
USANDO O VALIDADOR
CAPITULO 14
110
flow_to_ses [n]! I_buf [p], 0;
111
flow_to_dll [n]! ack, p;
112
p = (p + 1)% M
113
od
114}

```

Não saber nada sobre a complexidade do modelo que construímos para a validação, a melhor abordagem é executar uma análise rápida de supertrace (espaço de estado de bits) e verifique o fator de hash e o número de estados alcançáveis. Multiplicando o número de estados armazenados com o número de bytes necessários por estado, podemos então obter um estimativa da quantidade de memória que seria necessária para uma pesquisa exaustiva.

Por exemplo, uma análise de supertrace do modelo de validação da camada de controle de fluxo de A Figura 14.1 é realizada da seguinte forma, usando uma arena de memória de aproximadamente 4,5 Mbytes:

```

$ spin -a pftp.flow
$ cc -DMEMCNT = 23 -DBITSTATE -o pan pan.c
$ pan -w25
bit statespace pesquisa por:
violações de asserção e estados finais inválidos
vetor 128 byte, profundidade alcançou 3781, erros: 0
90843 estados, armazenados
317124 estados, vinculados
182422 estados, combinados
total: 590389
fator de hash: 369,363216 (melhor cobertura se > 100)
(tamanho máximo de 2^25 estados, stackframes: 0/418)

```

```

memória usada: 4463832
...
A pesquisa foi de boa qualidade (o fator de hash é alto), então o número de estados
alcançado deve ser uma boa aproximação do verdadeiro número de estados alcançáveis no
espaço de estado completo. Um cálculo rápido mostra que precisaríamos de  $90843 \cdot 128$ , ou
cerca de 12 Mbytes para armazenar o espaço de estado completo. Ter uma máquina com 64
Mbytes disponíveis, podemos decidir repetir a análise com uma verificação exaustiva.
$ cc -DMEMCNT = 24 -o pan pan.c # limite de memória  $2^{24}$ 
$ pan -w16
# tabela hash de  $2^{16}$  slots
pesquisa de espaço de estado completo para:
violações de asserção e estados finais inválidos
vetor 128 byte, profundidade atingiu 5580, erros: 0
90845 estados, armazenados
317134 estados, vinculados
182425 estados, combinados
total: 590404
conflitos de hash: 154271 (resolvido)
(tamanho máximo  $2^{16}$  estados, stackframes: 0/418)
memória usada: 12886356
não alcançado em proctype _init:
atingiu todos os 7 estados

```

Página 344

SEÇÃO 14.5 VALIDAÇÃO DE CONTROLE DE FLUXO

333
não alcançado no proctipo superior:
linha 13 (estado 9)
linha 39 (estado 29)
linha 48 (estado 36)
linha 54 (estado 43)
linha 55 (estado 45)
linha 57 (estado 49)
alcançado: 43 de 49 estados
não alcançado no proctype fc:
linha 63 (estado 28)
linha 93 (estado 50)
linha 96 (estado 53)
linha 95 (estado 55)
linha 113 (estado 73)
alcançado: 68 de 73 estados

O espaço estadual construído continha 90.845 estados do sistema alcançáveis, com 317.134 estados vinculados

(estados intermediários em sequências atômicas), e uma sequência de execução única mais longa de 3781 etapas. Um total de 182.425 vezes foi alcançado um estado que foi analisado anteriormente na profundidade primeira pesquisa. A análise de espaço de estado de bits anterior tinha cobertura de 99,997%.

A seguir, consideremos os estados relatados como inalcançáveis. Quatro dos seis estados inacessíveis no testador superior correspondem às violações de asserção que nós deseja estar inacessível: linhas 39, 48, 54 e 55. A linha 13 especifica a ação a ser tomada se ocorrer um tempo limite enquanto o testador superior está esperando por uma resposta ao seu inicial

mensagem de sincronização. É prontamente verificado que, de fato, este código também deve estar inacessível:

se não houver perda de mensagem, o tempo limite nunca deve ocorrer. A linha 57, finalmente, é o estado de parada normal do testador superior, no final de seu código. Já que o código para o testador superior é escrito como um loop infinito, também não esperaríamos que esse estado fosse alcançável.

Cinco estados são relatados como inalcançáveis no protocolo da camada de controle de fluxo. o código não alcançado nos diz que nenhum timeout pode ocorrer (linha 63). Isso está correto, no a ausência de tempo limite de perda de mensagem é redundante. Também confirma que, na ausência de todos os erros, as confirmações sempre chegam na ordem exata em que os dados sábios são enviados (linhas 93-97). A linha 113, finalmente, é o estado final normal do fluxo de con processo de camada trol. Uma vez que o processo nunca termina, ele também é rotulado corretamente como inacessível.

Ao examinar as listagens, lembre-se de que os números das linhas são aproximados, off-by-alguns erros às vezes são difíceis de evitar. Em caso de dúvida, os números dos estados indicados em

parênteses podem ser usados para procurar a declaração precisa do processo no arquivo pan.m .

Na ausência de perda de mensagem no link de dados subjacente, então, a camada de controle de fluxo atende aos seus requisitos de correção. Uma vez que as afirmações na reivindicação temporal não podem ser violada, nenhuma mensagem pode ser perdida ou reordenada.

Página 345

334

USANDO O VALIDADOR

CAPÍTULO 14

PERDA DE MENSAGEM E ERROS DE DUPLICAÇÃO

Nas próximas rodadas de validação, verificamos o funcionamento da camada de controle de fluxo no presente

presence de dois tipos diferentes de erros: perda de mensagens e mensagens duplicadas. Nós primeiro verifique a perda de mensagens atribuindo à diretiva do pré-processador LOSS um valor diferente de zero. isto

está apenas ao alcance de uma análise completa do espaço de estado.

```
$ spin -a pftp.flow1
$ cc -o pan pan.c
$ pan -w20
pesquisa de espaço de estado completo para:
violações de asserção e estados finais inválidos
vetor 128 byte, profundidade atingiu 4421, erros: 0
396123 estados, armazenados
1046768 estados, vinculados
748273 estados, correspondentes
total: 2191164
conflitos de hash: 186761 (resolvido)
(tamanho máximo 2^20 estados, stackframes: 0/543)
não alcançado em proctype _init:
atingiu todos os 7 estados
não alcançado no proctipo superior:
linha 39 (estado 29)
linha 48 (estado 36)
linha 54 (estado 43)
linha 55 (estado 45)
linha 57 (estado 49)
alcançado: 44 de 49 estados
não alcançado no proctype fc:
linha 113 (estado 74)
alcançado: 73 de 74 estados
```

A opção de tempo limite no testador superior agora foi exercida, e todos os estados do processo de camada de controle de fluxo foram alcançados. Todos os estados inacessíveis restantes no testador superior corresponde aos estados de erro que deveriam ser inacessíveis.

Um próximo teste é para mensagens duplicadas. Habilitemos este teste com a direção do pré-processador tive DUPS . Esse tipo de erro aumenta drasticamente a complexidade do modelo. UMA a validação agora está solidamente fora da gama de pesquisas exaustivas. Apenas um pouco de esforço é necessário para alcançar resultados razoáveis.

```
$ spin -a pftp.flow2
$ cc -DMEMCNT = 27 -DBITSTATE -o pan pan.c
$ pan -w29 -m100000
vetor 128 byte, profundidade alcançada 56089, erros: 0
8241456 estados, armazenados
22946550 estados, vinculados
21143649 estados, correspondentes
total: 52331655
fator de hash: 65,142718 (melhor cobertura se> 100)
(tamanho máximo 2^29 estados, stackframes: 0/7621)
memória usada: 70073429
não alcançado em proctype _init:
atingiu todos os 7 estados
```

Página 346

SEÇÃO 14.6

VALIDAÇÃO DA CAMADA DE SESSÃO

335

não alcançado no proctipo superior:

```
linha 13 (estado 9)
linha 39 (estado 29)
linha 48 (estado 36)
linha 54 (estado 43)
linha 55 (estado 45)
```

```

linha 57 (estado 49)
alcançado: 43 de 49 estados
não alcançado no proctype fc:
linha 63 (estado 31)
linha 113 (estado 76)
alcançado: 74 de 76 estados

```

Armazenar um espaço de estado completo de 8.241.456 estados de 128 bytes cada levaria um Gigabyte de memória. A pesquisa de espaço de estado de bits acima usou 70 Mbytes e completou com um fator de hash de 65, portanto, com uma garantia razoável de cobertura completa (ver Capítulo 13). A sequência de execução única mais longa agora cresceu para 56.089 etapas. Tudo profissionalmente estudos do protocolo, exceto aqueles correspondentes a erros e tempos limite de retransmissão têm exercido. A camada de controle de fluxo também passa neste teste, ou seja, na ausência de os outros tipos de erros, a camada de controle de fluxo parece capaz de lidar com sucesso com quantidades arbitrárias de erros de duplicação.

Este teste de validação é, obviamente, bastante drástico. Retransmissão prematura os tempos limite podem ocorrer várias vezes durante uma sessão de transferência de arquivos, mas muito improvável centenas de vezes ou mais. Muitas outras variações de execuções de validação são possível. Poderíamos, por exemplo, reduzir a complexidade da pesquisa contando e restringindo o número de erros de duplicação por sessão. Também podemos testar para combinações de erros de perda e duplicação, e poderíamos intercalar o envio de branco , vermelho , e azul mensagens com resynchronizations controle de fluxo. Nós consideramos apenas um variante de uma validação executada abaixo.

VIOLAÇÕES DO INVARIANTE DE JANELA

Para garantir que os erros sejam detectados de forma adequada nas execuções de validação, podemos tentar

alterar o tamanho da janela e substituir os parâmetros corretos:

```

#define M
4
/* números de sequência de intervalo */
#define W
2
/* tamanho da janela: M / 2
*/

```

no protocolo da camada de controle de fluxo, com, por exemplo

```

#define M
4
/* números de sequência de intervalo */
#define W
3
/* tamanho da janela:> M / 2
*/

```

Na presença de perda de mensagem, isso deve revelar erros, porque viola a Dow protocolo invariante que provamos anteriormente. Primeiro tentamos uma pesquisa sem a possibilidade de perda de mensagem:

```

336
USANDO O VALIDADOR
CAPÍTULO 14
$ spin -a pftp.flow3
$ cc -o pan pan.c
$ pan -m20000
pesquisa completa de espaço de estado para:
violações de asserção e estados finais inválidos
vetor 128 byte, profundidade alcançou 10194, erros: 0
287445 estados, armazenados
1181892 estados, vinculados
664505 estados, correspondentes
total: 2133842
conflictos de hash: 487165 (resolvido)
(tamanho máximo 2^18 estados, stackframes: 0/1130)

```

Existem mais estados do que antes, porque pode haver mais mensagens pendentes em ao mesmo tempo, mas, como esperado, nenhum erro ainda. Em seguida, ativamos a perda de mensagens por

definir a diretiva do compilador LOSS para 1.

```

$ spin -a pftp.flow4
$ cc -o pan pan.c
$ pan
afirmação violada 0

```

```

pan: abortado (na profundidade 656)
pan: escreveu pan.trail
pesquisa completa de espaço de estado para:
violações de asserção e estados finais inválidos
pesquisa não foi completada
vetor 128 byte, profundidade atingiu 1290, erros: 1
22469 estados, armazenados
45816 estados, vinculados
28041 estados, combinados
total:
96326
conflitos de hash: 3267 (resolvido)
(tamanho máximo 2^18 estados, stackframes: 0/199)
...

```

Como esperado, a adulteração da invariante do protocolo da janela introduz um erro que é descoberto na análise de alcançabilidade depois que apenas alguns milhares de estados são verificado. Ele pode ser rastreado com uma simulação guiada, usando a trilha de erro produzidas pelo analisador.

14.6 VALIDAÇÃO DA CAMADA DE SESSÃO

Tendo nos convencido de que, com os parâmetros de tamanho de janela certos, o fluxo camada de controle imita corretamente o comportamento de um canal de transmissão ideal para o camadas de protocolo superiores, podemos agora usar esse resultado para simplificar a validação do sessão-camada de ação. Podemos construir um modelo de validação para este teste da seguinte forma, omitindo todos os

coisa que foi testada antes:

```

/*
 * Modelo de Validação PROMELA
 * Camada de Sessão
 */

```

SEÇÃO 14.6
VALIDAÇÃO DA CAMADA DE SESSÃO

```

337
#include "define2"
#include "usuário"
#include "presente"
#include "sessão"
#include "fserver"
iniciar
{
atômica {
execute userprc (0); execute userprc (1);
executar presente (0); executar presente (1);
sessão de execução (0); sessão de execução (1);
execute fserver (0); execute fserver (1);
flow_to_ses [0] = ses_to_flow [1];
flow_to_ses [1] = ses_to_flow [0]
}
}
```

As camadas de sessão são conectadas diretamente, como se conectadas por um canal ideal que nunca perde, distorce ou reordena mensagens. Uma vez que nenhuma camada de controle de fluxo está presente, nós

pode comentar o código na camada de sessão que se destina especificamente para o gerenciamento dos números de sequência da camada de controle de fluxo. O código resultante se parece com o seguinte

baixos:

```

1 /*
2 * Modelo de validação de camada de sessão
3 */
4
5 sessão proctype (bit n)
6 {bit alternar;
7 tipo de byte, status;
8
9 endIDLE:
10
Faz
11
:: pres_to_ses [n]? digite ->
12
E se
13
```

```

:: (tipo == transferência) ->
14
ir para DATA_OUT
15
:: (digite! = transferir) /* ignore */
16
fi
17
:: flow_to_ses [n]? tipo, 0 ->
18
E se
19
:: (digite == conectar) ->
20
ir para DATA_IN
21
:: (digite! = conectar)
/* ignore */
22
fi
23
od;
24
25 DATA_IN:
/* 1. prepare o arquivo local fsrver */
26
ses_to_fsrv [n]! criar;
27
Faz
28
:: fsrv_to_ses [n]? rejeitar ->
29
ses_to_flow [n]! rejeitar, 0;

```

Página 349

```

338
USANDO O VALIDADOR
CAPÍTULO 14
30
ir para endIDLE
31
:: fsrv_to_ses [n]? aceitar ->
32
ses_to_flow [n]! aceitar, 0;
33
quebrar
34
od;
35
/* 2. Receba os dados, até o final de */
36
Faz
37
:: flow_to_ses [n]? dados, 0 ->
38
ses_to_fsrv [n]! dados
39
:: flow_to_ses [n]? eof, 0 ->
40
ses_to_fsrv [n]! eof;
41
quebrar
42
:: pres_to_ses [n]? transferir ->
43
ses_to_pres [n]! rejeitar (NON_FATAL)
44
:: flow_to_ses [n]? fechar, 0 ->
/* usuário remoto abortado */
45
ses_to_fsrv [n]! close;
46
quebrar
47
:: tempo limite ->
/* foi desconectado */
48
ses_to_fsrv [n]! close;
49
ir para endIDLE

```

```

50
od;
51
/* 3. Feche a conexão */
52
ses_to_flow [n]! fechar, 0;
53
goto endIDLE;
54
55 DATA_OUT:
/* 1. prepare o arquivo local fsrver */
56
ses_to_fsrv [n]! open;
57
E se
58
:: fsrv_to_ses [n]? rejeitar ->
59
ses_to_pres [n]! rejeitar (FATAL);
60
ir para endIDLE
61
:: fsrv_to_ses [n]? aceitar ->
62
pular
63
fi;
64
/* 2. inicializar o controle de fluxo *** desativado
65
ses_to_flow [n]! sincronizar, alternar;
66
Faz
67
:: atomic {
68
flow_to_ses [n]? sync_ack, digite ->
69
E se
70
:: (tipo! = alternar)
71
:: (tipo == alternar) -> pausa
72
fi
73
}
74
:: tempo limite ->
75
ses_to_fsrv [n]! close;
76
ses_to_pres [n]! rejeitar (FATAL);
77
ir para endIDLE
78
od;
79
toggle = 1 - alternar;
80
/* 3. preparar arquivo remoto fsrver */
81
ses_to_flow [n]! conectar, 0;
82
E se
83
:: flow_to_ses [n]? rejeitar, 0 ->

```

SEÇÃO 14.6
VALIDAÇÃO DA CAMADA DE SESSÃO
339
84
ses_to_fsrv [n]! close;
85
ses_to_pres [n]! rejeitar (FATAL);
86
ir para endIDLE
87
:: flow_to_ses [n]? conectar, 0 ->

```

88
ses_to_fsrv [n]! close;
89
ses_to_pres [n]! rejeitar (NON_FATAL);
90
ir para endIDLE
91
:: flow_to_ses [n]? aceitar, 0 ->
92
pular
93
:: tempo limite ->
94
ses_to_fsrv [n]! close;
95
ses_to_pres [n]! rejeitar (FATAL);
96
ir para endIDLE
97
fi;
98
/* 4. Transmita os dados, até o final de */
99
Faz
100
:: fsrv_to_ses [n]? dados ->
101
ses_to_flow [n]! dados, 0
102
:: fsrv_to_ses [n]? eof ->
103
ses_to_flow [n]! eof, 0;
104
status = COMPLETO;
105
quebrar
106
:: pres_to_ses [n]? abortar ->
/* usuário local abortado */
107
ses_to_fsrv [n]! close;
108
ses_to_flow [n]! fechar, 0;
109
status = FATAL;
110
quebrar
111
od;
112
/* 5. Feche a conexão */
113
Faz
114
:: pres_to_ses [n]? abortar
/* ignore */
115
:: flow_to_ses [n]? fechar, 0 ->
116
E se
117
:: (status == COMPLETO) ->
118
ses_to_pres [n]! aceitar, 0
119
:: (status != COMPLETO) ->
120
ses_to_pres [n]! rejeitar (status)
121
fi;
122
quebrar
123
:: tempo limite ->
124
ses_to_pres [n]! rejeitar (FATAL);
125
quebrar
126
od;
127
ir para endIDLE

```

O código do usuário é:

Página 351

```

340
USANDO O VALIDADOR
CAPÍTULO 14
1 /*
2 * Modelo de validação de camada de usuário
3 */
4
5 proctype userprc (bit n)
6 {
7
use_to_pres [n]! transferência;
8
E se
9
:: pres_to_use [n]? aceitar -> ir para Concluído
10
:: pres_to_use [n]? rejeitar -> ir para Concluído
11
:: use_to_pres [n]! abortar -> goto abortado
12
fi;
13 Abortado:
14
E se
15
:: pres_to_use [n]? aceitar -> ir para Concluído
16
:: pres_to_use [n]? rejeitar -> ir para Concluído
17
fi;
18 Feito:
19
pular
20}

```

E, finalmente, o código da camada de apresentação é:

Página 352

```

SECÃO 14.6
VALIDAÇÃO DA CAMADA DE SESSÃO
341
1 /*
2 * Modelo de validação da camada de apresentação
3 */
4
5 proctipo presente (bit n)
6 {status de byte, uabort;
7
8 endIDLE:
9
Faz
10
:: use_to_pres [n]? transferir ->
11
uabort = 0;
12
quebrar
13
:: use_to_pres [n]? abortar ->
14
pular
15
od;
16
17 TRANSFERÊNCIA:
18
pres_to_ses [n]! transferência;
19
Faz
20
:: use_to_pres [n]? abortar ->
21
E se
22
:: (! uabort) ->

```

```

23
uabort = 1;
24
pres_to_ses [n]! abortar
25
:: (uabort) ->
26
afirmar (1 + 1! = 2)
27
fi
28
:: ses_to_pres [n]? aceitar, 0 ->
29
vá para FEITO
30
:: ses_to_pres [n]? rejeitar (status) ->
31
E se
32
:: (status == FATAL || uabort) ->
33
ir para FALHA
34
:: (status == NON_FATAL && ! uabort) ->
35 progresso:
ir para TRANSFERÊNCIA
36
fi
37
od;
38 FEITO:
39
pres_to_use [n]! aceitar;
40
goto endIDLE;
41 FALHA:
42
pres_to_use [n]! rejeitar;
43
ir para endIDLE
44}

```

Faremos uma validação em duas etapas separadas. O servidor de arquivos, sessão e apresentação os processos da camada de cão são todos cílicos: eles nunca devem terminar. O processo inicial e os processos do usuário, no entanto, estão encerrando e, uma vez que concluíram seu execução, os outros processos devem ter atingido um estado final bem definido. Em primeiro validação, portanto, podemos tentar ter certeza de que o sistema não tem acesso inválido estados finais. Podemos fazer isso com uma validação exaustiva, da seguinte maneira:

Página 353

```

342
USANDO O VALIDADOR
CAPÍTULO 14
$ spin -a pftp.ses
$ cc -o pan pan.c
$ pan -w19
pesquisa de espaço de estado completo para:
violações de asserção e estados finais inválidos
vetor 144 byte, profundidade alcançada 451, erros: 0
509179 estados, armazenados
9 estados, vinculados
576192 estados, combinados
total: 1085380
conflitos de hash: 369417 (resolvido)
(tamanho máximo 2^19 estados, stackframes: 0/23)
não alcançado em proctype _init:
atingiu todos os 12 estados
não alcançado no proctype fserver:
linha 29 (estado 30)
alcançado: 29 de 30 estados
não alcançado na sessão proctype:
linha 48 (estado 37)
linha 94 (estado 64)
linha 95 (estado 65)
linha 124 (estado 93)
linha 128 (estado 99)
alcançado: 94 de 99 estados
não alcançado no proctipo presente:
linha 26 (estado 15)

```

```
linha 44 (estado 34)
alcançado: 32 de 34 estados
não alcançado em proctype userprc:
atingiu todos os 17 estados
```

O código não alcançado na camada de apresentação (linha 26) indica que nenhum caso foi encontrado no qual duas mensagens de aborto subsequentes são recebidas do processo do usuário. Verificando o processo do usuário, podemos ver rapidamente o porquê disso: o processo do usuário não permitir. O código não alcançado no protocolo da camada de sessão, no entanto, sinaliza um erro plenitude neste primeiro teste de validação. As linhas não alcançadas 48, 94, 95, 124 e linha 128 são respostas a condições de tempo limite que foram incluídas para permitir que a camada de sessão recuperar de uma perda repentina de comunicação com seu processo de par. Essa possibilidade, entretanto, não é modelado como parte do comportamento do canal e não pode ser exercido.

Para verificar também se essas condições de tempo limite não podem causar estragos, devemos revisar o

modelo de validação. Podemos fazer isso adicionando algumas linhas ao código de inicialização no processo de inicialização fornecido acima:

```
atômico
{
byte any;
chan foo = [1] de {byte, byte};
ses_to_flow [0] = foo;
ses_to_flow [1] = foo
};
```

Página 354

SEÇÃO 14.6
VALIDAÇÃO DA CAMADA DE SESSÃO

```
343
fim:
Faz
:: foo? any, any
od
}
```

A qualquer momento após a inicialização do protocolo, essas linhas extras agora podem ser executadas cortado. O efeito é que os dois processos da camada de sessão peer são desconectados. o loop no final remove todas as mensagens que as duas camadas de sessão produzem. o extensão aumenta a complexidade do teste um pouco mais, mas um pouco de espaço de estado a análise ainda é viável. O resultado é agora

```
$ spin -a pftp.sesi
$ cc -DBITSTATE -o pan pan.c
$ pan -w29
bit state space search por:
violações de asserção e estados finais inválidos
vetor 148 byte, profundidade alcançada 456, erros: 0
1686543 estados, armazenados
246135 estados, vinculados
1960294 estados, combinados
total: 3892972
fator de hash: 318,326063 (melhor cobertura se> 100)
(tamanho máximo de 2^29 estados, stackframes: 0/25)
não alcançado em proctype _init:
linha 31 (estado 19)
alcançado: 18 de 19 estados
não alcançado no proctype fserver:
linha 29 (estado 30)
alcançado: 29 de 30 estados
não alcançado na sessão proctype:
linha 128 (estado 99)
alcançado: 98 de 99 estados
não alcançado no proctipo presente:
linha 26 (estado 15)
linha 44 (estado 34)
alcançado: 32 de 34 estados
não alcançado em proctype userprc:
atingiu todos os 17 estados
```

Em comparação com o primeiro teste, agora exploramos mais de três vezes mais estados e efetivamente atingiu todos os estados de protocolo relevantes. O fator de hash é grande o suficiente para ser

confiante de que perto de 100% dos estados do sistema alcançáveis foram testados dentro a arena da memória que está disponível. Uma pesquisa exaustiva teria exigido pelo menos $1.686.543 \cdot 148$ ou 249 Mbytes de memória, quatro vezes mais do que usamos.

A REIVINDICAÇÃO TEMPORAL

Na segunda validação do protocolo da camada de sessão que empreendemos aqui, nós consideraremos a reivindicação temporal que foi formulada no Capítulo 7.

Página 355

344
USANDO O VALIDADOR
CAPÍTULO 14
Nunca {
Faz
::! pres_to_ses [n]? [transferir]
&&! flow_to_ses [n]? [conectar]
:: pres_to_ses [n]? [transferir] ->
ir para aceitar 0
:: flow_to_ses [n]? [conectar] ->
ir para aceitar 1
od;
aceitar 0:
Faz
::! ses_to_pres [n]? [aceitar]
&&! ses_to_pres [n]? [rejeitar]
od;
aceitar1:
Faz
::! ses_to_pres [1-n]? [aceitar]
&&! ses_to_pres [1-n]? [rejeitar]
od
}

Uma vez que o protocolo é simétrico, é suficiente validar esta afirmação para apenas um valor de n , por exemplo, zero. O resultado é o seguinte:

```
$ spin -a pftp.ses2
$ cc -o pan pan.c
$ pan
ciclo de comprimento 6 (99) 104
pan: aceitar o estado no ciclo (na profundidade 99)
pan: escreveu pan.trail
pesquisa total do statespace sobre o comportamento restrito à reivindicação de:
violações de asserção
e ausência de rótulos de aceitação em todos os ciclos
pesquisa não foi completada
vetor 148 byte, profundidade alcançada 100, erros: 1
151 estados, armazenados
9 estados, vinculados
16 estados, combinados
total:
176
conflitos de hash: 0 (resolvido)
(tamanho máximo 2^18 estados, stackframes: 0/4)
```

Foi detectado um ciclo de aceitação, o que significa que a reclamação pode ser violada. UMA um olhar mais atento com o simulador pode revelar a causa.

```
$ spin -t -r -s pftp.ses2
# -t: segue a trilha produzida pela panela
proc 3 (userprc) linha 8, Enviar transferência -> fila 6 (use_to_pres [1])
proc 5 (presente) linha 11, transferência de recebimento <- fila 6 (use_to_pres [1])
proc 5 (presente) linha 19, Enviar transferência -> fila 4 (pres_to_ses [1])
proc 7 (sessão) linha 11, Recv 13
<- fila 4 (pres_to_ses [1])
proc 7 (sessão) linha 56, Enviado aberto -> fila 8 (ses_to_fsrv [1])
...
<<<< INÍCIO DO CICLO >>>>
proc 9 (fserver) linha 13, dados de recebimento <- fila 8 (ses_to_fsrv [1])
proc 6 (sessão) linha 101, Enviar dados, 0 -> fila 1 (ses_to_flow [0])
```

Página 356

SEÇÃO 14.6
VALIDAÇÃO DA CAMADA DE SESSÃO

345
proc 8 (fserver) linha 23, Dados enviados -> fila 9 (fsrv_to_ses [0])
proc 6 (sessão) linha 100, dados de recebimento <- fila 9 (fsrv_to_ses [0])
proc 7 (sessão) linha 37, dados Recv, 0 <- fila 1 (flow_to_ses [1])
rotação: a trilha termina após 179 passos
etapa 179, #processos: 10
...

O validador descobriu aqui que o número de mensagens de dados que são trocadas durante uma sessão de transferência de arquivos não é limitado. Isso significa que o envio de uma final aceitar ou rejeitar mensagem para a camada de apresentação pode ser adiado indefinidamente,

o que é uma violação direta de nosso requisito de correção.
 Para corrigir esse problema, podemos tentar dizer à reivindicação temporal para ignorar as mensagens de dados,
 isto é, considerar apenas as transferências de arquivos de comprimento zero.

```
Nunca {
Faz
::! pres_to_ses [0]? [transferir]
&&! flow_to_ses [0]? [conectar]
:: pres_to_ses [0]? [transferir] ->
ir para aceitar 0
:: flow_to_ses [0]? [conectar] ->
ir para aceitar 1
od;
aceitar 0:
Faz
::! ses_to_pres [0]? [aceitar]
&&! ses_to_pres [0]? [rejeitar]
&&! ses_to_flow [0]? [dados]
od;
aceitar1:
Faz
::! ses_to_pres [1]? [aceitar]
&&! ses_to_pres [1]? [rejeitar]
&&! ses_to_flow [1]? [dados]
od
}
```

A validação com esta nova reivindicação prossegue da seguinte forma:

```
$ spin -a pftp.sess3
$ cc -o pan pan.c
$ pan
ciclo de comprimento 5 (99) 103
pan: aceitar o estado no ciclo (na profundidade 99)
pan: escreveu pan.trail
pesquisa total do espaço de estado em comportamento restrito à reivindicação de:
violações de asserção
e ausência de estados de aceitação em todos os ciclos
pesquisa não foi completada
vetor 148 byte, profundidade alcançada 132, erros: 1
21645 estados, armazenados
9 estados, vinculados
20316 estados, combinados
total:
41970
```

Página 357

346

USANDO O VALIDADOR

CAPÍTULO 14

conflictos de hash: 2293 (resolvido)
 (tamanho 2^{18} estados, stackframes: 0/5)

Novamente, o validador descobriu que o requisito de correção pode ser violado. A parte relevante da trilha é a seguinte:

```
$ spin -t -r -s pftp.sess3
...
proc 4 (presente) linha 19, Enviar transferência -> fila 3 (pres_to_ses [0])
proc 6 (sessão) linha 42, transferência de recebimento <- fila 3 (pres_to_ses [0])
<<<< INÍCIO DO CICLO >>>>
proc 6 (sessão) linha 43, Enviar rejeição, NON_FATAL -> \
fila 11 (ses_to_pres [0])
proc 4 (presente) linha 31, Recv rejeição, 15 <- fila 11 (ses_to_pres [0])
proc 4 (presente) linha 19, Enviar transferência -> fila 3 (pres_to_ses [0])
rotação: a trilha termina após 176 passos
...
```

Após o início de uma transferência de arquivo, pode haver um número ilimitado de conflitos solicitações de transferência do processo peer remoto. Novamente, processando essas solicitações como rejeições não fatais podem adiar por muito tempo o envio da aceitação final ou rejeitar mensagem para a transferência de arquivo ativa.

Desta vez, é muito mais difícil modificar a reivindicação temporal para remover este padrão de consideração. Um rótulo de estado de aceitação identifica eventos como potencialmente ruins. Nesse caso, no entanto, podemos trabalhar de forma mais eficaz com um método para rotular um pequeno conjunto de

eventos tão bons e foco nos outros. A ferramenta certa para isso é o rótulo de estado de progresso. Se pudermos reformular a reivindicação temporal como um requisito de correção na ausência de ciclos sem progresso, torna-se mais fácil excluir certos padrões da consideração.

Observe que, se desconsiderarmos os dois padrões descobertos anteriormente, todas as execuções do seu protocolo da camada de sessão deve terminar. Qualquer ciclo que possa ser identificado, portanto, será tornar-se um ciclo de não progresso e, portanto, uma violação detectável da correção requisitos. Rotulamos os estados DATA_IN e DATA_OUT no protocolo de camada de sessão col como estados de progresso. Para excluir os dois padrões descobertos acima, também rotulamos a troca de dados com o servidor de arquivos como estados de progresso, mais um estado no presente protocolo da camada de operação. Uma nova lista da camada de apresentação é fornecida abaixo:

```
proctype presente (bit n)
{
status de byte, uabort;
endIDLE:
Faz
:: use_to_pres [n]? transferir ->
uabort = 0;
quebrar
:: use_to_pres [n]? abortar ->
pular
od;
od;
TRANSFERIR:
pres_to_ses [n]! transferência;
```

Página 358

CAPÍTULO 14
NOTAS BIBLIOGRÁFICAS

347

```
Faz
:: use_to_pres [n]? abortar ->
E se
:: (! uabort) ->
uabort = 1;
pres_to_ses [n]! abortar
:: (uabort) ->
afirmar (1 + 1! = 2)
fi
:: ses_to_pres [n]? aceitar ->
vá para FEITO
:: ses_to_pres [n]? rejeitar (status) ->
progresso:
E se
:: (status == FATAL || uabort) ->
ir para FALHA
:: (status == NON_FATAL && ! uabort) ->
ir para TRANSFERÊNCIA
fi
od;
FEITO:
pres_to_use [n]! aceitar;
goto endIDLE;
FALHOU:
pres_to_use [n]! rejeitar;
ir para endIDLE
}
```

A validação é direta deste ponto em diante.

```
$ spin -a pftp.ses4
$ cc -DBITSTATE -o pan pan.c
$ pan -l -w28
bit state space search por:
violações de asserção e loops de não progresso
vetor 148 byte, profundidade alcançada 458, loops de não progresso: 0
847134 estados, armazenados
18 estados, vinculados
1104341 estados, correspondentes
total: 1951493
fator de hash: 316,874472 (melhor cobertura se > 100)
(tamanho 2^28 estados, stackframes: 0/489)
```

Uma análise de espaço de estado de bits concluída com boa cobertura. Sem ciclos de não progresso foram descobertos, o que significa que, com boa probabilidade, a correção exigeimentos são cumpridos.

OUTRAS REDUÇÕES

Para confirmar os resultados anteriores com uma validação exaustiva, poderíamos buscar vários opções. Composição incremental e generalização podem ser usadas para combinar o usuário e processos de camada de apresentação em um único processo de ambiente para a sessão camada. Este modelo pode ter a seguinte aparência, devidamente rotulado com tags de progresso:

```

348
USANDO O VALIDADOR
CAPÍTULO 14
/*
 * Modelo de Validação PROMELA
 * Apresentação e camada do usuário - combinada e reduzida
 */
proctype presente (bit n)
{
    status de byte;
    progress0:
    pres_to_ses [n]! transferência ->
Faz
:: pres_to_ses [n]! abortar;
progress1:
pular
:: ses_to_pres [n]? aceitar, status ->
quebrar
:: ses_to_pres [n]? rejeitar, status ->
E se
:: (status == NON_FATAL) ->
goto progress0
:: (status! = NON_FATAL) ->
quebrar
fi
od
}

```

O comportamento externo deste processo é indistinguível do comportamento externo dos dois processos separados, com uma exceção importante: o novo modelo é menos bem comportado. O modelo reduzido pode desencadear um número arbitrário de mensagens de `aberto` enquanto uma solicitação de transferência está pendente. Se o protocolo da camada de sessão estiver correto para isso

ambiente, também deve ser correto em relação ao original, simplesmente porque o comportamento original é um subconjunto do novo. A validação agora pode ser feita exaustivamente e produz o seguinte resultado:

```

$ spin -a pftp.sess
$ cc -DMEMCNT = 27 -o pan pan.c
$ pan -l -m2000
pesquisa de espaço de estado completo para:
violações de asserção e loops de não progresso
vetor 132 byte, profundidade alcançada 1783, loops de não progresso: 0
553987 estados, armazenados
8 estados, vinculados
798367 estados, combinados
total: 1352362
conflitos de hash: 990275 (resolvido)
(tamanho 2^18 estados, stackframes: 0/325)
memória usada: 70872461

```

A execução de validação confirma que o requisito de correção da camada de sessão pro o tocol é devidamente atendido. Se esta primeira redução fosse insuficiente, redução adicional etapas ainda podem ser tomadas para forçar uma validação exaustiva. Todas as interações da sessão com o servidor de arquivos, por exemplo, pode ser removida e substituída por escolhas não determinísticas equivalentes dentro da camada de sessão. Da mesma forma, o combinado

CAPÍTULO 14
NOTAS BIBLIOGRÁFICAS
349

o usuário e a camada de apresentação podem ser fundidos no protocolo da camada de sessão para produzir um único processo que representa o comportamento de uma entidade da camada de sessão de protocolo. As combinações podem ser feitas manualmente, preservando cuidadosamente a equivalência com o modelo original, ou automaticamente com um método de composição incremental como discutido nos capítulos 8 e 11.

14.7 RESUMO

Nossa admiração por programadores que podem projetar e depurar um protocolo usando apenas ferramentas desenvolvidas para sistemas sequenciais só podem crescer após a primeira experiência com um sistema de validação de protocolo automatizado. É claro que não é surpreendente que o

execuções de validação relatadas neste capítulo falharam em revelar erros graves no design do Capítulo 7. Os erros certamente estavam presentes nas versões iniciais do protocolo, mas foram encontrados com SPIN e removidos antes que esses testes finais fossem executados formado. A maioria dos erros encontrados nas fases anteriores do projeto foram casos de incompletude que são muito difíceis de encontrar pela inspeção manual do código.

Dada uma máquina de tamanho razoável, os protocolos básicos para controle de sessão e fluxo controle pode ser facilmente validado com pesquisas puramente exaustivas de todos os estados do sistema. Tudo isso está ao alcance das ferramentas automatizadas. As ferramentas são severamente testados pelas condições de exceção que devem ser validadas: perda de mensagem, erros de duplicação e travamentos. O aumento da complexidade torna impossível realizar as validações tradicionais completamente exaustivas. Hashing de espaço de estado de bits prova ser uma alternativa poderosa aqui. Por exemplo, um teste realizado para um a versão anterior do protocolo da camada de sessão gerou 15.462.939 estados de sistema de 472 bytes cada. Um espaço de estado completo que armazena todos esses estados teria mais de 7 Gigabytes (7.298.507.208 bytes), muito além do que pode ser efetivamente armazenado ou processado. Com um máquina com 64 Mbytes de memória disponível para a pesquisa, não mais que 142.179 de esses estados podem ser armazenados em uma pesquisa de espaço de estado completa: uma cobertura de menos de 1%.

o técnica de espaço de estado de bits, usando a mesma quantidade de memória, pode acomodar mais 250 milhões de estados, mais de 15 vezes o que é necessário. Com este método, poderíamos aumentar efetivamente a cobertura dessa pesquisa de menos de 1% para um que, com alta probabilidade, está perto de 100%. Nenhum outro método conhecido até agora pode fazer melhor.

EXERCÍCIOS

14-1. Valide o seu protocolo favorito com as ferramentas descritas aqui.

14-2. Desenvolver e implementar ferramentas mais específicas para automatizar a generalização ou aumentar composição mental de modelos PROMELA (projeto de pesquisa).

NOTAS BIBLIOGRÁFICAS

Um estudo de validação detalhado, conforme realizado neste capítulo, raramente é documentado. o primeiras validações automatizadas foram relatadas em West e Zafiropolo [1978], embora o o poder analítico de nossas ferramentas cresceu substancialmente desde então. A validação O método aplicado neste capítulo foi originalmente descrito em Holzmann [1987b, 1988]. Suas capacidades são comparadas com abordagens mais convencionais do protocolo

Página 361

350

USANDO O VALIDADOR
CAPÍTULO 14

problema de validação em Holzmann [1990]. Tem sido aplicado a sistemas que são ordinariamente bem fora do intervalo de validação exaustiva, conforme relatado em Holzmann e Patti [1989].

Página 362

CONCLUSÃO

Enquanto o desempenho dos computadores e a velocidade das redes de dados continuam a melhorar continuamente, nossa capacidade de utilizar esses recursos de forma eficaz não. Com de dados o software de comunicação tornou-se um gargalo em muitos sistemas de alto desempenho; isto costuma ser muito mais lento do que o permitido pelo hardware e pode ser difícil estabelecer sua lógica consistência cal.

É notoriamente difícil escrever software para um sistema distribuído. É ainda mais difícil para provar rigorosamente a exatidão de tal software. Em sua forma mais simples, o problema é projetar métodos que permitem a execução assíncrona de máquinas em comunicação para trocar informações de forma rápida e confiável e para provar que esses métodos, ou tocois, têm certas propriedades desejáveis.

Hoje, a maioria dos protocolos é projetada de maneira *ad hoc*. Existe um conjunto conhecido de padrões de tocol, cuja descrição é fielmente copiada na maioria dos livros didáticos. Há sim, no entanto, pouca compreensão de por que alguns protocolos funcionam e o que há de errado com outras. Um designer precisa saber como um protocolo correto pode ser construído a partir de scratch e como esse design pode ser combinado com critérios específicos de design e correção. As técnicas que podem ser usadas para provar que um novo projeto de protocolo é correto têm há muito tempo é considerado esotérico demais para o uso real no dia a dia. Este livro pretende mostrar

que as ferramentas amadureceram.

Os métodos e ferramentas de design que discutimos permitem que o designer ataque diversos problemas fundamentais de coordenação de processos de uma maneira rigorosa e prática. Para projetar protocolos confiáveis, não importa qual seja sua aplicação, você precisa de ferramentas para testar

suas ideias. Este livro deve convencê-lo de que as ferramentas certas estão disponíveis. os recursos das novas ferramentas de validação às vezes é justificadamente considerado com cismo. Um generoso número de páginas é, portanto, dedicado neste texto a uma distribuição detalhada dessas ferramentas. Pela primeira vez, o código-fonte completo dessas ferramentas agora é feito disponíveis, tanto neste texto quanto em meio eletrônico. Sua avaliação crítica, experimentos, aplicações e comparações são ansiosamente convidados.

351

Página 363

REFERÊNCIAS

- Adi, W.** [1984], "Fast burst error-correct scheme with Fire code," *IEEE Trans. em computadores*, vol. C-33, No. 7, julho de 1984, pp. 613-618.
- Agerwala, TKM** [1975], *Rumo a uma Teoria para a Análise e Síntese de Sistemas exibindo simultaneidade*, Ph.D. Tese, Universidade Johns Hopkins, Baltimore, Md., 241 págs.
- Aggarwal, S., Courcoubetis, C., e Wolper, P.** [1990], "Added liveness propriedades para máquinas de estado finito acopladas," *ACM Trans. na programação Languages and Systems*, Vol 12, No. 2, pp. 303-339.
- Aggarwal, S., Kurshan, RP e Sharma, D.** [1983], "A language for protocol especificação e verificação," *Proc. 3º IFIP WG 6.1 Int. Workshop sobre protocolo Especificação, Teste e Verificação*, Norte da Holanda Publ., Amsterdam, pp. 35-50
- Aho, AV, Dahbura, AT, Lee, D., e Uyar, MU** [1988], "Uma otimização técnica para geração de teste de conformidade de protocolo com base em sequências UIO e rural Chinese Postman Tours," *Proc. 8º IFIP WG 6.1 Int. Workshop sobre protocolo Especificação, Teste e Verificação*, North-Holland Publ., Amsterdam.
- Aho, AV, Hopcroft, JE, Ullman, JD** [1974], *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 470 pgs. ISBN 0-201-00029-6.
- Aho, AV, Hopcroft, JE, Ullman, JD** [1983], *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass., 427 pgs. ISBN 0-201-00023-7.
- Aho, AV, Sethi, R., Ullman, JD** [1986], *Compilers - Principles, Techniques and Ferramentas*, Addison-Wesley, Reading, Mass., 796 pgs. ISBN 0-201-10088-6.
- Aho, AV e Ullman, JD** [1977], *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 604 pgs. ISBN 0-201-00022-9.
- Apt, KR, and Kozen, DZ** [1986], "Limits for automatic verification of finite state sistemas concorrentes," *Inf. Processing Letters*, Vol. 22, No. 6, maio de 1986, pp. 307-309.
- Arthurs, E., Chesson, GL e Stuck, BW** [1983], "Theoretical performance

352

Página 364

análise de controle de fluxo de janela deslizante," *IEEE Journal on Selected Areas in Comm.*, Vol. SAC-1, No. 5, pp. 947-959.

Balkovic, MD, Klancer, HW, Klare, SW e McGruther, WG [1971], " 1969-1970 Pesquisa de conexão: desempenho de transmissão de dados de banda de voz em alta velocidade on the Switched Telecommunications Network," *Bell System Technical Journal*, Vol. 50, No. 4, 1349-1384.

Bartlett, KA, Scantlebury, RA e Wilkinson, PT [1969], "Uma nota sobre confiável transmissão full-duplex em linhas half-duplex," *Comm. do ACM*, Vol. 12, nº 5, 260-265.

Beeforth, TH, Grimsdale, RL, Halsall, F., e Woolons, DJ [1972], "Organização proposta para rede de comunicação de dados comutada por pacote." *Proc.*

- IEEE*, vol. 119, No. 12, Dez. 1972, pp. 1677-1682.
- Bennet, WR e Davey, JR** [1965], *Data Transmission*, McGraw-Hill, New Iorque.
- Berlekamp, ER** [1968], *Algebraic Coding Theory*, McGraw-Hill, New York.
- Bertsekas, D., e Gallager, R.** [1987], *Redes de Dados*, Prentice Hall, Englewood Cliffs, NJ, 486 pgs. ISBN 0-13-196825-4.
- Bochmann, G. von** [1983], *Concepts for Distributed Systems Design*, Springer-Verlag, Nova York.
- Bochmann, G. von** [1986], *Métodos e ferramentas para o projeto e validação de especificações e implementações de protocolo*, Relatório 596, outubro de 1986, Universidade de Montreal, Canadá, 56 pgs.
- Bochmann, G. von, e Sunshine, CA** [1980], "Formal methods in projeto de protocolo de comunicação," *IEEE Trans. on Communications*, vol. COM-28, No. 4, pp. 624-631.
- Bolognesi, T., e Brinksma, E.** [1987], "Introdução à especificação ISO linguagem Lotos," *Redes de Computadores e Sistemas ISDN*, Vol. 14, pp. 25-59.
- Bond, DJ** [1987], "Um estudo teórico de ruído de explosão", *British Telecom Technology Journal*, vol. 5, No. 4, outubro de 1987.
- Bose, RC e Ray-Chaudhuri, DK** [1960], "Em uma classe de correção de erros códigos de grupo binários," *Informar. Controle*, vol. 3, pp. 68-79, 279-290.
- Bourguet, A.** [1986], "Uma ferramenta de rede de Petri para verificação de serviço em protocolos, " *Proc.*
- 6º Workshop sobre Especificação, Teste e Verificação de Protocolo*, Montreal, Norte-Holland Publ., Amsterdam, pp. 281-292.
- Brand, D. e Joyner, WH, Jr.** [1978], "Verificação de protocolos usando símbolos execução," *Computer Networks*, Vol. 2, pp. 351-360.
- Brand, D., e Zafiropulo, P.** [1980], "Synthesis of protocol for an unlimited número de processos," *Proc. Conf. De protocolos de rede de computadores*, IEEE, pp. 29-40.
- Brand, D. e Zafiropulo, P.** [1983], "On comunicando máquinas de estado finito," *Journal of the ACM*, vol. 30, No. 2, pp. 323-342.
- Bredt, TH** [1970], *The Mutual Exclusion Problem*, Report SU-STAN-CS-70-173, Stanford University, Califórnia, agosto de 1970, 71 pgs.
- Brilliant, MB** [1978], "Observações de erros e taxas de erro em T1 digital linhas repetidas," *Bell System Technical Journal*, Vol. 57, No. 3, março de 1978, pp.

Página 365

- 711-747.
- Brinksma, E.** [1987], "Uma introdução a Lotos," *Proc. 7º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo, teste e verificação*, North-Holland Publ., Amsterdam.
- Brinksma, E.** [1988], *On the Design of Extended Lotos*, Ph.D. Tese, Universidade de Twente, Holanda, 240 pgs.
- Brinksma, E., Alderden, R., Langerak, R., Tretmans, J., e Lagemaat, J. van de,** [1989], *A Formal Approach to Conformance Testing*, Report 89-45, Dept. of Ciência da Computação, Universidade de Twente, Holanda, 18 pgs.
- Brown, GM, Gouda, MG e Miller, RE** [1989], "Reconhecimentos de bloco: redesenhandando o protocolo da janela," *Proc. ACM SIGCOMM '89*, Austin, Texas.
- Brown, GM, Gouda, MG e Wu, C.** [1989], "Token systems that self-estabilizar," *IEEE Trans. em computadores*, vol. 36, No. 6, pp. 845-852.
- Browne, MC, Clarke, EM, Dill, DL e Mishra, B.** "Verificação automática de circuitos sequenciais usando lógica temporal," *IEEE Trans. em computadores*, vol. C-35, No. 12, pp. 1035-1043.
- deBruyn, NG** [1967], "Comentários adicionais sobre um problema em simultâneo controle de programação," *Comm. do ACM*, Vol. 10, No. 3, pp. 137-138.
- Budkowski, S., e Dembinski, P.** [1987], "Uma introdução a Estelle: a linguagem de especificação para sistemas distribuídos," *Redes de Computadores e ISDN Systems*, vol. 14, pp. 3-23.
- Byte** [1989], Vol. 14, No. 1, janeiro de 1989, pp. 363-376.
- CCITT** [1977], *Orange Book VIII.2*, Recomendação X.25, "Interface entre

DTE e DCE para terminais operando no modo de pacote em redes públicas de dados,"
União Internacional de Telecomunicações (UIT), Genebra.

CCITT

[1988],

Livro Azul, Recomendação Z.100, Internacional
União das Telecomunicações (ITU), Genebra.

Campbell-Kelly, M. [1988], "Comunicações de dados em NPL," em *Annals of the History of Computing*, vol. 9, No. 3.

Cardelli, L., e Pike, R. [1985], "Squeak: a language for communication with
ratos," Proc. ACM Siggraph, Vol 19, No. 3, pp. 199-204.

Cerf, VG e Kahn, RE [1974], "Um protocolo para rede de pacotes
intercomunicação," *IEEE Trans. on Communications*, vol. COM-22, nº 5, pp.
637-648.

Chappe, C. [1798], *Lettres sur le nouveau te ' le ' graphe*, (em francês), Paris, França.

Chappe, IUJ [1824], *Histoire de la te ' le ' graphie*, do irmão de Claude Chappe
Ignace, (em francês), Paris, França.

Chesson, G. [1987], "The protocol engine project," *Unix Review*, Vol. 5, No. 9,
Setembro de 1987, pp. 70-77.

Choi, TY e Miller, RE [1986], "Análise e síntese de protocolo por
participamento," *Computer Networks and ISDN Systems*, Vol. 11, No. 5, pp. 367-383.

Chow, T. [1978], "Testing software design modeled by finite state machines," *IEEE Trans. em Engenharia de Software*, vol. SE-4, No. 3, pp. 178-187.

Chu, PM e Liu, MT [1988], "Síntese de protocolo em um modelo de transição de estado,"

Página 366

Proc IEEE Compsac Conf., Outubro de 1988, pp. 505-512.

Chu, PM [1989], *Towards Automating Protocol Synthesis and Analysis*, Ph.D.

Tese, Ohio State University, Dept. of Computer and Information Science, 168 pgs.

Clark, D. [1985], *NETBLT: A bulk data transfer protocol*, Report RFC-275, MIT,
Lab. for Computer Science, fevereiro de 1985.

Clark, D., Lambert, M. e Zhang, L. [1988], "NETBLT: A high throughput
protocolo de transporte," *ACM Computer Communication Review*, Vol 17., No. 5, pp.
353-359.

Clarke, EM, Emerson, EA, Sistla, AP [1983], "Automatic verification of finite
estado sistemas concorrentes usando especificações lógicas temporais: uma abordagem prática,"
Proc. 10º Simpósio ACM sobre Princípios de Linguagens de Programação, Austin, Tx.

Cole, R. [1987], *Computer Communications*, Springer Verlag, New York, ISBN 0-
387-91306-8, 173 pgs.

Cooke, WF [1842], *Telegraphic Railways ou a via única recomendada por
segurança, economia e eficiência, sob a proteção e controle do sistema elétrico
telégrafo - com referência particular à comunicação ferroviária com a Escócia, e
para a Irish Railways*, W. Lewis and Son, Finch-Lane, Londres, Inglaterra.

Courcoubetis, C., Vardi, M., Wolper, P., e Yannakakis, M. [1990], "Memory
algoritmos eficientes para a verificação de propriedades temporais," *2º Workshop sobre
Computer-Aided Verification*, Rutgers University, New Brunswick, NJ, 18 a 20 de junho,
1990.

Cunha, PRF, e Maibaum, TSE [1981], "A synchronization calculus for
programação orientada a mensagens," *Proc. Int. Conf. on Distributed Systems*, IEEE, pp.
433-445.

Dahbura, A., e Sabnani, K. [1988], "Experiência em estimar a cobertura de um
teste de protocolo," *Proc. IEEE INFOCOM '88*, março de 1988, pp.71-85.

Dahl, OJ, Dijkstra, EW, and Hoare, CAR [1972], *Structured Programming*,
Academic Press, New York.

Decina, M. e Julio, U. de [1982], "Desempenho de redes digitais integradas:
padrões internacionais," *IEEE 1982 Int. Communications Conference*, ICC, Vol. 1,
pp. 2D1.1-6

Dijkstra, EW [1959], "Uma nota sobre dois problemas em conexão com gráficos,"
Numerische Mathematik, Vol. 1, pp. 269-271.

Dijkstra, EW [1965], "Solução de um problema no controle de programação concorrente,"

- Com. do ACM*, Vol. 8, nº 9, p. 569.
- Dijkstra, EW** [1968], "Cooperating sequential process," in: *Programming Languages*, F. Genuys (Ed.), Academic Press, New York.
- Dijkstra, EW** [1968a], "Vá para considerado prejudicial," *Comm. do ACM*, Vol. 11, No. 3, pp. 147-148, 538, 541.
- Dijkstra, EW** [1968b], "The structure of the 'THE' multiprogramming system," *Com. do ACM*, Vol. 11, No. 5, pp. 341-346.
- Dijkstra, EW** [1969a], "Complexidade controlada por ordenação hierárquica de função e variabilidade," em *Software Engineering*, P. Naur e B. Randell (eds.), Sc. Aff. Div., NATO, Bruxelas, pp. 114-116.

Página 367

- Dijkstra, EW** [1969b], *Notes on Structured Programming*, THE Report EWD-249 (70-Wsk-03), University of Technology Eindhoven, Holanda. Também publicado em: Dahl et al. [1972].
- Dijkstra, EW** [1972], "Hierarchical order of sequential process," in: *Operating Systems Techniques*, Academic Press, New York.
- Dijkstra, EW** [1974], "Sistemas auto-estabilizantes apesar do controle distribuído," *Com. do ACM*, Vol. 17, No. 11, pp. 643-644.
- Dijkstra, EW** [1975], "Comandos guardados, não-determinação e derivação formal de programas," *Comm. do ACM*, Vol. 18, No. 8, pp. 453-457.
- Dijkstra, EW** [1976], *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ.
- Dijkstra, EW** [1986], "Uma prova tardia de auto-estabilização," *Distribuído Computing*, vol. 1, janeiro de 1986, pp. 5-6.
- Duke, R., Hayes, I., King, P. e Rose, G.** [1988], "Protocol verification and especificação usando Z," *Proc. 8º IFIP WG 6.1 Int. Workshop sobre protocolo Specification, Testing and Verification*, North-Holland Publ., Amsterdam, pp. 33-46.
- Duke, R., Hayes, I. e Rose, G.** [1988], *Verification of a cyclic retransmission protocol*, Relatório Técnico nº 92, julho de 1988, Departamento de Ciência da Computação, University of Queensland, Australia, 23 pgs.
- Edelcrantz, AN** [1976], *Avhandling om Telegrapher, och Fo "rso" k Til en ny Inra "tning da" raf*, Johan Pehr Lind, Estocolmo, (em sueco).
- Edmonds, J. e Johnson, EL** [1973], "Matching, Euler tours and the Chinese carteiro," *Mathematical Programming*, Vol. 5, pp. 88-124.
- Eijk, P. van, Vissers, CA, e Diaz, M.** [1989], *The Formal Description Technique Lotos*, North-Holland Publ., Amsterdam, 1989. 451 pgs.
- Eisenberg, MA, e McGuire, MR** [1972], "Comentários adicionais sobre o de Dijkstra problema de controle de programação simultânea," *Com. do ACM*, Vol. 15, nº 11, p. 999.
- Field, JA** [1976], "Efficient computer-computer communication," *Proc. IEEE*, Vol. 123, agosto de 1976, pp. 756-760.
- Finkel, A. e Rosier, L.** [1987], *A Survey of FIFO Nets*, Publ. 632, outubro de 1987, University of Montreal, Dept. d'Informatique et de Recherche Operationelle, 37 pgs.
- Fleming, HC e Hutchinson, RM, Jr.** [1971], "Pesquisa de conexão de 1969-1970: desempenho de transmissão de dados de baixa velocidade nas telecomunicações comutadas network," *Bell System Technical Journal*, Vol. 50, No. 4, 1385-1405.
- Fletcher, JG** [1982], "Um checksum aritmético para transmissão serial," *IEEE Trans. on Communications*, vol. COM-30, No. 1, janeiro de 1982.
- Floyd, RW** [1967], "Assigning meanings to programs," *Proc. Simpósios em Aplicada Mathematics*, American Mathematical Society, Vol. 19, pp. 19-32.
- Fraser, AG e Marshall, WT** [1989], "Data Transport in a Byte Stream Network," *IEEE Journal on Selected Areas in Comm.*, Vol. 7, No. 7, pp. 1020-1033.
- Friedman, AD, e Menon, PR** [1971], *Fault Detection in Digital Circuits*, Prentice Hall, Englewood Cliffs, NJ, 220 pgs.
- Garey, MG, e Johnson, DS** [1979], *Computers and Intractability: a Guide to theory of NP-completeness*, Freeman, San Francisco.

Página 368

- Gerla, M. e Kleinrock, L.** [1980], " Flow control: a comparative survey, " *IEEE Trans. on Communications* , vol. COM-28, No. 4, abril de 1980, pp. 553-574.
- Gibbons, A.** [1985], *Algorithmic Graph Theory* , Cambridge University Press, 260 pgs.
- Gill, A.** [1962], *Introdução à Teoria das Máquinas de Estados Finitos* , McGraw-Hill, Nova York.
- Gobershtain, SM** [1974], " Verificar palavras para os estados de um autômato finito, " *Kibernetika* , No. 1, pp. 46-49, (original em russo).
- Godefroid, P.** [1990], " Usando pedidos parciais para melhorar a verificação automática métodos, " *Proc. 2º Workshop sobre verificação auxiliada por computador* , RP Kurshan e EM Clarke (Eds.), Rutgers University, Springer Verlag, New York.
- Gotzheim, R., e Bochmann, G. von** [1986], *Deriving Protocol Specification de Especificações de Serviço* , Universidade de Montreal, Relatório # 562.
- Gouda, MG** [1983], " Um exemplo para a construção de máquinas comunicantes por refinamento passo a passo, " *Proc. 3º IFIP WG 6.1 Int. Workshop sobre protocolo Specification , Testing and Verification* , North-Holland Publ., Amsterdam, pp. 63-74.
- Gouda, MG** [1987], *The Stabilizing Philosopher: Asymmetry by Memory and by Ação* , Relatório TR-87-12, abril de 1987, Dept. of Computer Sciences, University of Texas em Austin.
- Gouda, MG e Han, JY** [1985], " Validação de protocolo por estado de progresso justo exploração, " *Computer Networks and ISDN Systems* , Vol. 9, pp. 353-361.
- Gouda, MG e Yu, YT** [1984], " Validação de protocolo por estado de progresso máximo exploração, " *IEEE Trans. on Communications* , vol. COM-32, No. 1, pp. 94-97.
- Griffiths, G., and Stones, GC** [1987], " O algoritmo do leitor de folhas de chá: um eficiente implementação de CRC-16 e CRC-32, " *Comm. do ACM* , Vol. 30, nº 7, pp. 617-620.
- Hajek, J.** [1978], " Automatically verification data transfer protocol, " *Proc. 4º ICCC* , Kyoto, pp. 749-756.
- Hamming, RW** [1950], " Detecção de erros e códigos de correção de erros ", *Sistema Bell Technical Journal* , vol. 29, pp. 147-160.
- Har'El, Zri e Kurshan, RP** [1990]. " Software para desenvolvimento analítico de protocolos de comunicação, " *AT&T Technical Journal* , Special issue on Protocol Teste e verificação. Vol 69, No 1, pp. 45-59.
- Harbison, SP e Steele, GL, Jr.** [1987], *C - a Reference Manual* , Prentice Hall, Englewood Cliffs, NJ, 2ª ed., ISBN 0-13-109802-0.
- Harrison, MA** [1965], *Introduction to Switching and Automata Theory* , McGraw-Hill, New York, Series in Systems Science.
- Hartley, RVL** [1928], " Transmission of information ", *Bell System Technical Journal* , Vol 7, pp. 535-563.
- Hartmanis, J., e Stearns, RE** [1966], *Algebraic Structure Theory of Sequential Máquinas* , Prentice Hall, Englewood Cliffs, NJ Int. Series in Appl. Matemática.
- Harvard University**, Computation Laboratory Staff, [1951], *Synthesis of Electronic Computing and Control Circuits* , Harvard University Press, Massachusetts.

Página 369

- Hayes, I., Mowbray, M., e Rose, GA** [1989], " Signaling system No. 7, the camada de rede, " *Proc. 9º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo , Testing and Verification* , North-Holland Publ., Amsterdam.
- Hennie, FC** [1964], " Fault detecting experiment for sequential circuits, " *Proc. 5º Symp. Teoria de circuitos de comutação e design lógico* , Princeton University, Princeton, NJ, novembro 11-13 1964, pp. 95-110.
- Herbarth, D.** [1978], *Die Entwicklung der Optischen Telegrafie in Preussen* , Arbeitsheft 15, Landeskonservator Rheinland, Rheinland-Verlag, Koln 1978, em Alemão, ISBN 3-7927-0247-9, 200 pgs.
- Hoare, CAR** [1978], " Communicating sequential process ', *Comm. do ACM* , Vol. 21, No. 8, pp. 666-677.
- Hocquenghem, A.** [1959], " Codes correcteurs d'erreurs, " *Chiffres (Paris)* , Vol. 2, pp. 147-156.
- Hodge, FW** [1910], *Handbook of American Indians North of Mexico* , Part 2,

- Smithsonian Institution, Bureau of Ethnology, Bulletin 30.
- Holzmann, GJ** [1979], *Coordination Problems in Multiprocessing Systems*, Ph.D. Tese, Delft University of Technology, Holanda, 315 pgs.
- Holzmann, GJ** [1982a], "A Theory for protocol Validation," *IEEE Trans. em Computers*, Vol. C-31, No. 8, pp. 730-738.
- Holzmann, GJ** [1982b], "Algebraic validation methods," *Proc. 2º IFIP WG 6.1 Int. Workshop sobre especificação, teste e verificação de protocolo*, North-Holland Publ., Amsterdam, pp. 383-390.
- Holzmann, GJ** [1984a], "O sistema Pandora - um sistema interativo para o design of data communication protocol," *Computer Networks*, Vol. 8, nº 2, pp. 71-81.
- Holzmann, GJ** [1984b], "Execução simbólica reversa de protocolos," *Proc. 4º IFIP WG 6.1 Int. Workshop sobre especificação, teste e verificação de protocolo*, North-Holland Publ., Amsterdam, pp. 19-30.
- Holzmann, GJ** [1985], "protocolos de rastreamento", *AT&T Technical Journal*, Vol 64, Dezembro de 1985, pp. 2413-2434.
- Holzmann, GJ** [1987a], "Automated protocol validation in 'Argos,' assertion prova e busca dispersa," *IEEE Trans. em Engenharia de Software*, vol. 13, No. 6, junho de 1987, pp. 683-697.
- Holzmann, GJ** [1987b], "Sobre os limites e possibilidades do protocolo automatizado análise," *Proc. 7º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo, teste, e Verification*, North-Holland Publ., Amsterdam, pp. 137-161.
- Holzmann, GJ** [1988], "Uma técnica de análise de acessibilidade de protocolo melhorada," *Software, Practice and Experience*, Vol 18, No. 2, fevereiro de 1988, pp. 137-161.
- Holzmann, GJ, e Patti, J.** [1989], "Validating SDLifications: an experimento," *Proc. 9º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo, Testing and Verification*, North-Holland Publ., Amsterdam.
- Holzmann, GJ** [1990], "Algorithms for Automated Protocol Validation," *AT&T Jornal técnico*, edição especial sobre teste e verificação de protocolo. Vol 69, No 1, pp. 32-44.

Página 370

- Hsieh, EP** [1971], "Checking experiment for sequential machines," *IEEE Trans. on Computers*, Vol C-20, No. 10, pp. 1152-1166.
- Hubbard, G.** [1965], *Cooke e Wheatstone and the Invention of the Electric Telegraph*, Routledge & Kegan Paul, Londres.
- Huffman, DA** [1954], "A síntese de circuitos sequenciais," *Jour. Franklin Inst.*, Vol. 257, março-abril de 1954, No. 3-4, pp. 161-190, 275-303.
- Huffman, DA** [1964], "Canonical forms for information-lossless finite state logic machines," em *Sequential Machines: Selected Papers*, EF Moore (Ed.), Addison-Wesley, Reading, Mass.
- Hutchinson, NC, Peterson, LL e Rao, H.** [1989], "The X-kernel: an open projeto do sistema operacional," *Proc. 2º Workshop sobre Sistemas Operacionais de Estação de Trabalho*, IEEE Computer Society, WWS-II, 27-29 de setembro de 1989, pp. 55-59.
- Hyman, H.** [1966], "Comentários sobre um problema no controle de programação simultânea," *Com. do ACM*, Vol 9, No. 1, p. 45
- IBM** [1964], *Error Control through Coding*, Relatório Técnico Documentário No. RADC-TDR-64-149, Vol. 1 a 9 de julho de 1964, Griffis AFB, NY
- IFIP** [1982-presente], *Proc. IFIP WG 6.1 Int. Workshops sobre especificação de protocolo, Teste e verificação*, os procedimentos desta conferência anual são publicados por: North-Holland Publ., Amsterdam.
- ISO** [1979], *Modelo de Referência de Interconexão de Sistemas Abertos*, Doc. ISO / TC97 / SC16 N227.
- ISO** [1983], *Connection Oriented Transport Protocol*, Doc. DP 8073.
- ISO** [1987], *Metodologia e Estrutura de Teste de Conformidade OSI*, ISO / TC97 / SC21, Parte 2: Especificação do conjunto de testes abstratos, "Anexo E: A árvore e notação tabular combinada."
- Jacobson, V.** [1988], "Congestion Avoidance and control," *ACM Computer*

- Revisão de comunicação* , vol. 18, No. 4, pp. 314-329.
- Jain, R.** [1986], " Um esquema de controle de congestionamento baseado em tempo limite para fluxo de janela redes controladas, " *IEEE Journal on Selected Areas in Comm.* , Vol. SAC-4, No. 7, Outubro de 1986, pp. 1162-1167.
- Jain, R., Ramakrishan, KK, e Chiu, DM** [1987], *Congestion Avoidance in Redes de computadores com uma camada de rede sem conexão* , relatório DEC-TR-506, Digital Equipment Corp., 17 págs.
- Kain, RY** [1972], *Automata Theory: Machines and Languages* , McGraw-Hill, New York, Computer Science Series.
- Kanellakis, PC, and Smolka, SA** [1990], " Expressões CCS, estado finito processos, e três problemas de equivalência, " *Informação e Computação* , Vol 86, No. 1, pp. 43-68.
- Karn, P., e Partridge, C.** [1987], " Melhorando as estimativas de tempo de ida e volta em protocolos de transporte confiáveis, " *Proc. ACM SIGCOMM '87* , pp. 2-7.
- Kendall, DG** [1951], " Alguns problemas na teoria das filas ", *J. Royal. Estatista. Sociedade.* , Ser. B, Vol 13, pág. 151
- Kernighan, BW e Pike, R.** [1984], *The UNIX Programming Environment* , Prentice Hall, Englewood Cliffs, NJ, ISBN 0-13-937699-2.

Página 371

- Kernighan, BW e Ritchie, DM** [1978], *The C Programming Language* , Prentice Hall, Englewood Cliffs, NJ, 2^a ed. 1988, ISBN 0-13-110362-8.
- Klee, V.** [1980], " Otimização combinatória: qual é o estado da arte ?, " *Math. Oper. Res.* , Vol. 5, pp. 1-26.
- Knudsen, HK** [1983], *Linked-state Machines* , Report LA-9770-MS, Los Alamos National Laboratory, Los Alamos, NM, julho de 1983, 84 pgs.
- Knuth, DE** [1966], " Comentários adicionais sobre um problema em simultâneo controle de programação, " *Comm. do ACM* , Vol. 9, No. 5, pp. 321-322.
- Knuth, DE** [1981], " Verification of link level protocol, " *BIT* , Vol. 21, pp. 31-36.
- Kohavi, Z.** [1978] *Switching and Finite Automata Theory* , 2^a ed., McGraw-Hill, Nova York, Computer Science Series, 658 págs.
- Krogdahl, S.** [1978], " Verification of a class of link level protocols, " *BIT* , Vol. 18, pp. 436-448.
- Kruijer, HSM** [1979], " Auto-estabilização (apesar do controle distribuído) em árvore-sistemas estruturados, " *Information Processing Letters* , Vol. 8, No. 2, pp. 91-95.
- Kuan, MK.** [1962], " Programação gráfica usando pontos ímpares ou pares, " *chinês Matemática.* , Vol. 1, pp. 273-277.
- Kuo, FF** [1981], *Protocols and Techniques for Data Communications Networks* , FF Kuo (Ed.), Prentice Hall, Englewood Cliffs, NJ
- Lam, SS e Shankar, AU** [1984], " Verificação de protocolo via projeções, " *IEEE Trans. em Engenharia de Software* , vol. 10, No. 4, pp. 325-342.
- Lamport, L.** [1974], " Uma nova solução para a programação concorrente de Dijkstra problema, " *Comm. do ACM* , Vol. 17, No. 8, pp. 453-455.
- Lamport, L.** [1976], " A sincronização de processos independentes, " *Acta Informatica* , vol. 7, pp. 15-34.
- Lamport, L.** [1977], " Proving the correctness of multiprocess programs " *IEEE Trans. on Software Engineering* , Vol SE-3, No. 2, pp 125-143.
- Lamport, L.** [1984], " Unsolved problems, and non-problems in concurrency, " Endereço convidado, *Proc. 3º Simpósio ACM sobre Princípios de Computação Distribuída* , Vancouver, Canadá, agosto de 1984, pp. 1-11.
- Lamport, L.** [1986], " O problema de exclusão mútua - partes I e II ", *Journal of ACM* , vol. 33, No. 2, abril de 1986, pp. 313-347.
- Lin, FJ, Chu, PM e Liu, MT** [1987], " Verificação de protocolo usando análise de acessibilidade, " *Computer Communication Review* , Vol. 17, No. 5, pp. 126-135
- Lin, S., e Rado, T.** [1965], " Computer studies of Turing machine problems, " *Journal of the ACM* , vol. 12, No. 2, pp. 196-212.
- Lindgren, B.** [1987], *The X.25 Handbook* , Kalmar Publ., Suécia.

- Linn, RJ e McCoy, WH** [1983], " Produzindo testes para implementações de OSI protocolos, " *Proc. 3º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo , testes, e Verification* , North-Holland Publ., Amsterdam, pp. 505-520.
- Lint, JH van** [1971], *Coding Theory* , Lecture Notes in Mathematics, Springer Verlag, New York, 136 pgs.
- Lynch, WC** [1968], " Reliable full duplex file transmission over half-duplex

Página 372

- linhas telefônicas, " *Comm. do ACM* , Vol. 11, No. 6, pp. 407-410.
- MacWilliams, FJ, e Sloane, NJA** [1977], *Theory of Error-correcting Codes* , Publicação da Holanda do Norte, Amsterdam.
- Mallery, J.** [1881], " Sign Language between North American Indians, " *Bureau of Relatório Anual de Etnologia* , vol. 1, 1879-1880, Washington, GPO.
- Malmgren, E.** [1964], " Den optiska telegrafen i Furusund, " in *Daedalus* , Annual Publ. do Museu Nacional de Ciência e Tecnologia da Suécia, Estocolmo, (em Sueco).
- Mandelbrot, B.** [1965], " Clusters de erros semelhantes em sistemas de comunicação e o conceito de estacionariedade condicional, " *IEEE Trans. em comunicações Technology* , Vol COM-13, No. 1, pp. 71-90.
- Manna, Z., e Pnueli, A.** [1987], " Specification and verification of concurrent programas por autômatos, " *Proc. ACM Conf. sobre princípios de programação Languages* , POPL'87, 21-23 de janeiro de 1987, Munich, W.Germany, pp. 1-12.
- Manna, Z., e Wolper, P.** [1984], " Synthesis of communication process from especificações lógicas temporais, " *ACM Trans. em linguagens de programação e Systems* , vol. 6, No. 1, janeiro de 1984, pp. 68-93.
- Marland, EA** [1964], *Early Electrical Communication* , Abelard-Schuman, New York, 220 págs.
- Maxemchuck, N., e Sabnani, K.** [1987], " Probabilistic verification of protocolos de comunicação, " *Proc. 7º IFIP WG 6.1 Int. Workshop sobre protocolo Specification , Testing, and Verification* , North-Holland Publ., Amsterdam, pp. 307-320
- McCulloch, WS e Pitts, W.** [1943], " Um cálculo lógico das idéias imanentes em atividade nervosa, " *Bulletin of Mathematical Biophysics* , Vol. 5, pp. 115-133.
- McIlroy, MD** [1982], " Development of a Spelling List, " *IEEE Trans. em Communications* , vol. 30, pp. 91-99.
- McNamara, JE** [1982], *Technical Aspects of Data Communication* , 2^a ed., Digital Press, DEC, Bedford, Mass.
- McQuillan, JM e Walden, DC** [1977], " The ARPA network design decisões, " *Computer Networks* , Vol. 1, pp. 243-289.
- Mealy, GH** [1955], " Um método para sintetizar circuitos sequenciais, " *Bell System Technical Journal* , vol. 34, setembro de 1955, pp. 1045-1079.
- Merlin, PM** [1979], " Especificação e validação de protocolos, " *IEEE Trans. em Communications* , vol. COM-27, 1761-1780.
- Merlin, PM e Bochmann, G. von** [1983], " On the construction of submodule especificações e protocolos de comunicação, " *ACM Trans. na programação Languages and Systems* , janeiro de 1983, pp. 1-25.
- Michaelis, AR** [1965], *From Semaphore to Satellite* , International União das Telecomunicações, Genebra.
- Milner, R.** [1980], " A calculus for communicating systems, " *Lecture Notes in Computer Science* , vol. 92
- Moitra, A.** [1985], " Automatic construction of CSP programs from sequential non-deterministic programs, " *Science of Computer Programming* , Vol. 5 (1985), pp. 277-307.

Página 373

- Moore, EF** [1956], " Gedanken-experiencias em máquinas sequenciais, " *Automata Studies, Annals of Mathematics Studies* , No. 34, Princeton University Press,

- Princeton, NJ, pp. 129-153.
- Moore, EF** [1964], *Sequential Machines: Selected Papers*, Addison-Wesley, Leitura, Missa.
- Morris, R.** [1968], "Scatter Storage Techniques," *Comm. do ACM*, Vol. 11, No. 1, pp. 38-44.
- Multari, N.** [1989], *Towards a Theory of Self-Stabilizing Protocols*, Ph.D. Tese, Departamento de Ciência da Computação da Universidade do Texas em Austin.
- Naito, S., e Tsunoyama, M.** [1981], "Detecção de falhas para máquinas sequenciais por tours de transição," *Proc. IEEE Fault Tolerant Computing Conf.*.
- Nakassis, A.** [1988], "Algoritmo de detecção de erro de Fletcher: como implementá-lo de forma eficiente e como evitar as armadilhas mais comuns," *Comunicação por Computador Review*, vol. 18, No. 5, outubro de 1988, pp. 63-88.
- Needham, RM e Herbert, AJ** [1982], *The Cambridge Distributed Computing System*, International Computer Science Series, Addison-Wesley, ISBN 0-201-14092-6, 170 pgs.
- Nightingale, JS** [1982], "Teste de protocolo usando uma implementação de referência," em: *Proc. 2º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo, teste e Verification*, North-Holland Publ., Amsterdam, pp. 513-522.
- Nock, OS** [1967], *Historic Railway Disasters*, Ian Allan, Londres.
- Nyquist, H.** [1924], "Certos fatores que afetam a velocidade do telégrafo," *Bell System Technical Journal*, vol. 3, pp. 324-346.
- Owicki, S., e Lamport, L.** [1982] "Proving liveness properties of concurrent programas," *ACM Trans. em Linguagens e Sistemas de Programação*, Vol. 4, No. 3, Julho de 1982, pp. 455-495.
- Pageot, JM e Jard, C.** [1988], "Experience in guiding simulação," *Proc. VIII Oficina de Especificação, Teste e Verificação de Protocolo*, Atlantic City, 1988, North-Holland Publ., Amsterdam.
- Perez, A.** [1983], "Byte-wise CRC calculations", *IEEE Micro*, junho de 1983, pp. 40-50
- Peterson, WW e Weldon, EJ, Jr.** [1972], *Error-correcting Codes*, 2ª ed., MIT Press, Cambridge, Mass.
- Petri, CA** [1962], *Kommunikation mit Automaten*, em alemão, Universidade de Bonn, Alemanha. Reimpresso em inglês como *Communication with Automata*, Suppl. 1 para Techn. Relatório RADC-TR-65-377, Vol. 1, Griffis AFB, New York, 1966.
- Pike, R. e Kernighan, BW** [1984]. "Projeto de programa no Unix ambiente," *Bell System Technical Journal*, Vol. 63, No. 8, outubro de 1984, pp. 1595-1605.
- Pnueli, A.** [1977], "A lógica temporal dos programas," *Proc. 18º Simpósio IEEE on Foundations of Computer Science*, Providence, RI, pp. 46-57.
- Pouzin, L.** [1976], "Controle de fluxo em redes de dados - métodos e ferramentas," *Proc. Int. Conf. Comp. Com.*, Toronto, Ont., Canadá, agosto de 1976.

Página 374

- Pouzin, L. e Zimmerman, H.** [1978], "A Tutorial on protocol," *Proc. IEEE*, Vol. 66, No. 11, 1346-1370.
- Prescott, GB** [1877], *Electricity and the Electric Telegraph*, Nova York, 1877.
- Price, WL** [1971], *Graphs and Networks*, Auerbach Publ., ISBN 0-8876-128-2, 108 pgs.
- Prineth, R.** [1982], "Um algoritmo para construir sistemas distribuídos de estado máquinas," *Proc. 2º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo, testes, e Verification*, North-Holland Publ., Amsterdam, pp. 261-282.
- Probert, RL e Ural, H.** [1983], "Requisitos para uma especificação de teste linguagem para teste de implementação de protocolo," em: *Proc. 3º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo, teste e verificação*, North-Holland Publ., Amsterdam. pp. 437-436.
- Probst, DK** [1990], "Usando semântica de ordem parcial para evitar a explosão de estado problema em sistemas assíncronos," *Proc. 2ª Oficina Assistida por Computador Verification*, RP Kurshan e EM Clarke (Eds.), Rutgers University, Springer Verlag, Nova York.

- Puzman, J. e Porizek, R.** [1980], *Communication Control in Computer Networks*, Wiley & Sons, Nova York.
- Queille, JP** [1982], *Le systeme Cesar: descrição, especificação e análise dos reparties de aplicações*, Ph.D. Tese, Departamento de Ciência da Computação, Universidade de Grenoble, França,
- Rado, T.** [1962], "On non-computable functions," *Bell System Technical Journal*, Vol. 41, maio de 1962, pp. 977-884.
- Rafiq, O., and Ansart, JP** [1983], "A protocol validator and its applications," *Proc. 3º Workshop sobre Especificação, Teste e Verificação de Protocolo*, Zurique, North-Holland Publ., Amsterdam, pp. 189-198.
- Rayner, D.** [1982], "A system for test protocol implementations," in: *Proc. 2º IFIP WG 6.1 Int. Workshop sobre especificação, teste e verificação de protocolo*, North-Holland Publ., Amsterdam, pp. 539-554.
- Rayner, D.** [1987], "OSI conformance testing," *Computer Networks and ISDN Systems*, edição especial sobre testes de conformidade, vol. 14, No. 1, pp. 79-98.
- Razouk, BB e Estrin, G.** [1980], "Modelagem e verificação de comunicação protocolos em SARA: a interface X.21," *IEEE Trans. em computadores*, vol. C-29, No. 12, pp. 1038-1052.
- Reid, JD** [1886], *Telegraph in America*, John Polhemus Publ., New York, 881 pgs.
- Reif, JH e Smolka, SA** [1988], "A complexidade de alcançabilidade em distribuído processos de comunicação," *Acta Informatica*, Vol. 25, pp. 333-354.
- Richier, JL, Rodriguez, C., Sifakis, J., e Viron, J.**, [1987], "Verification in Xesar do protocolo de janela deslizante," *Proc. 7º Workshop sobre Protocolo Specification, Testing and Verification*, Zurich, North-Holland Publ., Amsterdam, pp. 235-250.
- Ritchie, GR e Scheffler, PE** [1982], "Projecting the error performance of the Rede digital do sistema Bell," *IEEE 1982 Int. Conferência de comunicações*, ICC, Vol. 1, pp. 2D2.1-6

Página 375

- Rockstrom, A., e Saracco, R.** [1982], "SDL - CCITT Specification and Linguagem de descrição," *IEEE Trans. on Communications*, vol. COM-30, nº 6, pp. 1310-1318.
- Rolt, LTC** [1976], *Red for Danger, a History of Railway Accidents and Railway Segurança*, David & Charles, Londres.
- Rubin, J. e West, CH** [1982], "Uma técnica de validação de protocolo aprimorada," *Redes de Computadores*, vol. 6, Nr. 2, pp. 65-74.
- Rudie, K. e Wonham, WM** [1990], *Proc. 10º IFIP WG 6.1 Int. Workshop em Especificação de protocolo*, North-Holland Publ., Amsterdam.
- SDL** [1987], "Edição especial sobre a linguagem CCITT SDL," *Redes de Computadores e ISDN Systems*, Vol. 13, No. 2, pp. 65-134.
- Sabnani, K., e Dahbura, A.** [1985], "Uma nova técnica para gerar protocolo testes," *ACM Computer Communication Review*, Vol. 15, No. 4, setembro de 1985.
- Sabnani, K. e Dahbura, A.** [1988], "A protocol test generation procedure," *Redes de computadores e sistemas ISDN*, vol. 15, pp. 285-297.
- Saracco, R., Smith, JRW e Reed, R.** [1989], *Telecommunications Systems Engenharia usando SDL*, North-Holland Publ., Amsterdam, 633 pgs, ISBN 0 444 88084 4.
- Saracco, R., and Tilanus, PAJ** [1987], "CCITT SDL: visão geral da linguagem e suas aplicações," *Computer Networks and ISDN Systems*, Vol. 13, nº 2, pp. 65-74
- Sarikaya, B.** [1984], *Test Design for Computer Network Protocols*, Ph.D. Tese, Universidade McGill, Montreal, Canadá.
- Scantlebury, RA, Bartlett, KA** [1967], "Um protocolo para uso nos dados NPL rede de comunicações," Memorando Técnico, Laboratório Físico Nacional.
- Schneider, A., e Mase, A.** [1968], *Railway Accidents of Great Britain e Europe, their Causes and Consequences*, David & Charles, Londres.
- Schreiner, AT e Friedman, HG, Jr.** [1985], *Introdução ao Compilador Construction with UNIX*, Prentice Hall, Englewood Cliffs, NJ, 194 pgs. ISBN 0-13-

474396-2.

- Schwabe, D.** [1981], "Especificação formal e verificação de uma conexão protocolo de estabelecimento," *Proc. Seventh Data Comm. Symp.*, Cidade do México, (IEEE), pp. 11-26.
- Shannon, CE** [1948], "Uma teoria matemática da comunicação", *Bell System Technical Journal*, vol. 27, pp. 379-423, 623-656.
- Shannon, CE e McCarthy, J.** (eds.) [1956], *Automata Studies*, Princeton University Press, Princeton, NJ
- Shaw, RB** [1978], *A History of Railroad Accidents, Safety Precautions and Operating Practices*, Northern Press, Postdam, Nova York.
- Shen, YN e Lombardi, F.** [1989], "Teste de conformidade de protocolo usando vários Seqüências UIO," *Proc. 9º Workshop sobre Especificação de Protocolo, Teste e Verification*, Twente, The Netherlands, North-Holland Publ., Amsterdam.
- Shih, T., e Sidhu, D.** [1986], *Uma técnica para gerar sequências de teste para protocolos*, Relatório TR 86-23, Iowa State University.

Página 376

- Sidhu, D., e Leung, T.** [1989], "Formal methods for protocol testing: a detalhado estudo," *IEEE Trans. em Engenharia de Software*, vol. 15, No. 4, pp. 413-426.
- Sidhu, D.** [1990], "Teste de protocolo, os primeiros dez anos, os próximos dez anos," *Proc. 10º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo*, North-Holland Publ., Amsterdam.
- Slepian, D. (ed.)** [1973], *Key papers in the development of information theory*, IEEE Press, Nova York.
- Snepscheut, JLA van den** [1985], *Trace Theory and VLSI Design*, Ph.D. Tese, Universidade de Tecnologia de Eindhoven, Holanda, também em: Notas de aula em Computer Science, Vol 200, Springer Verlag, New York.
- Stallings, W.** [1985], *Data and Computer Communications*, Macmillan, New York, ISBN 0-02-415440-7, 594 pgs. 2ª ed., 1988, 653 págs.
- Stallings, W., Mockapetris, P., McLeod S., e Michel, T.** [1988], *Handbook of Computer Communications Standards*, Macmillan, New York, London, ISBN 0-02-948072-8, 206 pgs.
- Stenning, NV** [1976], "Data transfer protocol", *Computer Networks*, Vol 1, pp. 99-110.
- Ainda assim, A.** [1946], *Comunicação através dos tempos; da linguagem de sinais para Television*, Murray Hill Books, New York, 201 pgs.
- Sunshine, CA e Smallberg, DA** [1982], *Automated Protocol Verification*, Relatório USC / ISI RR-83-110, outubro de 1982.
- Tanenbaum, AS** [1981], *Redes de Computadores*, Prentice Hall, Englewood Cliffs, NJ, 2ª ed., 1988.
- Tarjan, RE** [1983], *Data Structures and Network Algorithms*, Soc. de industrial and Applied Mathematics (SIAM), Filadélfia, Pa., 131 pgs.
- Tugal, D. e Tugal, O.** [1982], *Data Transmission, Analysis, Design and Applications*, McGraw-Hill, New York, ISBN 0-07-065427-1.
- Turing, AM** [1936], "Em números computáveis, com uma aplicação para o Entscheidungsproblem," *Proc. London Math. Soc.*, Ser. 2, vol. 42, pp. 230-265; correção Vol. 43, pp. 544-546.
- Ural, H., e Probert, RL** [1986], "Step-wise Validation of communication protocolos e serviços," *Computer Networks and ISDN Systems*, Vol. 11, nº 3, pp. 367-383.
- Uyar, MU e Dahbura, AT** [1986], "Optimal test sequence generation for protocolos: o Algoritmo do Carteiro Chinês aplicado a Q.931," *Proc. IEEE Global Conferência de Telecomunicações, 1986*, Houston, Texas, Vol. 1, pp. 3.1.1-5.
- Valmari, A.** [1990], "A teimoso ataque à explosão de estado," *Proc. 2º Workshop sobre Computer-Aided Verification*, RP Kurshan e EM Clarke (Eds.), Rutgers University, Springer Verlag, New York.
- Vasilevskii, MP** [1973], "Diagnóstico de falha de autômatos," *Kibernetika*, No. 4, pp. 98-108, (original em russo).
- Vissers, C. e Logrippo, L.** [1985] "A importância do conceito de serviço no

design de protocolos de comunicação de dados, " *Proc. 5º IFIP WG 6.1 Int. Workshop em Especificação, teste e verificação de protocolo* , North-Holland Publ., Amsterdã,

Página 377

pp. 3-17.

Vissers, C. [1990] " FDT's para sistemas distribuídos abertos, uma retrospectiva e um

visão prospectiva, " *Proc. 10º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo* ,

Testing and Verification , North-Holland Publ., Amsterdam.

Wang, B., e Hutchinson, D. [1987], " Protocol test tests, " *Computer Communications* , Vol 10, No. 2, abril de 1987, pp. 79-87.

West, CH e Zafiropulo, P. [1978], " Validação automatizada de um protocolo de comunicações: a recomendação CCITT X.21, " *IBM J. Res. Desenvolve.* , Vol. 22, No. 1, pp. 60-71.

West, CH [1978], " Técnica geral para validação de protocolo de comunicações, " *IBM J. Res. Desenvolve.* , Vol. 22, No. 3, pp. 393-404.

West, CH [1982], " Aplicações e limitações da validação de protocolo automatizado, " *Proc. 2º IFIP WG 6.1 Int. Workshop sobre especificação de protocolo, teste e Verification* , North-Holland Publ., Amsterdam, pp. 361-372.

West, CH [1986a], " A validação do protocolo da camada de sessão OSI, " *Computer Networks and ISDN Systems* , Vol 11, No. 3, pp. 173-183.

West, CH [1986b], " Protocolo de validação por exploração aleatória de estado, " *Proc. 6º IFIP WG 6.1 Int. Workshop sobre especificação, teste e verificação de protocolo* , North-Holland Publ., Amsterdam, pp. 233-242.

West, CH [1989], " Protocol validation in complex systems, " *Proc. 8º ACM Symposium on Principles of Distributed Computing* , Austin, Texas, agosto de 1989.

Wirth, N. [1971], " Program development by stepwise refinement, " *Comm. do ACM* , vol. 14, No. 4, pp. 221-227.

Wirth, N. [1974], " Sobre a composição de programas bem estruturados, " *Computação Surveys* , vol. 6, No. 4, pp. 247-259.

Wolper, P. [1981], " Lógica temporal agora pode ser mais expressiva, " *Proc. 22º Simpósio IEEE sobre Fundamentos de Ciência da Computação* , pp. 340-348.

Wolper, P. [1986], " Especificando propriedades interessantes de programas em proposições lógica temporal, " *Proc. 13º Simpósio ACM sobre Princípios de Programação Languages* , St. Petersburg Beach, Fla., Janeiro de 1986, pp. 148-193.

Yannakakis, M., e Lee, D. [1990], " Testing finite state machines, " AT&T Bell Laboratórios, memorando interno.

Zafiropulo, P. [1978], " Validação de protocolo por análise de matriz duólogo, " *IEEE Trans. on Communications* , vol. COM-26, No. 8, pp. 1187-1194.

Zafiropulo, P., West, CH, Rudin, H., Cowan, DD e Brand, D. [1980], " Para analisar e sintetizar protocolos, " *IEEE Trans. em Comunicações* , Vol. COM-28, No. 4, pp. 651-661.

Zhang, L. [1986], " Por que os temporizadores TCP não funcionam bem, " *Proc. ACM SIGCOMM '86* , pp. 397-405.

Página 378

TRANSMISSÃO DE DADOS A

Para fins de projeto de protocolo, pode ser suficiente modelar um canal físico como um caixa preta com apenas um recurso interessante: pode distorcer os dados que passam isto. Entenderemos um *canal* como qualquer meio capaz de transferir sinais do remetente ao destinatário. A realização física do canal pode ser qualquer coisa desde um par trançado de fios de cobre para um link de satélite. A única coisa em que estamos interessados é o comportamento do canal na medida em que pode modificar os sinais que transfere.

Confável

Dados

Distorção

Caixa

Não confiável

Dados

Figura A.1 - Comportamento do canal

A distorção que é introduzida pelo canal é normalmente definida por uma distribuição de erro função de função com características conhecidas. Para um determinado meio, a probabilidade média biliidade dos erros de bit pode ser consultada em uma tabela (ver Tabela A.1 no final deste Appendix). Com essas informações, podemos conceber um esquema de detecção de erros que codifica o dados confiáveis de tal forma que sua integridade possa ser verificada depois de passar por a caixa de distorção (Capítulo 3). Esse esquema de detecção de erros intercepta a maioria dos distorções, mas é transparente para dados não distorcidos. Isso transforma a *caixa de distorção* em uma *caixa de exclusão*.

Erros de exclusão podem ser tratados diretamente em um protocolo de controle de fluxo que mensagens de números (Capítulo 4). Porque os esquemas de controle de erros são baseados em um estimativa da *taxa média* de erro de bit, há sempre alguma probabilidade de que distorceu os dados não são interceptados. O objetivo do controle de erros é fazer com que a probabilidade de esses eventos são aceitavelmente pequenos (Capítulo 3).

Para obter uma melhor compreensão da natureza dos erros de transmissão, no entanto, tomamos uma veja a caixa de distorção neste apêndice. Veremos como o comportamento do canal físico é influenciado por fatores como

O método de codificação de dados

A qualidade do canal (largura de banda, nível de ruído)

As dimensões físicas do canal

367

Página 379

A velocidade de sinalização

Com esse pano de fundo, será mais fácil fazer a avaliação certa sobre o protocolo requisitos para diferentes tipos de canais. Por exemplo, seria uma loucura absoluta desenvolver um protocolo elaborado com controle de erro direto (Capítulo 3) em um link de fibra óptica. Da mesma forma, não seria sensato tentar enviar dados a 100 Mbps em um cabo de par trançado, independentemente do esquema de controle de erro usado.

TIPOS DE CANAIS

Na prática, são usados três tipos diferentes de canais de transmissão de dados. Um *simplex* canal só pode ser usado para transferência de dados em uma direção. O remetente normalmente tem um "modulador" para traduzir dados binários em sinais analógicos, e o receptor tem um "demodulador" para a tradução reversa. Um canal *duplex* ou *full-duplex* pode transferir informações em ambas as direções simultaneamente. Cada estação tem um "modulador" e um "demodulador" combinados em um único instrumento chamado de "modem." Um canal *half-duplex*, finalmente, pode transferir dados em ambas as direções, mas não simultaneamente. As estações devem ser trocadas de envio para recebimento ou costas. A mudança geralmente leva cerca de 200 mseg.

SERIAL E PARALELO

Dependendo do hardware disponível, os bits de dados brutos podem ser transmitidos em um phy-canal sical com vários bits em paralelo ou um bit por vez em série.

A transmissão paralela normalmente é usada apenas em distâncias curtas, por exemplo, de uma máquina para um periférico. Em transmissões paralelas, uma linha extra é usada para carregar um relógio especial ou "strobe" sinal que irá indicar quando precisamente os sinais nas outras linhas consti-tute uma palavra de dados válida. Devido às variações nos atrasos de propagação, e a faixa de possíveis distorções, torna-se cada vez mais difícil em linhas mais longas sincronizar o sinal estroboscópico e os vários fluxos de bits. Para longas distâncias, a transmissão serial é portanto, mais comum.

ASSÍNCRONO E SÍNCRONO

Em uma linha serial, tanto o remetente quanto o receptor têm um relógio separado que define o taxa de transmissão. O remetente usa seu relógio para *conduzir* a linha (ou seja, para transmitir os bits), e o receptor usa seu relógio para fazer a *varredura*. Em transmissões assíncronas, os dois os relógios não precisam estar em perfeita sincronia quando nenhum dado é transmitido. Dados são transmitido em blocos de, por exemplo, 7 ou 8 bits, precedido por um símbolo especial de *início* e seguido por um símbolo de *parada*. O receptor usa o símbolo de início para sincronizar seu relógio com o remetente.

É suficiente se os dois relógios podem permanecer em sincronia apenas para os 7 ou 8 bits que inventar uma palavra de dados. O comprimento da palavra de dados às vezes é chamado de "syn-

intervalo de cronização, " o período de tempo que os dois relógios devem permanecer em sincronia. O símbolo de parada geralmente tem 1,5 ou 2 bits de comprimento, para permitir que o receptor processe o dados e alcançar o remetente e restaurar a sincronia no próximo símbolo de início. O período de tempo que passa entre o símbolo de parada e o próximo símbolo de início, entretanto, não precisa ser um número inteiro de vezes de bit.

Página 380

0
começar
Pare
I
0 1 2 3 4 5 6 P

Figura A.2 - Transmissão Assíncrona

A Figura A.2 mostra a transmissão assíncrona de um caractere ASCII de 8 bits: 7 bits de dados seguidos por um bit de paridade, denominado *P* (consulte o Capítulo 3). O estado ocioso da linha é indicada por uma alta tensão, lógica. O único símbolo é às vezes chamado de *marca* e o símbolo zero um *espaço*.

O método de transmissão assíncrona é auto-estabilizante, mesmo quando o receptor inicia erroneamente seu relógio em um bit de dados em vez do símbolo de início. O número de bits de dados digitalizados sairão errados, produzindo um "erro de enquadramento". O início assumido de uma palavra de dados só pode avançar no tempo, mais cedo ou mais tarde, o receptor será sincronizado novamente.

Na transmissão síncrona, o relógio do emissor e do receptor deve permanecer em sincronia em todas as vezes. Quando nenhum dado é transmitido, os dois relógios podem ser mantidos sincronizados com caracteres especiais " SYNC ".

Os dados também podem ser codificados de forma que o sinal sempre tenha um número suficiente de transições para manter o relógio do receptor sincronizado com o do remetente. Com isso, os bits são codificados nas *transições* de um sinal binário, ao invés de em absoluto níveis de sinal de alaúde. O método mais conhecido desse tipo é a codificação Manchester. Um símbolo é codificado no código Manchester por uma transição para baixo (um para zero) e um zero é codificado por uma transição ascendente (zero para um). Este método usa dois Baud (elementos de sinal) para codificar um bit de informação. A Figura A.3 ilustra isso

processo.

Sinal
I
I
0
0
I
0

Dados

Figura A.3 - Codificação Manchester

O código Manchester é chamado de " código de auto-clock. " O relógio do receptor pode sincronizar-se na transição que é garantida ocorrer no meio de cada símbolo.

O código Manchester tem outra propriedade importante: ele cria um sinal " equilibrado ".

O valor médio do sinal ao longo do tempo se aproxima de zero, mesmo se um contínuo

seqüência de bits iguais é transmitida. A distorção de um sinal balanceado no phy-

O link de dados físico é geralmente menor do que um sinal não balanceado. O elétrico propriedades da mídia, como um par trançado ou um cabo coaxial são relativamente desfavoráveis para sinais DC (corrente contínua), mas mais favorável para AC (corrente alternada), ou sinais equilibrados.

Experiência mostrou que a velocidade máxima de sinalização em um cabo de par trançado pode

Página 381

ser aumentado por um fator de *dez* se um código não balanceado for substituído por um balanceado.
VELOCIDADE DE SINALIZAÇÃO

Os sinais são normalmente transmitidos em canais como sequências de elementos de sinal de alguns duração fixa t . Cada elemento de sinal pode ter um valor finito escolhido de V distintos níveis de sinal. Quando $V = 2$, o sinal é chamado de sinal *binário*. A duração de cada

sinal determina a velocidade de sinalização. Esta velocidade é expressa na unidade *Baud*², que é definido como o número de elementos de sinal que podem ser transmitidos por segundo. A velocidade de sinalização de um canal, no entanto, é medida mais apropriadamente pelo taxa na qual "informações" podem ser transferidas. Um *bit*³ é a menor unidade de informação mação. Ele tem um de dois valores possíveis. Se um nível de sinal for usado para codificar um símbolo, V níveis de sinal discreto trivialmente permitem a codificação de $\log_2 V$ bits de informação por elemento de sinal, então

$$1 \text{ Baud} = \log_2 V \text{ bits por segundo (bps)}$$

Para sinais binários, a velocidade de sinalização em Baud, portanto, sempre é igual à sinalização velocidade em bps. Observe, no entanto, que no código Manchester uma sequência de dois sinais níveis é usado para codificar um único símbolo. Para os códigos de Manchester, portanto, 2 Baud = 1 bps.

É compreensível que essas unidades sejam facilmente confundidas. Observe cuidadosamente o que a diferença é entre uma velocidade de sinalização de, por exemplo, 1200 Baud, 1200 bps e 1200 car / s.

PROPAGAÇÃO DE SINAL

As informações podem ser transferidas por muitas portadoras de sinal diferentes, variando de fios de cobre, cabos coaxiais e fibras ópticas para links de satélite. Cada canal tem um comportamento característico e requer uma codificação específica da informação em elétrica ou sinais eletromagnéticos. Teoricamente, o tempo de propagação do sinal em cada canal nel irá definir um limite superior para a velocidade de sinalização máxima obtida. Na prática, nós verá que outros fatores, como "ruído" e limitações de largura de banda, têm uma maior efeito limitante. Para ondas eletromagnéticas, por exemplo, links de satélite e fibras ópticas, o tempo de propagação do sinal é de aproximadamente 3,10⁸ metros / seg. Para sinais elétricos em cabos, é cerca de um fator de dez a menos.

Considere, na Figura A.4, a projeção p de um ponto imaginário que se move ao redor do círculo. À direita é mostrado como a projeção no eixo y muda com o tempo quando o ponto se move com velocidade constante: uma curva *senoidal* perfeita. Um completo

-
1. Uma exceção notável é o código Morse. Os sinais familiares de ponto e traço têm comprimentos desiguais.
 2. A palavra "Baud" homenageia o operador telegráfico francês Emil Baudot, que inventou um código de cinco bits para transmissões telegráficas em 1874.
 3. O termo "bit" foi cunhado por JW Tukey da AT&T Bell Laboratories como uma abreviação para 'dígito binário' Shannon [1948].



Figura A.4 - Curva Senoidal

a travessia do círculo produz um " período " ou " ciclo " do seno. O máximo A " amplitude " da curva é igual ao raio do círculo a . Se a curva for interpretada como um sinal elétrico, a velocidade do ponto determina a " frequência " do sinal. a unidade para medir a frequência é Hertz (Hz). Um Hertz é igual a um ciclo por segundo. A Figura A.4 também mostra uma curva pontilhada que corresponderia à projeção de um segundo ponto que seguiria o primeiro a uma distância fixa, dada pelo ângulo ϕ . O ângulo é chamado de " mudança de fase " entre o primeiro e o segundo sinal. Obviamente geral, a mudança de fase máxima será uma travessia completa do círculo ou 2π radianos. Formalmente, uma curva seno é descrita por

$$y = a \sin(2\pi ft + \phi)$$

onde a é a amplitude, f a frequência, t o tempo e ϕ a mudança de fase.

A curva senoidal tem duas propriedades que a tornam atraente para os teóricos: é contínua e é periódico. O sinal na Figura A.5, por exemplo, não é nenhum dos dois, mas parece ser uma representação mais provável de um fluxo de bits binários.

uma
 t

Figura A.5 - Sinal Discreto, Não Periódico

SÉRIES DE FOURIER

Felizmente, quando estudamos as características dos canais de transmissão, não tem que considerar todas as formas de onda possíveis, como a complicada na Figura A.5. Nós pode obter uma aproximação muito boa considerando apenas ondas senoidais. Vamos considerar um sinal periódico arbitrário como o da Figura A.5. Existem dois problemas com este sinal: não é periódico e não é contínuo. O primeiro problema é fácil

Página 383

para corrigir, pelo menos para fins de modelagem. Se quisermos descrever este fragmento do sinal de forma elegante, podemos modelá-lo como parte de um sinal mais longo e periódico que é obtido por repetir o fragmento de sinal infinitamente frequentemente. O segundo problema não é problema: a onda quadrada descontínua ideal é apenas uma abstração. Na prática, qualquer mudança nos níveis de sinal levam um tempo diferente de zero e não existe nenhuma descontinuidade. Fourier descobriu que cada sinal periódico contínuo pode ser descrito por uma soma de ondas senoidais simples, cada uma com uma frequência que é um múltiplo inteiro de uma " base frequência " f .

$n = 1$

(C)

$$a_n \sin(2\pi f_n t + \phi_n)$$

Nesta fórmula, a_n é um coeficiente que determina a amplitude da n -ésima frequência componente e ϕ_n é o deslocamento de fase correspondente. Para sinais aperiódicos, o discreto série de componentes de frequência muda para um continuum de frequências, mas em princípio o mesmo tipo de análise pode ser executado.

A Figura A.6 dá um exemplo da aproximação de uma onda quadrada discreta pelo soma de dois componentes do seno.

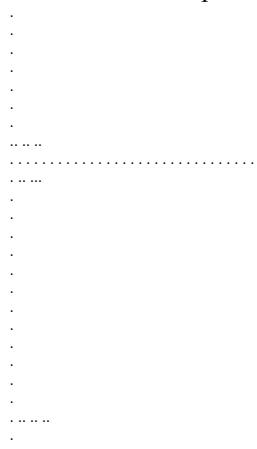


Figura A.6 - Série de Fourier

Quanto mais componentes sinusoidais adicionarmos, melhor será a aproximação. O sinal composto é novamente construído a partir de uma frequência base e uma gama de " harmônicos, " de que usamos apenas o primeiro. O composto completo é definido por

$$n=0$$

(C)

$$\frac{1}{\text{BANDWIDTH}} \sin ((2n+1) 2\pi ft)$$

Se definirmos a frequência do sinal ao longo do eixo x e amplitude ao longo do eixo y, podemos descrever este sinal no " domínio da frequência " conforme mostrado na Figura A.7.

Página 384

uma

1

2

3

4

5

f

Figura A.7 - Domínio de Freqüência

Se aumentarmos a velocidade de sinalização, a frequência base e todos os seus harmônicos também irão aumentar. Infelizmente, um canal de transmissão real só pode transferir um alcance limitado de frequências de sinal. Uma linha telefônica de voz, por exemplo, só pode transferir sinais entre 300 Hz e 3400 Hz. Se aumentarmos a velocidade de sinalização, quanto maior componentes de frequência podem cair fora da banda de sinalização e desaparecer do sinal transmitido. Se diminuirmos a velocidade de sinalização, o mesmo pode acontecer com o componentes de frequência mais baixa, tendo um efeito ainda mais prejudicial no sinal qualidade.

A " largura de banda " do canal determina sua qualidade. A largura de banda é definida como diferença entre a frequência mais alta e a mais baixa que o canal pode confiavelmente transferir. Quanto maior a largura de banda, mais informações o canal pode transportar.

300

3400

f

uma

Figura A.8 - Frequências de corte

Em geral, se transmitirmos um sinal composto por um canal de largura de banda limitada, alguns componentes de frequência serão atenuados mais do que outros, e alguns serão perdidos completamente. O resultado será um sinal distorcido. Se tentarmos transmitir o sinal binário final da Figura A.5 diretamente como um sinal elétrico, o sinal distorcido que chegará no receptor pode muito bem se parecer com a linha pontilhada na Figura A.6.

A Figura A.8 mostra a largura de banda de uma linha telefônica comutada padrão. Sem sinal

com uma frequência inferior a 300 Hz passará por ele, e nenhum sinal com uma frequência superior a 3400 Hz. A largura de banda é de 3,1 kHz. Para transferir um sinal binário arbitrário final através de um canal de telefone, deve ser traduzido em frequências que passam o canalizar sem esforço. Este processo, denominado *modulação*, é discutido a seguir. Por enquanto deve ser notado que todo meio de transmissão físico tem uma largura de banda finita, e, consequentemente, distorce os sinais transmitidos nele. Um par de fios comum tem um largura de banda de aproximadamente 250 kHz (consulte a Tabela A.1). A frequência de corte é aproximadamente em 200 kHz, com a atenuação dos sinais de maior frequência aumentando exponencialmente. Para cabos coaxiais, a alta frequência de corte é cerca de uma ordem de magnitude maior.

Página 385

A distorção aumentará com a velocidade de sinalização, simplesmente porque os dados superiores taxas causam frequências de sinal mais altas.

Uma sequência de sinais binários irá se deteriorar de uma onda quadrada limpa para um forma de onda suave na qual os bits individuais podem ser difíceis de reconhecer. O pontilhado linha na Figura A.9 mostra o "nível de decisão" abaixo do qual um sinal é classificado como um zero. A precisão do receptor é severamente testada pela distorção do sinal. Um pequeno quantidade de ruído pode causar imediatamente erros de classificação no receptor. Observe também que a presença dos dois sinais um em torno do sinal zero isolado na Figura A.9, contribui para a distorção do zero. Esta "interferência entre símbolos" piora conforme a velocidade de sinalização aumenta, e os "símbolos" estão mais próximos espaçado.

Receptor
Remetente

.

.

.

.

.

.

Figura A.9 - Sinal distorcido

MODULAÇÃO

A modulação é usada para adaptar os sinais às características de um canal. Em um telefone linha, por exemplo, podemos transmitir um binário como uma frequência (um seno) de 1270 Hz, e um zero como 1070 Hz. Para fazer um canal full-duplex, podemos escolher 2225 Hz e 2025 Hz para a transmissão de um e zero, respectivamente, no canal de retorno⁴.

Todas essas frequências estão dentro da faixa que é transmitida com pouco ou nenhum sinal atenuação em uma linha telefônica (Figura A.8), a fim de evitar alguns dos efeitos do harmonicas monicas na qualidade do sinal.

Este método de modulação é conhecido como *chaveamento de mudança de frequência*, ou também simplesmente como *frequency modulation*. Como observamos anteriormente, nem muitos canais podem transmitir sinais DC

nals convenientemente. Um sinal balanceado, ou AC, pode sobreviver aos danos causados pelo canal muito melhor. Se tomarmos uma onda senoidal padrão como um sinal portador básico para transmitir os dados, existem três maneiras diferentes em que podemos alterar (modular)

4. Essas são, de fato, as frequências usadas em um modem 300 Baud Bell 108.

Página 386

essa operadora para codificar as informações. Podemos usar o sinal de dados para variar a Amplitude

Frequência

Ângulo de fase

A modulação de amplitude para um sinal binário seria alcançada se escolhêssemos dois amplitudes representativas, por exemplo, 5 Volts e 10 Volts, para codificar dados binários. O fre-

frequência transmitida é constante e pode ser escolhida no meio da banda de frequência freqüências que são aceitas pelo canal. Qualquer ruído no canal, no entanto, é adicionado ao sinal conforme transmitido e pode causar erros de bit. Atenuação de sinal, especialmente variações nas atenuações principalmente dependentes do tempo podem causar erros extras. A modulação de freqüência é mais robusta contra ruído e atenuação direta do sinal. Mas agora, atrasos de propagação dependentes da freqüência e interferências sutis de freqüência padrões de presença causados por eco e diafonia podem causar problemas (veja abaixo). Por usando várias freqüências, no entanto, é fácil aumentar a velocidade de sinalização em bits por segundo, para uma determinada taxa de transmissão.

□

□

□

□

11

01

00

10

Figura A.10 - Modulação de amplitude em quadratura

O terceiro método, usando chaveamento de mudança de fase, ou modulação de fase, é o mais complicado um dos três. Cada elemento de sinal agora é codificado por uma mudança de fase do elemento de sinal anterior. Nessas técnicas de *modulação de amplitude em quadratura* (Figure A.10) uma combinação de modulação de amplitude e fase é usada. Uma versão simples disso usa quatro mudanças de fase diferentes: em incrementos de 90° : 45° , 135° , 225° e 315° . Uma vez que esta é uma das quatro opções, cada novo símbolo agora codifica dois bits de informação, e a taxa de dados em bits por segundo será o dobro da taxa de dados medida em Baud.

DISTORÇÃO

Um sinal transmitido em um canal de largura de banda limitada incorre em um dependente de freqüência atenuação. Este tipo de distorção de sinal é uma distorção linear. Pode ser medido e pode, até certo ponto, ser compensado com filtros especiais que achatam a resposta curva no domínio da freqüência.

O tempo de propagação do sinal pode ser diferente para cada componente de freqüência em um comp. sinal positivo. Isso causa uma mudança de fase não intencional entre os harmônicos: quanto mais alto as freqüências geralmente viajam mais rápido do que as outras. Para um determinado canal, esta fase

Página 387

a distorção também pode ser corrigida com filtros especiais.

Os canais de transmissão também podem adicionar novas formas de onda de várias freqüências a um sinal. Essas distorções não lineares podem ser completamente alheias ao sinal original e são muito mais difíceis de combater.

Os ecos de sinal são um exemplo de distorções não lineares. Onde quer que haja um súbito mudança de impedância no canal, por exemplo, nos terminais, o sinal pode retornar na linha e viajar na direção oposta, distorcendo o sinal original. UMA tipo semelhante de distorção não linear é causado por diafonia. A distorção vem de outros canais que estão fisicamente próximos o suficiente para causar sinais de sombra por eletricidade magnética. Em sinais modulados, o mesmo tipo de problema pode ocorrer como ruído de intermodulação.

Causas ainda mais drásticas de erro são picos e faíscas elétricas: curtas, poderosas e descargas elétricas imprevisíveis. Eles podem ser causados por interruptores, motores ou simplesmente por descargas espontâneas na atmosfera. Eles são difíceis de evitar, exceto por isolamento completo.

TEOREMA DE AMOSTRAGEM DE NYQUIST

A relação entre a velocidade de sinalização e largura de banda foi estudada pela primeira vez por H. Nyquist em 1924. Ele mostrou que se as amostras são retiradas de um sinal arbitrário que é transmitido em um canal com largura de banda B , o sinal original pode ser completamente reconstruído se pelo menos $2B$ amostras por segundo forem coletadas. Este *teorema de amostragem* pode ser usado para determinar a velocidade máxima de sinalização. $2B$ amostras por segundo definem ao máximo

$2B$ elementos de sinal diferentes. A velocidade máxima de sinalização em um canal com uma largura de banda de B Hz é então

$$2B \log_2 V \text{ bps}$$

De acordo com esta estimativa, a velocidade de sinalização pode ser aumentada arbitrariamente por um número de valores de sinal V . Abaixo veremos que há ainda outro fator que limita a velocidade de sinalização: ruído.

BARULHO

O ruído é uma causa fundamental e inevitável de distorção do sinal. O ruído térmico é causado por flutuações térmicas de elétrons em condutores. Não tem preferência por nenhuma frequência particular: está igualmente presente em todos. Portanto, às vezes é referido como *ruído branco*. Uma medida importante para a qualidade de um sinal é a relação sinal-ruído Razão.

A força, ou potência, de um sinal é expressa em watts (energia por segundo). Sinal as razões são mais convenientemente definidas em *decibéis*. Se P_1 e P_2 dão a potência de dois sinais em watts, então

$$10 \log_{10} \frac{P_2}{P_1} \text{ dB}$$

Página 388

é sua proporção em decibéis. Os decibéis são usados, por exemplo, para expressar a atenuação do sinal em um canal. Se R_1 é a atenuação do sinal em um canal em dB, e R_2 é o atenuação em outro canal, a perda combinada se ambos os canais forem usados em série será simplesmente $R_1 + R_2$.

LIMITE DE SHANNON-HARTLEY

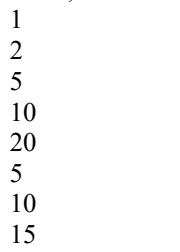
Em 1948, Claude E. Shannon estudou o efeito preciso da relação sinal-ruído sobre transmissão de dados. Ele mostrou, por exemplo, que a velocidade máxima de sinalização em um canal com largura de banda B e relação sinal-ruído S/N , com S e N em watts, é

$$C = B \log_2 \left(1 + \frac{S}{N} \right) \text{ bps}$$

Esse resultado é conhecido como limite de Shannon-Hartley. C é chamado de capacidade do canal.

Para uma linha telefônica, temos $B = 3100$ Hz e uma relação sinal-ruído de 30 dB (1000: 1), dando uma velocidade máxima de sinalização de 30 Kbit / seg. Acima deste limite está em geral, não é possível distinguir o sinal transmitido do ruído de fundo: o conteúdo de informação do sinal é muito baixo.

A Figura A.11 mostra os valores que podem ser calculados a partir do limite de Shannon-Hartley para uma linha telefônica, para relações sinal-ruído de 1 a 30 dB. A linha pontilhada mostra a assíntota $B \log_2 (S/N)$. As velocidades de sinalização acima da linha desenhada não podem ser realizadas, nem mesmo com a codificação de dados mais inteligente que se possa imaginar.



Não é viável

Factível

Kbit / s

Relação Sinal-Ruído (dB)

Figura A.11 - Limite de Shannon-Hartley

Para sinais binários, a taxa de Nyquist de $2B$ bps (cerca de 6 Kbit / s) pode ser alcançada, teoricamente, para uma relação sinal-ruído de apenas 2,5 dB.

Até mesmo para nos aproximarmos do limite de Shannon-Hartley, devemos fazer um uso ótimo da estatística

informações sobre os dados a serem transmitidos. A transferência do texto em inglês, para

Página 389

exemplo, pode ser otimizado tomando a frequência de ocorrência de certas letras e combinações de letras em consideração, atribuindo o código mais curto ao mais frequente uns.

A taxa de Nyquist para um canal de largura de banda limitada era $2B$, ou, para $B = 3100 \text{ Hz}$, 6200 Baud. Isso significa que para realizar uma velocidade de sinalização de 30 Kbit / s, devemos também usar 32

níveis de sinal diferentes: não pode ser realizado com um sinal binário.

Na prática, as velocidades de sinalização usadas são muito mais baixas do que as de Nyquist e o limite de Shannon-Hartley. Uma razão é que todas as outras causas de distorção (eco, cross-talk, distorções não lineares e assim por diante) não são levados em consideração nestes resultados. Além disso, nem sempre vale a pena ou é possível incluir muito elaborado esquemas de codificação que podem realmente otimizar as taxas de transmissão. Na prática, a máxima velocidade de sinalização é de 1200 a 2400 Baud.

A maneira mais simples de obter uma velocidade de sinalização mais alta em um canal de largura de banda limitada

é, claro, aumentar a largura de banda. Isso é precisamente o que a companhia telefônica faz com os novos serviços de voz sobre dados (Co-Lan) em um telefone especialmente equipado linhas, oferecendo serviço de telefone normal e transferências de dados duplex simultâneas em sinalização acelera até 19,2 Kbit / seg.

VISÃO GLOBAL

Um sinal que é transmitido em um canal físico pode ser afetado por dois tipos principais de distorção:

A transformação do sinal original

A adição de informações não relacionadas ao sinal original

Exemplos do primeiro tipo de distorção são a atenuação dependente da frequência e o perda de componentes de sinal de alta e baixa frequência devido a limitações de largura de banda.

Exemplos do segundo tipo de distorção são ruído, ecos, diafonia e interferência padrões de referência causados por distorções de sinal não linear.

O efeito do primeiro tipo de distorção pode ser reduzido usando a codificação de dados adequada, modulação e técnicas de filtragem de sinal.

Dados típicos e taxas de erro para três tipos comuns de mídia física são fornecidos em

Tabela A.1.

Tabela A.1

Cabo coaxial de fibra óptica de par trançado

Taxa de dados em Mbps

10

100

1000

Taxa de erro de bit

10⁻⁵

10⁻⁶

10⁻⁷

10⁻⁸

10⁻⁹

Largura de banda

250 kHz

350 MHz

1 GHz

Observe, no entanto, que muitos outros fatores além da largura de banda afetam os dados e erros taxas: o método particular de codificação de dados usado, o comprimento da linha de dados e

Página 390

daí sua suscetibilidade a ruído, ecos, diafonia, distorções não lineares, etc.

cabo de par trançado, por exemplo, a taxa cotada de 10 Mbps é válida para um comprimento de linha acima

a cerca de 30 pés, para " transmissões equilibradas " (por exemplo, com uma codificação Manchester ing). A 300 pés, a taxa de dados cai para 1 Mbps; a 3000 pés, cai para 100 Kbit / s.

A transmissão a 1 Mbps em um cabo de par trançado de 3000 pés, portanto, requer sinal regeneradores (repetidores).

NOTAS BIBLIOGRÁFICAS

Um estudo detalhado das características da linha e da teoria da transmissão de dados é fornecido no Bennett

e Davey [1965]. Um excelente tutorial sobre modems, linhas de dados e padrões de protocolo dards é McNamara [1982]; um livro de referência prática bem recomendado. Um aplicativo tratamento orientado a catiões de técnicas de transmissão de dados é apresentado em Tugal e Tugal [1982]. Outros tratamentos sólidos de teoria e técnicas de transmissão de dados podem ser encontrado em Bertsekas e Gallager [1987], [Stallings '85] e, é claro, Tanenbaum [1981, 1988]. Uma introdução agradável a alguns dos detalhes da transmissão de dados pode também pode ser encontrado em Byte [1989].

Página 391

FLUXOGRAMA IDIOMA B

A linguagem do fluxograma usada na Parte I é baseada em um pequeno subconjunto do CCITT Especificação e linguagem de descrição SDL, CCITT [1988], Rockstrom e Saracco [1982], SDL [1987], Saracco, Smith e Reed [1989]. Existem alguns desvios que aproximar sua semântica daquela da linguagem PROMELA discutida nos Capítulos 5, 6 e Apêndice C.

Cada fluxograma independente define um processo que, pelo menos conceitualmente, é executado cortado simultaneamente com todos os outros processos definidos de forma semelhante. Cada fluxograma tem um

ponto de entrada que é rotulado com um nome de processo ou com o símbolo de *início*.

Como em um fluxograma tradicional, as ações de um processo são especificadas com símbolos de várias formas ligadas por arcos direcionados. Seis tipos diferentes de símbolos são usados, como ilustrado na Figura B.1.

Declaração

Teste

Esperar

interno

Entrada

Resultado

Figura B.1 - Símbolos do fluxograma

Esses símbolos representam:

Declarações, por exemplo, atribuições

Testes booleanos, por exemplo, expressões

Condições de espera, por exemplo, recebe

Eventos internos, por exemplo, tempos limite

Entradas e saídas de mensagens

Os testes booleanos são avaliados sem demora. As condições de espera, no entanto, são usadas para modelo de sincronização de processos. Eles especificam que o processo de execução não prossiga além desse ponto no programa, a menos que uma condição específica seja mantida. Os dois

Os elementos restantes do fluxograma, usados para conectar os símbolos da Figura B.1, são:

Arcos direcionados

Conectores

Isso nos dá um total de oito blocos de construção básicos para construir gráficos.

380

Página 392

Os arcos direcionados indicando o fluxo de controle só podem convergir em conectores, como ilustrado tratado na Figura B.2. Eles podem divergir, sem conectores, em condições de espera e em testes booleanos.

Figura B.2 - Conector e arcos

Cada processo de fluxograma tem associado a ele uma fila de mensagens implícita, teorias de capacidade infinita, que é usada para armazenar as mensagens recebidas. Mensagens são anexados às filas nas instruções de saída e eles são recuperados das filas em declarações de entrada. Os nomes das mensagens devem identificar exclusivamente o processo de recebimento. Nota

que um nome de mensagem sempre pode ser estendido com o nome de um processo para garantir esta.

Saídas, instruções, condições de espera, eventos internos e testes booleanos podem aparecer em qualquer lugar em um fluxograma. As entradas só podem seguir um símbolo de espera identificado como *recebimento*.

Mais de uma entrada pode aparecer.

receber
ack
tempo esgotado

Figura B.3 - Entradas e tempos limite

Uma condição de espera marcada como *recebimento* irá atrasar o processo de execução até que o fila de mensagens desse processo contém, em seu primeiro slot, uma mensagem de um tipo especificado em uma das entradas que seguem o símbolo de espera no fluxograma. É um erro de protocolo se a mensagem no primeiro slot da fila for de outro tipo.

Um tempo limite é uma condição de sincronização interna representada como um evento. A condição correspondente sempre se tornará verdadeira. Se um tempo limite evento é especificado após um símbolo de espera rotulado *recebimento*, o processo de execução pode abortar a espera por uma mensagem recebida e continuar com a execução do estado mentos após o tempo limite.

O símbolo de espera também pode ser rotulado com uma expressão. Neste caso, a execução o processo será atrasado até que a expressão, quando avaliada, produza o valor booleano *verdadeiro* (ou qualquer valor inteiro diferente de zero).

Página 393

Um teste booleano deve ser rotulado com uma expressão, mas neste caso a expressão é avaliado uma vez e o valor resultante é usado para selecionar um link de saída com o etiqueta correspondente. O processo não está atrasado. É um erro se a avaliação do expressão produz um valor para o qual não há rótulo correspondente em qualquer um dos arcos. O efeito de tal erro é indefinido.

próximo: a, b
msg: a, b
msg: a, b
aceitar: a, b
tempo esgotado

Figura B.4 - Eventos internos

Duas ações internas especiais que modelam o acesso ao arquivo são predefinidas: *avançar* e *aceitar*. A notação *a seguir: a, b* indica a recuperação interna dos itens de dados *a* e *b* de um base de dados interna. Da mesma forma, *aceitar: a, b* indica o armazenamento dos itens de dados em um base de dados interna. As duas ações *a seguir* e *aceitar* incluem todo o processamento em segundo plano que está associado à recuperação e armazenamento de itens de dados, respectivamente. Seus uso é ilustrado na Figura B.4.

O uso de variáveis e tipos de dados abstratos não é restrito pelo fluxograma língua. Da mesma forma, o conteúdo de uma caixa de declaração pode ser qualquer coisa que não envolvem condições de espera, recebimento ou envio de mensagens, timeouts e testes booleanos. Para obter exemplos, consulte os fluxogramas nos Capítulos 2 e 4.

Página 394

PROMELA LANGUAGE REPORT C

Este apêndice é um manual de referência para PROMELA, a linguagem para descrever protótipos de validação col introduzidos neste livro. Ele oferece uma visão geral resumida das principais requisitos de sintaxe da linguagem. A semântica e o uso são explicados de forma mais completa em Capítulos 5 e 6. Este manual descreve a linguagem adequada. Não cobre possíveis restrições ou extensões de implementações específicas. Em caso de dúvida, por exemplo quando você tem que descobrir qual é o efeito preciso de uma expressão como

`(-10) % (- 9) ou (-10) << (- 2)` em sua máquina, a maneira mais rápida de aprender é executar um pequeno programa de teste PROMELA , como

```
init {printf ("%d \t% d \n", (-10) % (- 9), (-10) << (- 2))}
```

usando o simulador PROMELA (Capítulo 12). O significado de tudo binário, aritmético, e os operadores relacionais correspondem ao do padrão ANSI C.

C.1 CONVENÇÕES LÉXICAS

Existem cinco classes de tokens: identificadores, palavras-chave, constantes, operadores e estado-separadores de ment. Espaços em branco, guias, novas linhas e comentários servem apenas para separar tokens.

Se mais de uma interpretação for possível, um token é considerado a string mais longa de caracteres que podem constituir um token.

C.2 COMENTÁRIOS

Qualquer string iniciada com `/ *` e terminada com `* /` é um comentário. Comentários não podem ser aninhado.

C.3 IDENTIFICADORES

Um identificador é uma única letra ou sublinhado, seguido por zero ou mais letras, dígitos, ou sublinhados.

C.4 PALAVRAS-CHAVE

Os seguintes identificadores são reservados para uso como palavras-chave:

```
afirmar
atômico
mordeu
bool
quebrar
byte
chan
Faz
383
```

Página 395

```
fi
vamos para
E se
iniciar
int
len
mtype
Nunca
od
do
printf
proctipo
corre
curto
pular
tempo esgotado
```

C.5 CONSTANTES

Existem três tipos de constantes.

Constantes de string

Constantes de enumeração

Constantes inteiras

Constantes de string só podem ser usadas em instruções `printf` .

As constantes de enumeração podem ser usadas para definir nomes simbólicos para tipos de mensagem.

Eles podem ser definidos em declarações `mtype` do tipo

```
mtype = {namelist}
```

onde `namelist` é uma lista separada por vírgulas de nomes simbólicos. Apenas um tipo `m` declaração por programa pode ser usada.

Uma constante inteira é uma sequência de dígitos que representa um inteiro decimal. tem nenhum número de ponto flutuante em PROMELA .

C.6 EXPRESSÕES

A avaliação das expressões é definida na aritmética de inteiros. Dados não assinados, isto é todas as variáveis declaradas com o tipo `bit` ,`byte` , ou `bool` , são convertidos em números inteiros assinados antes

sendo usado em expressões. Por exemplo, o valor da expressão `(p-1)` , com `p` a variável capaz de tipo `byte` (`unsigned char`) e valor zero, é o valor assinado `-1` na PROMELA , e não o equivalente não assinado `255`. Em atribuições, no entanto, o tipo de destino

nação sempre prevalece. O valor -1 é convertido para 255 quando é armazenado em um não assinado variável, mas permanece -1 quando armazenado em uma variável assinada.

Os seguintes operadores podem ser usados para construir expressões.

+, -, *, /, %,
operadores aritméticos
>, >=, <, <=, ==, !=,
operadores relacionais
&&, ||, !
lógico AND, OR, NOT
&, |, ~, >>, <<
Operadores de bit estilo C
!, ?
enviar e receber operadores
(), []
agrupamento, indexação
len, corra
operadores especiais

A sintaxe, a semântica, os efeitos colaterais e as dependências da máquina de todos os operadores correspondem

Padrão ANSI C. A Tabela C.1 define os níveis de precedência. Os operadores no primeiro linha na tabela tem a precedência mais alta.

A maioria dos operadores, incluindo atribuição =, leva dois operandos. A negação booleana ! e o menos unário – operador pode ser unário e binário, dependendo do contexto.

O operador de atribuição leva uma expressão à direita e uma referência de variável em a esquerda:

Página 396

Tabela C.1 - Precedência e associatividade

Operadores	Associatividade
() []	da esquerda para direita
~ - (menos unário) ! (negação booleana)	da esquerda para direita
* / %	da esquerda para direita
da esquerda para direita	+ -
da esquerda para direita	>> <<
da esquerda para direita	> <> = <=
da esquerda para direita	== ! =
da esquerda para direita	E
da esquerda para direita	
da esquerda para direita	&
da esquerda para direita	&&
da esquerda para direita	
da esquerda para direita	! (enviar) ? (receber)
da esquerda para direita	len run
da esquerda para direita	=
direita para esquerda	

Variáveis globais e variáveis locais declaradas dentro do mesmo tipo de processo podem ser referido pelo nome. Por exemplo

```
byte glob;
proctipo mesmo ()
{
    bool loc;
    aqui: (loc + glob)
}
```

Variáveis locais de outros processos podem ser referidas da seguinte forma:

```
tipo outro ()
{
    assert (same [2] .loc > 3) /* NB !: não é mais válido na versão atual */
}
```

Aqui, um processo do tipo `outro` se refere à variável local `loc` do processo com `pid` dois, ou seja, o segundo processo que foi instanciado. É um erro em tempo de execução se o tipo de esse processo é diferente do `mesmo` tipo especificado.

Página 397

O estado do processo de um processo remoto pode ser testado com expressões de dois pontos booleanos. Por exemplo, a condição

```
mesmo [2]: aqui
/* NB! a sintaxe atual é: mesma [2] @aqui */
é verdadeiro se e somente se o processo referido estiver atualmente no estado que foi rotulado
aqui. A referência remota de variáveis e estados de fluxo de controle deve ser usada
apenas em afirmações e em reivindicações temporais. A definição da linguagem, no entanto, não
impedir outras aplicações.
```

C.7 DECLARAÇÕES

Processos e variáveis devem ser declarados antes de serem usados. Variáveis podem ser declarado localmente, dentro de um tipo de processo ou globalmente. Um processo só pode ser declarado globalmente em uma declaração `proctype`. Declarações `Proctype` não podem ser aninhado. As declarações locais podem aparecer em qualquer lugar em um corpo de processo. O escopo de um variável local é o corpo do processo completo, independentemente de onde sua declaração é colocada. Não está acessível, entretanto, até que a execução tenha passado do ponto de declaração pelo menos uma vez. Existem seis tipos de dados:

```
bit, bool, byte, chan, short, int
```

VARIÁVEIS

Uma declaração de variável começa com uma palavra-chave indicando o tipo de dados da variável seguido por uma lista de nomes de identificadores, cada um seguido opcionalmente por um inicializador.

```
byte nome1, nome2 = 4, nome3;
chan qname; chan a = [3] de {byte};
```

O inicializador deve ser uma expressão para uma variável de um tipo de dados básico e um canal especificação para variáveis do tipo `chan`. Por padrão, variáveis de todos os tipos, exceto `chan` são inicializados em zero. Variáveis do tipo `chan` devem ser inicializadas explicitamente antes eles podem ser usados para troca de mensagens. É indefinido qual é o resultado de usar um variável de canal não inicializada. Provavelmente, isso causa um erro fatal de tempo de execução.

A Tabela C.2 resume a largura e os atributos dos cinco tipos de dados básicos.

Tabela C.2 - Tipos de dados básicos

Tamanho do nome (bits)

Uso

mordeu

1

não assinado

bool

1

não assinado

byte

8

não assinado

curto

16

assinado

int

Os nomes `bit` e `bool` são sinônimos para um único bit de informação. Um `byte` é um quantidade não assinado que pode armazenar um valor entre 0 e 255. Curto `s` e `int` s são quantidades assinadas que diferem apenas na faixa de valores que podem conter.

Página 398

Uma matriz de variáveis é declarada da seguinte maneira:

```
int name1 [N];  
chan q [M];
```

onde `N` e `M` são constantes. Uma declaração de array pode ter um inicializador, que inicializa todos os elementos da matriz. Se a matriz for um canal, um canal de mensagem do determinado tipo por elemento do array é criado. No inicializador de canal

```
chan q [M] = [x] de {tipos}
```

`M` é uma constante, `x` é uma expressão que especifica o tamanho do canal e `tipos` é uma lista separada por vírgulas de um ou mais tipos de dados que define o formato de cada mensagem sóbrio que pode ser passado pelo canal. Todos os canais são inicializados como vazios.

Identificadores de canal inicializados podem ser passados de um processo para outro nas mensagens ou em instruções `run`.

C.8 PROCESSOS E RECLAMAÇÕES TEMPORAIS

Uma declaração de processo começa com a palavra-chave `proctype` seguida por um nome, uma lista de parâmetros formais entre colchetes e uma sequência de declarações e declarações de variáveis. O corpo de uma declaração de processo está entre parênteses.

```
nome do tipo de proc /* declarações de parâmetro */  
{  
/* declarações e declarações */  
}
```

As declarações de parâmetro não podem ter inicializadores. Uma declaração de processo é exigido em todo modelo PROMELA : o processo inicial. É declarado sem a chave palavra `proctype` e sem uma lista de parâmetros.

```
iniciar  
/* declarações e declarações */  
}
```

É o primeiro processo em execução e tem `pid` zero.

Uma reivindicação temporal começa com a palavra-chave `nunca` e pode conter qualquer texto PROMELA

```
Nunca {  
/* declarações e declarações */  
}
```

Pode haver no máximo uma reivindicação temporal por modelo PROMELA . É usado para especificar um exigência de exatidão sobre as execuções do sistema especificado. O temporal reivindicação especifica um comportamento que é considerado impossível. A reclamação normalmente conter apenas condições, embora seja válido permitir que a reivindicação temporal contenha vários declarações capazes, sequências atômicas e instruções de envio e recebimento. Para violar um afirmação de exatidão, deve ser possível executar uma instrução ou um atômico sequência de declarações, para cada declaração executada por qualquer um dos outros processos no modelo. Usando a reivindicação temporal em combinação com rótulos de estado de aceitação, qualquer fórmula lógica temporal proposicional de tempo linear no

Página 399

o comportamento do sistema pode ser expresso (consulte o Capítulo 6).

C.9 DECLARAÇÕES

Existem doze tipos de afirmações:

```
afirmação  
tarefa  
atômico  
quebrar  
expressão  
vamos para  
printf  
receber  
seleção  
repetição  
enviar  
tempo esgotado
```

Qualquer declaração pode ser precedida por uma ou mais declarações. Uma declaração só pode ser passado se for executável. Para determinar sua executabilidade, a instrução pode ser avaliado: se a avaliação retornar um valor zero, a instrução é bloqueada. Em todos os outros casos, a instrução é executável e pode ser passada. A avaliação de um composto a expressão é sempre indivisível. Isso significa que a declaração

(*a* == *b* && *a* != *b*)

sempre será não executável, mas a sequência

(*a* == *b*); (*a* != *b*)

pode ser executável nessa ordem.

O ato de passar a declaração após uma avaliação bem-sucedida é chamado de "execução" da declaração. Existe uma *pseudo* instrução, `skip`, que é sintaticamente equivalente à condição (1). `skip`, é uma instrução nula; é sempre executável e não tem efeito quando executado. Pode ser necessário para atender aos requisitos de sintaxe.

As instruções Goto podem ser usadas para transferir o controle para qualquer instrução rotulada dentro do mesmo processo ou procedimento. Eles são sempre executáveis. Atribuições e declarações também são sempre executáveis. As expressões só são executáveis se retornarem um valor diferente de zero. Ou seja, a expressão 0 (zero) nunca é executável e, da mesma forma, 1 é sempre executável.

Cada declaração pode ser precedida por um rótulo: um nome seguido por dois pontos. Cada etiqueta pode ser usado como o destino de um `goto`. Três tipos de rótulos têm uma média predefinida em validações: rótulos de estado final, rótulos de estado de progresso e rótulos de estado de aceitação.

A semântica é explicada no Capítulo 6.

As instruções restantes, seleção, repetição, enviar, receber, interromper, tempo limite e sequências atômicas, são discutidas abaixo.

SELEÇÃO

Uma declaração de seleção começa com a palavra-chave `if`, é seguida por uma lista de um ou mais `options` e termina com a palavra-chave `fi`. Cada opção começa com o sinalizador `::` seguido por qualquer sequência de declarações. Uma e apenas uma opção de uma declaração de seleção será selecionado para execução. A primeira declaração de uma opção determina se a opção pode ser selecionada ou não. Se mais de uma opção for executável, uma será selecionados aleatoriamente. Assim, a linguagem define máquinas não determinísticas como definidas na página 164.

Página 400

REPETIÇÃO E QUEBRA

A repetição ou fazer declaração é semelhante a uma instrução de seleção, mas é executado repetibilidade até que uma instrução `break` seja executada ou um `goto` jump transfira o controle para fora lado do ciclo. As palavras-chave da instrução de repetição são `do` e `od` em vez de `if` e `fi`. A instrução `break` encerrará a instrução de repetição mais interna em qual é executado. O uso de uma instrução `break` fora de uma instrução de repetição é ilegal.

SEQUÊNCIA ATÔMICA

A palavra-chave `atômica` introduz uma sequência atômica de declarações que devem ser executadas cortado como um passo indivisível. A sintaxe é a seguinte:

{sequência} `atômica;`

Logicamente, a sequência de instruções agora é equivalente a uma única instrução. É um erro em tempo de execução se qualquer instrução em uma sequência atômica diferente da primeira for encontrada

para ser inexecutável. A primeira afirmação é chamada de *guarda* da sequência. Se for executável, assim deve ser o resto da sequência. Em geral, portanto, o guarda de uma sequência atômica é seguida apenas com atribuições locais e condições locais, mas não com quaisquer declarações de envio ou recebimento.

ENVIAR

A sintaxe de uma instrução de envio é

`q! expr`

onde `q` é o nome de um canal, e a avaliação da expressão `expr` retorna um valor a ser anexado ao canal. A instrução de envio não é executável (blocos) se o canal está cheio ou não existe. Se mais de um valor deve ser passado de emissor para receptor, as expressões são escritas em uma lista separada por vírgulas:
`q! expr1, expr2, expr3`

Equivalentemente, isso pode ser escrito

```
q! expr1 (expr2, expr3)
```

RECEBER

A sintaxe da instrução de recebimento é

```
q? nome
```

onde `q` é o nome de um canal e `nome` é uma variável ou constante. Se uma constante é especificado, a instrução de recebimento só é executável se o canal existir e o mais antigo a mensagem armazenada no canal contém o mesmo valor. Se uma variável for especificada, o comando `receive` é executável se o canal existe e contém qualquer mensagem.

A variável nesse caso receberá o valor da mensagem que é recuperada. E se mais de um valor é enviado por mensagem, a instrução de recebimento também contém uma vírgula lista separada de variáveis e constantes,

Página 401

```
q? nome1, nome2, ...
```

que novamente é sintaticamente equivalente a

```
q? nome1 (nome2, ...)
```

Cada constante nesta lista coloca uma condição extra na executabilidade do recebimento: ele deve corresponder ao valor do campo da mensagem correspondente da mensagem a ser recuperado. Os campos variáveis recuperam os valores dos campos de mensagem correspondentes em uma recepção. É um erro tentar receber um valor quando nenhum foi transferido, e vice versa.

Qualquer instrução de recebimento pode ser usada como uma condição livre de efeitos colaterais, incluindo seu

lista de parâmetros entre colchetes:

```
q? [nome1, nome2, ...]
```

```
q? [nome1 (nome2, ...)]
```

A instrução é executável (retorna um resultado diferente de zero) apenas se o

A operação de recepção é executável, mas não tem efeito nas variáveis ou no canal.

O único outro tipo de operação permitida nos canais é

```
len (varref)
```

onde `varref` identifica um canal instanciado. A operação retorna o número de mensagens no canal especificado ou zero se o canal não existir.

TEMPO ESGOTADO

O tempo limite da palavra-chave representa uma condição que se torna verdadeira se e somente se não outra instrução no sistema é executável. Uma declaração de tempo limite não tem efeito quando executado. Timeouts podem ser incluídos em expressões.

C.10 MACROS E ARQUIVOS INCLUÍDOS

O texto de origem de uma especificação é processado pelo pré-processador C para macro-expansão e inclusão de arquivos, Kernighan e Ritchie [1978].

C.11 PROMELA GRAMMAR

A gramática está listada no estilo BNF. Parênteses são usados para agrupamento. A plus indica uma repetição de uma ou mais vezes da última unidade sintática; uma estrela indica uma repetição de zero ou mais vezes. Colchetes são usados para indicar elementos opcionais. Uma barra vertical separa as opções. Literais são citados. Terminais são escritos em maiúsculas, não terminais em minúsculas.

```
programa
```

```
::= {unidade} +
```

Página 402

```
unidade
 ::= NOME DO PROCTYPE '(' [decl_lst] ')' corpo
 | CLAIM body
 | Corpo INIT
 | one_decl
 | MTYPE ASGN '{' NAME ',' NAME } * '}'
 | ';'
corpo
 ::= '{' sequência '}'
sequência ::= etapa ';' degrau } *
degrau
 ::= [decl_lst] stmnt
one_decl ::= [TYPE ivar ',' ivar } *
decl_lst ::= one_decl ';' one_decl } *
ivar
 ::= var_dcl | var_dcl ASGN expr | var_dcl ASGN ch_init
```

```

ch_init
 ::= '[' CONST ']' OF '{' TYPE {' , ' TYPE} * '}'
var_dcl
 ::= NOME '[' CONST ']'
var_ref
 ::= NOME '[' expr ']'
stmtnt
 ::= var_ref ASGN expr
 | var_ref RCV margs
 | var_ref SND margs
 | PRINT '(' STRING {' , ' expr} * ')'
 | ASSERT expr
 | GOTO NAME
 | expr
 | NOME ':' stmtnt
 | IF opções FI
 | Opções DO OD
 | QUEBRAR
 | ATÔMICA '{' seqüência '}'
opções
 ::= {sequência SEP} +
binop
 ::= '+' | '-' | '*' | '/' | '%' | '&' | '||' | '>' | '<'
 | GE | LE | EQ | NE | E | OU | LSHIFT | RSHIFT
abrir
 ::= '(' | '[' | '{'
fechar
 ::= ')' | ']' | '}'

```

Página 403

```

expr
 ::= '(' expr ')'
 | expr binop expr
 | unop expr
 | RUN NAME '(' [arg_lst] ')'
 | LEN '(' var_ref ')'
 | var_ref RCV '[' margs ']'
 | var_ref
 | CONST
 | TEMPO ESGOTADO
 | var_ref '.' var_ref
 | var_ref ':' NOME
arg_lst
 ::= expr {' , ' expr} *
margs
 ::= arg_lst | expr '(' arg_lst ')'

```

Página 404

SPIN VERSÃO 0 FONTE DO SIMULADOR D

O *makefile* para a versão final do SPIN, discutido no Capítulo 12, é definido como segue baixos.

```

CC = cc
# Compilador ANSI C
CFLAGS = -O
# otimizador
YFLAGS = -v -d -D # criar y.output, y.debug e y.tab.h
OFILES = spin.o lex.o sym.o vars.o main.o debug.o \
mesg.o flow.o sched.o run.o dummy.o
rotação: $ (OFILES)
$ (CC) $ (CFLAGS) -o spin $ (OFILES) -lm
% .o:
% .c spin.h
$ (CC) $ (CFLAGS) -c % .c

```

O restante deste Apêndice lista o conteúdo dos 11 arquivos de origem (ver Tabela D.1) que são necessários para compilar o programa, mais um arquivo fictício como temporário espaço reservado para as rotinas de análise que são adicionadas no Capítulo 13 e listadas em Apêndice E. O programa deve ser executado em qualquer sistema UNIX com um padrão ANSI compilador C compatível.

Consulte a introdução do Apêndice E para obter informações sobre como recuperar uma cópia online da versão mais recente do SPIN.

394
APÊNDICE D
PROJETO E VALIDAÇÃO

Tabela D.1 - Índice do arquivo de origem

Arquivo
Número da linha

manequim.c
1907
flow.c
1137
lex.l
194
main.c
487
mesg.c
867
run.c
1378
sched.c
1553
spin.h
1
spin.y
283
sym.c
626
vars.c
717

FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN
395

Tabela D.2 - Procedimentos listados - Apêndice D

Procedimento
Procedimento de linha
Linha

a _ recv (q, n, completo)
1001 a _ snd (q, n)
969
adicionar _ el (e, s)
1232 add _ seq (n)
1256
adiciona símbolo (r, s)
1719 break _ dest ()
1348
elenco _ val (t, v)
791 verificar _ nome (s)
268
checkvar (s, n)
745 fechar _ seq ()
1158
cnt _ mpars (n)
882 _ rendez completo () 1688
doq (s, n)
1111 dumpglobals ()
815
dumplocal (s)
847 emalloc (n)
582
habilitar (s, n)
1602 eval (agora)

1455
eval_sub (e)
1385 eval_sync (agora)
1435
fatal (s1, s2)
573 find_lab (s, c)
1322
findloc (s, n)
1801 gensrc ()
1909
obter_laboratório (s)
1287 getglobal (s, n)
769
getlocal (s, n)
1821 getval (s, n)
725
tem_laboratório (e)
1298 hash (s)
634
if_seq (s, tok, lnno)
1209 interpret (n)
1511
ismtype (str)
703 pesquisa (ões)
649
principal (argc, argv)
502 make_atomic (s)
1355
corresponder_trilha ()
1914 mov_lab (z, e, y)
1309
naddsymbol (r, s, k)
1741 novo_el (n)
1195
nn (s, v, t, l, r)
592 aberto_seq (topo)
1148
p_falar (e)
1860 pushbreak ()
1335
q_é_sync (n)
935 qlen (n)
925
qmake (s)
893 qrecv (n, completo)
959
qsend (n)
945 pronto (n, p, s)
1589
rem_lab (a, b, c)
607 rem_var (a, b, c)
617
remotelab (n)
1868 remotevar (n)
1880
executável (s, n)
1575 s_snd (q, n)
1039
sched ()
1631 seqlist (s, r)
1184
set_lab (s, e)
1272 setglobal (v, m)
804
setlocal (p, m)
1831 setmtype (m)
686
setparams (r, p, q)
1778 settype (n, t)
670

setval (v, n)
 735 sr _ mesg (v, j)
 1098
 sr _ falar (n, v, s, a, j, mx, nomeado) 1073 iniciar _ reivindicação (n)
 1617
 falar (e, s)
 1848 typek (n, t, s)
 1761
 typex (n, t)
 783 andar _ atômico (a, b)
 1364
 quem corre()
 Finalização de 1842 ()
 1674
 yyerror (s1, s2)
 561

Página 407

396
 APÊNDICE D
 PROJETO E VALIDAÇÃO

Tabela D.3 - Procedimentos explicados - Capítulo 12

Procedimento	Procedimento de página	Página
a _ recv ()		275
a _ snd ()		274
adicionar _ el ()		280
add _ seq ()		280
adicionayymbol ()		289
break _ dest ()		283
elenco _ val ()		268
verifique _ nome ()		260
verifique _ nome ()		269
verifique _ nome ()		286
checkvar ()		267
fechar _ seq ()		279
complete _ rendez ()		276
emalloc ()		262
habilitar()		288
eval ()		254
eval ()		267
eval ()		272
eval ()		289
eval ()		291

eval_sub ()
281
eval_sub ()
290
eval_sync ()
277
findloc ()
291
get_lab ()
281
getglobal ()
268
getlocal ()
291
cerquilha()
263
if_seq ()
284
interpret ()
271
ismtype ()
286
olho para cima()
262
olho para cima()
263
a Principal()
293
novo_el ()
279
nn ()
253
nn ()
265
abrir_seq ()
279
pushbreak ()
282
qlen ()
273
qmake ()
268
qmake ()
273
qrecv ()
274
qsend ()
274
pronto()
287
rem_lab ()
292
rem_var ()
292
executável ()
287
s_snd ()
275
sched ()
289
seqlist ()
279
set_lab ()
281
setglobal ()
267
setlocal ()
291
setmtype ()
286

setparams ()
288
setparams ()
288

Página 408

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN
397
1 / ***** spin: spin.h ***** /
2
Símbolo de estrutura de 3 typedef {
4
Caracteres
*nome;
5
tipo curto;
/* variável ou tipo de canal
*/
6
int
nel;
/* 1 se escalar,> 1 se matriz */
7
int
* val;
/* valor (es) de tempo de execução, initl 0 */
8
Nó de estrutura
* ini; /* valor inicial ou chan-def */
9
contexto de símbolo * de estrutura; /* 0 se global ou procname */
10
struct Symbol * next; /* lista vinculada */
11} símbolo;
12
13 typedef struct Node {
14
int
nval;
/* atributo de valor
*/
15
short ntyp;
/* tipo de nó
*/
16
Símbolo * nsym;
/* novo atributo
*/
17
Símbolo * fname;
/* nome do arquivo de src
*/
18
struct SeqList * seql; /* lista de sequências
*/
19
Nó de estrutura
* lft, * rgt; /* filhos na árvore de análise */
20} Nó;
21
22 typedef struct Queue {
23
qid curto;
/* índice q de tempo de execução
*/
24
qlen curto;
/* nr mensagens armazenadas */
25
nslots curtos, nflds; /* capacidade, flds / slot */
26
short * fld_width;
/* tipo de cada campo */
27
int
* conteúdo;

```

/ * o buffer real
*
28
struct Queue
* nxt; /* lista vinculada */
29} fila;
30
31 typedef struct Element {
32
Nó
* n;
/* define o tipo e conteúdo */
33
int
seqno;
/* identifica exclusivamente este el */
34
status de char unsigned; /* usado pelo gerador do analisador */
35
struct SeqList * sub; /* subsequências, para compostos */
36
Elemento struct * nxt; /* lista vinculada */
37} Elemento;
38
39 typedef struct Sequence {
40
Elemento * primeiro;
41
Elemento * último;
42} Sequência;
43
44 typedef struct SeqList {
45
Seqüência
*esta; /* uma sequência */
46
struct SeqList * nxt; /* lista vinculada */
47} SeqList;
48
Rótulo de estrutura de 49 typedef {
50
Símbolo * s;
51
Símbolo * c;
52
Elemento * e;
53
etiqueta de estrutura
* nxt;

```

```

398
APÉNDICE D
PROJETO E VALIDAÇÃO
54} Label;
55
56 typedef struct Lbreak {
57
Símbolo * l;
58
struct Lbreak * nxt;
59} Lbreak;
60
61 typedef struct RunList {
62
Símbolo * n;
/* nome
*/
63
int
pid;
/* id do processo
*/
64
int
maxseq;
/* usado pelo gerador do analisador */
65
Elemento * pc;
/* stmnt atual */

```

```

66
Símbolo * symtab;
/* variáveis locais */
67
struct RunList * nxt; /* lista vinculada */
68} RunList;
69
70 typedef struct ProcList {
71
Símbolo * n;
/* nome
*/
72
Nó
* p;
/* parâmetros */
73
Sequência * s;
/* corpo
*/
74
struct ProcList * nxt; /* lista vinculada */
75} ProcList;
76
77 #ifdef GODEF
78 typedef struct atom_stack {/* todos os conjuntos de conflito atingidos na seq atômica. */
79
char * o quê;
/* what = "R_LOCK", "W_LOCK" */
80
int quando;
/* quando == Direto ou Indireto */
81
causa interna;
/* tipo de operação, por exemplo, 'r' */
82
Nó * n;
83
struct atom_stack * nxt;
84} atom_stack;
85
86 #define FEITO
1
/* bits de status dos elementos */
87 #define ATOM
2
/* parte de uma cadeia atômica */
88 #define L_ATOM
4
/* último elemento em uma cadeia */
89 #ifdef VARSTACK
90 # define HIT 8
/* hit em dflow.c search */
91 #define Nhash
255
/* tamanho da tabela de hash */
92
93 #define PREDEF
5
/* identificador predefinido */
94 # define BIT 1
/* tipos de dados
*/
95 #define BYTE
8
/* largura em bits */
96 #define SHORT
16
97 #define INT 32
98 #define
CHAN
64
99
100 #ifdef GODEF
101 #define Direct
1
102 #define Indireto
2
103
104 #define max (a, b) (((a) <(b))? (B): (a))
105
106 / ***** Old-Style C - definições de protótipo ***** /

```

```
107 extern char * malloc ();
```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```
399
108 extern char * memcpy ();
109 extern char * memset ();
110 extern char * mktemp ();
111 extern char * strcat ();
112 extern char * strcpy ();
113 extern long time ();
114 saída externa vazia ();
115 externo vazio srand ();
116
117 elemento externo
* d_eval_sub ();
118 elemento externo
* dois pontos ();
119 elemento externo
* eval_sub ();
120 elemento externo
* get_lab ();
121 elemento externo
* huntele ();
122 elemento externo
* if_seq ();
123 elemento externo
* new_el ();
124 elemento externo
* walk_sub ();
125 Nó externo * nn ();
126 Nó externo * rem_var ();
127 Nó externo * rem_lab ();
128 extern SeqList
* lista_seq ();
129 sequência externa
* close_seq ();
130 simbolo externo
* break_dest ();
131 simbolo externo
* findloc ();
132 simbolo externo
* has_lab ();
133 simbolo externo
* olho para cima();
134 extern char * emalloc ();
135 externo vazio add_el ();
136 extern void add_seq ();
137 extern void addsymbol ();
138 #ifdef DEBUG
139 externo vazio auto2 ();
140 externo vazio auto_atomic ();
141 extern void check_proc ();
142 extern void cnt_mpars ();
143 comentário vazio externo ();
144 extern void do_var ();
145 doglobal externo vazio ();
146 extern void do_init ();
147 vazio externo dolocal ();
148 extern void doq ();
149 extern void dumpglobals ();
150 dumplocal de vazio externo ();
151 extern void dumpskip ();
152 extern void dumpsrc ();
153 extern vazio end_labs ();
154 externo vazio explicar ();
155 externo vazio fatal ();
156 extern void genaddproc ();
157 extern void genaddqueue ();
158 extern void genaddclaim ();
159 genheader vazio externo ();
160 genother vazio externo ();
161 gensrc vazio externo ();
```

PROJETO E VALIDAÇÃO

```

162 genúnio vazio externo ();
163 extern void lost_trail ();
164 extern vazio principal ();
165 vazio externo make_atomic ();
166 extern void match_trail ();
167 extern vazio mov_lab ();
168 naddsymbol externo vazio ();
169 casos externos vazios ();
170 espaços vazios externos ();
171 extern void open_seq ();
172 vazio externo p_talk ();
173 extern void patch_atomic ();
174 pushbreak de vazio externo ();
175 extern void put_pinit ();
176 extern void put_ptype ();
177 extern void putname ();
178 externo vazio putnr ();
179 externo vazio putremoto ();
180 extern void putproc ();
181 progresso do vazio externo ();
182 extern void putseq ();
183 extern void putsed_el ();
184 extern void putsed_lst ();
185 extern void putskip ();
186 externo vazio putrc ();
187 extern void putstmnt ();
188 vazio externo pronto ();
189 extern void runnable ();
190 extern vazio sched ();
191 extern void set_lab ();
192 extern void setmtype ();
193 extern void setparams ();
194 extern void setttype ();
195 extern void sr_mesg ();
196 extern void sr_talk ();
197 extern void start_claim ();
198 extern void talk ();
199 externo vazio typ2c ();
200 tipo externo vazio ();
201 extern void undostmnt ();
202 externo vazio walk_atomic ();
203 prostitutas de vazio externo ();
204
205 / ***** spin: lex.l ***** /
206
207%
208 #inclui "spin.h"
209 #include "y.tab.h"
210
211 int
lineno = 1;
212 caracteres não assinados
in_comment = 0;
213 simbolo externo
* Fname;
214
215 #define Token
if (! in_comment) return

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

401
216%
217
218 %%
219 "/ **"
{in_comment = 1; }
220 "* /"
{in_comment = 0; }
221 \ n
{lineno ++; }
222 [\ t]
{/ * ignorar o espaço em branco * /}
223 [0-9] +
{yyval.val = atoi (yytext); Token CONST; }
224 \ # \ [0-9] + \ \ "[^ \"]" * \ "[0-9] * {
/* Directiva do pré-processador */
225

```

```

int i = 1;
226
while (yytext [i] == '') i++;
227
lineno = atoi (& yytext [i]) - 1;
228
while (yytext [i]! = '') i++;
229
Fname = lookup (& yytext [i + 1]);
230
}
231 \ "[^ \\]" * \ "{yyval.sym = lookup (yytext); Token STRING; }
232 "nunca" {yyval.sym = lookup (" : nunca:"); Token CLAIM; }
233 "init"
{yyval.sym = lookup (" _ init"); Token INIT; }
234 "int"
{yyval.val = INT; Token TYPE; }
235 "curto"
{yyval.val = SHORT; Token TYPE; }
236 "byte"
{yyval.val = BYTE; Token TYPE; }
237 "bool"
{yyval.val = BIT; Token TYPE; }
238 "bit"
{yyval.val = BIT; Token TYPE; }
239 "chan"
{yyval.val = CHAN; Token TYPE; }
240 "pular"
{yyval.val = 1; Token CONST; }
241 [a-zA-Z_] [a-zA-Z_0-9] * {Token check_name (yytext); }
242 "::"
{yyval.val = lineno; Token SEP; }
243 "="
{yyval.val = lineno; Token ASGN; }
244 "!"
{yyval.val = lineno; Token SND; }
245 "?"
{yyval.val = lineno; Token RCV; }
246 ">"
{ Símbolo ';' ; / * separador de instrução * /}
247 "<<"
{Token LSHIFT; / * shift bits left * /}
248 ">>"
{Token RSHIFT; / * shift bits right * /}
249 "<="
{ Símbolo
LE; /* menos que ou igual a */ }
250 "> ="
{ Símbolo
GE; /* Melhor que ou igual a */ }
251 "==""
{ Símbolo
EQ; /* igual a */ }
252 "!" =
{ Símbolo
NE; /* diferente de */ }
253 "&&"
{ Símbolo
E; /* lógico e */ }
254 "||"
{ Símbolo
OU; /* lógico ou */ }
255.
{Token yytext [0]; }
256 %%
257
258 struct estático {
259
char * s;
tok int;
260} Nomes [] = {
261
"Simetria",
SIMETRIA,
262
"afirmar",
AFIRMAR,
263
"atômico",
ATOMIC,
264
"quebrar",

```

```
QUEBRAR,  
265  
"Faz",  
FAZ,  
266  
"fi",  
FI,  
267  
"vamos para",  
VAMOS PARA,  
268  
"E se",  
E SE,  
269  
"len",  
LEN,
```

Página 413

```
402  
APÊNDICE D  
PROJETO E VALIDAÇÃO  
270  
"mtype",  
MTYPE,  
271  
"od",  
OD,  
272  
"do",  
DO,  
273  
"printf",  
IMPRESSÃO,  
274  
"proctype",  
PROCTYPE,  
275  
"corre",  
CORRE,  
276  
"tempo esgotado",  
TEMPO ESGOTADO,  
277  
0,  
0,  
0,  
278};  
279  
280 check_name (s)  
281  
char * s;  
282 {  
283  
registrar int i;  
284  
para (i = 0; nomes [i] .s; i ++)  
285  
if (strcmp (s, nomes [i] .s) == 0)  
286  
{  
yyval.val = lineno;  
287  
retornar nomes [i] .tok;  
288  
}  
289  
if (yyval.val = ismtype (s))  
290  
return CONST;  
291  
yyval.sym = pesquisa (s); /* tabela de simbolos */  
292  
retornar NOME;  
293}  
294  
295 / ***** spin: spin.y ***** /  
296  
297% {  
298 #include "spin.h"  
299 #define YYDEBUG  
0
```

```

300 #define Parada
nn (0, lineno, '@', 0,0)
301 simbolo externo
*contexto;
302 extern int lineno, u_sync, u_async;
303 char
* Claimproc = (char *) 0;
304%
305
306% sindicato {
307
int
308
val;
309
Nó
*nó;
309
Símbolo * sym;
310
Sequência * seq;
311
SeqList * seql;
312}
313
314% token
<val> EXECUTAR LEN DE
315% token
<val> TIPO DE CONST ASGN
316% token
<sym> NOME RECLAMAÇÃO
317% token
<sym> STRING INIT
318% token
<val> ASSERT SYMMETRY
319% token
<val> IR PARA INTERROMPER MTYPE SEP
Token de 320%
<val> SE FI DO OD ATOMIC
321% token
<val> SND RCV PRINT TIMEOUT
322% token
<val> PROCTYPE
323

```

Página 414

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

403
324% tipo
<sym> var ivar
325% tipo
<node> expr var_list stmnt
326% tipo
<node> args arg arglist typ_list decl
327% tipo
<node> decl_lst one_decl any_decl
328% tipo
<node> prargs margs varref passo ch_init
329% tipo
opções <seql>
330% tipo
<seq> corpo da opção
331
332% certo
ASGN
333% restantes
SND RCV
334% restantes
OU
335% restantes
E
336% restantes
'|
337% restantes
'&
338% restantes
EQ NE
339% restantes
'>' '<' GE LE
340% restantes

```

```

LSHIFT RSHIFT
341% restantes
'+' '-'
342% restantes
'*' '/' '%'
343% certo
'~' UMIN NEG
344 %%
345
346 / ** Regras de gramática PROMELA ** /
347
348 programa: unidades
{sched (); }
349
;
350 unidades: unidade | unidade de unidades
351
;
352 unidade
: proc
353
| afirmação
354
| iniciar
355
| one_decl
356
| mtype
357
;
358 proc
: NOME DO PROCTYPE
{contexto = $ 2; }
359
'(' decl ')'
360
corpo
{pronto ($ 2, $ 5, $ 7);
361
contexto = (simbolo *) 0;
362
}
363
;
364 reivindicação: CLAIM
{contexto = $ 1;
365
if (Claimproc)
366
yyerror ("reivindicação% s redefinida",
367
Claimproc);
368
Claimproc = $ 1-> nome;
369
}
370
corpo
{pronto ($ 1, (Nó *) 0, $ 3);
371
contexto = (simbolo *) 0;
372
}
373
;
374 init
: INICIAR
{contexto = $ 1; }
375
corpo
{executável ($ 3, $ 1);
376
contexto = (simbolo *) 0;
377
}

```

```

378
;
379 mtype: MTYPE ASGN '{' args '}' {setmtype ($ 4); }
380
| SYMMETRY ASGN arglist
{syms ($ 3); }
381
| ';;'
/ * opcional; como separador de unidades *
382
;
383 arglist
: '{' arg '}'
{$$ = $ 2; }
384
| '{' arg '}' ',' arglist
{$$ = nn (0, 0, ',', $ 2, $ 5); }
385
;
386 corpo
: '{'
{open_seq (1); }
387
seqüênci
{add_seq (parar); }
388
'}'
{$$ = close_seq ();
389
}
390
;
Sequênci 391: etapa
{add_seq ($ 1); }
392
| seqüênci ';' etapa {add_seq ($ 3); }
393
;
394 passo
: any_decl stmnt
{$$ = $ 2; }
395
;
396 any_decl: / * vazio *
{$$ = (NÓ *) 0; }
397
| one_decl ',' any_decl {$$ = nn (0, 0, ',', $ 1, $ 3); }
398
;
399 one_decl: TYPE var list
{settype ($ 2, $ 1); $$ = $ 2; }
400
;
401 decl_lst: one_decl
{$$ = nn (0, 0, ',', $ 1, 0); }
402
| one_decl ',' decl_lst {$$ = nn (0, 0, ',', $ 1, $ 3); }
403
;
404 decl
: / * vazio *
{$$ = (NÓ *) 0; }
405
| decl_lst
{$$ = $ 1; }
406
;
407 var_list: ivar
{$$ = nn ($ 1, 0, TIPO, 0, 0); }
408
| ivar ',' var_list {$$ = nn ($ 1, 0, TYPE, 0, $ 3); }
409
;
410 ivar
: var
{$$ = $ 1; }
411
| var ASGN expr
{$ 1-> ini = $ 3; $$ = $ 1; }
412
| var ASGN ch_init {$ 1-> ini = $ 3; $$ = $ 1; }
413 / * para compatibilidade com Unix v.10: */

```

```

414
| NOME '[' CONST ']' OF '{' typ_list '}' {
415
$ 1-> nel = 1;
416
if ($ 3) u_async++; else u_sync++;
417
$ 1-> ini = nn (0, $ 3, CHAN, 0, $ 7);
418
cnt_mpars ($ 7);
419
$$ = $ 1;
420
}
421
;
422 ch_init: '[' CONST ']' OF '{' typ_list '}'
423
(if ($ 2) u_async++; else u_sync++;
424
cnt_mpars ($ 6);
425
$$ = nn (0, $ 2, CHAN, 0, $ 6);
426
)
427
;
428 var
: NOME
{$ 1-> nel = 1; $$ = $ 1; }
429
| NOME '[' CONST ']' {$ 1-> nel = $ 3; $$ = $ 1; }
430
;
431 varref: NAME
{$$ = nn ($ 1, 0, NOME, 0, 0); }

```

Página 416

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

405
432
| NOME '[' expr ']' {$$ = nn ($ 1, 0, NOME, $ 3, 0); }
433
;
434 stmnt: varref ASGN expr {$$ = nn ($ 1-> nsym, $ 2, ASGN, $ 1, $ 3); }
435
| varref RCV margs {$$ = nn ($ 1-> nsym, $ 2, 'r', $ 1, $ 3); }
436
| varref SND margs {$$ = nn ($ 1-> nsym, $ 2, 's', $ 1, $ 3); }
437
| PRINT '(' STRING prargs ')' {$$ = nn ($ 3, $ 1, PRINT, $ 4, 0); }
438
| ASSERT expr
{$$ = nn (0, $ 1, ASSERT, $ 2, 0); }
439
| GOTO NAME
{$$ = nn ($ 2, $ 1, GOTO, 0, 0); }
440
| expr
{$$ = nn (0, lineno, 'c', $ 1, 0); }
441
| NOME ':' stmnt
{$$ = nn ($ 1, $ 3-> nval, ':', $ 3, 0); }
442
| IF opções FI
{$$ = nn (0, $ 1, SE, 0, 0);
443
$$ -> seql = $ 2;
444
}
445
| FAZ
{pushbreak (); }
446
opções OD
{$$ = nn (0, $ 1, DO, 0, 0);
447
$$ -> seql = $ 3;
448

```

```

}
449
| QUEBRAR
{$$ = nn (break_dest (), $ 1, GOTO, 0,0); }
450
| ATOMIC
451
'{'
{open_seq (0); }
452
seqüênciा
453
'}'
{$$ = nn (0, $ 1, ATÔMICO, 0, 0);
454
$$ -> seql = seqlist (close_seq (), 0);
455
make_atomic ($$ -> seql-> this);
456
}
457
;
458 opções: opção
{$$ = lista seq ($ 1, 0); }
459
| opções de opções
{$$ = lista seq ($ 1, $ 2); }
460
;
Opção 461: SEP
{open_seq (0); }
462
seqüênciা
{$$ = close_seq (); }
463
;
464 expr
: '(' expr ')'
{$$ = $ 2; }
465
| expr '+' expr
{$$ = nn (0, 0, '+', $ 1, $ 3); }
466
| expr '-' expr
{$$ = nn (0, 0, '-', $ 1, $ 3); }
467
| expr '*' expr
{$$ = nn (0, 0, '*', $ 1, $ 3); }
468
| expr '/' expr
{$$ = nn (0, 0, '/', $ 1, $ 3); }
469
| expr '%' expr
{$$ = nn (0, 0, '%', $ 1, $ 3); }
470
| expr '&' expr
{$$ = nn (0, 0, '&', $ 1, $ 3); }
471
| expr '!' expr
{$$ = nn (0, 0, '!', $ 1, $ 3); }
472
| expr '>' expr
{$$ = nn (0, 0, '>', $ 1, $ 3); }
473
| expr '<' expr
{$$ = nn (0, 0, '<', $ 1, $ 3); }
474
| expr GE expr
{$$ = nn (0, 0, GE, $ 1, $ 3); }
475
| expr LE expr
{$$ = nn (0, 0, LE, $ 1, $ 3); }
476
| expr EQ expr
{$$ = nn (0, 0, EQ, $ 1, $ 3); }
477
| expr NE expr
{$$ = nn (0, 0, NE, $ 1, $ 3); }
478
| expr AND expr
{$$ = nn (0, 0, AND, $ 1, $ 3); }
479

```

```

| expr OU expr
{$$ = nn (0, 0, OU, $ 1, $ 3); }
480
| expr LSHIFT expr {$$ = nn (0, 0, LSHIFT, $ 1, $ 3); }
481
| expr RSHIFT expr {$$ = nn (0, 0, RSHIFT, $ 1, $ 3); }
482
| `~` expr
{$$ = nn (0, 0, `~`, $ 2, 0); }
483
| `-` expr% prec UMIN
{$$ = nn (0, 0, UMIN, $ 2, 0); }
484
| SND expr% prec NEG
{$$ = nn (0, 0, `!`, $ 2, 0); }
485
| RUN NAME `(` args ')'
{$$ = nn ($ 2, $ 1, EXECUTAR, $ 4, 0); }

```

Página 417

```

406
APÊNDICE D
PROJETO E VALIDAÇÃO
486
| LEN `(` varref ')'
{$$ = nn ($ 3-> nsim, $ 1, LEN, $ 3, 0); }
487
| varref RCV `[' margs ']' {$$ = nn ($ 1-> nsym, $ 2, 'R', $ 1, $ 4); }
488
| varref
{$$ = $ 1; }
489
| CONST
{$$ = nn (0, $ 1, CONST, 0, 0); }
490
| TEMPO ESGOTADO
{$$ = nn (0, $ 1, TEMPO LIMITE, 0, 0); }
491
| NOME `[' expr ']' `.' varref {$$ = rem_var ($ 1, $ 3, $ 6); }
492
| NOME `[' expr ']' `:' NOME {$$ = rem_lab ($ 1, $ 3, $ 6); }
493
;
494 typ_list: TYPE
{$$ = nn (0, 0, $ 1, 0, 0); }
495
| TYPE `,' typ_list
{$$ = nn (0, 0, $ 1, 0, $ 3); }
496
;
497 args
: / * vazio * /
{$$ = (Nó *) 0; }
498
| arg
{$$ = $ 1; }
499
;
500 arg
: expr
{$$ = nn (0, 0, `,', $ 1, 0); }
501
| expr `,' arg
{$$ = nn (0, 0, `,', $ 1, $ 3); }
502
;
503 prargs: / * vazio * /
{$$ = (Nó *) 0; }
504
| `,' arg
{$$ = $ 2; }
505
;
506 margs: arg
{$$ = $ 1; }
507
| expr `(` arg `)'
{$$ = nn (0, 0, `,', $ 1, $ 3); }
508
;

```

```

509 %%
510
511 / ***** spin: main.c ***** /
512
513 #include <stdio.h>
514 #include "spin.h"
515 #include "y.tab.h"
516
517 simbolo externo
*contexto;
518 extern int lineno;
519 extern FILE * yyin;
520 Symbol
* Fname;
521 int verbose = 0;
522 análise interna = 0;
523 int s_trail = 0;
524 int m_loss = 0;
525 int Nomeado = 0;
526 int nr_errs = 0;
527
528 vazio
529 principal (argc, argv)
530
char * argv [];
531 {
532
Símbolo * s;
533
int T = (int) tempo ((longo *) 0);
534
535
while (argc> 1 && argv [1] [0] == '-')
536
{
switch (argv [1] [1]) {
537
caso 'a': analisar = 1; quebrar;
538
caso 'g': verboso += 1; quebrar;
539
caso 'l': verboso += 2; quebrar;

```

Página 418

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

407
540
caso 'm': m_loss = 1; quebrar;
541
case 'N': Nomeado
= 1; quebrar;
542
caso 'n': T = atoi (& argv [1] [2]); quebrar;
543
caso 'p': verboso += 4; quebrar;
544
caso 'r': verboso += 8; quebrar;
545
case 's': verboso += 16; quebrar;
546
caso 'v': verboso += 32; quebrar;
547
case 't': s_trail = 1; quebrar;
548
padrão: printf ("use: spin - [agmlpqrst] [-nN] arquivo \ n");
549
printf ("\ ta produzir um analisador \ n");
550
printf ("\ tg imprimir todas as variáveis globais \ n");
551
printf ("\ tl imprime todas as variáveis locais \ n");
552
printf ("\ tm mensagens perdidas enviadas para filas cheias \ n");
553
printf ("semente \ t-nN para gerador nr aleatório \ n");
554
printf ("\ tp imprimir todas as instruções \ n");
555
printf ("\ tr imprimir eventos de recebimento \ n");

```

```

556
printf ("\ ts eventos de envio de impressão \ n");
557
printf ("\ tv detalhado, mais avisos \ n");
558
printf ("\ tt segue uma trilha de simulação \ n");
559
printf ("\ tN forçar nomeação de tipos de mensagem em trilhas \ n");
560
saída (1);
561
}
562
argc--, argv++;
563
}
564
if (argc> 1)
565
{
arquivo de saída char [17], cmd [64];
566
Fname = lookup (argv [1]);
567
mktemp (strcpy (outfile, "/tmp/spin.XXXXXX"));
568
sprintf (cmd, "/ lib / cpp% s>% s", argv [1], arquivo de saída);
569
if (sistema (cmd))
570
{
desvincular (arquivo de saída);
571
saída (1);
572
} else if (! (yyin = fopen (outfile, "r")))
573
{
printf ("não é possível abrir% s \ n", arquivo de saída);
574
saída (1);
575
}
576
desvincular (arquivo de saída);
577
} outro
578
Fname = lookup ("<stdin>");
579
srand (T);
580
s = pesquisa ("_ p"); s-> tipo = PREDEF;
581
yyparse ();
582
sair (nr_errs);
583}
584
585 yywrap ()
/* rotina fictícia */
586 {
587
return 1;
588}
589
590 yyerror (s1, s2)
591
char * s1, * s2;
592 {
593
extern int yychar;

```

```

if (s2)
596
printf (s1, s2);
597
outro
598
printf (s1);
599
if (yychar) printf (""
viu '% d'. ", yychar);
600
printf ("\n"); fflush (stdout);
601
nr_errs++;
602}
603
604 vazio
605 fatal (s1, s2)
606
char * s1, * s2;
607 {
608
yyerror (s1, s2);
609
fflush (stdout);
610
saída (1);
611}
612
613 char *
614 emalloc (n)
615 (char * tmp = malloc (n);
616
617
if (! tmp)
618
fatal ("memória insuficiente", (char *) 0);
619
memset (tmp, 0, n);
620
return tmp;
621}
622
623 Nó *
624 nn (s, v, t, l, r)
625
Símbolo * s;
626
Nó * l, * r;
627 {
628
Nó * n = (Nó *) emalloc (sizeof (Nó));
629
n-> nval = v;
630
n-> ntyp = t;
631
n-> nsym = s;
632
n-> fname = Fname;
633
n-> lft = l;
634
n-> rgt = r;
635
return n;
636}
637
638 Nó *
639 rem_lab (a, b, c)
640
Símbolo * a, * c;
641
Nó * b;
642 {
643
if (! context || strcmp (context-> name, ": never:") != 0)
644
yyerror ("aviso: uso ilegal de ':' (fora nunca reclamar)", (char *) 0);
645
retornar nn ((símbolo *) 0, 0, EQ,
646

```

```
nn (lookup ("_ p"), 0, 'p', nn (a, 0, '?', b, (Nó *) 0), (Nó *) 0),
647 nn (c, 0, 'q', nn (a, 0, NOME, (Nó *) 0, (Nó *) 0), (Nó *) 0));
```

Página 420

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```
409
648}
649
650 Nó *
651 rem_var (a, b, c)
652 Símbolo * a;
653
654 {
655
656 * tmp;
657
if (! context || strcmp (context-> name, ": never:") != 0)
658 yyerror ("aviso: uso ilegal de '.' (fora nunca reclamar)", (char *) 0);
659 tmp = nn (a, 0, '?', b, (Nó *) 0);
660
661 retornar nn (c-> nsym, 0, 'p', tmp, c-> lft);
662
663 / ***** spin: sym.c ***** /
664 #include "spin.h"
665 #incluir "y.tab.h"
666
667 Symbol
* symtab [Nhash + 1];
668 Symbol
* contexto = (Símbolo *) 0;
669
670 hash (s)
671
char * s;
672 {
673
int h = 0;
674
675 enquanto (* s)
676
{
h += * s++;
677
h <<= 1;
678
if (h & (Nhash + 1))
679
h |= 1;
680
}
681
return h & Nhash;
682}
683
684 Symbol *
685 pesquisa (ões)
686
char * s;
687 {
688
Símbolo * sp;
689
int h = hash (s);
690
691 para (sp = symtab [h]; sp; sp = sp-> próximo)
692 if (strcmp (sp-> nome, s) == 0 && sp-> contexto == contexto)
693 return sp;
/* encontrado */
```

```

694
if (contexto)
/* local */
695
para (sp = symtab [h]; sp; sp = sp-> próximo)
696
if (strcmp (sp-> nome, s) == 0 &&! sp-> contexto)
697
return sp;
/* global */
698
sp = (símbolo *) emalloc (sizeof (símbolo));
/* adicionar
*/
699
sp-> nome = (char *) emalloc (strlen (s) + 1);
700
strcpy (sp-> nome, s);
701
sp-> nel = 1;

```

Página 421

```

410
APÊNDICE D
PROJETO E VALIDAÇÃO
702
sp-> contexto = contexto;
703
sp-> próximo = symtab [h];
704
symtab [h] = sp;
705
706
return sp;
707}
708
709 vazio
710 settype (n, t)
711
Nó * n;
712 {
713
enquanto (n)
714
{
if (n-> nsym-> tipo)
715
yyerror ("redeclaração de '% s'", n-> nsym-> nome);
716
n-> nsym-> tipo = t;
717
if (n-> nsym-> nel <= 0)
718
yyerror ("tamanho de array inválido para '% s'", n-> nsym-> nome);
719
n = n-> rgt;
720
}
721}
722
723 Nó * Tipo M = (Nó *) 0;
724
725 vazio
726 setmtype (m)
727
Nó * m;
728 {
729
Nó * n = m;
730
if (Mtype)
731
yyerror ("mtype redeclared", (char *) 0);
732
733
Mtype = n;
734
enquanto (n)
/* verificação de sintaxe */
735

```

```

{
if (! n-> lft ||! n-> lft-> nsym
736
|| (n-> lft-> ntyp! = NOME)
737
|| n-> lft-> lft)
/* variável indexada */
738
fatal ("definição de mtype ruim", (char *) 0);
739
n = n-> rgt;
740
}
741}
742
743 Node * Symnode = 0;
744
745 vazio
746 simbolos (m)
747
Nó * m;
748 {
749
if (Symnode)
750
yyerror ("Definição de simetria duplicada", (char *) 0);
751
752
Symnode = m;
753}
754 #include <stdio.h>
755

```

Página 422

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

411
756 int SymCnt = 0;
757 vazio
758 putsyms (fd, fh)
759
ARQUIVO * fd, * fh;
760 {
761
if (! Symnode ||! Symnode-> rgt ||! Symnode-> lft)
762
{
fprintf (fh, "#define Normalize 0 \ n");
763
Retorna;
764
}
765
fprintf (fh, "#ifndef WHICH \ n");
766
fprintf (fh, "#define WHICH
1
/* deve ser 1 ou -1 */ \ n ");
767
fprintf (fh, "#endif \ n");
768
769
putdescend (fd, Symnode-> lft, Symnode-> rgt);
770
771
fprintf (fh, "#define Normalize 1");
772
enquanto (SymCnt> 0)
773
fprintf (fh, "\\\ n \ t \ t && Symmer% d () == WHICH", --SymCnt);
774
fprintf (fh, "\ n");
775}
776
777 vazio
778 putdescend (fd, a, b)
779
ARQUIVO * fd;
780
Nó * a, * b;

```

```

781 {
782 if (b-> ntyp! = ';')
783 putasym (fd, a, b);
784 outro
785 {
786 putasym (fd, a, b-> lft);
787 putasym (fd, a, b-> rgt);
788 }
789
790 vazio
791 putasym (fd, m, n)
792
ARQUIVO * fd;
793
Nó * m, * n;
794 {
795
Nó * t1 = m;
796
Nó * t2 = n;
797
fprintf (fd, "Symmer% d () \\ n {
d longo; \\ n ", SymCnt ++);
798
para ( ; t1 && t2; t1 = t1-> rgt, t2 = t2-> rgt)
799
{
if (! t1-> lft ||! t2-> lft)
800
fatal ("lista de simetria ruim", (char *) 0);
801
802
fprintf (fd, "d =");
803
putstmtnt (fd, t1-> lft, 0);
804
fprintf (fd, "-");
805
putstmtnt (fd, t2-> lft, 0);
806
fprintf (fd, "; \\ n");
807
808
fprintf (fd, "if (d <0) return -1; \\ n");
809
fprintf (fd, "if (d> 0) return 1; \\ n \\ n");

```

Página 423

```

412
APÊNDICE D
PROJETO E VALIDAÇÃO
810
}
811
fprintf (fd, "return 0; \\ n} \\ n");
812}
813
814 ismstype (str)
815
char * str;
816 {
817
Nó * n;
818
int cnt = 1;
819
820
para (n = tipo M; n; n = n-> rgt)
821
{
if (strcmp (str, n-> lft-> nsym-> nome) == 0)
822
return cnt;

```

```

823
cnt++;
824
}
825
return 0;
826}
827
828 / ***** spin: vars.c ***** /
829
830 #include <stdio.h>
831 #include "spin.h"
832 #include "y.tab.h"
833
834 extern RunList
* X;
835 int Noglobal = 0;
836
837 getval (s, n)
838 Símbolo * s;
839 {
840
if (strcmp (s-> nome, "_p") == 0)
841
return (X && X-> pc)? X-> pc-> seqno: 0;
842
if (s-> contexto && s-> tipo)
843
retornar getlocal (s, n);
844
if (Noglobal)
845
845
return 0;
846
if (! s-> type) /* não declarado localmente */
847
s = pesquisa (s-> nome); /* try global */
848
return getglobal (s, n);
849}
850
851 setval (v, n)
852
853 {
854
if (v-> nsym-> contexto && v-> nsym-> tipo)
855
retornar setlocal (v, n);
856
if (! v-> nsym-> type)
857
v-> nsym = pesquisa (v-> nsym-> nome);
858
retornar setglobal (v, n);
859}
860
861 checkvar (s, n)
862
Símbolo * s;
863 {

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

413
864
int i;
865
866
if (n> = s-> nel || n <0)
867
{
yyerror ("erro de indexação de array, '% s'", s-> nome);
868
return 0;
869
}
870

```

```

if (s-> type == 0)
871
{
yyerror ("undecl var '% s' (assumindo int)", s-> nome);
872
s-> tipo = INT;
873
}
874
if (s-> val == (int *) 0)
/* não inicializado */
875
{
s-> val = (int *) emalloc (s-> nel * sizeof (int));
876
para (i = 0; i <s-> nel; i++)
877
{
if (s-> digite! = CHAN)
878
s-> val [i] = eval (s-> ini);
879
outro
880
s-> val [i] = qmake (s);
881
}
}
882
return 1;
883}
884
885 getglobal (s, n)
886
Símbolo * s;
887 {
888
int i;
889
if (s-> type == 0 && X && (i = find_lab (s, X-> n)))
890
return i;
891
if (checkvar (s, n))
892
retornar cast_val (s-> tipo, s-> val [n]);
893
return 0;
894}
895
896 vazio
897 typex (n, t)
898
Nó * n;
899 {
900
if (n-> ntyp == NAME && n-> nsym-> type! = t
901
&& (t == CHAN || n-> nsym-> type == CHAN))
902
yyerror ("digite clash (chan) em mesg pars", 0);
903}
904
905 cast_val (t, v)
906 {int i = 0; s curto = 0; caractere sem sinal u = 0;
907
908
if (t == INT || t == CHAN) i = v;
909
else if (t == SHORT) s = (short) v;
910
else if (t == BYTE) u = (unsigned char) v;
911
senão if (t == BIT) u = (unsigned char) (v & 1);
912
913
if (v! = i + s + u)
914
yyerror ("valor% d truncado na atribuição", v);
915
return (int) (i + s + u);
916}

```

414
APÉNDICE D
PROJETO E VALIDAÇÃO
918 setglobal (v, m)
919
Nó * v;
920 {
921
int n = eval (v-> lft);
922
923
if (checkvar (v-> nsym, n))
924
v-> nsym-> val [n] = m;
925
return 1;
926}
927
928 vazio
929 dumpglobals ()
930 {símbolo externo * symtab [Nhash + 1];
931
registrar símbolo * sp;
932
registrar int i, j, k, n, m;
933
934
para (i = 0; i <= Nhash; i ++)
935
para (sp = symtab [i]; sp; sp = sp-> próximo)
936
{
if (! sp-> digite || sp-> contexto)
937
continuar;
938
para (j = 0, m = -1; j <sp-> nel; j ++)
939
{
if (sp-> type == CHAN)
940
{
doq (sp, j);
941
k = 0;
942
continuar;
943
}
944
n = getglobal (sp, j);
945
if (j == 0 || n! = k)
946
{
if (m! = j-1)
947
printf ("\t \t ... \n");
948
if (sp-> nel> 1)
949
printf ("\t \t% s [% d] =% d \n",
950
sp-> nome, j, n);
951
outro
952
printf ("\t \t% s =% d \n",
953
sp-> nome, n);
954
m = j;
955
}
956
k = n;
957

```

    }
}
958}
959
960 vazio
961 dumplocal (s)
962
Símbolo * s;
963 {
964
Símbolo * z;
965
966
int i;
967
968 para (z = s; z; z = z-> próximo)
969
para (i = 0; i <z-> nel; i++)
970
{
if (z-> type == CHAN)
971
doq (z, i);
972
outro

```

Página 426

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

415
972
{
if (z-> nel> 1)
973
printf ("\t \t% s [% d] =% d \ n",
974
z-> nome, i, getval (z, i));
975
outro
976
printf ("\t \t% s =% d \ n",
977
z-> nome, getval (z, 0));
978
}
}
979}
980
981 / ***** spin: mesg.c ***** /
982
983 #include <stdio.h>
984 #include "spin.h"
985 #incluir "y.tab.h"
986
987 #define MAXQ
2500
/* número máximo padrão de filas */
988
989 externo
int lineno, verboso;
990 Queue
* qtab = (fila *) 0;
/* lista vinculada de filas */
991 Queue
* ltab [MAXQ];
/* lista linear de filas */
992 nqs int = 0;
993 int Mpars = 0;
/* número máximo de parâmetros de mensagem */
994
995 vazio
996 cnt_mpars (n)
997
Nó * n;
998 {
999
Nó * m;
1000
int i = 0;
1001

```

```

1002
para (m = n; m; m = m-> rgt)
1003
i++;
1004
Mpars = max (Mpars, i);
1005}
1006
1007 qmake (s)
1008
Símbolo * s;
1009 {
1010
Nó * m;
1011
Fila * q;
1012
int i; análise interna externa;
1013
1014
if (! s-> ini)
1015
return 0;
1016
if (s-> ini-> ntyp! = CHAN)
1017
fatal ("inicializador de canal inválido para% s \ n", s-> nome);
1018
if (nqs> = MAXQ)
1019
fatal ("muitas filas (% s)", s-> nome);
1020
1021
q = (Fila *) emalloc (sizeof (Fila));
1022
q-> qid = ++ nqs;
1023
q-> nslots = s-> ini-> nval;
1024
para (m = s-> ini-> rgt; m; m = m-> rgt)
10: 25h
q-> nflds++;

```

Página 427

```

416
APÊNDICE D
PROJETO E VALIDAÇÃO
1026
i = max (1, q-> nslots); /* 0-slot qs obtém 1 slot mínimo */
1027
1028
q-> conteúdo = (int *) emalloc (q-> nflds * i * sizeof (int));
1029
q-> fld_width = (short *) emalloc (q-> nflds * sizeof (short));
1030
para (m = s-> ini-> rgt, i = 0; m; m = m-> rgt)
1031
q-> fld_width [i ++] = m-> ntyp;
1032
q-> nxt = qtab;
1033
qtab = q;
1034
ltab [q-> qid-1] = q;
1035
1036
return q-> qid;
1037}
1038
1039 qlen (n)
1040
Nó * n;
1041 {
1042
int qualq = eval (n-> lft) -1;
1043
1044
if (qualq <MAXQ && qualq> = 0 && ltab [qualq])
1045
return ltab [whichq] -> qlen;

```

```

1046
return 0;
1047}
1048
1049 q_is_sync (n)
1050
Nó * n;
1051 {
1052
int qualq = eval (n-> lft) -1;
1053
1054
if (qualq <MAXQ && qualq> = 0 && ltab [qualq])
1055
return (ltab [whichq] -> nslots == 0);
1056
return 0;
1057}
1058
1059 qsend (n)
1060
Nó * n;
1061 {
1062
int qualq = eval (n-> lft) -1;
1063
if (whichq == -1)
1064
{
printf ("Erro: envio para um canal não inicializado \ n");
1065
qualq = 0;
1066
}
1067
if (qualq <MAXQ && qualq> = 0 && ltab [qualq])
1068
{
if (ltab [whichq] -> nslots> 0)
1069
retornar a_snd (ltab [whichq], n);
1070
outro
1071
retornar s_snd (ltab [qualq], n);
1072
}
1073
return 0;
1074}
1075
1076 qrecv (n, completo)
1077
Nó * n;
1078 {
1079
int qualq = eval (n-> lft) -1;

```

Página 428

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

417
1080
1081
if (whichq == -1)
1082
{
printf ("Erro: recebendo de um canal não inicializado \ n");
1083
qualq = 0;
1084
}
1085
if (qualq <MAXQ && qualq> = 0 && ltab [qualq])
1086
retornar a_rcv (ltab [whichq], n, completo);
1087
return 0;
1088}
1089

```

```

1090 a_snd (q, n)
1091
1092 Fila * q;
1093 Nó * n;
1094 {
1095 Nó * m; extern int m_loss;
1096 int i = q-> qlen * q-> nflds;
/* q offset */
1097
1098 int j = 0;
/* q campo */
1099
1100 if (q-> nslots > 0 && q-> qlen >= q-> nslots)
1101 return m_loss; /* q está cheio */
1102
1103 para (m = n-> rgt; m && j <q-> nflds; m = m-> rgt, j++)
1104
1105 {
1106 q-> conteúdo [i + j] =
1107 cast_val (q-> fld_width [j], eval (m-> lft));
1108 typex (m-> lft, q-> fld_width [j]); /* após avaliação (m-> lft) */
1109 if (detalhado e 16)
1110 sr_talk (n, eval (m-> lft), "Enviar", "->", j,
1111 q-> nflds, m-> lft && m-> lft-> ntyp == CONST);
1112
1113 if (detalhado e 16)
1114
1115 para (i = j; i <q-> nflds; i++)
1116 sr_talk (n, 0, "Enviar", "->", i, q-> nflds, 0);
1117 if (detalhado e 32)
1118 {if (j <q-> nflds)
1119 printf ("\twarning: parâmetros ausentes no envio \n");
1120 if (m)
1121 printf ("\twarning: muitos parâmetros no envio \n");
1122
1123 a_rcv (q, n, completo)
1124 Fila * q;
1125 Nó * n;
1126 {
1127 Nó * m;
1128 int j, k;
1129 if (q-> qlen == 0)
1130 return 0;
/* q está vazio */
1131
1132 para (m = n-> rgt, j = 0; m && j <q-> nflds; m = m-> rgt, j++)

```

```
{  
if (m-> lft-> ntyp == CONST)  
1133  
{  
if (q-> conteúdo [j] != m-> lft-> nval)  
  
418  
APÉNDICE D  
PROJETO E VALIDAÇÃO  
1134  
return 0;  
/* sem correspondência */  
1135  
} else if (m-> lft-> ntyp != NAME)  
1136  
fatal ("argumento incorreto no recebimento", (char *) 0);  
1137  
}  
1138  
if (verboso & 8 && verboso & 32)  
1139  
{  
if (j <q-> nflds)  
1140  
printf ("\twarning: parâmetros ausentes no próximo recebimento \n");  
1141  
else if (m)  
1142  
printf ("\twarning: muitos parâmetros no próximo recebimento \n");  
1143  
}  
1144  
para (m = n-> rgt, j = 0; j <q-> nflds; m = (m)? m-> rgt: m, j++)  
1145  
{  
if (verboso & 8)  
1146  
sr_talk (n, q-> conteúdo [j], (completo)? "Recv": "[Recv]", "<-", j,  
1147  
q-> nflds, m && m-> lft-> ntyp == CONST);  
1148  
if (m && m-> lft-> ntyp == NOME)  
1149  
{  
setval (m-> lft, q-> conteúdo [j]);  
1150  
typex (m-> lft, q-> fld_width [j]);  
1151  
}  
1152  
para (k = 0; completo && k <q-> qlen-1; k++)  
1153  
q-> conteúdo [k * q-> nflds + j] =  
1154  
q-> conteúdo [(k + 1) * q-> nflds + j];  
1155  
}  
1156  
if (completo) q-> qlen--;  
1157  
return 1;  
1158}  
1159  
1160 s_snd (q, n)  
1161  
Fila * q;  
1162  
Nó * n;  
1163 {  
1164  
Nó * m;  
1165  
int i, j = 0; /* q campo # */  
1166  
1167  
para (m = n-> rgt; m && j <q-> nflds; m = m-> rgt, j++)  
1168  
q-> conteúdo [j] = cast_val (q-> fld_width [j], eval (m-> lft));  
1169
```

```

1170
q-> qlen = 1;
1171
if (! complete_rendez ())
1172
{
q-> qlen = 0;
1173
return 0;
1174
}
1175
if (detalhado e 16)
1176
{
m = n-> rgt;
1177
para (j = 0; m && j <q-> nflds; m = m-> rgt, j++)
1178
{
sr_talk (n, eval (m-> lft), "Enviado", "->", j,
1179
q-> nflds, m-> lft && m-> lft-> ntyp == CONST);
1180
typex (m-> lft, q-> fld_width [j]);
1181
}
1182
para (i = j; i <q-> nflds; i++)
1183
sr_talk (n, 0, "Enviado", "->", i, q-> nflds, 0);
1184
if (detalhado e 32)
1185
{
if (j <q-> nflds)
1186
printf ("\twarning: parâmetros ausentes no envio \n");
1187
if (m)

```

Página 430

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

419
1188
printf ("\twarning: muitos parâmetros no envio \n");
1189
}
}
}
1190
return 1;
1191}
1192
1193 vazio
1194 sr_talk (n, v, s, a, j, mx, nomeado)
1195
Nó * n;
1196
char * s, * a;
1197 {
1198
extern int Named;
1199
if (j == 0)
1200
{
quem corre();
1201
printf ("linha% 3d,% s", n-> nval, s);
1202
sr_mesg (v, nomeado || Nomeado);
1203
} outro
1204
{
printf ",";
1205
sr_mesg (v, nomeado);
1206

```

```

}
1207
if (j == mx-1)
1208
{
printf ("\t% s fila% d", a, eval (n-> lft));
1209
if (n-> nsym-> type == CHAN)
1210
printf ("(% s", n-> nsym-> nome);
1211
outro
1212
printf ("(% s", lookup (n-> nsym-> nome) -> nome);
1213
if (n-> lft-> lft)
1214
printf ("[% d]", eval (n-> lft-> lft));
1215
printf ("") \ n");
1216
}
1217
fflush (stdout);
1218}
1219
1220 vazio
1221 sr_mesg (v, j)
1222 {Nó externo * Mtype;
1223
1224
int cnt = 1;
12: 25h
Nó * n;
1226
para (n = Mtype; n && j; n = n-> rgt, cnt++)
1227
if (cnt == v)
1228
{
printf ("% s", n-> lft-> nsym-> nome);
1229
Retorna;
1230
}
1231
printf ("% d", v);
1232}
1233
1234 vazio
1235 doq (s, n)
1236
Símbolo * s;
1237 {
1238
Fila * q;
1239
int j, k;
1240
if (! s-> val)
/* fila não inicializada */
1241
Retorna;

```

```

420
APÊNDICE D
PROJETO E VALIDAÇÃO
1242
para (q = qtab; q; q = q-> nxt)
1243
if (q-> qid == s-> val [n])
1244
{
if (s-> nel! = 1)
1245
printf ("\t \ tqueue% d (% s [% d]):", q-> qid, s-> nome, n);
1246
outro
1247

```

```

printf ("\t \tqueue% d (% s):", q-> qid, s-> nome);
1248
para (k = 0; k <q-> qlen; k++)
1249
{
printf ("[" );
1250
para (j = 0; j <q-> nflds; j++)
1251
{
if (j> 0) printf (",");
1252
sr_mesg (q-> conteúdo [k * q-> nflds + j], j == 0);
1253
}
1254
printf ("]");
1255
}
1256
printf ("\n");
1257
quebrar;
1258
}
1259}
1260
1261 / ***** spin: flow.c ***** /
1262
1263 #include "spin.h"
1264 #include "y.tab.h"
1265
Etiqueta 1266
* labtab = (Etiqueta *) 0;
1267 Lbreak
* breakstack = (Lbreak *) 0;
1268 SeqList
* cur_s = (SeqList *) 0;
1269 int
Elcnt, break_id = 0;
1270
1271 vazio
1272 open_seq (topo)
1273 {SeqList * t;
1274
Sequência * s = (Sequência *) emalloc (sizeof (Sequência));
1275
1276
t = seqlist (s, cur_s);
1277
cur_s = t;
1278
if (topo) Elcnt = 1;
1279}
1280
Sequência 1281 *
1282 close_seq ()
1283 {Sequência * s = cur_s-> isto;
1284
Símbolo * z;
1285
1286
if (s-> primeiro == s-> último)
1287
{
if ((z = has_lab (s-> primeiro))
1288
&& (strncmp (z-> nome, "progresso", 8) == 0
1289
|| strncmp (z-> nome, "aceitar", 6) == 0
1290
|| strncmp (z-> nome, "fim", 3) == 0))
1291
{
Elemento * y =
/* insira um salto */
1292
new_el (nn ((simbolo *) 0, s-> primeiro-> n-> nval, 'c',
1293
nn ((simbolo *) 0, 1, CONST, (nó *) 0,
1294
(Nó *) 0), (Nó *) 0));

```

1295
if (s-> primeiro-> n-> ntyp == GOTO

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN
421
1296 || s-> primeiro-> n-> ntyp == BREAK)
1297 {
1298 s-> primeiro = y;
1299 y-> nxt = s-> último;
1300 } outro
1301 {
1302 mov_lab (z, s-> primeiro, y);
1303 s-> primeiro-> nxt = y;
1304 s-> último = y;
1305 }
1306 }
1307
1308 SeqList *
1309 seqlist (s, r)
1310 Sequência * s;
1311 SeqList * r;
1312 {
1313
SeqList * t = (SeqList *) emalloc (sizeof (SeqList));
1314 t-> this = s;
1315 t-> nxt = r;
1316 return t;
1317 }
1318
1319 Elemento *
1320 new_el (n)
1321 Nó * n;
1322 {
1323
Elemento * m;
1324
1325 if (n && (n-> ntyp == IF || n-> ntyp == DO))
1326 retornar if_seq (n-> seql, n-> ntyp, n-> nval);
1327 m = (Elemento *) emalloc (sizeof (Elemento));
1328 m-> n = n;
1329 m-> seqno = Elcnt ++;
1330 return m;
1331 }
1332
1333 Elemento *
1334 if_seq (s, tok, lnno)
1335
SeqList * s;
1336 {
1337
Elemento * e = new_el ((Nó *) 0);
1338

```

Elemento * t = new_el (nn ((Símbolo *) 0, lnno, '.',
1339
(Nó *) 0, (Nó *) 0)); /* alvo */
1340
SeqList * z;
1341
1342
e-> n = nn ((Símbolo *) 0, lnno, tok, (Nó *) 0, (Nó *) 0);
1343
e-> sub = s;
1344
para (z = s; z; z = z-> nxt)
1345
add_el (t, z-> this);
1346
if (tok == DO)
1347
{
add_el (t, cur_s-> this);
1348
t = new_el (nn ((Símbolo *) 0, lnno, BREAK, (Nó *) 0, (Nó *) 0));
1349
set_lab (break_dest (), t);

```

Página 433

```

422
APÉNDICE D
PROJETO E VALIDAÇÃO
1350
breakstack = breakstack-> nxt; /* pop stack */
1351
}
1352
add_el (e, cur_s-> this);
1353
add_el (t, cur_s-> this);
1354
return e;
/* nó de destino para rótulo */
1355
1356
1357 vazio
1358 add_el (e, s)
1359
Elemento * e;
1360
Sequência * s;
1361 {
1362
if (e-> n-> ntyp == GOTO)
1363
{
Símbolo * z;
1364
if ((z = has_lab (e))
1365
&& (strncmp (z-> nome, "progresso", 8) == 0
1366
|| strncmp (z-> nome, "aceitar", 6) == 0
1367
|| strncmp (z-> nome, "fim", 3) == 0))
1368
{
Elemento * y =
/* insira um salto */
1369
new_el (nn ((Símbolo *) 0, e-> n-> nval, 'c',
1370
nn ((Símbolo *) 0, 1, CONST, (Nó *) 0,
1371
(Nó *) 0), (Nó *) 0));
1372
mov_lab (z, e, y); /* recebe seu rótulo */
1373
add_el (y, s);
1374
}
}
1375
if (! s-> primeiro)

```

```

1376
s-> primeiro = e;
1377
outro
1378
s-> último-> nxt = e;
1379
s-> último = e;
1380}
1381
1382 Nó
* mais interno;
1383
1384 Element *
1385 dois pontos (n)
1386
Nó * n;
1387 {
1388
if (! n)
1389
return (Element *) 0;
1390
if (n-> ntyp == ':')
1391
{
Elemento * e = dois pontos (n-> lft);
1392
set_lab (n-> nsym, e);
1393
return e;
1394
}
1395
mais interno = n;
1396
retornar novo_el (n);
1397}
1398
1399 vazio
1400 add_seq (n)
1401
Nó * n;
1402 {
1403
Elemento * e;

```

Página 434

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

423
1404
if (! n) return;
1405
mais interno = n;
1406
e = dois pontos (n);
1407
if (mais interno-> ntyp! = IF && mais interno-> ntyp! = DO)
1408
add_el (e, cur_s-> this);
1409}
1410
1411 vazio
1412 set_lab (s, e)
1413
Símbolo * s;
1414
Elemento * e;
1415 {
1416
Rótulo * l; contexto externo do símbolo *;
1417
if (! s) return;
1418
l = (Label *) emalloc (sizeof (Label));
1419
l-> s = s;
1420
l-> c = contexto;

```

```

1421
l-> e = e;
1422
1423
labtab = l;
1424}
1425
1426 Elemento *
1427 get_lab (s)
1428
Símbolo * s;
1429 {
1430
Rótulo * l;
1431
para (l = labtab; l; l = l-> nxt)
1432
if (s == l-> s)
1433
retorno (l-> e);
1434
fatal ("rótulo indefinido% s", s-> nome);
1435
return 0;
/* não chega aqui */
1436}
1437
1438 Symbol *
1439 has_lab (e)
1440
Elemento * e;
1441 {
1442
Rótulo * l;
1443
para (l = labtab; l; l = l-> nxt)
1444
if (e == l-> e)
1445
retorno (l-> s);
1446
retorno (símbolo *) 0;
1447}
1448
1449 vazio
1450 mov_lab (z, e, y)
1451
Símbolo * z;
1452
Elemento * e, * y;
1453 {
1454
Rótulo * l;
1455
para (l = labtab; l; l = l-> nxt)
1456
if (e == l-> e)
1457
{
l-> e = y;

```

Página 435

```

424
APÊNDICE D
PROJETO E VALIDAÇÃO
1458
1459
Retorna;
1460
}
1460
fatal ("não pode acontecer - mov_lab% s", z-> nome);
1461}
1462
1463 find_lab (s, c)
1464
1464
Símbolo * s, * c;
1465 {
1466
Rótulo * l;

```

```

1467
para (l = labtab; l; l = l-> nxt)
1468
{
if (strcmp (s-> nome, l-> s-> nome) == 0
1469
&& strcmp (c-> nome, l-> c-> nome) == 0)
1470
retorno (l-> e-> seqno);
1471
}
1472
return 0;
1473}
1474
1475 vazio
1476 pushbreak ()
1477 {Lbreak * r = (Lbreak *) emalloc (sizeof (Lbreak));
1478
Símbolo * l;
1479
char buf [32];
1480
1481
sprintf (buf, ": b% d", break_id++);
1482
l = pesquisa (buf);
1483
r-> l = l;
1484
r-> nxt = breakstack;
1485
breakstack = r;
1486}
1487
Símbolo 1488 *
1489 break_dest ()
1490 {if (! Breakstack)
1491
fatal ("declaração de interrupção perdida", (char *) 0);
1492
return breakstack-> l;
1493}
1494
1495 vazio
1496 make_atomic (s)
1497
Sequência * s;
1498 {
1499
walk_atomic (s-> primeiro, s-> último);
1500
s-> último-> status &= ~ATOM;
1501
s-> último-> status |= L_ATOM;
1502}
1503
1504 vazio
1505 caminhada_atômica (a, b)
1506
Elemento * a, * b;
1507 {
1508
Elemento * f;
1509
SeqList * h;
1510
para (f = a;; f = f-> nxt)
1511
{
f-> status |= ATOM;

```

```

1514
1515     if (f == b)
1516     quebrar;
1517 }
1518 / ***** spin: run.c ***** /
1519
1520 #include <stdio.h>
1521 #include "spin.h"
1522 #include "y.tab.h"
1523
1524 Elemento * 1525 *
1525 eval_sub (e)
1526
1527 Elemento * e;
1528 {
1529
1530     Elemento * f, * g;
1531     SeqList * z;
1532
1533     int i, j, k;
1534
1535     extern int Rvous, lineno;
1536
1537     if (! e-> n)
1538         return (Element *) 0;
1539
1540     if (e-> n-> ntyp == GOTO)
1541         return (! Rvous) ? get_lab (e-> n-> nsym) :( Elemento *) 0;
1542
1543     if (e-> sub)
1544     {
1545         para (z = e-> sub, j = 0; z; z = z-> nxt)
1546         j++;
1547         k = rand ()% j; /* não determinismo */
1548         para (i = 0, z = e-> sub; i <j + k; i++)
1549         {
1550             if (i>= k && (f = eval_sub (z-> this-> primeiro)))
1551                 return f;
1552             z = (z-> nxt)? z-> nxt: e-> sub;
1553         }
1554     } outro
1555     {
1556         if (e-> n-> ntyp == ATOMIC)
1557         {
1558             f = e-> n-> seql-> this-> primeiro;
1559             g = e-> n-> seql-> this-> last;
1560             g-> nxt = e-> nxt;
1561             if (! (g = eval_sub (f))) /* guarda atômica */
1562                 return (Element *) 0;
1563             Rvous = 0;
1564             while (g && (g-> status & (ATOM | L_ATOM)))
1565                 &&! (f-> status & L_ATOM)
1566             {
1567                 f = g;
1568             }

```

```

g = eval_sub (f);
1559
}
1560
if (! g)
1561
{
embrulhar();
1562
lineno = f-> n-> nval;
1563
fatal ("blocos seq atômicos", (char *) 0);
1564
}
1565
return g;

```

Página 437

```

426
APÊNDICE D
PROJETO E VALIDAÇÃO
1566
} else if (Rvous)
1567
{
if (eval_sync (e-> n))
1568
return e-> nxt;
1569
} outro
1570
return (eval (e-> n)) ? e-> nxt: (Element *) 0;
1571
}
1572
return (Element *) 0;
1573}
1574
1575 eval_sync (agora)
1576
Nó * agora;
1577 /* permitir apenas recebimentos síncronos
1578
/* e tipos de nós relacionados
*/
1579
1580
se (agora)
1581
switch (agora-> ntyp) {
1582
case TIMEOUT: case PRINT:
case ASSERT:
1583
case RUN:
case LEN:
case 's':
1584
case 'c':
caso ASGN:
case BREAK:
1585
case IF:
case DO:
case '.':
1586
return 0;
1587
case 'R':
1588
case 'r':
1589
if (! q_is_sync (agora))
1590
return 0;
1591
}
1592
return eval (agora);
1593}

```

```

1594
1595 eval (agora)
1596
Nó * agora;
1597 {
1598
extern int Tval;
1599
1600
se (agora)
1601
switch (agora-> ntyp) {
1602
case CONST: retorna agora-> nval;
1603
case '!': return! eval (now-> lft);
1604
case UMIN: return -eval (now-> lft);
1605
case '^': retorna ^eval (now-> lft);
1606
1607
case '/': return (eval (now-> lft) / eval (now-> rgt));
1608
case '*': return (eval (now-> lft) * eval (now-> rgt));
1609
case '-': return (eval (now-> lft) - eval (now-> rgt));
1610
case '+': return (eval (now-> lft) + eval (now-> rgt));
1611
case '%': return (eval (now-> lft)% eval (now-> rgt));
1612
case '<': return (eval (now-> lft) <eval (now-> rgt));
1613
case '>': return (eval (now-> lft)> eval (now-> rgt));
1614
case '&': return (eval (now-> lft) & eval (now-> rgt));
1615
case '|': return (eval (now-> lft) | eval (now-> rgt));
1616
caso
LE: return (eval (now-> lft) <= eval (now-> rgt));
1617
caso
GE: return (eval (now-> lft)> = eval (now-> rgt));
1618
caso
NE: return (eval (now-> lft) != Eval (now-> rgt));
1619
caso
EQ: return (eval (now-> lft) == eval (now-> rgt));

```

Página 438

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

427
1620
caso
OU: return (eval (now-> lft) || eval (now-> rgt));
1621
case AND: return (eval (now-> lft) && eval (now-> rgt));
1622
case LSHIFT: return (eval (now-> lft) << eval (now-> rgt));
1623
case RSHIFT: return (eval (now-> lft) >> eval (now-> rgt));
1624
1625
case TIMEOUT: return Tval;
1626
1627
case RUN: return enable (now-> nsym, now-> lft);
1628
case LEN: return qlen (agora);
1629
case 's': return qsend (agora);
/* enviar
 */
1630
case 'r': return qrecv (agora, 1);
/* full-receive */

```

```

1631
case 'R': return qrecv (agora, 0);
/* teste apenas
 */
1632
case 'c': return eval (now-> lft);
/* doença
 */
1633
case 'p': return remotevar (agora);
1634
case 'q': retorna remotelab (agora);
1635
case PRINT: return interprint (agora);
1636
case ASGN: return setval (now-> lft, eval (now-> rgt));
1637
case NAME: return getval (now-> nsym, eval (now-> lft));
1638
case ASSERT: if (eval (now-> lft)) return 1;
1639
yyerror ("assertion violated", (char *) 0);
1640
embrulhar(); saída (1);
1641
case IF: case DO: case BREAK: /* estrutura composta */
1642
case '.': return 1; /* rótulo de retorno para o composto */
1643
case '@': retorna 0; /* estado de parada */
1644
padrão: printf ("spin: tipo de nó inválido% d (executar) \ n", agora-> ntyp);
1645
fflush (stdout);
1646
saída (1);
1647
}
1648
return 0;
1649}
1650
1651 interprint (n)
1652
Nó * n;
1653 {
1654
Nó * tmp = n-> lft;
1655
char c, * s = n-> nsym-> nome;
1656
int i, j;
1657
1658
para (i = 0; i <strlen (s); i++)
1659
switch (s [i]) {
1660
padrão: putchar (s [i]); quebrar;
1661
case '\ ''': break; /* ignorar */
1662
case '\\':
1663
switch (s [++ i]) {
1664
case 't': putchar ('\ t'); quebrar;
1665
case 'n': putchar ('\ n'); quebrar;
1666
padrão: putchar (s [i]); quebrar;
1667
}
1668
quebrar;
1669
caso '%':
1670
if ((c = s [++ i]) == '%')
1671
{
putchar ('%'); /* literal */

```

```
1672  
quebrar;  
1673  
}
```

```
428  
APÊNDICE D  
PROJETO E VALIDAÇÃO  
1674  
if (! tmp)  
1675  
{  
yyerror ("muito poucos argumentos de impressão% s", s);  
1676  
quebrar;  
1677  
}  
1678  
j = eval (tmp-> lft);  
1679  
tmp = tmp-> rgt;  
1680  
switch (c) {  
1681  
case 'c': printf ("% c", j); quebrar;  
1682  
case 'd': printf ("% d", j); quebrar;  
1683  
case 'o': printf ("% o", j); quebrar;  
1684  
case 'u': printf ("% u", j); quebrar;  
1685  
case 'x': printf ("% x", j); quebrar;  
1686  
padrão: yyerror ("unrecognized print cmd %% '% c'", c);  
1687  
quebrar;  
1688  
}  
1689  
quebrar;  
1690  
}  
1691  
fflush (stdout);  
1692  
return 1;  
1693}  
1694  
1695 / ***** spin: sched.c ***** /  
1696  
1697 #include <stdio.h>  
1698 #include "spin.h"  
1699 #include "y.tab.h"  
1700  
1701 int nproc = 0;  
1702 int nstop = 0;  
1703 int Tval = 0;  
1704 int Rvous = 0;  
Profundidade de 1705 int = 0;  
1706  
1707 RunList  
* X = (RunList *) 0;  
1708 RunList  
* run = (RunList *) 0;  
1709 ProcList * rdy = (ProcList *) 0;  
Elemento 1710  
* eval_sub ();  
1711 extern int verbose, lineno, s_trail, analise;  
1712 simbolo externo  
* Fname;  
1713 caracteres externos  
* Claimproc;  
1714 extern int Noglobal;  
1715 int Have_claim = 0;  
1716  
1717 vazio  
1718 executável (s, n)  
1719
```

```

Sequência * s;
/* body */
1720
Símbolo * n;
/* nome */
1721 {
1722
RunList * r = (RunList *) emalloc (sizeof (RunList));
1723
r-> n = n;
1724
r-> pid = nproc++;
1725
r-> pc = s-> primeiro;
1726
r-> maxseq = s-> último-> seqno;
1727
r-> nxt = executar;

```

Página 440

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

429
1728
run = r;
1729}
1730
1731 vazio
1732 pronto (n, p, s)
1733
Símbolo * n;
/* nome do processo */
1734
Nó * p;
/* parâmetros formais */
1735
Sequência * s;
/* corpo do processo */
1736 {
1737
ProcList * r = (ProcList *) emalloc (sizeof (ProcList));
1738
r-> n = n;
1739
r-> p = p;
1740
r-> s = s;
1741
r-> nxt = rdy;
1742
rdy = r;
1743}
1744
1745 habilitar (s, n)
1746
Símbolo * s;
/* nome do processo */
1747
Nó * n;
/* parâmetros reais */
1748 {
1749
ProcList * p;
1750
para (p = rdy; p; p = p-> nxt)
1751
if (strcmp (s-> nome, p-> n-> nome) == 0)
1752
{
executável (p-> s, p-> n);
1753
setparams (executar, p, n);
1754
retorno (nproc-nstop-1); /* pid */
1755
}
1756
return 0; /* processo não encontrado */
1757}
1758

```

```

1759 vazio
1760 start_claim (n)
1761 {ProcList * p;
1762 int i;
1763
1764 para (p = rdy, i = 1; p; p = p-> nxt, i++)
1765
if (i == n)
1766 {
executável (p-> s, p-> n);
1767
Have_claim = 1;
1768
Retorna;
1769 }
1770 fatal ("não foi possivel encontrar reivindicação", (char *) 0);
1771}
1772
1773 vazio
1774 sched ()
1775 {Elemento * e;
1776
RunList * Y;
/* processo anterior na fila de execução */
1777
int i;
1778
1779 se (analisar)
1780 {
gensrc ();
1781
Retorna;

```

```

430
APÊNDICE D
PROJETO E VALIDAÇÃO
1782
} else if (s_trail)
1783
{
match_trail ();
1784
Retorna;
1785
}
1786
if (Claimproc)
1787
printf ("aviso: as reivindicações são ignoradas nas simulações \ n");
1788
1789
para (Tval = i = 0; Tval <2; Tval ++, i = 0)
1790
{
enquanto (i <nproc-nstop)
1791
para (X = executar, Y = 0, i = 0; X; X = X-> nxt)
1792
{
lineno = X-> pc-> n-> nval;
1793
Fname = X-> pc-> n-> fname;
1794
if (e = eval_sub (X-> pc))
1795
{
X-> pc = e; Tval = 0;
1796
talk (e, X-> symtab);
1797
} outro
1798

```

```

{
if (X-> pc-> n-> ntyp == '@'
1799
&& X-> pid == (nproc-nstop-1))
1800
{
if (Y)
1801
Y-> nxt = X-> nxt;
1802
outro
1803
executar = X-> nxt;
1804
nstop++; Tval = 0;
1805
if (detalhado e 4)
1806
{
quem corre();
1807
printf ("termina \ n");
1808
}
1809
} outro
1810
i++;
1811
}
1812
Y = X;
1813
}
}
1814
embrulhar();
1815}
1816
Finalização de 1817 ()
1818 {if (profundidade)
/* para simulações guiadas, Capítulo 12 */
1819
printf ("etapa% d,", profundidade);
1820
if (nproc! = nstop)
1821
{
printf ("# processos:% d \ n", nproc-nstop);
1822
dumpglobals ();
1823
verboso & = ~1; /* no more globals */
1824
verboso |= 4; /* adicionar estados de processo */
1825
para (X = executar; X; X = X-> nxt)
1826
talk (X-> pc, X-> symtab);
1827
}
1828
printf ("% d processos criados \ n", nproc);
1829}
1830
1831 complete_rendez ()
1832 {RunList * orun = X;
1833
Elemento * e;
1834
int res = 0;
1835

```

```

1837
return 1;
1838
Rvous = 1;
1839
para (X = executar; X; X = X-> nxt)
1840
if (X! = orun && (e = eval_sub (X-> pc)))
1841
{
X-> pc = e;
1842
if (detalhado e 4)
1843
{
printf ("rendezvous:% s", X-> n-> nome);
1844
printf ("<->% s \ n", orun-> n-> nome);
1845
printf ("= r ==:");
1846
talk (e, X-> symtab);
1847
printf ("= s ==:");
1848
X = orun;
1849
talk (X-> pc, X-> symtab);
1850
}
1851
res = 1;
1852
quebrar;
1853
}
1854
Rvous = 0;
1855
X = orun;
1856
return res;
1857}
1858
1859 / ***** Runtime - Local Variables ***** /
1860
1861 vazio
1862 adiciona simbolo (r, s)
1863
RunList * r;
1864
Simbolo * s;
1865 {
1866
Simbolo * t = (Simbolo *) emalloc (sizeof (Simbolo));
1867
int i;
1868
1869
t-> nome = s-> nome;
1870
t-> tipo = s-> tipo;
1871
t-> nel = s-> nel;
1872
t-> ini = s-> ini;
1873
if (s-> val)
/* se inicializado, copie-o */
1874
{
t-> val = (int *) emalloc (s-> nel * sizeof (int));
1875
para (i = 0; i <s-> nel; i++)
1876
t-> val [i] = s-> val [i];
1877
} outro
1878
checkvar (t, 0); /* inicializa-o */
1879
t-> proximo = r-> symtab;

```

```

/* * adicionar * /
1880
r-> symtab = t;
1881}
1882
1883 vazio
1884 naddsymbol (r, s, k)
1885
RunList * r;
1886
Símbolo * s;
1887 {
1888
Símbolo * t = (Símbolo *) emalloc (sizeof (Símbolo));
1889
int i;

```

Página 443

```

432
APÊNDICE D
1890
1891
t-> nome = s-> nome;
1892
t-> tipo = s-> tipo;
1893
t-> nel = s-> nel;
1894
t-> ini = s-> ini;
1895
t-> val = (int *) emalloc (s-> nel * sizeof (int));
1896
if (s-> nel! = 1)
1897
fatal ("array na lista de parâmetros formal,% s", s-> nome);
1898
para (i = 0; i <s-> nel; i++)
1899
t-> val [i] = k;
1900
t-> próximo = r-> symtab;
1901
r-> symtab = t;
1902}
1903
Typck 1904 (n, t, s)
1905
Nó * n;
1906
char * s;
1907 {
1908
if (! n ||! n-> lft
1909
|| (n-> lft-> ntyp == NOME && n-> lft-> nsym-> tipo! = t
1910
&& n-> lft-> nsym-> type! = 0
1911
&& (t == CHAN || n-> lft-> nsym-> type == CHAN))
1912
|| (n-> lft-> ntyp == NOME && n-> lft-> nsym-> tipo == 0
1913
&& lookup (n-> lft-> nsym-> nome) -> tipo! = t))
1914
{
yyerror ("erro nos parâmetros da execução% s (...)", s);
1915
return 0;
1916
}
1917
return 1;
1918}
1919
1920 vazio
Setparams 1921 (r, p, q)
1922
RunList * r;
1923
ProcList * p;

```

```

1924
Nó * q;
1925 {
1926
Nó * f, * a;
/* pars formal e real */
1927
Nó * t;
/* lista de pars de 1 tipo */
1928
1929
para (f = p->p, a = q; f; f = f->rgt) /* um tipo de cada vez */
1930
para (t = f->lft; t; t = t->rgt, a = (a)? a->rgt: a)
1931
{
int k;
1932
if (!a) fatal ("parâmetros reais ausentes: '% s'", p->n->nome);
1933
k = eval (a->lft);
/* deve ser inicializado */
1934
if (typck (a, t->nsym->tipo, p->n->nome))
1935
{
if (t->nsym->type == CHAN)
1936
naddsymbol (r, t->nsym, k); /* cópia de */
1937
outro
1938
{
t->nsym->ini = a->lft;
1939
adiciona símbolo (r, t->nsym);
1940
}
1941
}
1942
}
1943}

```

Página 444

DE PROTOCOLOS DE COMPUTADOR
FONTE DO SIMULADOR DA VERSÃO 0 DO SPIN

```

433
1944
Símbolo de 1945 *
1946 findloc (s, n)
1947
Símbolo * s;
1948 {
1949
Símbolo * r = (Símbolo *) 0;
1950
1951
if (n>= s->nel || n <0)
1952
{
yyerror ("erro de indexação de array% s", s->nome);
1953
retorno (símbolo *) 0;
1954
}
1955
1956
if (!x)
1957
{
se (analisar)
1958
fatal ("erro, não é possível avaliar a variável '% s'", s->nome);
1959
outro
1960
yyerror ("erro, não é possível avaliar a variável '% s'", s->nome);
1961
retorno (símbolo *) 0;

```

```

1962
}
1963
para (r = X-> symtab; r; r = r-> próximo)
1964
if (strcmp (r-> nome, s-> nome) == 0)
1965
quebrar;
1966
if (! r && ! Noglobal)
1967
{
adiciona símbolo (X, s);
1968
r = X-> symtab;
1969
}
1970
return r;
1971}
1972
1973 getlocal (s, n)
1974
Símbolo * s;
1975 {
1976
Símbolo * r;
1977
1978
r = findloc (s, n);
1979
if (r) retorna cast_val (r-> tipo, r-> val [n]);
1980
return 0;
1981}
1982
Setlocal 1983 (p, m)
1984
Nó * p;
1985 {
1986
int n = eval (p-> lft);
1987
Símbolo * r = findloc (p-> nsym, n);
1988
1989
se (r) r-> val [n] = m;
1990
return 1;
1991}
1992
Vazio de 1993
Putas de 1994 ()
1995 {if (! X) return;
1996
1997
if (Have_claim && X-> pid> = 1)

```

Página 445

```

434
APÊNDICE D
PROJETO E VALIDAÇÃO
1998
{
if (X-> pid == 1)
1999
printf ("proc - (% s)
", X-> n-> nome);
2000
outro
2001
printf ("proc% 2d (% s)", X-> pid-1, X-> n-> nome);
2002
} outro
2003
printf ("proc% 2d (% s)", X-> pid, X-> n-> nome);
2004}
2005
2006 void
Palestra de 2007 (e, s)

```

```

2008
Elemento * e;
2009
Símbolo * s;
2010 {
2011
if (detalhado & 4)
2012
{
p_talk (e);
2013
if (verboso & 1) dumpglobals ();
2014
if (verbose & 2) dumplocal (s);
2015
}
2016}
2017
2018 vazio
2019 p_talk (e)
2020
Elemento * e;
2021 {
2022
quem corre();
2023
printf ("linha% d (estado% d) \ n",
2024
(e && e-> n && e-> n-> nval) ? e-> n-> nval: -1, e-> seqno);
2025}
2026
2027 remotelab (n)
2028
Nó * n;
2029 {
2030
int i;
2031
2032
if (n-> nsym-> tipo)
2033
fatal ("não é um nome de rótulo: '% s'", n-> nsym-> nome);
2034
if ((i = find_lab (n-> nsym, n-> lft-> nsym)) == 0)
2035
fatal ("labelname desconhecido:% s", n-> nsym-> nome);
2036
return i;
2037}
2038
2039 remotevar (n)
2040
Nó * n;
2041 {
2042
int pno, i, j, truque = 0;
2043
RunList * Y, * oX = X;
2044
2045
if (! n-> lft-> lft)
2046
{
yyerror ("faltando pid em% s", n-> nsym-> nome);
2047
return 0;
2048
}
2049
pno = eval (n-> lft-> lft); /* pid */
2050 TryAgain:
2051
i = nproc - nstop;

```

```

2053
if (--i == pno)
2054
{
if (strcmp (Y-> n-> nome, n-> lft-> nsym-> nome))
2055
{
if (! trick && Have_claim)
2056
{
truque = 1; pno++;
2057
/* assume que o usuário adivinhou apenas o pid */
2058
goto TryAgain;
2059
}
2060
printf ("referência remota% s [% d] refere-se a% s \ n",
2061
n-> lft-> nsym-> nome, pno, Y-> n-> nome);
2062
yyerror ("proctype% s errado", Y-> n-> nome);
2063
}
2064
{extern int Noglobal;
2065
Noglobal = 1; /* certifique-se de que não é criado por padrão */
2066
if (n-> nsym-> tipo == 0) n-> nsym-> tipo = INT;
2067
X = Y; j = getval (n-> nsym, eval (n-> rgt)); X = oX;
2068
Noglobal = 0;
2069
}
2070
return j;
2071
}
2072
printf ("ref remoto:% s [% d]", n-> lft-> nsym-> nome, pno);
2073
yyerror ("variável% s não encontrada", n-> nsym-> nome);
2074
return 0;
2075}
2076
2077 / ***** spin: dummy.c ***** /
2078
2079 gensrc ()
2080 {
2081
printf ("analisar: não definido \ n");
2082}
2083
2084 match_trail ()
2085 {
2086
printf ("trilhas: não definidas \ n");
2087}

```

SPIN VERSÃO 0 FONTE DO VALIDADOR E

As listas de programas que se seguem são os segmentos do programa que são adicionados ao simulador do localizador descrito no Capítulo 12 e listado no Apêndice D. O código deste apêndice é usado para gerar um validador específico de protocolo para qualquer validação de protocolo modelo que é descrito na PROMELA . As extensões são discutidas no Capítulo 13.

O novo *makefile* para esta versão do SPIN tem a seguinte aparência.

```

CC = cc
# Compilador ANSI C
CFLAGS = -O
# otimizador
YFLAGS = -v -d -D # criar y.output, y.debug e y.tab.h

```

```

OFILES = spin.o lex.o sym.o vars.o main.o debug.o \
mesg.o flow.o sched.o run.o pangen1.o pangen2.o \
pangen3.o pangen4.o pangen5.o
rotação: $ (OFILES)
$ (CC) $ (CFLAGS) -o spin $ (OFILES) -lm
%.o:
%.c spin.h
$ (CC) $ (CFLAGS) -c %.c
pangen1.o:
pangen1.c pangen1.h pangen3.h
pangen2.o:
pangen2.c pangen2.h

```

O restante deste Apêndice lista o conteúdo dos 8 arquivos de origem adicionais (consulte Tabela E.1). Uma grande parte do código está contido em arquivos de cabeçalho e copiado em um programa

validador específico de tocol gerado com SPIN .

Duas diretivas de pré-processador são geradas para manipulação opcional pelo usuário. Por padrão, todos os validadores gerados pelo SPIN realizam uma pesquisa exaustiva. Se o nome BITSTATE é definido em tempo de compilação, esta estratégia de pesquisa é substituída por um super análise de traços (consulte o Capítulo 14 para exemplos). Da mesma forma, por padrão, não há máximo predefinido para a quantidade de memória que uma análise exaustiva pode usar. Se, no entanto, o nome MEMCNT for definido em tempo de compilação, seu valor numérico será usado para definir um limite superior. Se, por exemplo, MEMCNT = 20, o limite superior usado é 2^{20} bytes (veja também o Capítulo 14 para exemplos).

436

Página 448

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

437

Tabela E.1 - Índice do arquivo de origem

Arquivo	Número da linha
pangen1.c	
1148	
pangen1.h	
1	
pangen2.h	
909	
pangen2.c	
1574	
pangen3.c	
2096	
pangen3.h	
1038	
pangen4.c	
2201	
pangen5.c	
2381	

VERSÃO ONLINE DO SPIN

O código-fonte listado nos Apêndices D e E deste livro documenta a versão 0 fontes de SPIN . Essas fontes foram originalmente distribuídas apenas por meio da AT&T Sistema de distribuição de software Toolchest, por uma taxa. A versão mais recente e estendida de SPIN está disponível gratuitamente para pesquisa e uso educacional na web via Página inicial da SPIN : <http://netlib.bell-labs.com/netlib/spin/whatispin.html> Mais informações relacionadas ao SPIN , sobre workshops, boletins informativos e documentação online ção, está disponível nesta página.

Tabela E.2 - Procedimentos listados - Apêndice E

Procedimento
Procedimento de linha
Linha

qualquer _ proc (agora)
2298 qualquer _ desfazer (agora)
2281
blurb (fd, t, n)
2071 verificar _ proc (agora, m)
2308
d _ eval _ sub (s, pno, nst)
2501 do _ init (sp)
1325
faça _ var (dowhat, s, sp)
1299 doglobal (dowhat)
1288
docal (dowhat, p, s)
1271 dumpskip (n, m)
2143
dumpsr (n, m)
2167 fim _ laboratórios (s, i)
1240
genaddproc ()
1190 genaddqueue ()
1475
genheader ()
1171 genother (cnt)
1210
genunio ()
2322 getweight (n)
2048
tem _ tau (n)
2060 huntele (f, o)
1403
huntstart (f)
1388 perdeu _ trilha ()
2459
corresponder _ trilha ()
2395 casos (fd, p, n, m, c) 1462
ntimes (fd, n, m, c)
1258 put _ pinit (e, s, p, i)
1363
put _ ptype (s, p, i, m0, m1) 1343 putnr (n)
2192
putstmt (fd, agora, m)
1825 typ2c (sp)
1432
undostmnt (agora, m)
2214 caminhada _ sub (e, pno, nst) 2469

Página 449

438

APÊNDICE E
PROJETO E VALIDAÇÃO

Tabela E.3 - Procedimentos explicados - Capítulo 13

Procedimento
Procedimento de página
Página

addproc ()
306
afirmar()
307
checkchan ()
309
d _ hash ()
300
d _ hash ()
307

delproc ()
306
endstate ()
307
gensrc ()
298
gensrc ()
308
hstore ()
306
huntini ()
308
corresponder _ trilha ()
298
corresponder _ trilha ()
310
novo _ estado ()
306
novo _ estado ()
300
novo _ estado ()
300
novo _ estado ()
305
p _ restor ()
306
putproc ()
308
putseq ()
308
putstmtnt ()
308
putstmtnt ()
309
q _ restor ()
305
qrecv ()
305
qsend ()
305
r _ ck ()
307
retrans ()
307
s _ hash ()
300
s _ hash ()
307
Configurável()
307
uerror ()
300
uerror ()
303
undostmnt ()
308
undostmnt ()
309
unrecv ()
306
cancelar o envio ()
306

```

3 char * Cabeçalho [] = {
4
5 #define qptr (x)
6 ((uchar *) e agora) + q_offset [x]) ",
7
8 #define pptr (x)
9 ((uchar *) e agora) + proc_offset [x]) ",
10
11 #define Pptr (x)
12 ((proc_offset [x])? pptr (x): noptr) ",
13
14 #define q_sz (x)
15 ((Q0 *) qptr (x)) -> Qlen) \ n ",
16
17 #define MAXQ
18 255 ",
19
20 #define MAXPROC
21 255 ",
22
23 #define WS
24 sizeof (long) / * tamanho da palavra em bytes * / ",
25
26 #ifndef VECTORSZ",
27
28 #define VECTORSZ
29 1024
30 / * tamanho sv em bytes * / ",
31
32 #fim se",
33
34
35 extern char * malloc (), * memcpy (), * memset ();
36
37
38 extern void exit ();
39
40
41 extern int abort (); \ n",
42
43
44 typedef struct Stack {
45 / * para filas e processos * / ",
46
47
48 short o_delta; ",
49
50
51 short o_offset; ",
52
53
54 curto o_skip; ",
55
56
57 short o_delqs; ",
58
59
60 char * body; ",
61
62
63 struct Stack * nxt; ",
64
65
66 struct Stack * lst; ",
67
68
69 } Pilha; \ n",
70
71
72 typedef struct Svtack {/ * para vetor de estado completo * /",
73
74
75 short o_delta; / * tamanho atual do quadro * / ",
76
77
78 short m_delta; / * tamanho máximo do quadro * / ",
79
80
81 #if SYNC",
82
83
84 short o_boq; ",
85
86
87 #fim se",
88
89
90
91 int j1, j2;
92 / * detecção de loop * / ",
93
94

```

```

"
char * body; ",
34
"
struct Svtack * nxt; ",
35
"
struct Svtack * lst; ",
36
"} Svtack; \ n",
37 #ifndef VARSTACK
38
"typedef struct Varstack {",
39
"
int val; ",
40
"
int cksum;
/* debugging only */ ",
41
"
struct Varstack * nxt; ",
42
"
struct Varstack * lst; ",
43
"} Varstack; \ n",
44 #endif
45 #ifdef GODEF
46
"#define NÃO UTILIZADO 0",
47
"#define R_LOCK
0",
48
"#define W_LOCK
1",
49
"#define Snd_LOCK
2",
50
"#define Rcv_LOCK
3",
51
"#define NLOCKS
4",
52
"#define BLOCK 1",
53
"#define REL
2",

```

Página 451

```

440
APÊNDICE E
PROJETO E VALIDAÇÃO
54
"typedef struct CS_stack {",
55
"
status curto; /* -1,0,1,2 = pendente, não utilizado, bloqueado, liberado */ ,
56
"
razão curta; /* 0..NLOCKS = bloqueado por R, W, Snd ou Rcv */ ,
57
"
delta curto;
/* a quantidade de um incremento ou decremento */ ,
58
"
short pid, stmnt, cs; ",
59
"
profundidade interna; ",
60
"
struct CS_stack * nxt; ",
61
"
```

```

struct CS_stack * lst; ",
62
"} CS_stack; \ n",
63 #endif
64
"typedef struct Trans {",
65
"
átomo curto;
/* é uma transição atômica */ ,
66
"
st curto;
/* the nextstate */ ,
67
"
short ist;
/* estado intermediário */ ,
68 #ifdef GODEF
69
"
local curto;
/* 1 se esta opção for local */ ,
70
"
local curto;
/* 1 iff todas as outras opções também são locais */ ,
71 #endif
72
"
char * tp;
/* texto fonte do movimento para a frente */ ,
73
"
char ntp;
/* ntipo do estado, por exemplo, 'r', 'c' etc */ ,
74
"
int forw;
/* índice para transição direta */ ,
75
"
int back;
/* índice para transição de retorno */ ,
76
"
struct Trans * nxt; ",
77
"} Trans; \ n",
78
"Trans *** trans;
/* 1 ptr por estado por proctype */ \ n ",
79
"
int depthfound = -1; /* detecção de loop */ ,
80
"
short proc_offset [MAXPROC], proc_skip [MAXPROC];",
81
"
short q_offset [MAXQ], q_skip [MAXQ];",
82
"
short vsizer;
/* tamanho do vetor em bytes */ ,
83
"
short boq = -1;
/* locked_on_queue status */ ,
84 #ifdef GODEF
85
"
tratável curto [MAXPROC]; /* nº da 1ª trans de cada proctipo */ ,
86 #endif
87
"
typedef struct State {",
88
"
uchar _nr_pr; ",
89
"
uchar _nr_qs; ",
90
"
uchar _p_t; /* detecção de loop */ ,
91
"
uchar _a_t; /* detecção do ciclo de aceitação */ ,

```

```

92
0,
93};
94
95 char * Addp0 [] = {
96 /* addproc (.... parlist ... * / ")",
97 /*
98 /*
99 int j, h = now._nr_pr; ",
100 "
101 "
102 switch (n) {",
103 0,
104};
105
106 char * Addp1 [] = {
107 "
padrão: Uerror (\ "bad proc - addproc \"); ",

```

Página 452

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

441
108 "
}
109 "
if (vsize %% WS && (j> WS- (vsize %% WS))) ",
110 "
{
proc_skip [h] = WS- (vsize %% WS); ",
111 "
vsize += proc_skip [h]; ",
112 "
} outro",
113 "
proc_skip [h] = 0; ",
114 "
proc_offset [h] = vsize; ",
115 "
agora._nr_pr += 1; ",
116 "
vsize += j; ",
117 "
hmax = max (hmax, vsize); ",
118 "
if (vsize> = VECTORSZ) ",
119 "
Uerror (\ "VECTORSZ é muito pequeno, edite pan.h \"); ",
120 "
memset ((char *) pptr (h), 0, j); ",
121 "
switch (n) {",

```

```

122
0,
123};
124
125 char * Addq0 [] = {
126 "addqueue (n)",
127 "
128 int j = 0, i = agora._nr_qs; ",
129 "
130 if (i> = MAXQ) ",
131 "
132};
133
134 char * Addq1 [] = {
135 "
136 padrão: Uerror (\ "fila inválida - adicionar fila \"); ",
137 "
138 if (vsize %% WS && (j> WS- (vsize %% WS))) ",
139 "
140 q_skip [i] = WS- (vsize %% WS); ",
141 "
142 vsize += q_skip [i]; ",
143 "
144 agora._nr_qs += 1; ",
145 "
146 vsize += j; ",
147 "
148 hmax = max (hmax, vsize); ",
149 "
150 if (vsize> = VECTORSZ) ",
151 "
152 Uerror (\ "VECTORSZ é muito pequeno, edite pan.h \"); ",
153 "
154 memset ((char *) qptr (i), 0, j); ",
155 "
156 ((Q0 *) qptr (i)) -> _t = n; ",
157 "
158 return i + 1; ",
159 "
160 "} \ n",
161 "
162 0,
163 };
164
165 char * Addq11 [] = {
166 "

```

```

int j; uchar * z; \ n",
157
"
if (! into--) ",
158
"
uerror (\ "referência a nome de canal não inicializado (envio) \"); ",
159
"
if (into> = now._nr_qs || into <0) ",
160
"
Uerror (\ "qenviar fila inválida # \"); ",
161
"
z = qptr (em); ",

```

Página 453

```

442
APÊNDICE E
PROJETO E VALIDAÇÃO
162
"
switch (((Q0 *) qptr (into)) -> _ t) {",
163
0,
164};
165
166 char * Addq2 [] = {
167
"
caso 0: printf (\ "fila foi excluída \\ n \"); ",
168
"
padrão: Uerror (\ "fila inválida - qsend \"); ",
169
"
}
170
"#fim se",
171
"} \ n",
172
"#if SYNC == 0",
173
"q_zero (from) /* para compiladores exigentes */",
174
"#fim se",
175
"#if SYNC",
176
"q_zero (de)",
177
"{",
178
"
if (! from--) ",
179
"
uerror (\ "referência a nome de canal não inicializado (recebendo) \"); ",
180
"
switch (((Q0 *) qptr (from)) -> _ t) {",
181
0,
182};
183
184 char * Addq3 [] = {
185
"
caso 0: printf (\ "fila foi excluída \\ n \"); ",
186
"
}
187
"
Uerror (\ "fila inválida q-zero \"); ",
188
"}",
189

```

```

"#fim se",
190
"q_len (x)",
191
"{
if (! x--) uerror (\ "referência ao nome chan não inicializado \"); ",
192
"
return ((Q0 *) qptr (x)) -> Qlen; ",
193
"} \ n",
194
"q_full (de)",
195
"{
if (! from--) ",
196
"
uerror (\ "referência a nome de canal não inicializado (envio) \"); ",
197
"
switch (((Q0 *) qptr (from)) -> _ t) {",
198
0,
199};
200
201 char * Addq4 [] = {
202
"
caso 0: printf (\ "fila foi excluída \\ n \"); ",
203
"
} ",
204
"
Uerror (\ "fila inválida - q_full \"); ",
205
"} \ n",
206
"qrecv (from, slot, fld, done)",
207
"{
uchar * z; ",
208
"
int j, k, r = 0; ",
209
"
if (! from--) ",
210
"
uerror (\ "referência a nome de canal não inicializado (recebendo) \"); ",
211
"
if (de> = agora._nr_qs || de <0) ",
212
"
Uerror (\ "qrecv bad queue # \"); ",
213
"
z = qptr (de); ",
214
"
switch (((Q0 *) qptr (from)) -> _ t) {",
215
0,

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN
443
216};
217
218 char * Addq5 [] = {
219
"
caso 0: printf (\ "fila foi excluída \\ n \"); ",
220
"
padrão: Uerror (\ "fila inválida - qrecv \"); ",

```

221
"
} ",
222
"
return r; ",
223
"} \ n",
224
0,
225};
226
227 char * Code0 [] = {
228
"corre()", 
229
"{
memset ((char *) & now, 0, sizeof (State)); ",
230
"
vsize = sizeof (estado) - VECTORSZ; ",
231
"
Configurável();",
232
0,
233};
234 char * Código1 [] = {
235
"#define CONNECT
% d / * aceitar rótulos * / ",
236
0,
237};
238 char * Code2 [] = {
239
"
Desbloquear;
/* desativa rendez-vous * / ",
240
"#ifdef BITSTATE",
241
"
SS = (uchar *) emalloc (1 << (ssize-3)); ",
242
"
if (loops) ",
243
"
LL = (uchar *) emalloc (1 << (ssize-3)); ",
244
"#outro",
245
"
hinit ();",
246
"#fim se",
247
"
stack = (Stack *) emalloc (sizeof (Stack)); ",
248
"
svtack = (Svtack *) emalloc (sizeof (Svtack)); ",
249 #ifdef VARSTACK
250
"
varstack = (Varstack *) emalloc (sizeof (Varstack)); ",
251 #endif
252 #ifdef GODEF
253
"
cs_stack = (CS_stack *) emalloc (sizeof (CS_stack)); ",
254
"
pilha_cs-> profundidade = -1; /* evitar uma correspondência falsa */ ",
255 #endif
256
"
/* um lugar para apontar para Pptr de procs não em execução: */ ",
257
"
noptr = (uchar *) emalloc (Maxbody * sizeof (char)); ",

```

```

258
"
addproc (0);
/* iniciar */,
259
"
profundidade = mreached = 0; ",
260
"
trpt = & trail [profundidade]; ",
261
"
new_state (); ",
262
"} \ n",
263
264
"#ifdef JUMBO",
265
"/** EXPERIMENTAL **/",
266
"Trans *",
267
"jumbostep (curto II)",
268
"{",
registrar Trans * t, * T = 0; char m, ot; short tt; ",
269
"
/ * assume que isso já foi definido * / ,

```

Página 455

444

APÊNDICE E
PROJETO E VALIDAÇÃO

```

270
"#ifdef ALG3",
271
"
printf (\ "desculpe: não é possível combinar -DJUMBO com -DALG3 \\n \"); ",
272
"
sair (1); ",
273
"#fim se",
274
275
"
sv_save ();
/* lembre-se de onde viemos * / ,
276
"
tt = (curto) ((P0 *) this) -> _ p; ",
277
"
ot = (uchar) ((P0 *) this) -> _ t; ",
278
"corrente:",
279
"
para (t = trans [ot] [tt]; t; t = t-> nxt) ",
280
"#include \" pan.m \ """",
281
"P999:",
282
"
if (m == 0) ",
283
"
{
printf (\ "não pode acontecer - jumbostep \\n \"); ",
284
"
Retorna;",
285
"
}
",
286
"
```

```

se (! T) T = t; ",
287
"
if (t-> st) ",
288
"
{
tt = ((P0 *) this) -> _ p = t-> st; ",
289
"
atingiu [ot] [t-> st] = 1; ",
290
"
if (trans [ot] [tt] -> Local> 1) ",
291
"
ir para cadeia; ",
292
"
},
293
"
return T; ",
294
"}",
295
"/** FIM **/",
296
"#fim se",
297
298
"new_state ()",
299
"{",
300
"register Trans * t; ",
301
"
char n, m, ot, match_type; ",
302
"
curto II, tt; \ n ",
303
"
curto De = agora._nr_pr-1; ",
304 #ifdef GODEF
305
"
char presel; ",
306 #endif
307
"Baixa:",
308 #ifdef GODEF
309
"
presel = 0; ",
310 #endif
311
"
if (now._p_t && prognow ()) / * detecção de loop * / ",
312 #ifdef GODEF
313
"
{
trpt-> tau |= 16; / * pm para 1 nível acima * / ",
314
"
ir para cima; ",
315
"
},
316 #else
317
"
ir para cima; ",
318 #endif
319
"
if (profundidade> = profundidade máxima) ",
320

```

```

"
{
truncs++;
321
#ifndef SYNC,
322
"
(trpt + 1) -> o_n = 1; /* não é um impasse */ ,
323
#endif se",

```

Página 456

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

445
324
"
ir para cima; ,
325
"
}
326
#ifndef VERI,
327
"
if (! (trpt-> tau & 4))
/* se nenhuma reivindicação mover */ ,
328
#endif se",
329
#ifndef SYNC> 0",
330
"
if (boq == -1)
/* se não mid-rv */ ,
331
#endif se",
332
"
if (! (trpt-> tau & 8))
/* se não houver movimento atômico */ ,
333
"
{",
334
#ifndef BITSTATE",
335
"
d_hash ((uchar *) & now, vsize); ,
336
"
j3 = (1 << (J1 e 7)); j1 = J1 >> 3; ",
337
"
j4 = (1 << (J2 e 7)); j2 = J2 >> 3; ",
338
"
if ((SS [j2] e j3) && (SS [j1] e j4)) ",
339
"#outro",
340
341
#ifndef CACHE",
342
"
if ((match_type = nh_store ((char *) & now, vsize))!= 0) ",
343
"#outro",
344
"
if ((match_type = hstore ((char *) & now, vsize))!= 0) ",
345
#endif se",
346
347
#endif se",
348
"
{
truncs++;

```

```

349
"
if (match_type == 2) ",
350
"
trpt-> tau |= 16; /* pm para 1 nível acima */",
351
352
"#if CONNECT> 0",
353
"
if (now._a_t && depth> A_depth) ",
354
"
{
if (memcmp ((char *) & A_Root, (char *) & now, vsiz) == 0) ",
355
"
{
if (fair_cycle ()) ",
356
"
uerror (\\"ciclo de aceitação \"); ",
357
"
se (profundidade> 0) vá para cima; else return; ",
358
"
}
359
"
}
360
"#fim se",
361
362
"#ifdef BITSTATE",
363
"
if (loops && now._p_t ",
364
"
& LL [j1] && LL [j2] && onstack ()) ",
365
"
{
if (fair_cycle ()) ",
366
"
uerror (\\"ciclo de não progresso \"); ",
367
"
}
368
"#fim se",
369
"
se (profundidade> 0) vá para cima; else return; ",
370
"
}
371
"#ifdef BITSTATE",
372
"
SS [j2] |= j3; SS [j1] |= j4; ",
373
"
if (loops) ",
374
"
{
sv_save (); ",
375
"
LL [j1]++; LL [j2]++; ",
376
"
svtack-> j1 = J1; ",
377
"
svtack-> j2 = J2; ",

```

```
446
APÊNDICE E
PROJETO E VALIDAÇÃO
378
"
}
",
379
"#fim se",
380
"
nstates++;
381
"
}
",
382
"
if (depth > mreached) ",
383
"
mreached = profundidade; ",
384
"
n = 0; ",
385
"#if SYNC",
386
"
(trpt + 1) -> o_n = 0; ",
387
"#fim se",
388
"#ifdef VERI",
389
"
if (now._nr_pr < 2 ",
390
"
|| ((P0 *) pptr (1)) -> _p == endclaim) ",
391
"
uerror (\ "alegação violada! \"); ",
392
"
if (stopstate [VERI] [((P0 *) pptr (1)) -> _p]) ",
393
"
uerror (\ "endstate na reivindicação atingida \"); ",
394
"Gaguejar:",
395
"
if (trpt-> tau & 4)
/* deve fazer uma reclamação */ ,
396
"
{
II = 1; ",
397
"
goto Verio; ",
398
"
},
399
"#fim se",
400 #ifdef GODEF
401
"
if (boq != -1) nlinks++;
/* compatibilidade com patrice */ ,
402
"#ifndef NOALG2",
403
"
if (boq == -1 && From! = To) ",
404
"
para (II = De; II >= Para; II -= 1)
```

```

/ * pré-varredura * / ",
405
"
{
406
"Curriculo:
/ * pegar aqui quando uma primeira pré-seleção falhou * / ",
407
"#ifdef VERI",
408
"
se (II == 1) continuar; ",
409
"
#endif se",
410
"
este = pptr (II); ",
411
"
tt = (curto) ((P0 *) this) -> _p; ",
412
"
ot = (uchar) ((P0 *) this) -> _t; ",
413
"
para (t = trans [ot] [tt]; t; t = t-> nxt) ",
414
"
{
if (! t-> local) ",
415
"
goto Trynext; ",
416
"
}
417
"
De = Para = II; /* todos os movimentos são locais * / ",
418
"
presel = 1;
/* no caso de ficarmos presos * / ",
419
"
quebrar;",
420
"
Trynext:
; ",
421
"
}
422
"
#endif se",
423 #endif
424
"\ nMais uma vez:",
425
"
para (II = De; II> = Para; II -= 1) ",
426
"
{
"
427
"#ifdef VERI",
428
"
se (II == 1) continuar; ",
429
"
#endif se",
430
"
Veri0:
este = pptr (II); ",
431
"
tt = (curto) ((P0 *) this) -> _p; ",

```

```

447
432
"
ot = (uchar) ((P0 *) this) -> _t; ",
433
"#ifdef JUMBO",
434
"/** EXPERIMENTAL **/",
435
"
if (trans [ot] [tt] -> Local> 1) ",
436
"
{
t = jumbostep (II); ",
437
"
m = 3; ",
438
"
profundidade ++; trpt ++; ",
439
"
trpt-> pr = II; ",
440
"
trpt-> st = tt; ",
441
"
goto Q999; ",
442
"
},
443
"** FIM **/",
444
"#fim se",
445
"
para (t = trans [ot] [tt]; t; t = t-> nxt) ",
446
"
",
447 #ifdef GODEF
448
"#ifdef ALG3",
449
"
if (now._p_t == 0) ",
450
"
if (csets [II] [t-> forw]> 0) ",
451
"
",
452
"
continuar;",
453
"
}
",
454
"#fim se",
455 #endif
456
"#include \" pan.m \ "\",
457
"P999:
/* pula aqui quando o movimento é bem-sucessido */",
458
"#ifdef ALG3",
459
"
if (Nwait> 0) ",
460
"
rel_all_blocks (II); ",
461
"#fim se",
462
"#ifdef VERBOSE",
463

```

```

"
printf (\% 3d: proc \% d exec \% d, de \% d para \% d \% s \\ n \", ",
464
"
profundidade, II, t-> forw, tt, t-> st, Move [t-> forw]); ",
465
#ifndef ALG3",
466
"
dumpsleep (\% novo_estado \"); ",
467
"#fim se",
468
"#fim se",
469
"
profundidade ++; trpt ++; ",
470
"
trpt-> pr = II; ",
471
"
trpt-> st = tt; ",
472
"
if (t-> st) ",
473
"
{
(P0 *) isto) -> _p = t-> st; ",
474 #if 0
475
XXXXX ESCREVENDO _p XXXXX
476 #endif
477
"
atingiu [ot] [t-> st] = 1; ",
478
"
} ",
479
"Q999:
trpt-> o_t = t; trpt-> o_n = n; ",
480
"
trpt-> o_ot = ot; trpt-> o_tt = tt; ",
481
"
trpt-> o_To = To; trpt-> o_m = m; ",
482
"
trpt-> tau = 0; ",
483
"
if (t-> átomo & 2) ",
484
"
{
trpt-> tau |= 8; ",
485
"#ifdef VERI",

```

```

448
APÊNDICE E
PROJETO E VALIDAÇÃO
486
"
if ((trpt-1) -> tau & 4) ",
487
"
trpt-> tau |= 4; ",
488
"
outro",
489
"
trpt-> tau &= ~4; ",
490
"
```

```

    } outro",
491
"
{
if ((trpt-1) -> tau & 4) ",
492
"
trpt-> tau & = ~4; ",
493
"
outro",
494
"
trpt-> tau | = 4; ",
495
"
},
496
"#outro",
497
"
} outro",
498
"
trpt-> tau & = ~8; ",
499
"#fim se",
500
"
if (boq == -1 && t-> átomo & 2) ",
501
"
{
De = Para = II; nlinks ++; ",
502
"
} outro",
503
"
{
De = agora._nr_pr-1; Para = 0; ",
504
"
},
505 #ifdef GODEF
506
"
if (presel) ",
507
"
{
",
508
"
(trpt-1) -> tau | = 32; ",
509
"
} outro {",
510
"
(trpt-1) -> tau & = ~32; ",
511
"
},
512 #endif
513
"
goto Down;
/* pseudo-recursão * / ",
514
"Acima:",
515 #ifdef GODEF
516
"
presel = 0; ",
517 #endif
518
"#if CONNECT> 0",
519
"
if (now._a_t && depth <= A_depth) ",
520
"

```

```

",
521
"
Retorna; /* viemos de checkaccept () */ ,
522
"
},
523
"#fim se",
524
"
t = trpt->o_t; n = trpt->o_n; ",
525
"
ot = trpt->o_ot; II = trpt->pr; ",
526
"
tt = trpt->o_tt; este = pptr (II); ",
527
"
To = trpt->o_To; m = trpt->o_m; ",
528
"#ifdef VERI",
529
"#if SYNC",
530
"/ * preservar status de conclusão de rendez-vous: * /",
531
"/ * se o próximo nível for uma reclamação, copie através de * /",
532
"
if (trpt->tau & 4) ",
533
"
trpt->o_n = (trpt + 1) ->o_n; ",
534
"#fim se",
535
"#fim se",
536
537
"#ifdef JUMBO",
538
"/** EXPERIMENTAL **/",
539
"
if (trans [ot] [tt] -> Local> 1) ",

```

Página 460

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

449
540
"
{
541
"
sv_restor (); ",
542
"
goto R999; ",
543
"
}
544
"/** FIM **/",
545
"#fim se",
546
547
"#include \"pan.b \\ \"",
548
"
R999:
/* pula aqui quando pronto */ ,
549
"#ifdef VERBOSE",
550
"
printf (\\"%% 3d: proc %% d reverte %% d, de %% d para %% d \\", ",
551

```

```

"
profundidade, II, t-> forw, tt, t-> st); ",
552
"
printf (\% s tau \% d tau-1 \% d \\ n \", Move [t-> forw], ",
553
"
trpt-> tau, (trpt-1) -> tau); ",
554
"#fim se",
555 #ifdef GODEF
556
"#ifdef ALG3",
557
"
unrelease (); /* desfazer o status 2 versões futuras */ ,
558
"#fim se",
559
"
/* truncado na pilha ou em um
* / ,
560
"
/* estado de progresso com now._p_t == 1 */ ,
561
"
if (trpt-> tau & 16) ",
562
"
{
if ((trpt-1) -> tau & 8)
/* atômico */ ,
563
"
{
(trpt-1) -> tau |= 16; ",
564
"
} ",
565
"
} outro",
566
"
{
(trpt-1) -> tau |= 64;
/* lembre se */ ,
567
"
} ",
568 #endif
569
"
profundidade--; trpt--; ",
570
"
se (m> n) n = m; ",
571
"
((P0 *) isto) -> _ p = tt; ",
572
"
} /* todas as opções */ ,
573 #ifdef GODEF
574
"
push_commit (); /* ativar blocos de processo */ ,
575 #endif
576
"#ifdef VERI",
577
"
if (II == 1) quebra; ",
578
"#fim se",
579
"
} /* todos os processos */ ,
580 #ifdef GODEF
581
"#ifdef ALG3",

```

```

582
"
unpush ();
/* unpush status 1 blocks */ ,
583
"#fim se",
584
#ifndef NOALG2",
585
586
"
if (! (trpt-> tau & 64) /* no nxtstates fora da pilha */ ,
587
"
&&
trpt-> tau & 32) /* últimos movimentos foram pré-selecionados */ ,
588
"
{
",
589
"
presel = 0; ",
590
"
De = agora._nr_pr-1; Para = 0; ",
591
"
II--; /* próxima vítima de pré-seleção */ ,
592
"
se (II> = 0) ",
593
"
goto Resume; ",

```

Página 461

```

450
APÊNDICE E
PROJETO E VALIDAÇÃO
594
"
outro",
595
"
goto novamente; ",
596
"
},
597
598
"
if (presel == 1) ",
599
"
{
if (n == 0) /* processo pré-selecionado não pôde se mover */ ,
600
"
{
",
601
"
presel = 0; ",
602
"
De = agora._nr_pr-1; Para = 0; ",
603
"
II--; /* próxima vítima de pré-seleção */ ,
604
"
se (II> = 0) ",
605
"
goto Resume; ",
606
"
outro",
607
"
goto novamente; ",

```

```

608
"
} else if (loops && now._p_t == 0) ",
609
"
{
/* ainda deve executar o verificador de progresso */",
610
"
De = Para = 1; /* tem pid 1 */",
611
"
goto novamente; ",
612
"
},
613
"
},
614
"#fim se",
615
"#ifdef ALG3",
616
"
if (Nwait == nwait [CS_timeout]) ",
617
"#fim se",
618 #endif
619
"
if (n == 0) ",
620
"
{",
621
"#ifdef VERI",
622
"
if (trpt-> tau & 4) goto Concluido;
/* ok se uma reivindicação bloquear */",
623
"#fim se",
624
"#if SYNC",
625
"
if (boq == -1) ",
626
"#fim se",
627
"
if (! endstate () && now._nr_pr ",
628
"
&& profundidade <maxdepth-1) ",
629
"
{
if (! ((trpt-> tau) & 1)) /* tempo limite */",
630
"
{
trpt-> tau |= 1; ",
631
"
push_act (0, W_LOCK, REL, 0, CS_timeout); ",
632
"
/* se isso libera qualquer procs - eles são automaticamente ",
633
"
não liberado pelo primeiro processo retornando a este nível ",
634
"
*/
635
"
goto novamente; ",
636
"
}
",

```

```

637
"#ifdef VERI",
638
"
se (n> = 0)
/* Claim Stutter */ ,
639
#ifndef NOSTUTTER",
640
"
{
trpt-> tau |= 4; ",
641
"
{
trpt-> tau |= 128; ",
642
"
goto Stutter; ",
643
"
} ",
644
"#outro",
645
"
goto Done;
/* ou seja, sempre */ ,
646
"#fim se",
647
"#outro",

```

Página 462

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

451
648
"
if (loops) goto Done; /* do loop det. só */ ,
649
"#fim se",
650
"
if (! (trpt-> tau & 8))
/* não é um movimento atômico */ ,
651
"
{",
652
"#ifdef VERI",
653
"
printf (\ "reclamar em \");
654
"
xrefsrc (linha de reivindicação, 1, ((P0 *) pptr (1)) -> _ p); ",
655
"#fim se",
656
"
uerror (\ "endstate inválido \");
657
"
} outro",
658
"
Uerror (\ "blocos seq atômicos \");
659
"
} ",
660
"#ifdef VERI",
661
#ifndef NOSTUTTER",
662
"
outro",
663
"
```

```

{
trpt-> tau |= 4; ",
664
"
trpt-> tau |= 128;
/* Marca de gagueira */",
665
"
goto Stutter; ",
666
"
}
",
667
"#fim se",
668
"#fim se",
669
"
}
",
670
"Feito:",
671 #ifdef GODEF
672
"
if (! (trpt-> tau & 8)) ",
673 #else
674
"#ifdef CACHE",
675
"
if (! (trpt-> tau & 8)) ",
676
"#outro",
677
"
if (loops &&! (trpt-> tau & 8)) ",
678
"#fim se",
679 #endif
680
"#ifdef VERI",
681
"
if (! (trpt-> tau & 4)) ",
682
"#fim se",
683
"#if SYNC> 0",
684
"
if (boq == -1) ",
685
"#fim se",
686
"
{",
687
"#ifdef BITSTATE",
688
"
LL [(svtack-> j1) >> 3] -; ",
689
"
LL [(svtack-> j2) >> 3] -; ",
690
"
svtack = svtack-> lst; ",
691
"
if (trpt-> tau & 2)
/* estado marcado como sujo: remover */",
692
"
{
SS [(svtack-> j2) >> 3] &= ~ (1 << ((svtack-> j1) & 7)); "
693
"
SS [(svtack-> j1) >> 3] &= ~ (1 << ((svtack-> j2) & 7)); "
694
"
} ",
695

```

```

"#outro",
696
"
htag ((char *) & now, vsizze); ",
697
"#fim se",
698
"
} ",
699
"#if CONNECT> 0",
700
"#ifdef VERI",
701
"
if ((! (trpt-> tau & 4)) ",

```

Página 463

```

452
APÊNDICE E
PROJETO E VALIDAÇÃO
702
"
|| (trpt-> tau & 128))
/* nenhum movimento de reivindicação, a menos que Stutter */ ,
703
"#fim se",
704
"
if (aciclos
/* -uma opção é usada */ ,
705
"
&&! (trpt-> tau & 8))
/* não é um movimento atômico */ ,
706
"
checkaccept (); /* verificar os ciclos de aceitação */ ,
707
"#fim se",
708
"
if (profundidade> 0) goto Up; ",
709
"} \ n",
710 #ifdef GODEF
711
"#ifdef ALG3",
712
"rel_all_blocks (pid)
/* não completamente testado */ ,
713
"
{
int kk, mm, k, s, r, F, T, Cn, efeito = 0;
714
"
F = tratável [((P0 *) pptr (pid)) -> _ t]; ",
715
"
T = tratável [((P0 *) pptr (pid)) -> _ t + 1]; ",
716
"
para (s = F; s <T; s ++)
717
"
{
if (csets [pid] [s] == 0) continue; ",
718
"
para (kk = 1; kk <1 + Csels_c [s] [0]; kk ++) ",
719
"
{
if (Csels_p [s] [kk] != pid) continue; ",
720
"
k = Csels_c [s] [kk]; ",
721
"
r = Csels_r [s] [kk]; ",

```

```

722
"
Cn = Csels_c [s] [0] -; ",
723
"
if (Cn <1) Uerror (\ "não pode acontecer - rel_all \"); ",
724
"
para (mm = kk; mm <Cn; mm++) ",
725
"
{
Csels_c [s] [mm] = Csels_c [s] [mm + 1]; ",
726
"
Csels_r [s] [mm] = Csels_r [s] [mm + 1]; ",
727
"
Csels_p [s] [mm] = Csels_p [s] [mm + 1]; ",
728
"
} ",
729
"
csems [pid] [k] -; ",
730
"
csets [pid] [s] -; ",
731
"
if (nwait [k] <= 0) ",
732
"
{
printf (\ "nwait [%% d] = %% d (%% d) \\n \", ",
733
"
k, nwait [k], Nwait); ",
734
"
Uerror (\ "nAguarde \"); ",
735
"
}
",
736
"
nwait [k] -; Nwait--; efeito = 1; ",
737
"
push_cs_el (pid, s, k, profundidade + 1,2, r, 1); ",
738
"
kk--; ",
739
"
}
",
740
"
}
",
741
"#ifdef VERBOSE",
742
"
if (efeito) dumpsleep (\ "rel_blocks \"); ",
743
"#fim se",
744
"}",
745
"
char * LCK [] = {\ "Read \", \ "Write \", \ "Send \", \ "Recv \"); ",
746
"dumpsleep (str)",
747
"
char * str; ",
748
"{
int pid, xx, yy, zz, kk, F, T; ",
749
"
para (pid = 0; pid <agora._nr_pr; pid++) ",

```

```

750
"
{F = tratável [((P0 *) pptr (pid)) -> _ t]; ",
751
"
T = tratável [((P0 *) pptr (pid)) -> _ t + 1]; ",
752
"
para (xx = F; xx <T; xx ++) ",

{
if (csets [pid] [xx] == 0) continue; ",
754
"
printf (\ "sleepset proc %% d: \", pid); ",
755
"
printf (\ "trans %% 2d, cs {\\", xx); ",

```

Página 464

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

453
756
"
para (kk = 1; kk <1 + Csels_c [xx] [0]; kk ++) ",
757
"
{
yy = Csels_r [xx] [kk]; ",
758
"
zz = Csels_c [xx] [kk]; ",
759
"
if (pid == Csels_p [xx] [kk]) ",
760
"
{
if (zz <MAXCONFL) ",
761
"
printf (\ %% s em var %% s, \",
762
"
LCK [yy], CS_names [zz]); ",
763
"
else if (zz == MAXCONFL) ",
764
"
printf (\ <local>, \");
765
"
outro",
766
"
printf (\ %% s em qid %% d, \",
767
"
LCK [yy], zz); ",
768
"
} ",
769
"
}
770
"
} }",
771
"}",
772
"push_cs_el (pid, stmnt, cs, dp, st, rs, dt)",
773
"{" ,
774
"
if (stmnt == 0) return; /* timeouts mapeados para 0 */ ",

```

```

775
"
if (cs_stack-> depth> dp) ",
776
"
{
push2_cs_el (pid, stmnt, cs, dp, st, rs, dt); ",
777
"
Retorna;",
778
"
}
",
779
"
if (! cs_stack-> nxt) ",
780
"
{
cs_stack-> nxt = (CS_stack *) ",
781
"
emalloc (sizeof (CS_stack)); ",
782
"
cs_stack-> nxt-> lst = cs_stack; ",
783
"
cs_max ++; ",
784
"
}
",
785
"
cs_stack = cs_stack-> nxt; ",
786
"
cs_stack-> pid
= pid; ",
787
"
cs_stack-> stmnt = stmnt; ",
788
"
cs_stack-> cs
= cs; ",
789
"
cs_stack-> delta = dt; ",
790
"
cs_stack-> depth = dp; ",
791
"
cs_stack-> status = st; ",
792
"
cs_stack-> reason = rs; ",
793
"} \ n",
794
/* alcance cs_stack e insira na profundidade correta */
795
"push2_cs_el (pid, stmnt, cs, dp, st, rs, dt)",
796
"{",
CS_stack * k, * twiddle; ",
797
"
cs_max ++; ",
798
"
twiddle = (CS_stack *) emalloc (sizeof (CS_stack)); ",
799
"
twiddle-> pid
= pid; ",
800
"
twiddle-> stmnt = stmnt; ",
801
"

```

```

twiddle-> cs
= cs; ",
802
"
twiddle-> delta = dt; ",
803
"
twiddle-> depth = dp; ",
804
"
twiddle-> status = st; ",
805
"
twiddle-> reason = rs; ",
806
"
para (k = cs_stack; k && k-> profundidade> dp; k = k-> lst) ",
807
"
;
",
808
"
if (k) ",
809
"
{
twiddle-> nxt = k-> nxt; ",

```

Página 465

```

454
APÊNDICE E
PROJETO E VALIDAÇÃO
810
"
k-> nxt-> lst = twiddle; ",
811
"
twiddle-> lst = k; ",
812
"
k-> nxt = twiddle; ",
813
"
} outro",
814
"
cs_stack = twiddle; ",
815
"} \ n",
816
"push_commit () /* commit para um bloqueio pendente */",
817
"{
CS_stack * k; int mv, Cn, efeito = 0; ",
818
"
para (k = cs_stack; k && k-> profundidade == profundidade + 1; k = k-> lst) ",
819
"
{
if (k-> status! = -1) continue; ",
820
"
k-> status = 1; mv = k-> stmnt; ",
821
"
Cn = ++ Csels_c [mv] [0]; ",
822
"
if (Cn> MULT_MAXCS) ",
823
"
{
printf (\ "erro: recompilar com MULT> %% d \\ n \", MULT); ",
824
"
sair (1); ",
825
"
}
",
```

```

826
"
Csels_c [mv] [Cn] = k-> cs; ",
827
"
Csels_r [mv] [Cn] = k-> razão; ",
828
"
Csels_p [mv] [Cn] = k-> pid; ",
829
"
csems [k-> pid] [k-> cs] ++; ",
830
"
csets [k-> pid] [mv] ++; ",
831
"
nwait [k-> cs] ++; ",
832
"
Nwait ++; efeito = 1; ",
833
"
},
834
"#ifdef VERBOSE",
835
"
if (efeito) dumpsleep (\ "push_commit \");
836
"#fim se",
837
"} \ n",
838
"char Conflict [NLOCKS] [NLOCKS] = {/ * 1 == DEP, 0 == IND * /",
839
"/ *
R_LOCK, W_LOCK, Snd_LOCK, Rcv_LOCK * /",
840
"/ * R_LOCK * /
{0,
1,
1,
1,
1},
841
"/ * W_LOCK * /
{1,
1,
1,
1},
842
"/ * Snd_LOCK * /
{1,
1,
1,
1,
M LOSS},
843
"/ * Rcv_LOCK * /
{1,
1,
1,
M LOSS,
1},
844
"};",
845
"/ * quando m_loss é definido (no sinalizador -m do SPIN) envia e recebe",
846
"*
na mesma fila são realmente dependentes apenas quando a fila",
847
"*
está completo - a versão acima é, portanto, um pouco conservadora",
848
"*
",
849
"push_act (pid, what, when, stmnt, cs) / * log a global action * /",
850
"{
int i, j, k, r, F, R, T, maxk, delta, kk, próprio; efeito int = 0; ",
851
"",
852
"
if (quando == BLOCK)

```

```

/* definir um bloqueio pendente */ ,
853
#ifndef NOPELED",
854
"
{
if (! (trpt-> tau & 16))
/* PELED's PROVISO */ ,
855
"#outro",
856
"
{
857
"#fim se",
858
"
push_cs_el (pid, stmnt, cs, profundidade, -1, o quê, 1); ",
859
"
Retorna;",
860
"
} /* else release */ ,
861
"
maxk = 1 + MAXCONFL + agora._nr_qs; ",
862
"
if (nwait [cs]> 0) ",
863
"
para (i = 0; i <agora._nr_pr; i ++) ",

```

Página 466

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

455
864
"
if (csems [i] [cs]> 0) ",
865
"
{F = tratável [((P0 *) pptr (i)) -> _ t]; ",
866
"
T = tratável [((P0 *) pptr (i)) -> _ t + 1]; ",
867
"
para (j = F; j <T; j ++) ",
868
"
{
para (kk = 1; kk <1 + Csels_c [j] [0]; kk++) ",
869
"
{
if (Csels_c [j] [kk] == cs ",
870
"
&& Conflito [what] [Csels_r [j] [kk]]) ",
871
"
{
/* limpar todos os blocos em j */ ,
872
"
para (kk = 1; kk <1 + Csels_c [j] [0]; kk++) ",
873
"
{
r = Csels_r [j] [kk]; ",
874
"
k = Csels_c [j] [kk]; ",
875
"
próprio = Csels_p [j] [kk]; ",
876
"

```

```

csems [próprio] [k] -; ",
877
"
csets [próprios] [j] -; ",
878
"
nwait [k] -; ",
879
"
Nwait--; ",
880
"
push_cs_el (próprio, j, k, profundidade + 1,2, r, 1); ",
881
"
efeito = 1; ",
882
"
},
883
"
Csels_c [j] [0] = 0; ",
884
"
quebrar;",
885
"
}
886
"
},
887
"
},
888
"
"out: if (nwait [cs] <0)",
889
"
Uerror (\ "nesperar negativo \");
890
"#ifdef VERBOSE",
891
"
if (efeito) dumpsleep (\ "act \");
892
"#fim se",
893
"} \ n",
894
"unrelease ()",
895
"
{
int k, p, s, dt, Cn, efeito = 0; ",
896
"
CS_stack * K; ",
897
"
para (K = cs_stack; K && K-> profundidade == profundidade; K = K-> lst) ",
898
"
{
k = K-> cs; ",
899
"
p = K-> pid; ",
900
"
s = K-> stmnt; ",
901
"
if (K-> status == 2) ",
902
"
{
",
903
"
para (dt = 0; dt <K-> delta; dt++) ",
904
"
{

```

```

Cn = ++ Csels_c [s] [0]; ",
905
"
if (Cn> MULT_MAXCS) ",
906
"
Error (\ "não pode acontecer - Csels1 \");
907
"
Csels_c [s] [Cn] = k; ",
908
"
Csels_r [s] [Cn] = K-> motivo; ",
909
"
Csels_p [s] [Cn] = p; ",
910
"
csems [p] [k] ++;
911
"
csets [p] [s] ++;
912
"
nwait [k]++;
Nwait++;
913
"
},
914
"
K-> status = 3;
915
"
efeito = 1;
916
"
},
917
"
},
918
",
919
"
ifefeito = 1;
920
"
#fim se",
921
"} \ n",
922
"unpush ()",
923
"{
int k, p, r, s, kk, mm, Cn, oCn, efeito = 0;
924
"
enquanto (cs_stack && cs_stack-> profundidade == profundidade + 1) ",
925
"
{
k = pilha_cs-> cs;
926
"
p = cs_stack-> pid;
927
"
s = cs_stack-> stmnt;
928
"
r = pilha_cs-> razão;
929
"
if (cs_stack-> status == 1),
930
"

```

456
APÊNDICE E
PROJETO E VALIDAÇÃO
918
"#ifdef VERBOSE",
919
"
if (efeito) dumpsleep (\ "unrelease \"); ",
920
"#fim se",
921
"} \ n",
922
"unpush ()",
923
"{
int k, p, r, s, kk, mm, Cn, oCn, efeito = 0;
924
"
enquanto (cs_stack && cs_stack-> profundidade == profundidade + 1) ",
925
"
{
k = pilha_cs-> cs;
926
"
p = cs_stack-> pid;
927
"
s = cs_stack-> stmnt;
928
"
r = pilha_cs-> razão;
929
"
if (cs_stack-> status == 1),
930
"

```

",
931
"
oCn = Csels_c [s] [0]; ",
932
"
para (kk = 1; kk <1 + Csels_c [s] [0]; kk ++) ",
933
"
if (Csels_r [s] [kk] == r ",
934
"
&& Csels_c [s] [kk] == k ",
935
"
&& Csels_p [s] [kk] == p) ",
936
"
{
Cn = Csels_c [s] [0] -; ",
937
"
if (Cn <1) ",
938
"
Uerror (\ "não pode acontecer - Csels2 \"); ",
939
"
para (mm = kk; mm <Cn; mm++) ",
940
"
{Csels_c [s] [mm] = Csels_c [s] [mm + 1]; ",
941
"
Csels_r [s] [mm] = Csels_r [s] [mm + 1]; ",
942
"
Csels_p [s] [mm] = Csels_p [s] [mm + 1]; ",
943
"
}
",
944
"
quebrar;",
945
"
}
",
946
"
if (oCn == Csels_c [s] [0]) ",
947
"
{
",
948
"
printf (\ "não é possível encontrar %% d, %% d em \\ n \", r, k); ",
949
"
para (kk = 1; kk <1 + Csels_c [s] [0]; kk ++) ",
950
"
printf (\ "t %% d, %% d \\ n \", Csels_r [s] [kk], Csels_c [s] [kk]); ",
951
"
Uerror (\ "não pode acontecer Cs unpush \"); ",
952
"
}
",
953
"
csems [p] [k] -; ",
954
"
csets [p] [s] -; ",
955
"
if (nwait [k] <= 0) ",
956
"
{
printf (\ "nwait [%% d] = %% d (%% d) \\ n \", ",
957
"

```

```

"
k, nwait [k], Nwait); ",
958
"
Uerror (\ "esperar \");
959
"
}
960
"
nwait [k] -; Nwait--; efeito = 1;
961
"
} else if (cs_stack-> status! = 3) ",
962
"
{
printf (\ "cs = %% d, mv = %% d \\n \", ",
963
"
cs_stack-> cs, cs_stack-> stmnt); ",
964
"
printf (\ "Status ruim: %% d \\n \", cs_stack-> status); ",
965
"
Uerror (\ "unpush \");
966
"
}
967
"
cs_stack-> status = cs_stack-> reason = 0;
968
"
cs_stack = cs_stack-> lst; ",
969
"
}
970
"#ifdef VERBOSE",
971
"
if (efeito) dumpsleep (\ "unpush \");

```

Página 468

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

457
972
"#fim se",
973
"} \\ n",
974
"#fim se",
975 #endif
976
"assert (a, s, ii, tt, t)",
977
"
char * s; ",
978
"
Trans * t; ",
979
"{\nse um)\",
980
"
{
printf (\ "afirmação violada %% s \", s); ",
981
"
profundidade ++; trpt ++; ",
982
"
trpt-> pr = ii; ",
983
"
trpt-> st = tt; ",

```

```

984
"
trpt-> o_t = t; ",
985
"
uerror (\ "abortado \"); ",
986
"
profundidade--; trpt--; ",
987
"
} ",
988
"}",
989
"#ifndef NOBOUNDCHECK",
990
"Boundcheck (x, y, a1, a2, a3)",
991
"
Trans * a3; ",
992
"{  

assert ((x> = 0 && x <y), \ "- índice de matriz inválido \\ n \", a1, a2, a3); ",
993
"
return x; ",
994
"}",
995
"#fim se",
996
"#ifdef MEMCNT",
997
"int memcnt = 0;",
998
"#fim se",
999
"vazio",
1000
"embrulhar()",  

1001
"{{",
1002
"#ifdef BITSTATE",
1003
"
duplo a, b; \ n ",
1004
"
printf (\ "pesquisa de espaço de estado de bits \"); ",
1005
"#outro",
1006
"
printf (\ "pesquisa de espaço total de estados \"); ",
1007
"#fim se",
1008
"#ifdef VERI",
1009
"
printf (\ "no comportamento restrito à reivindicação \"); ",
1010
"#fim se",
1011
"
printf (\ "para: \\ n \ violações de asserção \"); ",
1012
"#ifndef VERI",
1013
"
if (loops) ",
1014
"
printf (\ "e %% s loops sem progresso \", ",
1015
"
justiça? \ "JUSTO \": \ \"\"); ",
1016
"
outro",

```

```

1017
"
printf (\ "e estados finais inválidos \"); ",
1018
"#fim se",
1019
"#if CONNECT> 0",
1020
"
if (aciclos &&! loops) ",
1021
"
printf (\ \"\\ n \\ t e % s ciclos de aceitação \", ",
1022
"
justiça? \ "JUSTO \": \ "\"); ",
1023
"#fim se",
1024
"
if (! done) printf (\ \"\\ nprocura não foi concluída \"); ",
10: 25h
"
printf (\ \"\\ nvetor %% d byte, profundidade alcançada %% d \", ",

```

Página 469

```

458
APÊNDICE E
PROJETO E VALIDAÇÃO
1026
"
hmax, mreached); ",
1027
"
if (loops) ",
1028
"
{printf (\ ", loops sem progresso: %% d \\ n \", erros); ",
1029
"
} outro",
1030
"
printf (\ ", erros: %% d \\ n \", erros); ",
1031
"
printf (\ \"% 8d estados, armazenados \", nstates - reciclado); ",
1032
"
if (reciclado) printf (\ "(% d reciclado) \", reciclado); ",
1033
"
printf (\ \"\\ n %% 8d estados, vinculados \\ n \", nlinks); ",
1034
"
printf (\ "estados %% 8d, correspondido \\ t total: %% 8d \\ n \", ",
1035
"
trunks, nstates + nlinks + trunks); ",
1036
"#ifdef BITSTATE",
1037
"
a = (duplo) (1 << ssize); ",
1038
"
b = (duplo) nestados + 1 .; ",
1039
"
printf (\ "fator de hash: %% f \", a / b); ",
1040
"
printf (\ "(melhor cobertura se> 100) \\ n \"); ",
1041
"#outro",
1042
"
printf (\ "conflitos de hash: %% d (resolvido) \\ n \", hcmp); ",
1043
"#fim se",

```

```

1044
"
printf (\ " (tamanho máximo 2^ %% d estados, \", ssize); ",
1045 #ifdef VARSTACK
1046 "
printf (\ "varstack: %% d, \", vmax); ",
1047 #endif
1048 #ifdef GODEF
1049 "
printf (\ "cs_stack: %% d, \", cs_max); ",
1050 #endif
1051 "
printf (\ "stackframes: %% d / %% d) \\n \\n \", smax, svmax); ",
1052 "
if (M_LOSS) printf (\ "total de mensagens perdidas: %% d \\n \\n \", perda); ",
1053
"#ifdef MEMCNT",
1054 "
printf (\ "memória usada: %% d \\n \", memcnt); ",
1055
"#fim se",
1056 "
if (loops && feitos) do_reach (); ",
1057 #ifdef GODEF
1058 "
#endif ALG3",
1059 "
#endif VERBOSE",
1060 "
se (feito) ",
1061 "
{
int i, j, k, r; ",
1062 "
para (j = 0; j <MAXSTATE; j++) ",
1063 "
{
if (Csels_c [j] [0]! = 0) ",
1064 "
printf (\ "Csels_c [%% d] [0] = %% d \\n \", ",
1065 "
j, Csels_c [j] [0]); ",
1066 "
}
1067 "
para (i = 0; i <MAXPROC; i++) ",
1068 "
para (k = 0; k <TOPQ; k++) ",
1069 "
if (csems [i] [k]! = 0) ",
1070 "
printf (\ "\ tcsem %% d, %% d = %% d \\n \", ",
1071 "
i, k, csems [i] [k]); ",
1072 "
para (j = 0; j <TOPQ; j++) ",
1073 "
if (nwait [j]! = 0) ",
1074 "
printf (\ "\ tnwait %% d = %% d \\n \", ",
1075 "

```

```

1076
"
j, nwait [j]); ",
1077
"
if (Nwait! = 0) ",
1078
"
printf (\ "Nwait = %% d \\n \", Nwait); ",
1079
"
}
",

```

Página 470

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

459
1080
"#fim se",
1081
"#fim se",
1082 #endif
1083
"
sair (0); ",
1084
"} \ n",
1085
"d_hash (cp, om)",
1086
"
uchar * cp; ",
1087
"{",
1088
"
registrar longo z = 0x88888EEFL; ",
1089
"
registrar long * q, * r; ",
1090
"
registrar int h; ",
1091
"
registrar m, n; \ n ",
1092
"
h = (om + 3) / 4; ",
1093
"
m = n = -1; ",
1094
"
q = r = (longo *) cp; ",
1095
"
r += (longo) h; ",
1096
"
Faz {",
1097
"
m += m; ",
1098
"
if (m <0) ",
1099
"
m ^ = z; ",
1100
"
m ^ = * q ++; ",
1101
"
n += n; ",
1102
"
if (n <0) ",
1103

```

```

"
n ^ = z; ",
1104
"
n ^ = * - r; ",
1105
"
} enquanto (--h> 0); ",
1106
"
J1 = (m ^ (m >> (8 * sizeof (unsigned) -ssize))) & mask; ",
1107
"
J2 = (n ^ (n >> (8 * sizeof (unsigned) -ssize))) & mask; ",
1108
"} \ n",
1109
"s_hash (cp, om)",
1110
"
uchar * cp; ",
1111
"{",
1112
"#ifdef ALTHASH",
1113
"
d_hash (cp, om); ",
1114
"
j1 = (J1^J2) & mask; ",
1115
"#outro",
1116
"
registrar longo z = 0x88888EEFL; ",
1117
"
registrar long * q; ",
1118
"
registrar int h; \ n ",
1119
"
registrar m = -1; ",
1120
"
h = (om + 3) / 4; ",
1121
"
q = (longo *) cp; ",
1122
"
Faz {",
1123
"
m += m; ",
1124
"
if (m <0) ",
11: 25h
"
m ^ = z; ",
1126
"
m ^ = * q ++; ",
1127
"
} enquanto (--h> 0); ",
1128
"
j1 = (m ^ (m >> (8 * sizeof (unsigned) -ssize))) & mask; ",
1129
"#fim se",
1130
"} \ n",
1131
"main (argc, argv)",
1132
"
char * argv []; ",
1133

```

```

"",

460
APÊNDICE E
PROJETO E VALIDAÇÃO
1134
"
while (argc> 1 && argv [1] [0] == '-') ",
1135
"
{
switch (argv [1] [1]) {" ,
1136
"#if CONNECT> 0",
1137
"
caso 'a': aciclos = 1; quebrar;",
1138
"#fim se",
1139
"
caso 'c': até = atoi (& argv [1] [2]); quebrar;",
1140
"
case 'd': state_tables ++; quebrar;",
1141
"
caso 'f': justiça = 1; quebrar;",
1142
"
caso 'H': homomorfismo = 1; ",
1143
"
if (argc <4) {uso (); Saída(); } ",
1144
"
hom_target = argv [2]; origem_home = argv [3]; ",
1145
"
printf (\ "trans curto; \\n \"); ",
1146
"
quebrar;",
1147
"#ifndef VERI",
1148
"
caso 'l': loops = 1; quebrar;",
1149
"#fim se",
1150
"
caso 'm': profundidade máxima = atoi (& argv [1] [2]); quebrar;",
1151
"
caso 'w': ssize = atoi (& argv [1] [2]); quebrar;",
1152 #ifdef PARES
1153
"
case 't': tree_before = 1; quebrar;",
1154 #endif
1155
"
padrão: uso (); sair (1); ",
1156
"
} ,
1157
"
argc--; argv ++; ",
1158
"
} ,
1159
"
if (acycles && loops) ",
1160
"
{

```

```

fprintf (stderr, \ "desculpe: não é possível combinar -a e -l \\ n \"); ",
1161
"
uso(); sair (1); ",
1162
"
},
1163
"
if (justiça && acycles && loops) ",
1164
"
{
fprintf (stderr, \ "desculpe: a opção -f só tem efeito quando \");
1165
"
fprintf (stderr, \ "combinado com -a ou -l \\ n \"); ",
1166
"
uso(); sair (1); ",
1167
"
},
1168
"
sinal (SIGINT, finalização); ",
1169
"
máscara = ((1 << ssize) -1); /* hash init */ ,
1170
"
trilha = (trilha *) emalloc ((profundidade máxima + 2) * sizeof (trilha)); ",
1171
"
corre();",
1172
"
concluido = 1; ",
1173
"
embrulhar();",
1174
"} \\ n",
1175
"uso()", 
1176
"{
fprintf (stderr, \ "opção desconhecida \\ n \");
1177
"#if CONNECT> 0",
1178
"
fprintf (stderr, \ "- a encontrar ciclos de aceitação \\ n \");
1179
"#outro",
1180
"
fprintf (stderr, \ "- um desativado (nenhum rótulo de aceitação é definido) \\ n \");
1181
"#fim se",
1182
"
fprintf (stderr, \ "- cN parar no enésimo erro \");
1183
"
fprintf (stderr, \ "(padrão = 1) \\ n \");
1184
"
fprintf (stderr, \ "- d imprimir tabelas de estado e parar \\ n \");
1185
"
fprintf (stderr, \ "- d -d imprimir tabelas de estado não otimizadas \\ n \");
1186
"
fprintf (stderr, \ "- f impõe justiça fraca em ciclos \\ n \");
1187
"
fprintf (stderr, \ "- H target_proctype source_proctype \\ n \");
1188
"

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

461
1188
"
fprintf (stderr, \
produzir um modelo para provar homomorfismo \\ n \ ");
1189
#ifndef VERI",
1190
"
fprintf (stderr, \
"- l encontrar ciclos de não progresso \\ n \");
1191
"#outro",
1192
"
fprintf (stderr, \
"- l desabilitado (pela presença de nunca reclamar) \\ n \");
1193
"#fim se",
1194
"
fprintf (stderr, \
"- profundidade máxima de mN N (padrão = 10k) \\ n \");
1195
"
fprintf (stderr, \
"- wN hashtable de 2^N entradas \");
1196
"
fprintf (stderr, \
"(padrão = %% d) \\ n \", ssize); ",
1197
"} \ n",
1198 #if 0
1199
"Caracteres *",
1200
"emalloc (n)",
1201
|{
char * tmp = malloc (n); ,
1202
#endif MEMCNT",
1203
"
if (! tmp || memcnt > 1 << MEMCNT) ",
1204
"#outro",
1205
"
if (! tmp) ,
1206
"#fim se",
1207
"
{
printf (\ "pan: sem memória \\ n \");
1208
"
embrulhar(); ,
1209
"
} ,
1210
#endif MEMCNT",
1211
"
memcnt += n; ,
1212
"#fim se",
1213
"
memset (tmp, 0, n); ,
1214
"
return tmp; ,
1215
"} \ n",
1216 #else
1217
/* * inclui realloc e livre para manter sysV libc",
1218
/* de incluí-los e",
1219
/* encontrar referências múltiplas",

```

```

1220
" * /",
1221
"Caracteres *",
1222
"reallocar (s)",
1223
"
char * s; ",
1224
"{
printf (\\"abortando: não pode acontecer - chamar realloc () \\n \"); ",
12: 25h
"
embrulhar();",
1226
"}",
1227
"",
1228
"grátis (s)",
1229
"
char * s; ",
1230
"{
/* nunca chamado - simplesmente ignore */",
1231
"}",
1232
"",
1233
"Caracteres *",
1234
"malloc (n)",
1235
"
n sem sinal; ",
1236
"|,
1237
"
char * tmp; ",
1238
"
extern char * sbrk ();",
1239
"
tmp = sbrk (n);",
1240
"#ifdef MEMCNT",
1241
"
if ((int) tmp == -1 || memcnt > 1 << MEMCNT) ",

```

```

462
APÊNDICE E
PROJETO E VALIDAÇÃO
1242
"#outro",
1243
"
if ((int) tmp == -1) ",
1244
"#fim se",
1245
"
{
printf (\\"abortando: sem memória \\n \"); ",
1246
"
embrulhar();",
1247
"
} ",
1248
"#ifdef MEMCNT",
1249
"

```

```

memcnt += n; ",
1250
"#fim se",
1251
"
return tmp; ",
1252
"}",
1253
"",
1254
"#define CHUNK 4096",
1255
"",
1256
"Caracteres *",
1257
"emalloc (n)
/* a memória nunca é liberada ou realocada */ ",
1258
"
n sem sinal; ",
1259
"{",
1260
"
char * tmp; ",
1261
"
static char * have; ",
1262
"
esquerda longa estática = 0L; ",
1263
"
fragmento longo estático = 0L; ",
1264
"",
1265
"
if (n == 0) ",
1266
"
return (char *) NULL; ",
1267
"
if (n & 3) ",
1268
"
n += 4 - (n & 3); /* para o alinhamento adequado */ ",
1269
"
if (left <n) ",
1270
"
{
crescer sem sinal = (n <CHUNK) ? CHUNK: n; ",
1271
"
have = malloc (crescer); ",
1272
"
fragmento += esquerdo; ",
1273
"
esquerda = crescer; ",
1274
"
} ",
1275
"
tmp = ter; ",
1276
"
tem += (longo) n; ",
1277
"
esquerda -= (longo) n; ",
1278
"
memset (tmp, 0, n); ",
1279

```

```

"
return tmp; ",
1280
"}",
1281 #endif
1282
"Uerror (str)",
1283
"
char * str; ",
1284
"{
/* sempre fatal */ ",
1285
"
erros = até-1; ",
1286
"
uerror (str); ",
1287
"
embrulhar();",
1288
"} \ n",
1289
"uerror (str)",
1290
"
char * str; ",
1291
"{
1292
"
if (++ erros == upto) ",
1293
"
{
printf (\ "pan: %% s (na profundidade %% d) \\n \", str, ",
1294
"
(profundidade encontrada == - 1)? profundidade: profundidade encontrada); ",
1295
"
putrail (); ",

```

Página 474

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

463
1296
"
embrulhar();",
1297
"
} ",
1298
"
return 1; ",
1299
"} \ n",
1300
"r_ck (que, N, M, src)",
1301
"
uchar * que; ",
1302
"
short * src; ",
1303
"{
int i, m = 0; \ n ",
1304
"#ifdef VERI",
1305
"
if (M == VERI) return; /* nenhuma informação útil lá */ ",
1306
"#fim se",
1307
"
```

```

printf (\ "não alcançado em proctype %% s: \\ n \\", procname [M]); ",
1308
"
para (i = 1; i <N; i ++) ",
1309
"
if (que [i] == 0) ",
1310
"
xrefsrc (src [i], M, i); ",
1311
"
outro",
1312
"
m ++; ",
1313
"
printf (\ " t (%% d de %% d estados) \\ n \\", N-1-m, N-1); ",
1314
"} \\ n",
1315
"xrefsrc (lno, M, i)",
1316
" {",
1317
"
printf (\ "\\ tline %% d (estado %% d) \\", lno, i); ",
1318
"
xrefstmtnt (M, i); ",
1319
"}",
1320
"xrefstmtnt (M, i)",
1321
" {",
1322
"
if (trans [M] [i] && trans [M] [i] -> tp) ",
1323
"
{
if (strcmp (trans [M] [i] -> tp, \\ "!) != 0) ",
1324
"
printf (\ ", \\\\" %% s \\\\ \"\\", trans [M] [i] -> tp); ",
1325
"
else if (stopstate [M] [i]) ",
1326
"
printf (\ ", -endstate- \\"); ",
1327
"
} outro",
1328
"
printf (\ ",? \\"); ",
1329
"
printf (\ "\\ n \\"); ",
1330
"} \\ n",
1331
"putrail (),
1332
"{
int fd, i, j, q; ",
1333
"
char snap [64]; \\ n ",
1334
"
if ((fd = creat (\ "pan.trail \\", 0666)) <= 0) ",
1335
"
{
printf (\ "não pode criar pan.trail \\ n \\"); ",
1336
"
Retorna;",

```

```

1337
"
}
1338
"#ifdef VERI",
1339
"
sprintf (snap, \ "- 2: %% d: -2: -2 \\ n \\", VERI); ",
1340
"
write (fd, snap, strlen (snap)); ",
1341
"#fim se",
1342
"
para (i = 1, j = 0; i <= profundidade; i++) ",
1343
"
{
q = trilha [i] .pr; ",
1344
"
if (i == depthfound) ",
1345
"
escrever (fd, \ "- 1: -1: -1: -1 \\ n \\", 12); ",
1346
"
if (loops) ",
1347
"#ifdef VERI",
1348
"
{
if (q == 2) continue; ",
1349
"
if (q> 2) q -= 2; ",

```

```

464
APÊNDICE E
PROJETO E VALIDAÇÃO
1350
"
}
1351
"#outro",
1352
"
{
if (q == 1) continue; ",
1353
"
if (q> 1) q--; ",
1354
"
}
1355
"#fim se",
1356
"
if (trilha [i] .o_t-> ist) ",
1357
"
{sprintf (snap, \ %% d: %% d: %% d: %% d \\ n \\", j ++, ",
1358
"
q, trilha [i] .o_t-> ist, i); ",
1359
"
write (fd, snap, strlen (snap)); ",
1360
"
}
1361
"
sprintf (snap, \ %% d: %% d: %% d: %% d \\ n \\", j ++, ",
1362
"

```

```

q, trilha [i] .o_t-> st, i); ",
1363
"
write (fd, snap, strlen (snap)); ",
1364
"
} ",
1365
"
printf (\ "pan: escreveu pan.trail \\n \"); ",
1366
"
fechar (fd); ",
1367
"} \ n",
1368
"sv_save ()
/* colocar vetor de estado na pilha de salvamento * /
1369
|{
if (! svtack-> nxt) ",
1370
"
{svtack-> nxt = (Svtack *) emalloc (sizeof (Svtack)); ",
1371
"
svtack-> nxt-> body = emalloc (vsize * sizeof (char)); ",
1372
"
svtack-> nxt-> lst = svtack; ",
1373
"
svtack-> nxt-> m_delta = vsize; ",
1374
"
svmax ++; ",
1375
"
} else if (vsize> svtack-> nxt-> m_delta) ",
1376
"
{svtack-> nxt-> body = emalloc (vsize * sizeof (char)); ",
1377
"
svtack-> nxt-> lst = svtack; ",
1378
"
svtack-> nxt-> m_delta = vsize; ",
1379
"
svmax ++; ",
1380
"
} ",
1381
"
svtack = svtack-> nxt; ",
1382
"#if SYNC",
1383
"
svtack-> o_boq = boq; ",
1384
"#fim se",
1385
"
svtack-> o_delta = vsize; ",
1386
"
memcpy ((char *) (svtack-> body), (char *) & now, vsize); ",
1387
"} \ n",
1388
"sv_restor ()
/* vetor de estado pop da pilha de salvamento * /
1389
|{
memcpy ((char *) & now, svtack-> body, svtack-> o_delta); ",
1390
"#if SYNC",
1391
"

```

```

boq = svtack-> o_boq; ",
1392
"#fim se",
1393
"
if (vsize! = svtack-> o_delta) ",
1394
"
Uerror (\ "sv_restor \");
1395
"
if (! svtack-> lst) ",
1396
"
Uerror (\ "erro: v_restor \");
1397
"
svtack = svtack-> lst; ",
1398
"} \ n",
1399
"p_restor (h)",
1400
|{
int i; char * z = (char *) & now; ",
1401
"
proc_offset [h] = pilha-> o_offset; ",
1402
"
proc_skip [h] = pilha-> o_skip; ",
1403
"
vsize += stack-> o_skip; ",

```

Página 476

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

465
1404
"
memcpy (z + vsize, stack-> body, stack-> o_delta); ",
1405
"
vsize += pilha-> o_delta; ",
1406
"
i = pilha-> o_delqs; ",
1407
"
agora._nr_pr += 1; ",
1408
"
if (! stack-> lst)
/* debugging */ ",
1409
"
Uerror (\ "erro: p_restor \");
1410
"
pilha = pilha-> lst; ",
1411
"
this = pptr (h); ",
1412
"
enquanto (i--> 0) ",
1413
"
q_restor (); ",
1414
"} \ n",
1415
"q_restor ()",
1416
|{
char * z = (char *) & now; ",
1417
"
q_offset [now._nr_qs] = stack-> o_offset; ",

```

```

1418
"
q_skip [agora._nr_qs] = pilha-> o_skip; ",
1419
"
vsize += stack-> o_skip; ",
1420
"
memcpy (z + vsize, stack-> body, stack-> o_delta); ",
1421
"
vsize += pilha-> o_delta; ",
1422
"
agora._nr_qs += 1; ",
1423
"
if (! stack-> lst)
/* debugging */ / ",
1424
"
Uerror (\ "erro: q_restor \"); ",
14: 25h
"
pilha = pilha-> lst; ",
1426
"} \ n",
1427
"delproc (sav, h)",
1428
"{
int d, i = 0; ",
1429
"",
1430
"
if (h + 1! = now._nr_pr) retorna 0; ",
1431
"",
1432
"
while (now._nr_qs ",
1433
"
&&
q_offset [now._nr_qs-1]> proc_offset [h]) ",
1434
"
{
delq (sav); ",
1435
"
i ++; ",
1436
"
},
1437
"
d = vsize - proc_offset [h]; ",
1438
"
if (sav) ",
1439
"
{
if (! stack-> nxt) ",
1440
"
{
pilha-> nxt = (pilha *) ",
1441
"
emalloc (sizeof (Stack)); ",
1442
"
stack-> nxt-> body = ",
1443
"
emalloc (Maxbody * sizeof (char)); ",
1444
"
stack-> nxt-> lst = stack; ",

```

```

1445
"
smax ++; ",
1446
"
} ",
1447
"
pilha = pilha-> nxt; ",
1448
"
pilha-> o_offset = proc_offset [h]; ",
1449
"
pilha-> o_skip = proc_skip [h]; ",
1450
"
pilha-> o_delta = d; ",
1451
"
pilha-> o_delqs = i; ",
1452
"
memcpy (pilha-> corpo, (char *) pptr (h), d); ",
1453
"
} ",
1454
"
vsize = proc_offset [h]; ",
1455
"
now._nr_pr = now._nr_pr - 1; ",
1456
"
memset ((char *) pptr (h), 0, d); ",
1457
"
vsize -= proc_skip [h]; ,

```

```

466
APÊNDICE E
PROJETO E VALIDAÇÃO
1458
"
return 1; ",
1459
"} \ n",
1460 #ifdef VARSTACK
1461
"pushvarval (v, ck)",
1462
"{
if (! varstack-> nxt) ",
1463
"
{
varstack-> nxt = (Varstack *) ",
1464
"
emalloc (sizeof (Varstack)); ",
1465
"
varstack-> nxt-> lst = varstack; ",
1466
"
vmax ++; ",
1467
"
} ",
1468
"
varstack = varstack-> nxt; ",
1469
"
varstack-> val = v; ",
1470
"
varstack-> cksum = ck; ",

```

```

1471
"} \ n",
1472
"popvarval (ck)",
1473
"{
if (! varstack-> lst) ",
1474
"
Uerror (\ "erro: popvar \");
1475
"
if (varstack-> cksum! = ck) ",
1476
"
{
printf (\ %% d <-> %% d \\ n \", varstack-> cksum, ck); ",
1477
"
Uerror (\ "mismatch varstack \");
1478
"
},
1479
"
varstack = varstack-> lst; ",
1480
"
return varstack-> nxt-> val; ",
1481
"} \ n",
1482 #endif
1483
"delq (sav)",
1484
"{
int h = now._nr_qs - 1; ",
1485
"
int d = vsize - q_offset [now._nr_qs - 1]; ",
1486
"
if (sav) ",
1487
"
{
if (! stack-> nxt) ",
1488
"
{
pilha-> nxt = (pilha *) ",
1489
"
emalloc (sizeof (Stack)); ",
1490
"
stack-> nxt-> body = ",
1491
"
emalloc (Maxbody * sizeof (char)); ",
1492
"
stack-> nxt-> lst = stack; ",
1493
"
smax ++; ",
1494
"
},
1495
"
pilha = pilha-> nxt; ",
1496
"
stack-> o_offset = q_offset [h]; ",
1497
"
stack-> o_skip = q_skip [h]; ",
1498
"
pilha-> o_delta = d; ",
1499
"

```

```

"
memcpy (pilha-> corpo, (char *) qptr (h), d); ",
1500
"
}
1501
"
vsize = q_offset [h]; ",
1502
"
now._nr_qs = now._nr_qs - 1; ",
1503
"
memset ((char *) qptr (h), 0, d); ",
1504
"
vsize -= q_skip [h]; ",
1505
"} \ n",
1506
"prognow ()",
1507
"｛",
1508
"
int i; P0 * ptr; ",
1509
"
para (i = 0; i < agora._nr_pr; i++) ",
1510
"
{
ptr = (P0 *) pptr (i); ",
1511
"
if (progstate [ptr -> _ t] [ptr -> _ p]) ",

```

Página 478

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

467
1512
"
return 1; ",
1513
"
}
1514
"
return 0; ",
1515
"} \ n",
1516
"endstate ()",
1517
"{
int i; P0 * ptr; ",
1518
"
para (i = 0; i < agora._nr_pr; i++) ",
1519
"
{
ptr = (P0 *) pptr (i); ",
1520
"#ifdef VERI",
1521
"
if (i == 1) continue; ",
1522
"#fim se",
1523
"
if (! stopstate [ptr -> _ t] [ptr -> _ p]) ",
1525
"
return 0; ",
1526

```

```

"
} ",
1527
"
if (loops) ",
1528
"
uerror (\ "sequência sem progresso \");
1529
"
return 1;
1530
"} \ n",
1531
"onstack ()",
1532
"{",
1533
"registrar Svtack * ptr; ",
1534
"
1535
"registrar int j = profundidade; ",
1536
"
para (ptr = svtack; ptr; ptr = ptr-> lst, j--) ",
1537
"
if (ptr-> o_delta == vsize ",
1538
"
&& ptr-> j1 == J1 && ptr-> j2 == J2 ",
1539
"
&& memcmp (ptr-> body, won, vsize) == 0) ",
1540
"
profundidade encontrada = j;
1541
"
return 1;
1542
"
1543
"
return 0;
1544
"} \ n",
1545
"fair_cycle ()",
1546
"{",
1547
"int i, j, q, II; ",
1548
"
Trans * t; ",
1549
"
short tt; ",
1550
"""
1551
"
se (! justiça) retornar 1;
1552
"
memset (movido, 0, MAXPROC);
1553
"#ifdef VERI",
1554
"
movido [1] = 1;
1555
"

```

```

"
if (loops) moveu [2] = 1; ",
1556
"#outro",
1557
"
if (loops) moveu [1] = 1; ",
1558
"#fim se",
1559
"
para (i = profundidade encontrada; i <= profundidade; i++) ",
1560
"
{
q = trilha [i] .pr; ",
1561
"#ifdef VERI",
1562
"
if (q == 1 || (loops && q == 2)) continue; ",
1563
"#outro",
1564
"
if (loops && q == 1) continue; ",
1565
"#fim se",

```

```

468
APÊNDICE E
PROJETO E VALIDAÇÃO
1566
"
movido [q] = 1; ",
1567
"
}
",
1568
"
para (II = 0; II < agora._nr_pr; II++) ",
1569
"
{
if (! mudou-se [II]) ",
1570
"
{
este = pptr (II); ",
1571
"
tt = (curto) ((P0 *) this) -> _p; ",
1572
"
ot = (uchar) ((P0 *) this) -> _t; ",
1573
"
para (t = trans [ot] [tt]; t; t = t-> nxt) ",
1574
"
{
",
1575
"#include \" pan.f \\"",
1576
"
goto not_fair; ",
1577
"
}
",
1578
"
}
",
1579
"
}
",
1580
"
/* um ciclo justo foi detectado */",
1581

```

```

"
para (i = profundidade encontrada-1; i <= profundidade; i ++) ",
1582
"
trilha [i] .tau & = ~2;
/* desmarcar estados no SCC */ ,
1583
"
return 1;
1584
"não é justo:",
1585
"
/* marca todos os estados no SCC sujo - para evitar a falta de * / ",
1586
"
/* travessias do mesmo SCC que poderiam ser geradas posteriormente * / ",
1587
"
para (i = profundidade encontrada-1; i <= profundidade; i ++) ",
1588
"
trilha [i] .tau |= 2; ",
1589
"
return 0;
1590
"} \ n",
1591
1592
"#if CONNECT> 0",
1593
"checkaccept ()",
1594
"{{
int i;
1595
"
para (i = 0; i < agora._nr_pr; i ++) ",
1596
"
{
P0 * ptr = (P0 *) pptr (i); ",
1597
"
if (accpstate [ptr -> _t] [ptr -> _p]) ",
1598
"
quebrar;",
1599
"
}
1600
"
if (i == now._nr_pr) ",
1601
"
Retorna;",
1602
"
if (now._a_t) ",
1603
"
{",
1604
"
Retorna;",
1605
"
}
1606
"
agora._a_t = 13; /* 13 para ajudar o hasher */ ,
1607
"
A_depth = profundidade; ",
1608
"
memcpy ((char *) & A_Root, (char *) & now, vsiz);
1609
"
profundidade encontrada = profundidade; ",

```

```

1610
"
new_state ();
/* a 2ª pesquisa * /
1611
"
profundidade encontrada = -1; ",
1612
"
agora._a_t = 0; ",
1613
")",
1614
#endif \ n",
1615
1616
#ifndef BITSTATE",
1617
"struct H_el {",
1618
"
struct H_el * nxt; ",
1619 #if 0

```

Página 480

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

469
1620
"#ifdef CACHE",
1621
"
struct H_el * lst, * unxt; ",
1622
"
hslot sem sinal; ",
/* slot de tabela de hash */
1623
"#fim se",
1624 #endif
1625
"
sz sem sinal; ",
/* tamanho do vetor de estado */
1626
"
unsigned tagged; ",
/* bits 30 e 31 são especiais */
1627
"
estado sem sinal; ",
1628
"} ** H_tab; \ n",
1629
1630
"hinit ()",
1631
|{
H_tab = (struct H_el **) ",
1632
"
emalloc ((1 << ssize) * sizeof (struct H_el *)); ",
1633 #if 0
1634
"
printf (\ "a tabela de hash usa % d bytes \\n \", ",
1635
"
(1 << ssize) * sizeof (struct H_el *)); ",
1636
"
fflush (stdout); ",
1637 #endif
1638
"} \ n",
1639
"#ifdef CACHE",
1640
"#include \" nh_store.c \ """",
1641

```

```

"#fim se",
1642
1643
"struct H_el * Free_list = 0;
/* recicla estados removidos */",
1644
"",
1645
"recycle_state (v, n)",
1646
"
struct H_el * v; ",
1647
"
curto n; ",
1648
"{
1649
"
struct H_el * tmp, * last = 0; ",
1650
"
v-> marcado = n; ",
1651
"
v-> nxt = 0; ",
1652
"
para (tmp = lista_free; tmp; último = tmp, tmp = tmp-> nxt) ",
1653
"
{
if (tmp-> tagged> = n) ",
1654
"
{
if (last) ",
1655
"
{
v-> nxt = tmp-> nxt; ",
1656
"
último-> nxt = v; ",
1657
"
} outro",
1658
"
{
v-> nxt = Free_list; ",
1659
"
Free_list = v; ",
1660
"
} ",
1661
"
Retorna;",
1662
"
}
",
1663
"
if (! tmp-> nxt) ",
1664
"
{
tmp-> nxt = v; ",
1665
"
Retorna;",
1666
"
}
",
1667
"
Free_list = v; ",
1668
"}",

```

```
1669
"",
1670
"struct H_el *",
1671
"grab_state (n)",
1672
"{
struct H_el * tmp, * last = 0; ",
1673
"",
1674
```

```
470
APÊNDICE E
PROJETO E VALIDAÇÃO
1674
"
para (tmp = lista_free; tmp; último = tmp, tmp = tmp-> nxt) ",
1675
"
if (tmp-> tagged == n) ",
1676
"
{
if (last) ",
1677
"
último-> nxt = tmp-> nxt; ",
1678
"
outro",
1679
"
Free_list = tmp-> nxt; ",
1680
"
tmp-> nxt = 0; ",
1681
"
reciclado ++; ",
1682
"
return tmp; ",
1683
"
}
1684
"
return (struct H_el *) ",
1685
"
emalloc (sizeof (struct H_el) + n-sizeof (unsigned)); ",
1686
"}",
1687
"",
1688
1689
"htag (V, N)",
1690
"
char * V; ",
1691
"
curto N; ",
1692
"{",
1693
"
registrar struct H_el * tmp, * last = 0; ",
1694
"
char * v; curto n; ",
1695
"#ifdef COMPRESS",
1696
"
n = comprimir (& v, V, N); ",
1697
```

```

"#outro",
1698
"
n = N; v = V; ",
1699
"#fim se",
1700
"
s_hash (v, n); ",
1701
"
para (tmp = H_tab [j1]; tmp, último = tmp, tmp = tmp-> nxt) ",
1702
"
{
E se (",
1703
"#ifdef CACHE",
1704
"
tmp-> sz == n && ",
1705
"#fim se",
1706
"
memcmp (((char *) & (tmp-> state)), v, n) == 0) ",
1707
"
{",
1708
"#ifdef CACHE",
1709
"
if (tmp-> marcado & (1 << 31)) Uerror (\ "Double Htag \"); ",
1710
"#fim se",
1711
"
tmp-> marcado &= (1 << 30); /* preservar apenas o bit 30 */ ,
1712
"#if CONNECT == 0",
1713
"#ifdef CACHE",
1714
"
tmp-> marcado |= (1 << 31); /* definir bit 31 */ ,
1715
"#fim se",
1716
"#fim se",
1717
"
if (trpt-> tau & 2) /* estado marcado como sujo: remover */ ,
1718
"
{
if (last) ",
1719
"
último-> nxt = tmp-> nxt; ",
1720
"
outro",
1721
"
H_tab [j1] = tmp-> nxt; ",
1722
"
recycle_state (tmp, n); ",
1723
"
} ",
1724
"
Retorna;",
1725
"
} ",
1726
"
} ",
1727

```

"
para (tmp = H_tab [j1], n = 0; tmp; tmp = tmp-> nxt) ",

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN
471
1728
"
n ++; ",
1729
"
printf (\ "não pode acontecer, htag, o comprimento da lista é % % d \\ \ n \ ", n); ",
1730
"
fflush (stdout); ",
1731
"/ *
Uerror (\ "não pode acontecer, htag \ ");
* / ",
1732
"} \ n",
1733
"#ifdef COMPRESS",
1734 #include "compress.h"
1735
"#fim se",
1736
1737
"#ifndef CACHE",
1738
"hstore (V, N)",
1739
"
char * V; ",
1740
"
curto N; ",
1741
"{",
1742
"
registrar struct H_el * tmp; \ n ",
1743
"
char * v; curto n; ",
1744
"#ifdef COMPRESS",
1745
"
n = comprimir (& v, V, N); ",
1746
"#outro",
1747
"
n = N; v = V; ",
1748
"#fim se",
1749
"
if (Normalizar) retorna 1; ",
1750
"
s_hash ((uchar *) v, n); ",
1751
"
tmp = H_tab [j1]; ",
1752
"
if (! tmp) ",
1753
"
{tmp = grab_state (n); ",
1754
"
H_tab [j1] = tmp; ",
1755
"
} outro",
1756

```

"
{para (++; hcmp ++) ",  

1757
"
{if (memcmp (& (tmp-> estado), v, n) == 0) ",  

1758
"
{
if (tmp-> marcado & ~ ((1 << 30) | (1 << 31))) ",  

1759
"
{
if (loops && now._p_t) ",  

1760
"
{
deepfound = tmp-> marcado & ~ (1 << 30); ",  

1761
"
if (fair_cycle ()) ",  

1762
"
uerror (\ "ciclo de não progresso \"); ",  

1763
"
} ",  

1764
"
return 2; /* combinar na pilha */ ",  

1765
"
} outro",
1766
"
return 1; /* corresponder fora da pilha */ ",  

1767
"
} ",  

1768
"
if (! tmp-> nxt) break; ",  

1769
"
tmp = tmp-> nxt; ",  

1770
"
} ",  

1771
"
tmp-> nxt = grab_state (n); ",  

1772
"
tmp = tmp-> nxt; ",  

1773
"
} ",  

1774
"
tmp-> marcado = profundidade + 1; /* diferente de zero enquanto na pilha */ ",  

1775
"
memcpy (((char *) & (tmp-> estado)), v, n); ",
1776 #ifdef PARES
1777
"#ifdef PARES",
1778
"
if (boq == -1) pares (); ",
1779
"#fim se",
1780 #endif
1781
"
return 0; ",

```

```

    "}",
1783 "#fim se",
1784 "#fim se",
1785 "#include \" pan.t \\ \"",
1786 "",
1787 "do_reach ()",
1788 "{",
1789 0,
1790 };
1791 1792 / ***** spin: pangen2.h ***** /
1793 1794 char * Preâmbulo [] = {
1795 "#incluir
<stdio.h> ",
1796 "#incluir
<signal.h> ",
1797 "#incluir
\\ \" \\ n ",
1798 "#define max (a, b) (((a) <(b)) ? (b) : (a))",
1799 "typedef struct Trail {",
1800 "
curto pr;
/* id do processo
* /",
1801 "
st curto;
/* Estado atual */",
1802 "
uchar tau;
/* sinalizadores de status */",
1803 "
char o_n, o_ot, o_m; /* para salvar locais */
1804 "
curto o_tt, o_To; /* usado em new_state () */"
1805 "
Trans * o_t; /* transição fct, próximo estado */
1806 "
int oval; /* valor de backup de uma variável */
1807 "
} Trilha;",
1808 "
Trilha * trilha, * trpt;",
1809 "
uchar * this; \\ n",
1810 "
int maxdepth = 10000;",
1811 "
uchar * SS, * LL;",
1812 "
char * emalloc (), * malloc (), * memset ();",
1813 "
int mreached = 0, done = 0, errors = 0;",
1814 "
nstates longos = 0, reciclado = 0;",
1815 "
nlinks longos = 0, truncs = 0, perda = 0;",
1816 "
máscara interna, hcmp = 0, loops = 0, aciclos = 0, até = 1;",
1817 "
int state_tables = 0, fairness = 0, homomorphism = 0;",
1818 "

```

```

"char * hom_target, * hom_source;",
1819 #ifdef PARES
1820
"int tree_before = 0;",
1821 #endif
1822
"#ifdef BITSTATE",
1823
"int ssize = 22;",
1824
"#outro",
1825
"int ssize = 18;",
1826
"#fim se",
1827
"int hmax = 0, svmax = 0, smax = 0;",
1828 #ifdef VARSTACK
1829
"int vmax = 0;",
1830 #endif
1831 #ifdef GODEF
1832
"int cs_max = 0;",
1833 #endif
1834
"int Maxbody = 0;",
1835
"uchar * nopr;
/* usado pela macro Pptr (x) */",

```

Página 484

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

473
1836
"Estado A_Root;
/* raiz dos ciclos de aceitação */,
1837
"Declare agora;
/* o vetor de estado completo */,
1838
"Pilha * pilha;
/* para filas, processos */,
1839
"Svtack * svstack;
/* para vetores de estado antigo */,
1840 #ifdef VARSTACK
1841
"Varstack * varstack;
/* para variáveis antigas vals */,
1842 #endif
1843 #ifdef GODEF
1844
"CS_stack * cs_stack;
/* conjuntos de conflito */,
1845 #endif
1846
"int J1, J2, j1, j2, j3, j4;",
1847
"int A_depth = 0; \ n",
1848
"profundidade interna = 0; \ n",
1849
"#if SYNC",
1850
"#define IfNotBlocked if (boq! = -1) continue;",
1851
"#define Desbloquear
boq = -1 ",
1852
"#outro",
1853
"#define IfNotBlocked /* não pode bloquear */",
1854
"#define Desbloquear
/* não incomoda */",
1855
"#endif \ n",
1856

```

```

0,
1857};
1858 char * Tail [] = {
1859
"Trans *",
1860
"settr (a, b, c, d, t, l, ntp)",
1861
"
gráfico;",
1862
"{
Trans * tmp = (Trans *) emalloc (sizeof (Trans)); \n",
1863
"
tmp-> átomo = a & 6; ",
1864
"
tmp-> st = b; ",
1865
"
tmp-> local = l; ",
1866
"
tmp-> tp = t; ",
1867
"
tmp-> ntp = ntp; ",
1868
"
tmp-> forw = c; ",
1869
"
tmp-> back = d; ",
1870 #ifdef GODEF
1871
"#ifdef VERBOSE",
1872
"
Move [c] = t; ",
1873
"#fim se",
1874 #endif
1875
"
return tmp; ",
1876
"} \ n",
1877 #ifdef PARES
1878
"#define Visitados 1 << 12",
1879
"dfs (p, t, srcln)",
1880
"
short srcln [];",
1881
"{
Trans * n; char * wtyp (); ",
1882
"",
1883
"
if (t == 0 || (trans [p] [t] -> átomo e visitado)) ",
1884
"
{
printf (\% dl \", t); ",
1885
"
Retorna;",
1886
"
}
",
1887
"
trans [p] [t] -> átomo |= visitado; ",
1888
"
printf (\% d \\% \ % d \% s \% s \\% "d \", ",
1889
"

```

t, srcln [t], ",

```
474
APÊNDICE E
PROJETO E VALIDAÇÃO
1890
"
(trans [p] [t] -> átomo & 2)? \ "* \": \ "\\", ",
1891
"
trans [p] [t] -> tp); ",
1892
"
if (trans [p] [t] -> st) ",
1893
"
dfs (p, trans [p] [t] -> st, srcln); ",
1894
"
para (n = trans [p] [t] -> nxt; n; n = n-> nxt) ",
1895
"
dfs (p, n-> st, srcln); ",
1896
"
printf (\ "u \"); ",
1897
"} \ n",
1898
"Tree (p, strt, srcln)",
1899
"
short srcln []",
1900
"{",
printf (\ "echo \"); ",
1901
"
dfs (p, strt, srcln); ",
1902
"
printf (\ "| 2.árvore \\ n \"); ",
1903
"} \ n",
1904 #endif
1905
"#ifdef JUMBO",
1906
"#define Visitados 1 << 12",
1907
"#define Concluído 1 << 13",
1908
"int",
1909
"jumbo_list (p, t)",
1910
"{",
int all_local; Trans * n; ",
1911
"
int nl, nxt_local; ",
1912
1913
"
if (t == 0) retorna 0; ",
1914
"
if (trans [p] [t] -> átomo e visitado) ",
1915
"
{
if (trans [p] [t] -> átomo e Completo) ",
1916
"
return trans [p] [t] -> Local; ",
1917
"
outro",
1918
```

```

"
return 0; ",
1919
"
} ",
1920
"
n = trans [p] [t]; ",
1921
"
n-> átomo |= visitado; ",
1922
1923
"
nxt_local = lista_jumbo (p, n-> st); ",
1924
1925
"
if (n-> nxt && n-> ntp! = 'c') ",
1926
"
all_local = 0; ",
1927
"
outro",
1928
"
all_local = n-> local; ",
1929
1930
"
para (n = n-> nxt; n; n = n-> nxt) ",
1931
"
{
nl = lista_jumbo (p, n-> st); ",
1932
"
if (nl <nxt_local) nxt_local = nl; ",
1933
"
if (! n-> local || n-> ntp! = 'c') all_local = 0; ",
1934
"
}
1935
"
if (all_local! = 0) ",
1936
"
all_local = nxt_local + 1; ",
1937
"
trans [p] [t] -> Local = all_local; ",
1938
"
trans [p] [t] -> átomo |= Concluido; ",
1939
"
return all_local; ",
1940
"}",
1941
"#fim se",
1942
"Trans *",
1943
"cpytr (a)",

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

475
1944
"
Trans * a; ",
1945
"{  
Trans * tmp = (Trans *) emalloc (sizeof (Trans)); \ n ",  
1946

```

```

"
tmp-> átomo = a-> átomo; ",
1947
"
tmp-> st = a-> st; ",
1948
"
tmp-> ist = a-> ist; ",
1949
"
tmp-> local = a-> local; ",
1950
"
tmp-> forw = a-> forw; ",
1951
"
tmp-> back = a-> back; ",
1952
"
tmp-> tp = a-> tp; ",
1953
"
tmp-> ntp = a-> ntp; ",
1954
"
return tmp; ",
1955
"} \ n",
1956
"int cnt;",
1957
"retrans (n, m, is, srcln, alcance) / * proc n, m estados, é = estado inicial * /",
1958
"
short srcln []; ",
1959
"
uchar alcance []; ",
1960
"{ 
Trans * T0, * T1, * T2, * T3; ",
1961
"
int i, j = 0; ",
1962
"
if (state_tables == 2) ",
1963
"
{
printf (\ "RAW proctype %% s \\n \\n", ",
1964
"
procname [n]); ",
1965
"
para (i = 1; i <m; i ++) ",
1966
"
alcance [i] = 1; ",
1967
"#ifdef JUMBO",
1968
"
lista_jumbo (n, é); ",
1969
"#fim se",
1970
"
tagtable (n, m, is, srcln, alcance); ",
1971
"
Retorna;",
1972
"
}
",
1973
"
Faz {
j ++; ",
1974
"

```

```

para (i = 1, cnt = 0; i <m; i ++) ",
1975
"
{
T1 = trans [n] [i] -> nxt; ",
1976
"
T2 = trans [n] [i]; ",
1977
"/ * pré-digitalização: * /
para (T0 = T1; T0; T0 = T0-> nxt) ",
1978
"/ * escolha dentro da escolha * /
if (trans [n] [T0-> st] -> nxt) ",
1979
"
quebrar;",
1980
"
if (T0) ",
1981
"
para (T0 = T1; T0; T0 = T0-> nxt) ",
1982
"
{
T3 = trans [n] [T0-> st]; ",
1983
"
if (! T3-> nxt) ",
1984
"
{
T2-> nxt = cpytr (T0); ",
1985
"
T2 = T2-> nxt; ",
1986
"
imed (T2, T0-> st, n); ",
1987
"
continuar;",
1988
"
},
1989
"
Faz {
T3 = T3-> nxt; ",
1990
"
T2-> nxt = cpytr (T3); ",
1991
"
T2 = T2-> nxt; ",
1992
"
imed (T2, T0-> st, n); ",
1993
"
} while (T3-> nxt); ",
1994
"
cnt++; ",
1995
"
},
1996
"
},
1997
"
} while (cnt); ",

```

```

"
para (i = 1; i <m; i ++) ",
1999
"
if (trans [n] [i] -> nxt) /* otimize um pouco a lista */ ,
2000
"
{
T1 = trans [n] [i] -> nxt; ",
2001
"
T0 = trans [n] [i] = cpytr (trans [n] [T1-> st]); ",
2002
"
imed (T0, T1-> st, n); ",
2003
"
para (T1 = T1-> nxt; T1; T1 = T1-> nxt) ",
2004
"
{
T0-> nxt = cpytr (trans [n] [T1-> st]); ",
2005
"
T0 = T0-> nxt; ",
2006
"
imed (T0, T1-> st, n); ",
2007
"
}
} ",
2008
"Mais:",
2009
"
if (state_tables ",
2010
"
|| (homomorfismo == 1 && strcmp (hom_target, procname [n]) == 0) ",
2011
"
|| (homomorfismo == 2 && strcmp (hom_source, procname [n]) == 0)) ",
2012
"
{
if (n == 0 && homomorfismo == 1) ",
2013
"
printf (\ \"\"); ",
2014
"
printf (\ "proctype %% s %% s %% s %% s \", ",
2015
"
homomorfismo == 1? \ "O \": \ "\\", ",
2016
"
homomorfismo == 2? \ "R \": \ "\\", ",
2017
"
procname [n], ",
2018
"
homomorfismo? \ "() \\ n {\\ n \": \ "\\ n \\"); ",
2019
"
para (i = 1; i <m; i ++) ",
2020
"
alcance [i] = 1; ",
2021
"#ifdef JUMBO",
2022
"
lista_jumbo (n, é); ",
2023
"#fim se",
2024
"
tagtable (n, m, is, srcln, alcance); ",
2025

```

```

"
if (! state_tables) ",
2026
"
printf (\ "S0:
pular \\ n} \\ n \ ");",
2027
"
}
2028
"
switch (homomorfismo) {",
2029
"
caso 1: homomorfismo = 2; vá para mais; ",
2030
"
caso 2: homomorfismo = 1; quebrar;",
2031
"
padrão: break; ",
2032
"
}
2033
"}",
2034
"imed (T, v, n) /* definir estado intermediário */",
2035
"
Trans * T; ",
2036
"{
uchar estático avisado = 0; ",
2037
"
if (T-> ist &&! warned) ",
2038
"
{
avisado = 1; ",
2039
"
printf (\ "aviso: %% s has \", procname [n]); ",
2040
"
printf (\ "estruturas ctl de fluxo ambíguo, \");
2041
"
printf (\ "revisar modelo \\ n \"); ",
2042
"
}
2043
"
progstate [n] [T-> st] |= progstate [n] [v]; ",
2044
"
accpstate [n] [T-> st] |= accpstate [n] [v]; ",
2045
"
stopstate [n] [T-> st] |= stopstate [n] [v]; ",
2046
"
T-> ist = v; ",
2047
"}",
2048
"tagtable (n, m, is, srcln, alcance)",
2049
"
short srcln []; ",
2050
"
uchar alcance []; ",
2051
" {",

```

FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```
477
2052
"
Trans * z; ",
2053
"
if (is> = m ||! trans [n] [is] ",
2054
"
|| é <= 0 || alcance [é] == 0) ",
2055
"
Retorna;",
2056
"
alcance [é] = 0; ",
2057
"
if (homomorfismo) ",
2058
"
{
if (accpstate [n] [is]) ",
2059
"
printf (\ "aceitar _ %% d: \\ n \", é); ",
2060
"
if (stopstate [n] [is]) ",
2061
"
printf (\ "end _ %% d: \\ n \", é); ",
2062
"
if (progstate [n] [is]) ",
2063
"
printf (\ "progress _ %% d: \\ n \", é); ",
2064
"
printf (\ "S %% d: \", é); ",
2065
"
if (homomorfismo == 1) ",
2066
"
printf (\ "t! trans -> \\ n \"); ",
2067
"
printf (\ " tif \\ n \"); ",
2068
"
para (z = trans [n] [é]; z; z = z-> nxt) ",
2069
"
{
printf (\ " t :: \"); ",
2070
"
Rachadura (n, is, z, srcln); ",
2071
"
} ",
2072
"
printf (\ " tfi; \\ n \"); ",
2073
"
} outro",
2074
"
if (state_tables) ",
2075
"
para (z = trans [n] [é]; z; z = z-> nxt) ",
2076
"
crack (n, is, z, srcln); ",
2077
"
para (z = trans [n] [é]; z; z = z-> nxt) ",
```

```
2078
"
tagtable (n, m, z-> st, srcln, alcance); ",
2079
"}",
2080
"uniq_trans (str)",
2081
"
char * str; ",
2082
"{",
int j; ",
2083
"
static int n_have = 0; ",
2084
"
typedef struct HAVE {",
2085
"
char * s; ",
2086
"
struct HAVE * n; ",
2087
"
} TER;",
2088
"
HAVE * t, * tt; ",
2089
"
estatico HAVE * h = 0; ",
2090
"
para (t = h, tt = 0, j = 0; t; tt = t, t = t-> n, j ++) ",
2091
"
if (strcmp (t-> s, str) == 0) ",
2092
"
return j; ",
2093
"
t = (HAVE *) emalloc (sizeof (HAVE)); ",
2094
"
t-> s = str; ",
2095
"
if (! h) ",
2096
"
h = t; ",
2097
"
outro",
2098
"
tt-> n = t; ",
2099
"
return j; ",
2100
"
"}",
2101
"putsource (s)",
2102
"
char * s; ",
2103
"{",
int i; ",
2104
"
para (i = 0; s [i]; i ++) ",
2105
"
if (s [i] == \\ n \\") ",
```

```
478
APÉNDICE E
PROJETO E VALIDAÇÃO
2106
"
printf (\\"\\n \\"); ",
2107
"
outro",
2108
"
putchar (s [i]); ",
2109
"}",
2110
"Crack (n, j, z, srcln)",
2111
"
Trans * z; ",
2112
"
short srcln [] ;",
2113
"{
int i; ",
2114
"
if (! z) return; ",
2115
"
i = 1 + uniq_trans (z-> tp); ",
2116
"
se (z-> átomo & 6) i = -i; ",
2117
"
if (strcmp (z-> tp, \"(1) \") == 0) ",
2118
"
{
printf (\\"ignorar; ir para S %% d
\", z-> st);",
2119
"
printf (\\"/* line %% 3d */ \\n \\", srcln [j]); ",
2120
"
Retorna;",
2121
"
} ",
2122
"#if 0",
2123
"
if (z-> local && strcmp (z-> tp, \"@ \") != 0) ",
2124
"
{
putsource (z-> tp); ",
2125
"
printf (\\"; ir para S %% d
\", z-> st);",
2126
"
printf (\\"/* line %% 3d */ \\n \\", srcln [j]); ",
2127
"
Retorna;",
2128
"
} ",
2129
"#fim se",
2130
"
if (homomorfismo == 1) ",
2131
"
```

```

printf (\ atomic {trans = %% 2d; \", i); ",
2132
"
else / * homomorfismo == 2 * / ",
2133
"
printf (\ atômico {(trans == %% 2d); trans = 0; \", i); ",
2134
"
printf (\ goto S %% d} \", z-> st); ",
2135
"
printf (\ "
/ * linha %% 3d, \", srcln [j]);",
2136
"
putsource (z-> tp); ",
2137
"
printf (\ * / \\ n \"); ",
2138
"
fflush (stdout); ",
2139
"}",
2140
"crack (n, j, z, srcln)",
2141
"
Trans * z; ",
2142
"
short srcln []; ",
2143
"{{
int i; ",
2144
"
if (! z) return; ",
2145
"printf (\ tstate %% 2d - [%% 2d] -> estado %% 2d [%% s %% s %% s %% s %% s] (%% d) linha
%% 3d => \",",
2146
"
j, z-> forw, z-> st, ",
2147
"
z-> átomo e 6? \ "A \": \ "- \", ",
2148
"
z-> local? \ "L \": \ "- \", ",
2149
"
accpstate [n] [j]? \ "a \": \ "- \", ",
2150
"
stopstate [n] [j]? \ "e \": \ "- \", ",
2151
"
progstate [n] [j]? \ "p \": \ "- \", ",
2152
"
z-> Local, ",
2153
"
srcln [j]); ",
2154
"
putsource (z-> tp); ",
2155
"
printf (\ \\ n \"); ",
2156
"
fflush (stdout); ",
2157
"}",
2158
0,
2159};


```

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN
479
2160
2161 / ***** spin: pangen3.h ***** /
2162
2163 char * R0 [] = {
2164 "
Maxbody = max (Maxbody, sizeof (P% d)); ",
2165 "
atingiu [% d] = atingiu% d; ",
2166 "
accpstate [% d] = (uchar *) emalloc (nstates% d); ",
2167 "
progstate [% d] = (uchar *) emalloc (nstates% d); ",
2168 "
stopstate [% d] = (uchar *) emalloc (nstates% d); ",
2169 "
stopstate [% d] [endstate% d] = 1; ",
2170
0,
2171};
2172 char * R0a [] = {
2173 #ifdef PARES
2174 "
if (tree_before) Tree (% d, start% d, src_ln% d); ",
2175 #endif
2176 "
retransmitir (% d, nstates% d, iniciar% d, src_ln% d, alcançado% d); ",
2177
0,
2178};
2179 char * R0b [] = {
2180 "
if (state_tables) ",
2181 "
{
printf (\\"\\nTipos de transição:
\\");",
2182 "
printf (\\"A = atômico; L = local; \\n \");",
2183 "
printf (\\"Source-State Labels: \");",
2184 "
printf (\\"p = progresso; e = fim; a = aceitar; \\n \");",
2185 "
}
",
2186 "
if (homomorfismo) ",
2187 "
printf (\\"init {atomic {executar O %% s (); executar R %% s ()}} \\n \", ",
2188 "
hom_target, hom_source); ",
2189 "
if (state_tables || homomorphism) ",
2190 "
Saída();",
2191
0,
2192};
2193 char * R1 [] = {

```

```

2194
"
atingiu [% d] = (uchar *) emalloc (4 * sizeof (uchar)); ",
2195
"
stopstate [% d] = (uchar *) emalloc (4 * sizeof (uchar)); ",
2196
"
progstate [% d] = stopstate [% d]; ",
2197
"
accpstate [% d] = stopstate [% d]; ",
2198
0,
2199};
2200 char * R2 [] = {
2201
"uchar * accpstate [% d];",
2202
"uchar * progstate [% d];",
2203
"uchar * atingiu [% d];",
2204
"uchar * stopstate [% d];",
2205
0,
2206};
2207 char * R3 [] = {
2208
"
Maxbody = max (Maxbody, sizeof (Q% d)); ",
2209
0,
2210};
2211 char * R4 [] = {
2212
"
r_ck (alcançado% d, nstates% d,% d, src_ln% d); ",
2213
0,

```

```

480
APÊNDICE E
PROJETO E VALIDAÇÃO
2214};
2215 char * R5 [] = {
2216
"
caso% d: j = sizeof (P% d); quebrar;",
2217
0,
2218};
2219 char * R6 [] = {
2220
"
caso% d: /* verificador de progresso */",
2221
"
((P% d *) pptr (h)) -> _ t ==% d; ",
2222
"
((P% d *) pptr (h)) -> _ p = 1; ",
2223
"
agora._p_t = 0; ",
2224
"
quebrar;",
2225
"
} ,
2226
"#ifdef VERI",
2227
"
if (h == 0 &&! addproc (VERI)) ",
2228
"
return 0; ",

```

```

2229
"#fim se",
2230
"
if (h == 0 && loops &&! addproc (% d)) ",
2231
"
return 0;
2232
"#ifdef VERI",
2233
"
return (h> 0)? h-loops-1: 0;
2234
"#outro",
2235
"
return (h> 0)? h-loops: 0;
2236
"#fim se",
2237
"} \ n",
2238
0,
2239};
2240 char * R8 [] = {
2241
"
caso% d: j = sizeof (Q% d); quebrar;",
2242
0,
2243};
2244 char * R9 [] = {
2245
"typedef struct Q% d {",
2246
"
uchar Qlen;
/* q_size */ / ",
2247
"
uchar _t;
/* q_type */ / ",
2248
"
struct {",
2249
0,
2250};
2251 char * R10 [] = {
2252
"typedef struct Q0 {\ \ t / * q genérico * /",
2253
"
uchar Qlen, _t; ",
2254
"} Q0;",
2255
0,
2256};
2257 char * R12 [] = {
2258
"\ t \ tcase% d: r = ((Q% d *) z) -> conteúdo [slot] .fld% d; break;",
2259
0,
2260};
2261 char * R13 [] = {
2262
"unsend (into)",
2263
|{
int m = 0, j; uchar * z; ",
2264
"
if (! into--) uerror (\ "referência ao nome chan não inicializado \");
2265
"
z = qptr (em); ",
2266
"
j = ((Q0 *) z) -> Qlen; ",
2267

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```
481
2268
"
switch (((Q0 *) qptr (into)) -> _ t) {" ,
2269
0,
2270};
2271 char * R14 [] = {
2272
"
padrão: Uerror (\ "fila inválida - não enviar \"); ",
2273
"
} ,
2274
"
return m; ,
2275
"} ,
2276
"",
2277
"unrecv (from, slot, fld, fldvar, strt)",
2278
"{
int j; uchar * z; ,
2279
"
if (! from--) uerror (\ "referência ao nome chan não inicializado \"); ",
2280
"
z = qptr (de); ,
2281
"
j = ((Q0 *) z) -> Qlen; ,
2282
"
if (strt) ((Q0 *) z) -> Qlen = j + 1; ,
2283
"
switch (((Q0 *) qptr (from)) -> _ t) {" ,
2284
0,
2285};
2286 char * R15 [] = {
2287
"
padrão: Uerror (\ "fila inválida - qrecv \"); ",
2288
"
} ,
2289
"} ,
2290
0,
2291};
2292
2293 / ***** spin: pangen1.c ***** /
2294
2295 #include <stdio.h>
2296 #include <math.h>
2297 #include "spin.h"
2298 #include "y.tab.h"
2299 #include "pangen1.h"
2300 #include "pangen3.h"
2301
2302 extern FILE
* tc, * th;
2303 Nô externo
* Mtype;
2304 extern ProcList
* rdy;
2305 fila externa
* qtab;
```

```

2306 extern RunList
*corre;
2307 simbolo externo
* symtab [Nhash + 1];
2308 extern int nqs, nps, mst, Mpars;
2309 caracteres externos
* Claimproc;
2310
2311 enum {INIV, PUTV};
2312
2313 Tipos curtos [] = {BIT, BYTE, CHAN, SHORT, INT};
2314 int Npars = 0, u_sync = 0, u_async = 0;
2315 aceitadores int = 0;
2316
2317 vazio
2318 genheader ()
2319 {ProcList * p;
2320
int i;
2321

```

```

482
APÊNDICE E
2322
fprintf (th, "#define SYNC
% d \ n ", u_sync);
2323
fprintf (th, "#define ASYNC
% d \ n \ n ", u_async);
2324
fprintf (tc, "char * procname [] = {\n");
2325
put_ptype (run-> n-> name, (Node *) 0, 0, mst, nps);
2326
para (p = rdy, i = 1; p; p = p-> nxt, i++)
2327
put_ptype (p-> n-> nome, p-> p, i, mst, nps);
2328
put_ptype ("_ progress", (Nó *) 0, i, mst, nps);
2329
fprintf (tc, "}; \ n \ n");
2330
n vezes (th, 0, 1, Cabeçalho);
2331
doglobal (PUTV);
2332
fprintf (th, "uchar sv [VECTORSZ]; \ n");
2333
fprintf (th, "} Estado; \ n");
2334 #ifdef GODEF
2335
{
Símbolo * sp; extern int uniq, Maxcs;
2336
int j, k = 0;
2337
fprintf (th, "\ n / *** Números de conjuntos de conflitos *** / \ n");
2338
fprintf (th, "#define CS_timeout \ t% d \ n", k++);
2339
para (j = 0; j <5; j++) / * para cada tipo de dados * /
2340
para (i = 0; i <= Nhash; i++)
2341
para (sp = symtab [i]; sp; sp = sp-> próximo)
2342
if (sp-> type == Tipos [j])
2343
{
if (sp-> contexto)
2344
continuar;
2345
fprintf (th, "#define CS_% s \ t% d \ n", sp-> nome, k);
2346
k += sp-> nel;
2347
}
2348

```

```

fprintf (th, "\nchar * CS_names [] = {\n");
2349
fprintf (th, "\" tempo limite \", \n");
2350
if (k> 1)
2351
{
int a = 0;
2352
para (j = 0; j <5; j++)
2353
para (i = 0; i <= Nhash; i++)
2354
para (sp = symtab [i]; sp; sp = sp-> próximo)
2355
if (sp-> type == Tipos [j])
2356
{
if (sp-> contexto)
2357
continuar;
2358
if (sp-> nel == 1)
2359
fprintf (th, "% s \ ", \ n", sp-> nome);
2360
outro
2361
para (a = 0; a <sp-> nel; a++)
2362
fprintf (th, "% s [% d] \ ", \ n",
2363
sp-> nome, a);
2364
}
2365
}
2366
fprintf (th, "}; \ n");
2367
fprintf (th, "#define MAXSTATE% d \ n", uniq + 2);
2368
/* adicionou 2 para os dois estados dos verificadores de progresso */
2369
fprintf (th, "#define TOPQ
(1 + MAXCONFL + MAXQ) \ n ");
2370
fprintf (th, "/ * Maxcs = \ n");
2371
fprintf (th, "* nr máximo de cs que qualquer 1 instrução \ n");
2372
fprintf (th, "* pode estar esperando a qualquer momento \ n");
2373
fprintf (th, "* / \ n");
2374
fprintf (th, "#define MAXCS
% d \ n ", Maxcs);
2375
fprintf (th, "#define MAXCONFL% d \ n", k);

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

483
2376
fprintf (th, "#ifndef MULT \ n");
2377
fprintf (th, "#define MULT
1 \ t / * max nr garfos de um proc * / \ n ");
2378
fprintf (th, "#endif \ n");
2379
fprintf (th, "#if SYNC == 0 \ n");
2380
fprintf (th, "#define MULT_MAXCS (MULT * MAXCS) \ n");
2381
fprintf (th, "#else \ n");
2382
fprintf (th, "#define MULT_MAXCS (2 * MULT * MAXCS) \ n");
2383

```

```

fprintf (th, "#endif \ n");
2384
2385
fprintf (tc, "#ifdef ALG3 \ n");
2386
fprintf (tc, "unsigned char Csels_c [MAXSTATE] [MULT_MAXCS + 1]; \ n");
2387
fprintf (tc, "unsigned char Csels_r [MAXSTATE] [MULT_MAXCS + 1]; \ n");
2388
fprintf (tc, "unsigned char Csels_p [MAXSTATE] [MULT_MAXCS + 1]; \ n");
2389
fprintf (tc, "char csems [MAXPROC] [TOPQ]; \ n");
2390
fprintf (tc, "csets curtos [MAXPROC] [MAXSTATE]; \ n");
2391
fprintf (tc, "short Nwait = 0, nwait [TOPQ]; \ n \ n");
2392
fprintf (tc, "#endif \ n");
2393
fprintf (th, "#ifdef VERBOSE \ n");
2394
fprintf (th, "char * Move [MAXSTATE]; \ n");
2395
fprintf (th, "#endif \ n");
2396
fprintf (th, "#ifndef ALG3 \ n");
2397
fprintf (th, "#define push_act (p, s, w, h, t)
/* pular */ \ n ");
2398
fprintf (th, "#define unrelease ()
/* pular */ \ n ");
2399
fprintf (th, "#define unpush ()
/* pular */ \ n ");
2.400
fprintf (th, "#define push_commit ()
/* pular */ \ n ");
2401
fprintf (th, "#define un_commit (p)
/* pular */ \ n ");
2402
fprintf (th, "#endif \ n");
2403
}
2404 #endif
2405
2406
2407 vazio
2408 genaddproc ()
2409 {ProcList * p;
2410
int i;
2411
2412
fprintf (tc, "addproc (n");
2413
para (i = 0; i <Npars; i++)
2414
fprintf (tc, ", par% d", i);
2415
2416
n vezes (tc, 0, 1, Addp0);
2417
n vezes (tc, 1, npes, R5);
2418
n vezes (tc, 0, 1, Addp1);
2419
2420
put_pinit (executar-> pc, executar-> n, (Nó *) 0, 0);
2421
para (p = rdy, i = 1; p; p = p-> nxt, i++)
2422
put_pinit (p-> s-> primeiro, p-> n, p-> p, i);
2423
2424
n vezes (tc, i, i + 1, R6);
2425}
2426
2427 vazio
2428 genother (cnt)
2429 {ProcList * p;

```

484
APÊNDICE E
PROJETO E VALIDAÇÃO
2430
int i;
2431
2432
n vezes (tc, 0, 1, Código0);
2433
n vezes (tc, 0, cnt, R0);
2434
n vezes (tc, cnt, cnt + 1, R1);
2435
end_labs (run-> n, 0);
2436
para (p = rdy, i = 1; p; p = p-> nxt, i ++)
2437
end_labs (p-> n, i);
2438
n vezes (tc, 0, cnt, R0a);
2439
n vezes (tc, 0, 1, R0b);
2440
2441 #ifdef GODEF
2442
fprintf (tc, "\ttratable [% d] = _TRA_% d; /* progresso * / \ n", i, i);
2443 #endif
2444 #ifdef PARES
2445
fprintf (tc, "if (tree_before) exit (0); \ n");
2446 #endif
2447
ntimes (th, aceitadores, aceitadores + 1, Código1);
2448
n vezes (th, i + 1, i + 2, R2);
2449
2450
doglobal (INIV);
2451
n vezes (tc, 1, nqs + 1, R3);
2452
n vezes (tc, 0,
1, Código 2);
2453
n vezes (tc, 0,
i, R4);
2454
fprintf (tc, "} \ n \ n");
2455 #ifdef PARES
2456
putpairs ();
2457}
2458
2459 putpairs ()
2460 {ProcList * p; int i;
2461
fprintf (tc, "#ifdef PAIRS \ n");
2462
fprintf (tc, "pares () \ n");
2463
fprintf (tc, "{int i, j; P0 * ptr; \ n");
2464
fprintf (tc, "para (i = 1, j = 0; i <know._nr_pr; i ++) \ n");
2465
fprintf (tc, "{
ptr = (P0 *) pptra (i); \ n ");
2466
fprintf (tc, "#ifdef VERI \ n");
2467
fprintf (tc, "
if (i == 1) continue; \ n ");
2468
fprintf (tc, "
if (i > 2) printf (\\"\\"); \ n ");
2469
fprintf (tc, "#else \ n");
2470
fprintf (tc, "

```

if (i> 1) printf (\ \" ); \ n ");
2471
fprintf (tc, "
else printf (\ "NOVO estado % d: \", nstates); \ n ");
2472
fprintf (tc, "#endif \ n");
2473
fprintf (tc, "
switch (ptr -> _ t) {\n ";
2474
para (p = rdy, i = 1; p; p = p-> nxt, i++)
2475
{
fprintf (tc, "\ t \ t \ tcase% d:", i);
2476
fprintf (tc, "printf (\% % d \ ", ptr -> _ p / * src_ln% d [ptr -> _ p] * /);", i);
2477
fprintf (tc, "j ++; break; \ n");
2478
}
2479
fprintf (tc, ")
} \ n ";
2480
fprintf (tc, "if (j) printf (\\" \\ n \\ "); \ n");
2481
fprintf (tc, "} \ n");
2482
fprintf (tc, "#endif \ n \ n");
2483 #endif

```

Página 496

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

485
2484}
2485
2486 estruturas estáticas {
2487
char * s, * t; int n, m;
2488} ln [] = {
2489
"fim",
"stopstate",
3,
0,
2490
"progresso",
"progstate",
8,
0,
2491
"aceitar",
"accpstate",
6,
1,
2492
0,
0,
0,
0,
0,
2493};
2494
2495 vazio
2496 end_labs (s, i)
2497
Símbolo * s;
2498 {
2499
extern Label * labtab;
2500
Rótulo * l;
2501
int j;
2502 #ifdef GODEF
2503
fprintf (tc, "\ ttratable [% d] = _TRA_% d; /*% s * / \ n", i, i, s-> nome);
2504 #endif
2505
para (l = labtab; l; l = l-> nxt)

```

```

2506
para (j = 0; ln [j] .n; j++)
2507
if (strncmp (l-> s-> nome, ln [j] .s, ln [j] .n) == 0
2508
&& strcmp (l-> s-> contexto-> nome, s-> nome) == 0)
2509
{
fprintf (tc, "\t% s [% d] [% d] = 1; \n",
2510
ln [j] .t, i, l-> e-> seqno);
2511
aceitadores += ln [j] .m;
2512
}
2513}
2514
2515 vazio
2516 n vezes (fd, n, m, c)
2517
ARQUIVO * fd;
2518
char * c [];
2519 {
2520
int i, j;
2521
para (j = 0; c [j]; j++)
2522
para (i = n; i <m; i++)
2523
{
fprintf (fd, c [j], i, i, i, i, i);
2524
fprintf (fd, "\n");
2525
}
2526}
2527
2528 vazio
2529 dolocal (dowhat, p, s)
2530
char * s;
2531 {
2532
int i, j;
2533
Símbolo * sp;
2534
char buf [64];
2535
2536
para (j = 0; j <5; j++)
2537
para (i = 0; i <= Nhash; i++)

```

```

486
APÊNDICE E
PROJETO E VALIDAÇÃO
2538
para (sp = symtab [i]; sp; sp = sp-> próximo)
2539
if (sp-> contexto && sp-> type == Tipos [j]
2540
&& strcmp (s, sp-> contexto-> nome) == 0)
2541
{
sprintf (buf, "((P% d *) pptr (h)) ->", p);
2542
do_var (dowhat, buf, sp);
2543
}
2544}
2545
2546 vazio
2547 doglobal (dowhat)
2548 {Símbolo * sp;
2549
int i, j;

```

```

2550
2551
para (j = 0; j <5; j++)
2552
para (i = 0; i <= Nhash; i++)
2553
para (sp = symtab [i]; sp; sp = sp-> próximo)
2554
if (! sp-> contexto && sp-> type == Tipos [j])
2555
do_var (dowhat, "agora.", sp);
2556}
2557
2558 vazio
2559 do_var (dowhat, s, sp)
2560
char * s;
2561
Símbolo * sp;
2562 {
2563
int i;
2564
2565
switch (dowhat) {
2566
case PUTV:
2567
typ2c (sp);
2568
quebrar;
2569
case INIV:
2570
if (! sp-> ini)
2571
quebrar;
2572
if (sp-> nel == 1)
2573
{
fprintf (tc, "\t \t% s% s =", s, sp-> nome);
2574
do_init (sp);
2575
} outro
2576
para (i = 0; i <sp-> nel; i++)
2577
{
fprintf (tc, "\t \t% s% s [% d] =", s, sp-> nome, i);
2578
do_init (sp);
2579
}
2580
quebrar;
2581
}
2582}
2583
2584 vazio
2585 do_init (sp)
2586
Símbolo * sp;
2587 {
2588
int i;
2589
2590
if (sp-> type == CHAN && ((i = qmake (sp))> 0))
2591
fprintf (tc, "adicionar fila (% d); \n", i);

```

```

2593
fprintf (tc, "% d; \ n", eval (sp-> ini));
2594}
2595
2596 blog (n)
/* para log2 pequeno sem problemas de arredondamento */
2597 {int m = 1, r = 2;
2598 enquanto (r <n) {m ++; r * = 2; }
2599 retornar 1 + m;
2600}
2601
2602 vazio
2603 put_ptype (s, p, i, m0, m1)
2604
char * s;
2605
Nó * p;
2606 {
2607
Nó * fp, * fpt;
2608
int j;
2609
fprintf (tc, "\% s \ ", \ n", s);
2610
fprintf (th, "typedef struct P% d {/* % s */ \ n", i, s);
2611
fprintf (th, "não assinado _t:% d; /* proctype */ \ n", blog (m1));
2612
fprintf (th, "unsigned _p:% d; /* estado
* / \ n ", blog (m0));
2613
dolocal (PUTV, i, s);
/* inclui pars */
2614
fprintf (th, "} P% d; \ n", i);
2615
2616
para (fp = p, j = 0; fp; fp = fp-> rgt)
2617
para (fpt = fp-> lft; fpt; fpt = fpt-> rgt)
2618
j++;
/* contagem # de parâmetros */
2619
Npars = max (Npars, j);
2620}
2621
2622 vazio
2623 put_pinit (e, s, p, i)
2624
Elemento * e;
2625
Símbolo * s;
2626
Nó * p;
2627 {
2628
Nó * fp, * fpt;
2629
int ini, j;
2630
2631
ini = huntele (e, e-> status) -> seqno;
2632
fprintf (th, "#define start% d
% d \ n ", i, ini);
2633
2634
fprintf (tc, "\ tcase% d: /* % s */ \ n", i, s-> nome);
2635
fprintf (tc, "\ t \ t ((P% d *) pptr (h)) -> _ t =% d; \ n", i, i);
2636
fprintf (tc, "\ t \ t ((P% d *) pptr (h)) -> _ p =% d;", i, ini);
2637
fprintf (tc, "alcançado% d [% d] = 1; \ n", i, ini);
2638
dolocal (INIV, i, s-> nome);
2639
para (fp = p, j = 0; fp; fp = fp-> rgt)

```

```

2640
para (fpt = fp-> lft; fpt; fpt = fpt-> rgt, j++)
2641
{
if (fpt-> nsym-> nel! = 1)
2642
fatal ("array na lista de parâmetros,% s", fpt-> nsym-> nome);
2643
fprintf (tc, "\t \t ((P% d *) pptr (h)) ->% s = par% d; \n",
2644
i, fpt-> nsym-> nome, j);
2645
}

```

Página 499

```

488
APÊNDICE E
PROJETO E VALIDAÇÃO
2646
fprintf (tc, "\t break; \n");
2647}
2648
2649 huntstart (f)
2650
Elemento * f;
2651 {
2652
Elemento * e = f;
2653
2654
if (e-> n)
2655
{
if (e-> n-> ntyp == '.' && e-> nxt)
2656
e = e-> nxt;
2657
else if (e-> n-> ntyp == ATOMIC)
2658
e-> n-> seql-> this-> last-> nxt = e-> nxt;
2659
}
2660
return e-> seqno;
2661}
2662
Elemento 2663 *
2664 huntele (f, o)
2665
Elemento * f;
2666 {
2667
Elemento * g, * e = f;
2668
int cnt; /* uma precaução contra loops */
2669
para (cnt = 0; cnt < 10 && e-> n; cnt++)
2670
{
switch (e-> n-> ntyp) {
2671
case GOTO:
2672
g = get_lab (e-> n-> nsym);
2673
quebrar;
2674
case '.':
2675
case BREAK:
2676
if (! e-> nxt)
2677
return e;
2678
g = e-> nxt;
2679
quebrar;
2680
case ATOMIC:

```

```

2681
e-> n-> seq1-> this-> last-> nxt = e-> nxt;
2682
padrão:
/* Cair em */
2683
return e;
2684
}
2685
if ((o & ATOM) && !(g-> status & ATOM))
2686
return e;
2687
e = g;
2688
}
2689
return e;
2690}
2691
2692 vazio
2693 typ2c (sp)
2694
Símbolo * sp;
2695 {
2696
switch (sp-> tipo) {
2697
case BIT:
2698
if (sp-> nel == 1)
2699
{
fprintf (th, "\ tunsigned% s: 1", sp-> nome);

```

Página 500

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

489
2700
quebrar;
2701
} /* outra falha */
2702
case CHAN:
/* bom para até 255 canais */
2703
caso BYTE:
2704
fprintf (th, "\ tuchar% s", sp-> nome);
2705
quebrar;
2706
case SHORT:
2707
fprintf (th, "\ tshort% s", sp-> nome);
2708
quebrar;
2709
case INT:
2710
fprintf (th, "\ tint% s", sp-> nome);
2711
quebrar;
2712
case PREDEF:
2713
Retorna;
2714
padrão:
2715
fatal ("variável% s não declarada", sp-> nome);
2716
}
2717
if (sp-> nel != 1)
2718
fprintf (th, "[% d]", sp-> nel);
2719

```

```

fprintf (th, "; \ n");
2720}
2721
2722 vazio
2723 casos (fd, p, n, m, c)
2724
ARQUIVO * fd;
2725
char * c [];
2726 {
2727
int i, j;
2728
para (j = 0; c [j]; j++)
2729
para (i = n; i <m; i++)
2730
{
fprintf (fd, c [j], i, p, i);
2731
fprintf (fd, "\ n");
2732
}
2733}
2734
2735 vazio
2736 genaddqueue ()
2737 {char * buf0;
2738
int j;
2739
Fila * q;
2740
2741
buf0 = (char *) emalloc (32);
2742
n vezes (tc, 0, 1, Addq0);
2743
para (q = qtab; q; q = q-> nxt)
2744
{
n vezes (tc, q-> qid, q-> qid + 1, R8);
2745
n vezes (th, q-> qid, q-> qid + 1, R9);
2746
para (j = 0; j <q-> nflds; j++)
2747
{
switch (q-> fld_width [j]) {
2748
case BIT:
2749
if (q-> nflds! = 1)
2750
{
fprintf (th, "\ t \ tunsigned");
2751
fprintf (th, "fld% d: 1; \ n", j);
2752
quebrar;
2753
} /* mais falha: dá uma estrutura menor */

```

490
APÊNDICE E
PROJETO E VALIDAÇÃO
2754
case CHAN:
2755
caso BYTE:
2756
fprintf (th, "\ t \ tuchar fld% d; \ n", j);
2757
quebrar;
2758
case SHORT:
2759
fprintf (th, "\ t \ tshort fld% d; \ n", j);
2760

```

quebrar;
2761
case INT:
2762
fprintf (th, "\t \ tint fld% d; \ n", j);
2763
quebrar;
2764
padrão:
2765
fatal ("especificações de canal incorretas", "");
2766
}
2767
}
2768
fprintf (th, "} conteúdo [% d]; \ n", max (1, q-> nslots));
2769
fprintf (th, "} Q% d; \ n", q-> qid);
2770
}
2771
n vezes (th, 0, 1, R10);
2772
n vezes (tc, 0, 1, Addq1);
2773
2774
fprintf (tc, "qsend (em");
2775
para (j = 0; j <Mpars; j++)
2776
fprintf (tc, ", fld% d", j);
2777
fprintf (tc, ") \ n");
2778
n vezes (tc, 0, 1, Addq11);
2779
2780
para (q = qtab; q; q = q-> nxt)
2781
{
sprintf (buf0, "((Q% d *) z) ->", q-> qid);
2782
fprintf (tc, "\tcase% d: j == sQlen; \ n", q-> qid, buf0);
2783
fprintf (tc, "\t \ t% sQlen = j + 1; \ n", buf0);
2784
if (q-> nslots == 0)
/* redefinir ponto de handshake */
2785
fprintf (tc, "\t \ t (trpt + 2) -> o_m = 0; \ n");
2786
sprintf (buf0, "((Q% d *) z) -> conteúdo [j] .fld", q-> qid);
2787
para (j = 0; j <q-> nflds; j++)
2788
fprintf (tc, "\t \ t% s% d = fld% d; \ n", buf0, j, j);
2789
fprintf (tc, "\t \ tbreak; \ n");
2790
}
2791
n vezes (tc, 0, 1, Addq2);
2792
2793
para (q = qtab; q; q = q-> nxt)
2794
fprintf (tc, "\tcase% d: return% d; \ n", q-> qid, (! q-> nslots));
2795
2796
n vezes (tc, 0, 1, Addq3);
2797
2798
para (q = qtab; q; q = q-> nxt)
2799
fprintf (tc, "\tcase% d: return (q_sz (from) ==% d); \ n",
2800
q-> qid, max (1, q-> nslots));
2801
2802
n vezes (tc, 0, 1, Addq4);
2803

```

```

para (q = qtab; q; q = q-> nxt)
2804
{
sprintf (buf0, "((Q% d *) z) ->", q-> qid);
2805
fprintf (tc, "caso% d:", q-> qid);
2806
if (q-> nflds == 1)
2807
{
fprintf (tc, "\ tif (fld == 0) r =% s", buf0);

```

Página 502

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

491
2808
fprintf (tc, "conteúdo [slot] .fld0; \ n");
2809
} outro
2810
{
fprintf (tc, "\ tswitch (fld) {\n");
2811
ncasos (tc, q-> qid, 0, q-> nflds, R12);
2812
fprintf (tc, "\ t \ t} \ n");
2813
}
2814
fprintf (tc, "\ t \ tif (concluído) \ n");
2815
fprintf (tc, "\ t \ t {
j =% sLen; \ n ", buf0);
2816
fprintf (tc, "\ t \ t
% sLen = --j; \ n ", buf0);
2817
fprintf (tc, "\ t \ t
para (k = 0; k <j; k ++) \ n ", q-> qid);
2818
fprintf (tc, "\ t \ t
{\n ");
2819
2820
sprintf (buf0, "\ t \ t \ t ((Q% d *) z) -> conteúdo", q-> qid);
2821
para (j = 0; j <q-> nflds; j++)
2822
{
fprintf (tc, "\ t% s [k] .fld% d = \ n", buf0, j);
2823
fprintf (tc, "\ t \ t% s [k + 1] .fld% d; \ n", buf0, j);
2824
}
2825
fprintf (tc, "\ t \ t
} \ n ");
2826
para (j = 0; j <q-> nflds; j++)
2827
fprintf (tc, "% s [j] .fld% d = 0; \ n", buf0, j);
2828
fprintf (tc, "\ t \ t \ tif (fld + 1! =% d) \ n \ t \ t \ t", q-> nflds);
2829
fprintf (tc, "\ turror (\\" pars faltando no recebimento \ "); \ n");
2830
/* mensagens recebidas incompletamente não podem ser recuperadas */
2831
fprintf (tc, "\ t \ t} \ n");
2832
fprintf (tc, "\ t \ tbreak; \ n");
2833
}
2834
n vezes (tc, 0, 1, Addq5);
2835}
2836
2837 / ***** spin: pangen2.c ***** /
2838

```

```

2839 #include <stdio.h>
2840 #include "spin.h"
2841 #include "y.tab.h"
2842 #include "pangen2.h"
2843
2844 extern ProcList
* rdy;
2845 extern RunList
*corre;
2846 simbolo externo
* Fname;
2847 caracteres externos
* Claimproc;
2848 extern int lineno;
2849 extern int Mpars;
2850 extern int m_loss;
2851 int
Globalname;
2852
2853 #ifdef GODEF
2854 vazio push_cs ();
2855 vazio push_loss ();
2856 vazio putbase ();
2857 void putindex ();
2858 void coll_global ();
2859 vazio col_base ();
2860 vazio coll_cs ();
2861 vazio coll_idx ();

```

```

492
APÉNDICE E
PROJETO E VALIDAÇÃO
2862 int aMarked = 0, Marked = 0, Countm = 0, Maxcs = 0;
2863 #endif
2864
2865 ARQUIVO
* tc, * th, * tt, * tm, * tb, * tf;
2866 int
uniq = 1;
2867 int
nocast = 0;
/* para desligar as conversões em lvalues */
2868 int
conciso = 0;
/* impressão concisa de nomes de variáveis */
2869 int
nps = 0;
/* número de processos */
2870 int
mst = 0;
/* número máximo de estado / processo */
2871 int
reivindicação nr = -1; /* processo de reivindicação, se houver */
2872 int
Pid;
/* proc processado atualmente */
2873 int
EVAL_runs = 0; /* usado em verificações de imparcialidade */
2874
2875 #ifdef VARSTACK
2876 int
Cksum;
/* depuração apenas */
2877 #endif
2878
2879 fproc (s)
2880 char * s;
2881 {
2882
ProcList * p;
2883
int i;
2884
2885 if (strcmp ("_ init", s) == 0)
2886
return 0;
2887

```

```

para (p = rdy, i = 1; p; p = p-> nxt, i++)
2888
if (strcmp (p-> n-> nome, s) == 0)
2889
return i;
2890
fatal ("proctype% s não encontrado", s);
2891}
2892
2893 vazio
2894 gensrc ()
2895 {ProcList * p;
2896
int i;
2897
2898
if (! (tc = fopen ("pan.c", "w"))
/* rotinas principais */
2899
|| ! (th = fopen ("pan.h", "w"))
/* arquivo de cabeçalho */
2900
|| ! (tt = fopen ("pan.t", "w"))
/* matriz de transição */
2901
|| ! (tm = fopen ("pan.m", "w"))
/* move para frente */
2902
|| ! (tf = fopen ("pan.f", "w"))
/* verificações de imparcialidade */
2903
|| ! (tb = fopen ("pan.b", "w")))
/* retrocede */
2904
{
printf ("spin: não é possível criar pan. [chtmb] \ n");
2905
saída (1);
2906
}
2907
fprintf (th, "/ ***% s *** / \ n", Fname-> nome);
2908
fprintf (th, "#define uchar
char não assinado \ n ");
2909
if (Claimproc)
2910
{
reivindicação nr = fproc (reivindicaçãoproc);
2911
fprintf (th, "#define VERI
% d \ n ", Claimnr);
2912
fprintf (th, "#define linha de reivindicação");
2913
fprintf (th, "src_ln% d [((P0 *) pptr (1)) -> _ p] \ n",
2914
Claimnr);
2915
}

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

493
2916
fprintf (th, "#define M_LOSS
% d \ n ", m_loss);
2917
fprintf (th, "#define endclaim endstate% d \ n", Claimnr);
2918
n vezes (tc, 0, 1, Preâmbulo);
2919
2920
fprintf (tc, "#ifndef NOBOUNDCHECK \ n");
2921
fprintf (tc, "#define o índice (x, y)
Boundcheck (x, y, II, tt, t) \ n ");
2922

```

```

fprintf (tc, "#else \ n");
2923
fprintf (tc, "#define o índice (x, y)
x \ n ");
2924
fprintf (tc, "#endif \ n");
2925
2926
mst = (executar)? executar-> maxseq: 0;
2927
para (p = rdy, i = 1; p; p = p-> nxt, i++)
2928
mst = max (p-> s-> último-> seqno, mst);
2929
nps = i + 1;
/* adicionar verificador de progresso */
2930
2931
fprintf (tt, "setable () \ n {\ \ tTrans * T, * settr (); \ n \ n");
2932
fprintf (tt, "#ifdef VERBOSE \ n");
2933
fprintf (tt, "\ tMoves [0] = \" jogada errada \ "; \ n");
2934
fprintf (tt, "#endif \ n");
2935
fprintf (tt, "\ ttrans = (Trans ***)");
2936
fprintf (tt, "emalloc (% d * sizeof (Trans **)); \ n", nps);
2937
2938
fprintf (tm, "switch (t-> forw) {\ \ n");
2939
fprintf (tm, "padrão: Uerror (\\" movimento incorreto para frente \ "); \ n");
2940
2941
fprintf (tb, "switch (t-> back) {\ \ n");
2942
fprintf (tb, "default: Uerror (\\" bad return move \ "); \ n");
2943
fprintf (tb, "case 0: goto R999; /* nada para desfazer */ \ n");
2944
2945
fprintf (tf, "switch (t-> forw) {\ \ n");
2946
fprintf (tf, "padrão: continuar; \ n");
2947
2948
if (! run) fatal ("nenhum processo executável", (char *) 0);
2949
2950
putproc (executar-> n, executar-> pc, 0, executar-> maxseq);
2951
para (p = rdy, i = 1; p; p = p-> nxt, i++)
2952
putproc (p-> n, p-> s-> primeiro, i, p-> s-> último-> seqno);
2953
putprogress (i, 2);
2954 #ifdef GODEF
2955
fprintf (th, "#define _TRA_ % d
% d
/* progresso */ \ n ", i, uniq);
2956
fprintf (th, "#define _TRA_ % d
% d
/* fim */ \ n ", i + 1, uniq + 2);
2957 #endif
2958
n vezes (tt, 0, 1, cauda);
2959
genheader ();
2960
genaddproc ();
2961
genother (i);
2962
genaddqueue ();
2963
genunio ();
2964
2965

```

```
putsyms (tc, th);
2966}
2967
2968 vazio
2969 putproc (n, e, i, j)
```

```
494
APÊNDICE E
PROJETO E VALIDAÇÃO
2970
Símbolo * n;
2971
Elemento * e;
2972 {
2973
Pid = i;
2974 #ifdef GODEF
2975
fprintf (th, "#define _TRA_% d
% d
/ *% s * / \ n ", i, uniq, n-> nome);
2976 #endif
2977
fprintf (th, "\ nestados curtos% d =% d; \ t / *% s * / \ n", i, j + 1, n-> nome);
2978
fprintf (tm, "\ n
/ * PROC% s * / \ n ", n-> nome);
2979
fprintf (tb, "\ n
/ * PROC% s * / \ n ", n-> nome);
2980
fprintf (tt, "\ n / * proctype% d:% s * / \ n", i, n-> nome);
2981
fprintf (tt, "\ n trans [% d] = (Trans **)", i);
2982
fprintf (tt, "emalloc (% d * sizeof (Trans *));
\ n \ n", j + 1);
2983
putseq (e, 0);
2984
dumpsrc (j, i);
2985}
2986
2987 vazio
2988 putprogress (i, j) / * detector de loop *
2989 {
2990
fprintf (th, "\ nshort nestados% d =% d; \ t / * _progress * / \ n", i, j + 1);
2991
2992
fprintf (tt, "\ n / * proctype% d: _progress * / \ n", i);
2993
fprintf (tt, "\ n trans [% d] = (Trans **)", i);
2994
fprintf (tt, "emalloc (% d * sizeof (Trans *));
\ n \ n", j + 1);
2995
fprintf (tt, "trans [% d] [1]
= settr (1,2,% d,% d, \ "- \", 0,0); \ n ",
2996
i, uniq, uniq);
2997
fprintf (tt, "trans [% d] [2]
= settr (1,0,% d,% d, \ "- \", 0,0); \ n ",
2998
i, uniq + 1, uniq + 1);
2999
fprintf (tt, "} \ n");
3000
3001
fprintf (tm, "\ n
/ * _progress * / \ n ");
3002
fprintf (tm, "case% d:
/ * progresso * / \ n ", uniq);
3003
fprintf (tm, "
IfNotBlocked \ n ");
3004
fprintf (tm, "
agora._p_t = 13; / * 13 para ajudar o hasher * / \ n ");
```

```

3005
fprintf (tm, "
m = 3; vá para P999; \ n ");
3006
fprintf (tm, "case% d: \ n", uniq + 1);
3007
fprintf (tm, "
continue; \ n ");
3008
fprintf (tm, "} \ n \ n");
3009
3010
fprintf (tb, "\ n
/* _progress * / \ n ");
3011
fprintf (tb, "case% d:
/* progresso * / \ n ", uniq);
3012
fprintf (tb, "
agora._p_t = 0; \ n ");
3013
fprintf (tb, "
goto R999; \ n ");
3014
fprintf (tb, "case% d: \ n", uniq + 1);
3015
fprintf (tb, "
goto R999; \ n ");
3016
fprintf (tb, "} \ n \ n");
3017
3018
fprintf (tf, "} \ n");
3019}
3020
3021 vazio
3022 putseq (f, nível)
3023
Elemento * f;

```

Página 506

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

495
3024 {
3025
Elemento * e;
3026
3027
para (e = f; e; e = e-> nxt)
3028
putseq_el (e, nível + 1);
3029}
3030
3031 vazio
3032 putseq_lst (s, nível)
3033
Seqüência * s;
3034 {
3035
Elemento * g;
3036
3037
para (g = s-> primeiro; g = g-> nxt)
3038
{
if (! g)
3039
{
fprintf (stderr, "não pode acontecer seq_lst \ n");
3040
saída (1);
3041
}
3042
putseq_el (g, nível + 1);
3043
if (g == s-> last)
3044
quebrar;

```

```

3045
}
3046}
3047
3048 vazio
3049 putseq_el (e, nível)
3050
Elemento * e;
3051 {
3052
SeqList * h;
3053
3054 int n, a, bu;
3055
3056 if (e-> status & DONE)
3057 Retorna;
3058 e-> status |= FEITO;
3059 if (e-> n-> nval)
3060 putrc (e-> n-> nval, e-> seqno);
3061 if (e-> sub)
3062 {
int oMarked, oaMarked; atom_stack * oCS, * save_ast ();
3063 fprintf (tt, "\tT = trans [%d] [%d] =", Pid, e-> seqno);
3064 fprintf (tt, "setattr (%d, 0,0,0, \" ", e-> status);
3065 comentário (tt, e-> n, e-> seqno);
3066 fprintf (tt, "\", 0,%d); \t / * linha% d (%d,%d) * / \n ",
3067 e-> n-> ntyp,
3068 e-> n-> nval, Marcado, nível);
3069 para (h = e-> sub; h; h = h-> nxt)
3070 {
putskip (h-> this-> primeiro-> seqno);
3071 a = huntstart (h-> this-> primeiro);
3072 fprintf (tt, "\tT = T-> nxt \t =");
3073 fprintf (tt, "setattr (%d,%d, 0,0, \" ",
3074 e-> status, a, e-> n-> ntyp);
3075 comentário (tt, e-> n, e-> seqno);
3076 fprintf (tt, "\",%d,%d); \t / * linha% d (%d,%d) * / \n ",
3077 1-Marcado, e-> n-> ntyp,

```

496
APÉNDICE E
PROJETO E VALIDAÇÃO
3078 e-> n-> nval, Marcado, nível);
3079 }
3080 #if 0
3081 oMarked = Marcado;
3082 oCS = save_ast ();
3083 #endif
3084 oaMarked = aMarked;
3085

```

para (h = e-> sub; h; h = h-> nxt)
3086
{
3087
aMarked = oaMarked;
3088 #if 1
3089
if (aMarked)
3090
{
clear_ast ();
3091
coll_global (h-> this, 0);
3092
Marcado = has_ast ();
3093
}
3094 #else
3095
Marcado = oMarcado;
3096
restor_ast (oCS);
3097 #endif
3098
putseq_lst (h-> this, level);
3099
}
3100
} outro
3101
{
if (e-> n && e-> n-> ntyp == ATOMIC)
3102
{
patch_atomic (e-> n-> seql-> this);
3103
putskip (e-> n-> seql-> this-> primeiro-> seqno);
3104
a = huntstart (e-> n-> seql-> this-> primeiro);
3105
fprintf (tt, "\tT = trans [% d] [% d] =", Pid, e-> seqno);
3106
fprintf (tt, "setattr (% d, 0,0,0, \" ", ATOM, e-> n-> ntyp);
3107
comentário (tt, e-> n, e-> seqno);
3108
fprintf (tt, "\", 0,% d); \t / * linha% d * / \n ",
3109
e-> n-> ntyp, e-> n-> nval);
3110
fprintf (tt, "\t
T-> nxt \t = ");
3111
3112
fprintf (tt, "setattr (% d,% d, 0,0, \" ", ATOM, a, e-> n-> ntyp);
3113
comentário (tt, e-> n, e-> seqno);
3114
fprintf (tt, "\", 0,% d); \t / * linha% d (% d,% d) * / \n ",
3115
e-> n-> ntyp, e-> n-> nval, Marcado, nível);
3116
e-> n-> seql-> this-> last-> nxt = e-> nxt;
3117 #ifdef GODEF
3118
/*
3119
* se as declarações em uma sequência atômica
3120
* tocar objetos globais, o (s) guarda (s) deve (m)
3121
* ser rotulado com todos os conjuntos de conflito tocados
3122
*/
3123
if (has_ast ())
3124
{
fprintf (stderr, "erro interno: pilha ast \n");
3125
pop_ast (stderr, 0);
3126

```

```

saída (1);
3127
}
3128
coll_global (e-> n-> seql-> this, 0);
3129
Marcado = has_ast ();
3130
aMarcado = 1;
3131 #endif

```

Página 508

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

497
3132
putseq_lst (e-> n-> seql-> este, nível);
3133 #ifdef GODEF
3134
Marcado = 0; aMarcado = 0;
3135 #endif
3136
Retorna;
3137
}
3138
if (e-> n-> ntyp == GOTO)
3139
a = huntele (get_lab (e-> n-> nsym),
3140
e-> status) -> seqno;
3141
else if (e-> nxt)
3142
a = huntele (e-> nxt, e-> status) -> seqno;
3143
outro
3144
a = 0;
3145
fprintf (tt, "\tT = trans [% d] [% d] \t =",
3146
Pid, e-> seqno);
3147
3148
putfair (tf, e-> n, e-> seqno, uniq);
3149
3150
fprintf (tm, "\tcase% d: /* STATE", uniq++);
3151
fprintf (tm, "% d", e-> seqno);
3152
comentário (tm, e-> n, e-> seqno);
3153
fprintf (tm, ", linha% d (marcado como% d, nível% d, status% d /% d) * / \n \t \t",
3154
e-> n-> nval, Marcado, nível, e-> status, e-> status e ATOM);
3155
if (e-> n && e-> n-> ntyp! = 'r' && Pid! = Claimnr)
3156
fprintf (tm, "IfNotBlocked \n \t \t");
3157
putstmtnt (tm, e-> n, e-> seqno);
3158 #ifdef VARSTACK
3159 /*
3160 * aviso: checklast () em dfollow.c
3161 * é uma otimização não confiável
3162 *
Cksum = rand ();
3163 *
if (Pid! = Claimnr)
3164 *
bu = checklast (tm, e-> n, e-> nxt, e-> seqno, 1);
3165 *
outro
3166 */
3167 #endif
3168
bu = 0;
3169

```

```

n = getweight (e-> n);
3170
fprintf (tm, "; \ n \ t \ tm =% d; \ n \ t \ t", n);
3171 #ifdef GODEF
3172
Countm = 0;
3173
if (any_cs (e-> n)) push_cs (tm, e-> n, 0);
3174
Maxcs = max (Countm, Maxcs);
3175 #endif
3176
if (bu == 0) bu = 2;
3177
fprintf (tm, "goto P999; \ n", n);
3178
if (bu || any_undo (e-> n))
3179 {
3180
fprintf (tb, "\ tcase% d:", uniq-1);
3181
fprintf (tb, "/ * ESTADO");
3182
fprintf (tb, "% d,", e-> seqno);
3183
comentário (tb, e-> n, e-> seqno);
3184
fprintf (tb, ", linha% d (marcado como% d, nível% d, status% d /% d) * / \ n \ t \ t",
3185
e-> n-> nval, Marcado, nível, e-> status, e-> status e ATOM);

```

Página 509

498
APÊNDICE E
PROJETO E VALIDAÇÃO
3186 #ifdef VARSTACK
3187
if (bu == 1)
3188
checklast (tb, e-> n, e-> nxt, e-> seqno, 0);
3189 #endif
3190
if (any_undo (e-> n))
3191
undostmnt (e-> n, e-> seqno);
3192
fprintf (tb, "; \ n \ t \ t");
3193 #ifdef GODEF
3194
Countm = 0;
3195
/* Combine os conjuntos de conflito no retrocesso */
3196
if (marcado)
/* guarda de uma sequência atômica -with globals- */
3197
pop_ast (tb, 1);
3198
outro
3199
{
if ((e-> status & ATOM) == 0 && (e-> status & L_ATOM) == 0)
3200
{
if (any_cs (e-> n))
/* globais tocaram */
3201
push_cs (tb, e-> n, 1);
3202
outro
/* somente locais */
3203
{
Countm++;
3204
fprintf (tb, "push_act (II, R_LOCK, BLOCK,");
3205
fprintf (tb, "t-> forw, MAXCONFL); \ n \ t \ t");
3206

```

}
3207
} else if (aMarked)
/* guarda de at.seq local. */
3208
{
Countm++;
3209
fprintf (tb, "push_act (II, R_LOCK, BLOCK,");
3210
fprintf (tb, "t-> forw, MAXCONFL); \ n \ t \ t");
3211
}
3212
}
3213
Maxcs = max (Countm, Maxcs);
3214 #endif
3215
fprintf (tb, "goto R999; \ n");
3216
fprintf (tt, "setattr (% d,% d,% d,% d, \" ",
3217
e-> status, a, uniq-1, uniq-1, e-> n-> ntyp);
3218
} outro
3219
fprintf (tt, "setattr (% d,% d,% d, 0, \" ",
3220
e-> status, a, uniq-1, e-> n-> ntyp);
3221
comentário (tt, e-> n, e-> seqno);
3222
if (marcado)
/* globais são tocados posteriormente em um s atômico. */
3223
Globalname = 1;
3224
fprintf (tt, "\",% d,% d); \ t / * linha% d (% d,% d) * / \ n ",
3225
1-Globalname, e-> n-> ntyp, e-> n-> nval, Marcado, nível);
3226
Marcado = 0; aMarcado = 0;
3227
}
3228}
3229
3230 vazio
3231 patch_atomic (s)
3232
Sequência * s;
3233 /* catch goto's that break the chain */
3234
Elemento * f, * g;
3235
SeqList * h;
3236
para (f = s-> primeiro; f = f-> nxt)
3237
{
if (f-> n && f-> n-> ntyp == GOTO)
3238
{
g = get_lab (f-> n-> nsym);
3239
if ((f-> status & ATOM)

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN
499
3240
&&! (g-> status & ATOM))
3241
{
f-> status & = ~ATOM;
3242
f-> status |= L_ATOM;
3243
}

```

3244
} outro
3245
para (h = f-> sub; h; h = h-> nxt)
3246
patch_atomic (h-> this);
3247
if (f == s-> last)
3248
quebrar;
3249
}
3250}
3251
3252 #ifdef GODEF
3253
3254 int Mustwrite = 0;
3255
3256 any_cs (agora)
3257
Nó * agora;
3258 {
3259
Nó * v;
3260
3261
if (! now) {return; }
3262
switch (agora-> ntyp) {
3263
3264
case CONST: case 'q': case '.':
3265
case BREAK: case GOTO: case '@':
3266
case ATOMIC: case IF: case DO:
3267
return 0;
3268
3269
case 'p':
/* XXXXX probe rem ref of locals - handle _p */
3270
return 0;
3271
3272
case 'c': case '!': case LEN:
3273
case UMIN: case ASSERT:
3274
caso '^':
3275
return any_cs (now-> lft);
3276
3277
case '/': case '*': case '-': case '+':
3278
case '%': case '<': case '>': case '&':
3279
case '|': case LE: case GE: case NE:
3280
case EQ: case OR: case AND: case LSHIFT: case RSHIFT: case ASGN:
3281
return any_cs (now-> lft) | any_cs (now-> rgt);
3282
3283
case RUN:
3284
case PRINT:
para (v = agora-> lft; v; v = v-> rgt)
3285
if (any_cs (v-> lft))
3286
return 1;
3287
return 0;
3288
3289
case TIMEOUT:
3290
case 'r':
3291
case 's':

```

```
3291
case 'R':
return 1;
3292
3293
caso NAME:
if (! now-> nsym-> contexto
500
APÊNDICE E
PROJETO E VALIDAÇÃO
3294
|| agora-> nsym-> type == CHAN)
/* global ou chan */
3295
return 1;
3296
if (now-> nsym-> nel! = 1)
3297
return any_cs (now-> lft);
3298
return 0;
3299
}
3300
fprintf (stderr, "não pode acontecer% d \ n", agora-> ntyp);
3301
return 0;
3302}
3303
3304 vazio
3305 coll_cs (agora)
3306
Nó * agora;
3307 {
3308
Nó * v;
3309
3310
if (! now) {return; }
3311
switch (agora-> ntyp) {
3312
case 'c': case '!':
3313
case UMIN: case ASSERT:
3314
caso '^':
3315
coll_cs (agora-> lft);
3315
quebrar;
3316
3317
case '/': case '*': case '-': case '+':
3318
case '%': case '<': case '>': case '&':
3319
case '|': case LE: case GE: case NE:
3320
case EQ: case OR: case AND: case LSHIFT: case RSHIFT:
3321
coll_cs (agora-> lft);
3322
coll_cs (agora-> rgt);
3323
quebrar;
3324
3325
case PRINT:
3326
case RUN:
para (v = agora-> lft; v; v = v-> rgt)
3327
coll_cs (v-> lft);
3328
quebrar;
3329
3330
caso ASGN:
```

```

coll_base ("W_LOCK", direto, agora-> ntyp, agora-> lft);
3331
coll_idx (agora-> lft);
3332
coll_cs (agora-> rgt);
3333
quebrar;
3334
3335
case 'r':
coll_base ("R_LOCK", direto, agora-> ntyp, agora-> lft);
3336
coll_base ("Rcv_LOCK", indireto, agora-> ntyp, agora-> lft);
3337
coll_idx (agora-> lft);
3338
3339
Mustwrite = 1;
3340
para (v = agora-> rgt; v; v = v-> rgt)
3341
coll_cs (v-> lft);
3342
Mustwrite = 0;
3343
quebrar;
3344
3345
case 's':
coll_base ("R_LOCK", direto, agora-> ntyp, agora-> lft);
3346
3347
coll_base ("Snd_LOCK", Indireto, agora-> ntyp, agora-> lft);

```

Página 512

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

501
3348
coll_idx (agora-> lft);
3349
3350
para (v = agora-> rgt; v; v = v-> rgt)
3351
coll_cs (v-> lft);
3352
quebrar;
3353
3354
case 'R':
coll_base ("R_LOCK", direto, agora-> ntyp, agora-> lft);
3355
3356
coll_base ("R_LOCK", indireto, agora-> ntyp, agora-> lft);
3357
coll_idx (agora-> lft);
3358
3359
para (v = agora-> rgt; v; v = v-> rgt)
3360
coll_cs (v-> lft);
3361
quebrar;
3362
case TIMEOUT: coll_base ("R_LOCK", direto, agora-> ntyp, 0);
3363
quebrar;
3364
case LEN:
coll_base ("R_LOCK", direto, agora-> ntyp, agora-> lft);
3365
3366
coll_base ("R_LOCK", indireto, agora-> ntyp, agora-> lft);
3367
coll_idx (agora-> lft);
3368
quebrar;
3369
3370
caso NAME:

```

```

coll_base ((Mustwrite) ? "W_LOCK": "R_LOCK", Direto, agora-> ntyp, agora);
3371
coll_idx (agora);
3372
quebrar;
3373
3374
case CONST:
3375
case 'p':
3376
case 'q':
3377
padrão :
quebrar;
3378
}
3379}
3380
3381 vazio
3382 push_loss (fd, agora, como)
3383
ARQUIVO * fd;
3384
Nó * agora;
3385 {
3386
Nó * v;
3387
/* caso especial: atualiza os conjuntos de conflito quando
3388
* ocorre uma perda de mensagem na opção -m
3389
* conta como uma leitura no id do canal
3390
*/
3391
putbase (fd, "R_LOCK", How, now-> lft);
3392
fprintf (fd, "push_act (II, R_LOCK,% s, t-> forw,",
3393
(Como == 0)? "REL": "BLOCK");
3394
putname (fd, "1 + MAXCONFL +", agora-> lft, 0, "); \ n \ t \ t");
3395
putindex (fd, now-> lft, How);
3396
para (v = agora-> rgt; v; v = v-> rgt)
3397
push_cs (fd, v-> lft, How);
3398}
3399
3400 vazio
3401 push_cs (fd, agora, como)
/* Como = 0, antes; Como = 1, depois */

```

502
APÊNDICE E
PROJETO E VALIDAÇÃO
3402
ARQUIVO * fd;
3403
Nó * agora;
3404 {
3405
Nó * v;
3406
3407
if (! now) {return; }
3408
switch (agora-> ntyp) {
3409
case 'c': case '!':
3410
case UMIN: case ASSERT:
3411
caso '^':
push_cs (fd, now-> lft, How);
3412

```

quebrar;
3413
3414
case '/': case '*': case '-': case '+':
3415
case '%': case '<': case '>': case '&':
3416
case '|': case LE: case GE: case NE:
3417
case EQ: case OR: case AND: case LSHIFT: case RSHIFT:
3418
push_cs (fd, now-> lft, How);
3419
push_cs (fd, now-> rgt, How);
3420
quebrar;
3421
3422
case PRINT:
3423
case RUN:
para (v = agora-> lft; v; v = v-> rgt)
3424
push_cs (fd, v-> lft, How);
3425
quebrar;
3426
3427
caso ASGN:
putbase (fd, "W_LOCK", How, now-> lft);
3428
if (now-> lft-> nsym-> nel! = 1)
3429
push_cs (fd, now-> lft-> lft, How);
3430
push_cs (fd, now-> rgt, How);
3431
quebrar;
3432
3433
case 'r':
fprintf (fd, "{int L_typ = Rcv_LOCK; \ n");
3434
fprintf (fd, "#if SYNC \ n");
3435
fprintf (fd, "\ t \ tint od = profundidade; \ n");
3436
fprintf (fd, "#se ASSÍNC \ n");
3437
putname (fd, "\ t \ tif (q_zero (", agora-> lft, 0, ")) \ n");
3438
fprintf (fd, "#endif \ n");
3439
fprintf (fd, "\ t \ t {
profundidade--; L_typ = Snd_LOCK; } \ n ");
3440
/*
3441
* a profundidade é diminuída aqui para garantir que estes
3442
* os blocos são comprometidos e não empurrados pelo
3443
* enviar metade do encontro
3444
*/
3445
fprintf (fd, "#endif \ n \ t \ t");
3446
putbase (fd, "R_LOCK", How, now-> lft);
3447
Countm++;
3448
fprintf (fd, "push_act (II, L_typ,% s, t-> forw,",
3449
(Como == 0)? "REL": "BLOCK");
3450
putname (fd, "1 + MAXCONFL +", agora-> lft, 0, "); \ n \ t \ t");
3451
putindex (fd, now-> lft, How);
3452
Mustwrite = 1;
3453

```

```

para (v = agora-> rgt; v; v = v-> rgt)
3454
push_cs (fd, v-> lft, How);
3455
Mustwrite = 0;

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

503
3456
fprintf (fd, "\ n # if SYNC \ n");
3457
fprintf (fd, "
profundidade = od; \ n ");
3458
fprintf (fd, "#endif \ n");
3459
fprintf (fd, "
} \ n \ t \ t ");
3460
quebrar;
3461
3462
case 's':
if (How == 1) /* lock rv's apenas no ponto de recebimento */
3463
{
fprintf (fd, "if (SYNC == 0 ||! q_zero");
3464
putname (fd, ("", agora-> lft, 0, ""));
3465
}
3466
putbase (fd, "R_LOCK", How, now-> lft);
3467
Countm++;
3468
fprintf (fd, "push_act (II, Snd_LOCK,% s, t-> forw",
3469
(Como == 0)? "REL": "BLOCK");
3470
putname (fd, "1 + MAXCONFL +", agora-> lft, 0, ");
3471
putindex (fd, now-> lft, How);
3472
para (v = agora-> rgt; v; v = v-> rgt)
3473
push_cs (fd, v-> lft, How);
3474
if (Como == 1)
3475
fprintf (fd, "} \ n \ t \ t ");
3476
quebrar;
3477
3478
case 'R':
putbase (fd, "R_LOCK", How, now-> lft);
3479
Countm++;
3480
fprintf (fd, "push_act (II, R_LOCK,% s, t-> forw",
3481
(Como == 0)? "REL": "BLOCK");
3482
putname (fd, "1 + MAXCONFL +", agora-> lft, 0, ");
3483
putindex (fd, now-> lft, How);
3484
para (v = agora-> rgt; v; v = v-> rgt)
3485
push_cs (fd, v-> lft, How);
3486
quebrar;
3487
3488
case LEN:
putbase (fd, "R_LOCK", How, now-> lft);
3489

```

```

Countm++;
3490
fprintf (fd, "push_act (II, R_LOCK,% s, t-> forw,",
3491
(Como == 0)? "REL": "BLOCK");
3492
putname (fd, "l + MAXCONFL +", agora-> lft, 0, "); \ n \ t \ t");
3493
putindex (fd, now-> lft, How);
3494
quebrar;
3495
3496
caso NAME:
putbase (fd, (Mustwrite)? "W_LOCK": "R_LOCK", como, agora);
3497
if (now-> nsym-> nel! = 1)
3498
push_cs (fd, now-> lft, How);
3499
quebrar;
3500
3501
case TIMEOUT: fprintf (fd, "push_act (II, R_LOCK,% s, t-> forw,",
3502
(Como == 0)? "REL": "BLOCK");
3503
fprintf (fd, "CS_timeout); \ n \ t \ t");
3504
quebrar;
3505
case 'p':
3506
case 'q':
3507
case CONST:
3508
padrão :
quebrar;
3509
}
3509}

```

Página 515

```

504
APÊNDICE E
PROJETO E VALIDAÇÃO
3510 #endif
3511
3512 # define cat0 (x)
putstmtnt (fd, agora-> lft, m); fprintf (fd, x); \
3513
putstmtnt (fd, now-> rgt, m)
3514 # define cat1 (x)
fprintf (fd, "("); cat0 (x); fprintf (fd, ")")
3515 # define cat2 (x, y)
fprintf (fd, x); putstmtnt (fd, y, m)
3516 # define cat3 (x, y, z)
fprintf (fd, x); putstmtnt (fd, y, m); fprintf (fd, z)
3517
3518 vazio
3519 putstmtnt (fd, agora, m)
3520
ARQUIVO * fd;
3521
Nó * agora;
3522 {
3523
Nó * v;
3524
int i, j;
3525
3526
if (! now) {fprintf (fd, "0"); Retorna; }
3527
if (now-> ntyp! = CONST) lineno = now-> nval;
3528
switch (agora-> ntyp) {
3529
case CONST:
fprintf (fd, "% d", agora-> nval); quebrar;

```

```

3530
case '!':
cat3 ("! (", agora-> lft, ")"); quebrar;
3531
case UMIN:
cat3 ("- (", agora-> lft, ")"); quebrar;
3532
caso '^':
cat3 ("^ (", agora-> lft, ")"); quebrar;
3533
3534
case '/':
cat1 ("/"); quebrar;
3535
case '*':
cat1 ("*"); quebrar;
3536
3537
case '-':
cat1 ("-"); quebrar;
3538
case '+':
cat1 ("+"); quebrar;
3539
caso '%':
cat1 ("%"); quebrar;
3540
case '<':
cat1 ("<"); quebrar;
3541
case '>':
cat1 (">"); quebrar;
3542
case '&':
cat1 ("&"); quebrar;
3543
case '|':
cat1 ("|"); quebrar;
3544
case LE:
cat1 ("<="); quebrar;
3545
caso GE:
cat1 ("> ="); quebrar;
3546
caso NE:
cat1 ("! ="); quebrar;
3547
caso EQ:
cat1 ("=="); quebrar;
3548
caso OU:
cat1 ("||"); quebrar;
3549
case AND:
cat1 ("&&"); quebrar;
3550
case LSHIFT:
cat1 ("<<"); quebrar;
3551
case RSHIFT:
cat1 (">>"); quebrar;
3552
case TIMEOUT: fprintf (fd, "((trpt-> tau) & 1)"); quebrar;
3553
3554
case RUN:
if (Claimproc
3555
&& strcmp (agora-> nsym-> nome, Claimproc) == 0)
3556
fatal ("% s é reivindicação, não executável",
3557
Claimproc);
3558
if (EVAL_runs)
3559
{
fprintf (fd, "(agora._nr_pr <MAXPROC)");
3560
quebrar;
3561

```

```

}
3562
fprintf (fd, "addproc (% d", fproc (agora-> nsym-> nome));
3563
para (v = agora-> lft; v; v = v-> rgt)

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

505
3564
{
cat2 (",", v-> lft);
3565
}
3566
fprintf (fd, ")");
3567
quebrar;
3568
case LEN:
putname (fd, "q_len (", agora-> lft, m, ")");
3569
quebrar;
3570
3571
case 's':
fprintf (fd, "\ n # if (SYNC> 0 && ASYNC == 0) \ n \ t \ t");
3572
putname (fd, "if (q_len (", agora-> lft, m, ")) \ n");
3573
fprintf (fd, "#else \ n \ t \ t");
3574
putname (fd, "if (q_full (", agora-> lft, m, ")) \ n");
3575
fprintf (fd, "#endif \ n");
3576
if (m_loss)
3577
{
fprintf (fd, "\ t \ t {\n \ t \ tm = 3; perda ++; \ n");
3578
push_loss (fd, agora, 0);
3579
fprintf (fd, "goto P999; \ n \ t \ t} \ n \ t \ t");
3580
} outro
3581
fprintf (fd, "\ t \ t \ tcontinue; \ n \ t \ t");
3582
putname (fd, "qsend (", now-> lft, m, "");
3583
para (v = agora-> rgt, i = 0; v; v = v-> rgt, i++)
3584
{
cat2 (",", v-> lft);
3585
}
3586
if (i> Mpars)
3587
fatal ("muitos pars no envio", "");
3588
para (; i <Mpars; i++)
3589
fprintf (fd, ", 0");
3590
fprintf (fd, "); \ n");
3591
fprintf (fd, "#if SYNC \ n # if ASYNC == 0 \ n");
3592
putname (fd, "\ t \ tboq =", agora-> lft, m, "; \ n");
3593
fprintf (fd, "#else \ n \ t \ t");
3594
putname (fd, "if (q_zero (", now-> lft, m, "))");
3595
putname (fd, "boq =", agora-> lft, m, "; \ n");
3596
fprintf (fd, "#endif \ n # endif \ n \ t \ t");

```

```

3597
quebrar;
3598
case 'r':
fprintf (fd, "\ n # if SYNC \ n # if ASYNC == 0 \ n");
3599
putname (fd, "\ t \ tif (boq! =", agora-> lft, m, ")");
3600
fprintf (fd, "continue; \ n # else \ n");
3601
putname (fd, "\ t \ tif (q_zero (", agora-> lft, m, ")"));
3602
fprintf (fd, "\ n \ t \ t");
3603
putname (fd, "{if (boq! =", agora-> lft, m, ")");
3604
fprintf (fd, "continue; \ n \ t \ t} else \ n \ t \ t");
3605
fprintf (fd, "{if (boq! = -1) continue; \ n \ t \ t");
3606
fprintf (fd, "} \ n # endif \ n # endif \ n \ t \ t");
3607
putname (fd, "if (q_len (", agora-> lft, m, ") == 0)");
3608
fprintf (fd, "continuar");
3609
/* teste */
para (v = agora-> rgt, i = j = 0; v; v = v-> rgt, i++)
3610
{
if (v-> lft-> ntyp! = CONST)
3611
{
j++; continuar;
3612
}
3613
fprintf (fd, "; \ n \ t \ t");
3614
cat3 ("if (", v-> lft, " != ");
3615
putname (fd, "qrecv (", agora-> lft, m, ")");
3616
fprintf (fd, "0,% d, 0)) continue", i);
3617
}

```

Página 517

```

506
APÊNDICE E
PROJETO E VALIDAÇÃO
3618
if (j> 0)
3619
fprintf (fd, "; \ n \ t \ tsv_save ()");
3620
/* set */
para (v = agora-> rgt, i = 0; v; v = v-> rgt, i++)
3621
{
if (v-> lft-> ntyp == CONST && v-> rgt)
3622
continuar;
3623
fprintf (fd, "; \ n \ t \ t");
3624
if (v-> lft-> ntyp! = CONST)
3625
{
nocast = 1;
3626
putstmtnt (fd, v-> lft, m);
3627
nocast = 0; fprintf (fd, "=");
3628
}
3629
putname (fd, "qrecv (", agora-> lft, m, ")");
3630
fprintf (fd, ", 0,% d,", i);

```

```

3631
fprintf (fd, "% d", (v-> rgt)? 0: 1);
3632
}
3633
fprintf (fd, "; \ n # if SYNC \ n");
3634
putname (fd, "\ t \ tif (q_zero (", agora-> lft, m, ""));
3635
fprintf (fd, ")) boq = -1; \ n # endif \ n \ t \ t");
3636
quebrar;
3637
case 'R':
putname (fd, "(q_len (", agora-> lft, m, ")> 0");
3638
/* teste */
para (v = agora-> rgt, i = j = 0; v; v = v-> rgt, i++)
3639
{
if (v-> lft-> ntyp! = CONST)
3640
{
j ++; continuar;
3641
}
3642
fprintf (fd, "\ n \ t \ t && qrecv ());
3643
putname (fd, "", agora-> lft, m, "");
3644
fprintf (fd, ", 0,% d, 0) ==", i);
3645
putstmtnt (fd, v-> lft, m);
3646
}
3647
fprintf (fd, ")");
3648
quebrar;
3649
3650
case 'c':
cat3 ("if (! (", now-> lft, ")) \ n");
3651
fprintf (fd, "\ t \ t \ tcontinue");
3652
quebrar;
3653
caso ASGN:
cat3 ("(trpt + 1) -> oval =", agora-> lft, "; \ n \ t \ t");
3654
nocast = 1; putstmtnt (fd, agora-> lft, m); nocast = 0;
3655
fprintf (fd, "=");
3656
putstmtnt (fd, agora-> rgt, m);
3657
quebrar;
3658
case PRINT:
fprintf (fd, "printf (% s", agora-> nsym-> nome);
3659
para (v = agora-> lft; v; v = v-> rgt)
3660
{
cat2 (",", v-> lft);
3661
}
3662
fprintf (fd, ")");
3663
quebrar;
3664
caso NAME:
if (! nocast && now-> nsym
3665
&& now-> nsym-> digite <SHORT)
3666
putname (fd, "((int)", agora, m, ")");
3667
outro

```

```

3668
putname (fd, "", agora, m, "");
3669
quebrar;
3670
case 'p':
putremote (fd, agora, m);
3671
quebrar;



---



```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

507
3672
case 'q':
se (conciso)
3673
fprintf (fd, "% s", agora-> nsym-> nome);
3674
outro
3675
fprintf (fd, "% d", remotelab (agora));
3676
quebrar;
3677
case ASSERT:
cat3 ("assert (", agora-> lft, ",");
3678
conciso = nocast = 1;
3679
cat3 ("\n ", agora-> lft, "\\\ n \ ", II, tt, t));
3680
conciso = nocast = 0;
3681
quebrar;
3682
case '.':
3683
case BREAK:
3684
case GOTO:
putskip (m);
3685
quebrar;
3686
case '@':
if (EVAL_runs)
3687
{
fprintf (fd, "if (i + 1! = now._nr_pr) continue");
3688
quebrar;
3689
}
3690
fprintf (fd, "if (! delproc (1, II)) continue");
3691
fprintf (th, "#define endstate% d% d \ n", Pid, m);
3692
quebrar;
3693
padrão :
printf ("spin: tipo de nó inválido% d (.m) \ n", agora-> ntyp);
3694
fflush (tm); fflush (tb);
3695
fflush (tc); fflush (th); fflush (tt);
3696
saída (1);
3697
}
3698}
3699
3700 putfair (fd, agora, m, u)
3701
ARQUIVO * fd;
3702
Nó * agora;
3703 {

```

```

3704
Nó * v;
3705
int i, j;
3706
3707
3708
if (! now) {fprintf (fd, "0"); Retorna; }
3709
switch (agora-> ntyp) {
3710
padrão:
fprintf (fd, "\ tcase% d: \ n \ t \ t", u);
3711
EVAL_runs = 1; /* não executa um RUN ou @ */
3712
putstmtnt (fd, agora, m);
3713
EVAL_runs = 0;
3714
fprintf (fd, "; \ n \ t \ tbreak; \ n");
3715
quebrar;
3716
case ASSERT:
3717
case '.':
3718
case BREAK:
3719
case GOTO:
3720
caso ASGN:
3721
case PRINT:
fprintf (fd, "\ tcase% d: break; \ n", u);
3722
quebrar;
3723
3724
case 's':
if (m_loss)
3725
{
fprintf (fd, "\ tcase% d: break; \ n", u);

```

Página 519

508

APÊNDICE E
PROJETO E VALIDAÇÃO

```

3726
3726
quebrar;
3727
}
3728
fprintf (fd, "\ tcase% d:", u);
3729
fprintf (fd, "\ n # if (SYNC> 0 && ASYNC == 0) \ n \ t \ t");
3730
putname (fd, "if (q_len (", now-> lft, m, ")"));
3731
fprintf (fd, "\ n # else \ n \ t \ t");
3732
putname (fd, "if (q_full (", now-> lft, m, ")"));
3733
fprintf (fd, "\ n # endif \ n \ t \ t \ t");
3734
fprintf (fd, "continue; \ n");
3735
fprintf (fd, "\ t \ tbreak; \ n");
3736
quebrar;
3737
case 'r':
fprintf (fd, "\ tcase% d:", u);
3738
fprintf (fd, "\ n # if SYNC \ n # if ASYNC == 0 \ n");
3739
putname (fd, "\ t \ tif (bog! =", agora-> lft, m, ")");
3740

```

```

fprintf (fd, "continue; \ n # else \ n");
3741
putname (fd, "\ t \ tif (q_zero (", agora->lft, m, ")"));
3742
fprintf (fd, "\ n \ t \ t");
3743
putname (fd, "{if (boq! =", agora->lft, m, ")");
3744
fprintf (fd, "continue; \ n \ t \ t} else \ n \ t \ t");
3745
fprintf (fd, "{if (boq! = -1) continue; \ n \ t \ t");
3746
fprintf (fd, "} \ n # endif \ n # endif \ n \ t \ t");
3747
putname (fd, "if (q_len (", agora->lft, m, ") == 0)");
3748
fprintf (fd, "continuar");
3749
/* teste */
para (v = agora->rgt, i = j = 0; v; v = v->rgt, i++)
3750
{
if (v->lft->ntyp! = CONST)
3751
{
j++; continuar;
3752
}
3753
fprintf (fd, "; \ n \ t \ t");
3754
cat3 ("if (", v->lft, "!=");
3755
putname (fd, "qrecv (", agora->lft, m, ",");
3756
fprintf (fd, "0,% d, 0)) continue", i);
3757
}
3758
fprintf (fd, "; \ n \ t \ tbreak; \ n");
3759
quebrar;
3760
}
3761}
3762
3763 vazio
3764 putbase (fd, what, when, n)
3765
ARQUIVO * fd;
3766
char * o quê;
3767
Nó * n;
3768 {
3769
if (n->nsym->contexto)
3770
Retorna; /* não é global */
3771
3772
Countm++;
3773
fprintf (fd, "push_act (II,% s,% s, t->forw, CS_",
3774
o que, (quando == 0)? "REL": "BLOCK");
3775
fprintf (fd, "% s", n->nsym->nome);
3776
if (n->nsym->ncl> 1)
3777
{
fprintf (fd, "+");
3778
putstmtnt (fd, n->lft, 0);
3779
}

```

FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

509
3780
fprintf (fd, "\n \t \t");
3781}
3782
3783 vazio
3784 putindex (fd, n, How)
3785
ARQUIVO * fd;
3786
Nó * n;
3787 {
3788
if (Mustwrite! = 0)
3789
fprintf (stderr, "não pode acontecer putindex \n");
3790
if (n-> nsym-> nel! = 1)
3791
push_cs (fd, n-> lft, How);
3792
3793}
3794
3795 vazio
3796 coll_indx (n)
3797
Nó * n;
3798 {
3799
if (n-> nsym-> nel! = 1)
3800
coll_cs (n-> lft);
3801}
3802
3803 vazio
3804 col_base (o que, quando, causa, n)
3805
char * o quê;
3806
Nó * n;
3807 {
3808
if (n && n-> nsym-> contexto)
3809
Retorna; /* não é global */
3810
push_ast (o que, quando, causa, n);
3811}
3812
3813 vazio
3814 putname (fd, pre, n, m, suff)
/* varref */
3815
3816
ARQUIVO * fd;
3817
Nó * n;
3818
char * pre, * suff;
3819
Símbolo * s = n-> nsym;
3820
se (! s)
3821
fatal ("sem nome - putname", "");
3822
if (! s-> type)
3823
yyerror ("nome não declarado '% s'", s-> nome);
3824
3825
if (! s-> contexto || s-> type == CHAN)
3826
Globalname = 1;
3827
fprintf (fd, pré);
3828
if (s-> contexto ||! strcmp (s-> nome, "_p"))
3829
{
if (! conciso) fprintf (fd, "((P% d *) this) ->", Pid);

```

```

3830
fprintf (fd, "% s", s-> nome);
3831
} outro
3832
{
if (! conciso) fprintf (fd, "agora.");
3833
fprintf (fd, "% s", s-> nome);

```

```

510
APÊNDICE E
PROJETO E VALIDAÇÃO
3834
}
3835
if (s-> nel! = 1)
3836
{
cat3 ("[Index (", n-> lft, ","); /* BOUNDCHECK */
3837
fprintf (fd, "% d]", s-> nel); /* BOUNDCHECK */
3838
}
3839
fprintf (fd, suf);
3840
3841
3842 vazio
3843 putremoto (fd, n, m)
/* referência remota */
3844
ARQUIVO * fd;
3845
Nó * n;
3846 {
3847
promovido interno = 0;
3848
3849
se (conciso)
3850
{
fprintf (fd, "% s [", n-> lft-> nsym-> nome);
3851
putstmtnt (fd, n-> lft-> lft, m);
3852
if (strcmp (n-> nsym-> nome, "_p") == 0)
3853
fprintf (fd, "]:");
3854
outro
3855
fprintf (fd, "].% s", n-> nsym-> nome);
3856
} outro
3857
{
if (n-> nsym-> digite <SHORT && ! nocast)
3858
{
promovido = 1;
3859
fprintf (fd, "((int)");
3860
}
3861
fprintf (fd, "((P% d *) Pptr (loops +",
3862
fproc (n-> lft-> nsym-> nome));
3863
if (Claimproc) fprintf (fd, "1+");
3864
putstmtnt (fd, n-> lft-> lft, m);
3865
fprintf (fd, ") ->% s", n-> nsym-> nome);
3866 #if 0
3867
if (strcmp (n-> nsym-> nome, "_p") == 0)

```

```

3868
XXXXX LEITURA _p XXXXX
3869 #endif
3870 }
3871 if (n-> rgt)
3872 {
3873     fprintf (fd, "[");
3874     /* não pode fazer BOUNDCHECK */
3875     putstmnt (fd, n-> rgt, m);
3876     fprintf (fd, "]");
3877 }
3878 if (promovido) fprintf (fd, ")");
3879 getweight (n)
3880
Nó * n;
3881 {
3882 switch (n-> ntyp) {
3883 case 'r':
3884     return 4;
3885 case 's':
3886     return 2;
3887 case TIMEOUT: return 1; /* prioridade mais baixa */
3888 case 'c':
3889     if (has_typ (n-> lft, TIMEOUT)) retorna 1;
3890 }

```

Página 522

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

511
3888
return 3;
3889}
3890
3891 has_typ (n, m)
3892
Nó * n;
3893
m curto;
3894 {
3895
if (! n) retorna 0;
3896
se (n-> ntyp == m) retornar 1;
3897
return (has_typ (n-> lft, m) || has_typ (n-> rgt, m));
3898}
3899
3900 / ***** spin: pangen3.c **** */
3901
3902 #include <stdio.h>
3903 #include <ctype.h>
3904 #include "spin.h"
3905 #include "y.tab.h"
3906
3907 extern FILE
*;
3908 #ifdef GODEF
3909 extern int Globalname;
3910 extern int Countm;
3911 #endif
3912
3913 typedef struct SRC {
3914
short ln, st;

```

```

3915
struct SRC * nxt;
3916} SRC;
3917
3918 SRC
* primeiro = (SRC *) 0;
3919 SRC
* pular = (SRC *) 0;
3920 int
col;
3921
3922 vazio
3923 putskip (m) /* afirma que não precisa ser alcançado */
3924 {SRC * tmp;
3925
3926
para (tmp = ignorar; tmp; tmp = tmp-> nxt)
3927
if (tmp-> st == m)
3928
Retorna;
3929
tmp = (SRC *) emalloc (sizeof (SRC));
3930
tmp-> st = (curto) m;
3931
tmp-> nxt = ignorar;
3932
skip = tmp;
3933}
3934
3935 vazio
3936 putrc (n, m)
/* corresponder estados às linhas de origem */
3937 {SRC * tmp;
3938
3939
para (tmp = primeiro; tmp; tmp = tmp-> nxt)
3940
if (tmp-> st == m)
3941
{
if (tmp-> ln! = n)

```

Página 523

```

512
APÊNDICE E
PROJETO E VALIDAÇÃO
3942
printf ("putsrc mismatch% d -% d \ n");
3943
Retorna;
3944
}
3945
tmp = (SRC *) emalloc (sizeof (SRC));
3946
tmp-> ln = (curto) n;
3947
tmp-> st = (curto) m;
3948
tmp-> nxt = primeiro;
3949
primeiro = tmp;
3950}
3951
3952 vazio
3953 dumpskip (n, m)
3954 {SRC * tmp, * lst;
3955
int j;
3956
3957
fprintf (th, "uchar atingiu% d [] = {\n \ t", m);
3958
para (j = 0, col = 0; j <= n; j++)
3959
{
lst = (SRC *) 0;
3960

```

```

para (tmp = ignorar; tmp; lst = tmp, tmp = tmp-> nxt)
3961
if (tmp-> st == j)
3962
{
putnr (1);
3963
if (lst)
3964
lst-> nxt = tmp-> nxt;
3965
outro
3966
pular = tmp-> nxt;
3967
quebrar;
3968
}
3969
if (! tmp)
3970
putnr (0);
3971
}
3972
fprintf (th, "}; \ n");
3973
pular = (SRC *) 0;
3974}
3975
3976 vazio
3977 dumpsrc (n, m)
3978 {SRC * tmp, * lst;
3979
int j;
3980
3981
fprintf (th, "short src_ln% d [] = {\n \ t", m);
3982
para (j = 0, col = 0; j <= n; j++)
3983
{
lst = (SRC *) 0;
3984
para (tmp = primeiro; tmp; lst = tmp, tmp = tmp-> nxt)
3985
if (tmp-> st == j)
3986
{
putnr (tmp-> ln);
3987
if (lst)
3988
lst-> nxt = tmp-> nxt;
3989
outro
3990
frst = tmp-> nxt;
3991
quebrar;
3992
}
3993
if (! tmp)
3994
putnr (0);
3995
}

```

```

4000
4001 vazio
4002 putnr (n)
4003 {
4004
if (col ++ == 8)
4005
{
fprintf (th, "\n \t");
4006
col = 1;
4007
}
4008
fprintf (th, "% 3d,", n);
4009}
4010
4011 # define Cat0 (x)
comwork (fd, agora-> lft, m); fprintf (fd, x); \
4012
comwork (fd, now-> rgt, m)
4013 # define Cat1 (x)
fprintf (fd, "("; Cat0 (x); fprintf (fd, ")")
4014 # define Cat2 (x, y)
fprintf (fd, x); comwork (fd, y, m)
4015 # define Cat3 (x, y, z)
fprintf (fd, x); comwork (fd, y, m); fprintf (fd, z)
4016
4017 vazio
4018 simbólico (fd, v)
4019
ARQUIVO * fd;
4020 {
4021
Nó * n; Nó externo * Mtype;
4022
int cnt = 1;
4023
4024
para (n = tipo M; n; n = n-> rgt, cnt++)
4025
if (cnt == v)
4026
{
fprintf (fd, "% s", n-> lft-> nsym-> nome);
4027
quebrar;
4028
}
4029
if (! n)
4030
fprintf (fd, "% d", v);
4031}
4032
4033 vazio
4034 comwork (fd, agora, m)
4035
ARQUIVO * fd;
4036
4037 {
4038
Nó * v;
4039
int i, j; extern int Mpars;
4040
4041
if (! now) {fprintf (fd, "0"); Retorna; }
4042
switch (agora-> ntyp) {
4043
case CONST:
fprintf (fd, "% d", agora-> nval); quebrar;
4044
case '!':
Cat3 ("! (", Agora-> lft, ")"); quebrar;
4045
case UMIN:
Cat3 ("- (", agora-> lft, ")"); quebrar;
4046
caso '^':

```

```
Cat3 ("~ (", agora-> lft, ")"); quebrar;
4047
4048
case '/':
Cat1 ("/"); quebrar;
4049
case '*':
Cat1 ("*"); quebrar;
```

Página 525

```
514
APÊNDICE E
PROJETO E VALIDAÇÃO
4050
4050
case '-':
Cat1 ("-"); quebrar;
4051
case '+':
Cat1 ("+"); quebrar;
4052
caso '%':
Cat1 ("%%"); quebrar;
4053
case '<':
Cat1 ("<"); quebrar;
4054
case '>':
Cat1 (">"); quebrar;
4055
case '&':
Cat1 ("&"); quebrar;
4056
case '|':
Cat1 ("|"); quebrar;
4057
case LE:
Cat1 ("<="); quebrar;
4058
caso GE:
Cat1 ("> ="); quebrar;
4059
caso NE:
Cat1 ("! ="); quebrar;
4060
case EQ:
Cat1 ("=="); quebrar;
4061
caso OU:
Cat1 ("||"); quebrar;
4062
case AND:
Cat1 ("&&"); quebrar;
4063
case LSHIFT:
Cat1 ("<<"); quebrar;
4064
case RSHIFT:
Cat1 (">>"); quebrar;
4065
4066
caso RUN:
fprintf (fd, "execute% s (", agora-> nsym-> nome);
4067
para (v = agora-> lft; v; v = v-> rgt)
4068
if (v == agora-> lft)
4069
{
Cat2 ("", v-> lft);
4070
} outro
4071
{
Cat2 (",", v-> lft);
4072
}
4073
fprintf (fd, ")");
4074
quebrar;
```

```

4075
4076
case LEN:
putname (fd, "len (", now-> lft, m, ")");
4077
quebrar;
4078
4079
case 's':
putname (fd, "", agora-> lft, m, "!");
4080
para (v = agora-> rgt, i = 0; v; v = v-> rgt, i++)
4081
{
if (v! = now-> rgt) fprintf (fd, ",");
4082
if (v-> lft-> ntyp == CONST)
4083
simbólico (fd, v-> lft-> nval);
4084
outro
4085
4085
comwork (fd, v-> lft, m);
4086
}
4087
para ( ; i <Mpars; i++)
4088
fprintf (fd, ", 0");
4089
quebrar;
4090
4090
case 'r':
putname (fd, "", agora-> lft, m, "?");
4091
para (v = agora-> rgt, i = j = 0; v; v = v-> rgt, i++)
4092
{
if (v! = now-> rgt) fprintf (fd, ",");
4093
if (v-> lft-> ntyp == CONST)
4094
simbólico (fd, v-> lft-> nval);
4095
outro
4096
4096
comwork (fd, v-> lft, m);
4097
}
4098
quebrar;
4099
4099
case 'R':
putname (fd, "", now-> lft, m, "? [");
4100
para (v = agora-> rgt, i = j = 0; v; v = v-> rgt, i++)
4101
{
if (v! = now-> rgt) fprintf (fd, ",");
4102
if (v-> lft-> ntyp == CONST)
4103
simbólico (fd, v-> lft-> nval);

```

```

515
4104
outro
4105
4105
comwork (fd, v-> lft, m);
4106
}
4107
fprintf (fd, "]");
4108
quebrar;
4109
4110

```

```

case 'c':
Cat3 ("(", agora-> lft, ")");
4111
quebrar;
4112
caso ASGN:
comwork (fd, agora-> lft, m);
4113
fprintf (fd, "=");
4114
comwork (fd, agora-> rgt, m);
4115
quebrar;
4116
caso PRINT:
{
char buf [512];
4117
strncpy (buf, agora-> nsym-> nome, 510);
4118
para (i = strlen (buf) -1; i> = 0; i--)
4119
if (buf [i] == '\\ ')
4120
buf [i] = '\\ ';
4121
fprintf (fd, "printf (% s", buf);
4122
}
4123
para (v = agora-> lft; v; v = v-> rgt)
4124
{
Cat2 (",", v-> lft);
4125
}
4126
fprintf (fd, ")");
4127
quebrar;
4128
caso NAME:
putname (fd, "", agora, m, "");
4129
quebrar;
4130
caso 'p':
putremote (fd, agora, m);
4131
quebrar;
4132
caso 'q':
fprintf (fd, "% s", agora-> nsym-> nome);
4133
quebrar;
4134
caso ASSERT:
Cat3 ("assert (", agora-> lft, ")");
4135
quebrar;
4136
caso '.':
4137
caso BREAK:
4138
caso GOTO:
fprintf (fd, "goto", m); quebrar;
4139
caso '@':
4140
fprintf (fd, "@", m); quebrar;
4141
4142
caso ATOMIC:
4143
fprintf (fd, "ATOMIC");
4144
quebrar;
4145
caso IF:
4146
fprintf (fd, "IF");

```

```

4147
quebrar;
4148
case DO:
4149
fprintf (fd, "DO");
4150
quebrar;
4151
case TIMEOUT:
4152 #ifdef GODEF
4153
Globalname = 1; /* não considere isso local */
4154 #endif
4155
fprintf (fd, "tempo limite");
4156
quebrar;
4157
padrão:
if (isprint (now-> ntyp))

```

Página 527

```

516
APÊNDICE E
PROJETO E VALIDAÇÃO
4158
fprintf (fd, "% c", agora-> ntyp);
4159
outro
4160
fprintf (fd, "% d", agora-> ntyp);
4161
quebrar;
4162
}
4163}
4164
4165 vazio
4166 comentário (fd, agora, m)
4167
ARQUIVO * fd;
4168
Nó * agora;
4169 {
4170
extern int conciso, nocast;
4171 #ifdef GODEF
4172
Globalname = 0;
4173 #endif
4174
conciso = nocast = 1;
4175
comwork (fd, agora, m);
4176
conciso = nocast = 0;
4177}
4178
4179 #ifdef GODEF
4180
4181 atom_stack * top_ast = 0;
4182
4183 push_ast (o que, quando, causa, n)
4184
char * o quê;
4185
Nó * n;
4186 {
4187
atom_stack * tmp, * lst;
4188
4189
para (tmp = top_ast; tmp; tmp = tmp-> nxt)
4190
if (strcmp (tmp-> what, what) == 0
4191
&& tmp-> quando == quando
4192
&& tmp-> n && n

```

```

4193
&& tmp-> n-> nsym == n-> nsym)
4194
Retorna;
4195
tmp = (atom_stack *) emalloc (sizeof (atom_stack));
4196
tmp-> what = (char *) emalloc (strlen (what) + 1);
4197
strcpy (tmp-> o quê, o quê);
4198
tmp-> quando = quando;
4199
tmp-> causa = causa;
4200
tmp-> n
= n;
4201
if (causa == 'r' || causa == 's' ||! top_ast)
4202
{
tmp-> nxt = top_ast;
4203
ao passado
= tmp;
4204
} else /* tail add */
4205
{
para (lst = top_ast; lst-> nxt; lst = lst-> nxt)
4206
;
4207
lst-> nxt = tmp;
4208
}
4209}
4210
4211 static int rHeader;

```

Página 528

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

517
4212 static int sHeader;
4213
4214 lastfirst (fd, tmp)
4215
ARQUIVO * fd;
4216
atom_stack * tmp;
4217 {
4218
if (! tmp) return;
4219
lastfirst (fd, tmp-> nxt);
4220
switch (tmp-> causa) {
4221
case 'r':
4222
if (! rHeader) quebra;
4223
rHeader = 0;
4224
fprintf (fd, "\n # if SYNC \n");
4225
fprintf (fd, "
profundidade = od; \n ");
4226
fprintf (fd, "#endif \n");
4227
fprintf (fd, "
} / * rH * / \n \t \t ");
4228
quebrar;
4229
case 's':
4230
if (! sHeader) quebra;

```

```

4231
sHeader = 0;
4232
fprintf (fd, "} / * sh * / \ n \ t \ t");
4233
padrão :
4234
quebrar;
4235
}
4236}
4237
4238 pop_ast (fd, como)
4239
ARQUIVO * fd;
4240 {
4241
atom_stack * tmp;
4242
4243
rHeader = 0;
4244
sHeader = 0;
4245
para (tmp = top_ast; tmp; tmp = tmp-> nxt)
4246
{
Countm++;
4247
switch (tmp-> causa) {
4248
case 'r':
4249
if (rHeader == 0)
4250
{
fprintf (fd, "{int L_typ = Rcv_LOCK; \ n");
4251
fprintf (fd, "#if SYNC \ n");
4252
fprintf (fd, "\ t \ tint od = profundidade; \ n");
4253
fprintf (fd, "#se ASSÍNC \ n");
4254
putname (fd, "\ t \ tif (q_zero (", tmp-> n, 0, ""));
4255
fprintf (fd, "#endif \ n");
4256
fprintf (fd, "\ t \ t {\ \ tdepth--; L_typ = Snd_LOCK;} \ n");
4257
fprintf (fd, "#endif \ n \ t \ t");
4258
rHeader++;
4259
}
4260
pop_common (fd, tmp, como);
4261
quebrar;
4262
4263
case 's':
4264
if (sHeader == 0 && how == 1)
4265
{
fprintf (fd, "if (SYNC == 0 ||! q_zero");

```

518
APÊNDICE E
PROJETO E VALIDAÇÃO
4266
putname (fd, "(" , tmp-> n, 0, ""));
4267
sHeader++;
4268
4269
pop_common (fd, tmp, como);

```

4270
quebrar;
4271
case TIMEOUT:
4272
fprintf (fd, "push_act (II, R_LOCK,% s, t-> forw,",
4273
(como == 0)? "REL": "BLOCK");
4274
fprintf (fd, "CS_timeout); / * + * / \ n \ t \ t");
4275
quebrar;
4276
padrão:
4277
pop_common (fd, tmp, como);
4278
quebrar;
4279
}
4280
}
4281
lastfirst (fd, top_ast);
4282
if (how == 1) clear_ast ();
4283}
4284
4285 pop_common (fd, tmp, como)
4286
ARQUIVO * fd;
4287
atom_stack * tmp;
4288{
4289
if (tmp-> quando == Direto)
4290
{
fprintf (fd, "push_act (II,% s,", tmp-> o quê);
4291
fprintf (fd, "% s, t-> forw, CS_",
4292
(como == 0)? "REL": "BLOCK");
4293
fprintf (fd, "% s", tmp-> n-> nsym-> nome);
4294
if (tmp-> n-> nsym-> nel> 1)
4295
{
fprintf (fd, "+");
4296
putstmtnt (fd, tmp-> n-> lft, 0);
4297
}
4298
fprintf (fd, "); / * + * / \ n \ t \ t");
4299
} else if (tmp-> when == Indireto)
4300
{
if (tmp-> cause == 'r')
4301
fprintf (fd, "push_act (II, L_typ,% s, t-> forw,",
4302
(como == 0)? "REL": "BLOCK");
4303
outro
4304
fprintf (fd, "push_act (II,% s,% s, t-> forw,",
4305
tmp-> o quê, (como == 0)? "REL": "BLOCK");
4306
putname (fd, "1 + MAXCONFL +", tmp-> n, 0, ""); / * + * / \ n \ t \ t");
4307
} outro
4308
{
fprintf (stderr, "não pode acontecer pop_ast \ n");
4309
abortar();
4310
}

```

```

4311}
4312
4313 has_ast ()
4314 {
4315
return (top_ast! = 0);
4316}
4317
4318 clear_ast ()
4319 {

```

Página 530

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

519
4320
/* não ligue grátis, evite perder tempo no malloc */
4321
top_ast = (atom_stack *) 0;
4322}
4323
4324 atom_stack *
4325 save_ast ()
4326 {
4327
return top_ast;
4328}
4329
4330 restor_ast (oCS)
4331
atom_stack * oCS;
4332 {
4333
top_ast = oCS;
4334}
4335
4336 coll_global (s, como)
4337
Sequência * s;
4338 {
4339
Elemento * f, * g;
4340
SeqList * h;
4341
4342
if (! s) return;
4343
para (f = s-> primeiro; f = f-> nxt)
4344
{
coll_cs (f-> n);
4345
para (h = f-> sub; h; h = h-> nxt)
4346
coll_global (h-> isso, como);
4347
if (f == s-> last)
4348
quebrar;
4349
}
4350}
4351 #endif
4352
4353 / ***** spin: pangen4.c ***** /
4354
4355 #include <stdio.h>
4356 #include "spin.h"
4357 #include "y.tab.h"
4358
4359 extern FILE
* tc, * tb;
4360 fila externa
* qtab;
4361 extern int nocast;
4362 extern int lineno;
4363 caracteres externos
* R13 [], * R14 [], * R15 [];
4364

```

```

4365 vazio
4366 undostmnt (agora, m)
4367
Nó * agora;
4368 {
4369
Nó * v;
4370
int i, j; extern int m_loss;
4371
4372
se (! agora)
4373
{
fprintf (tb, "0");

```

Página 531

```

520
APÊNDICE E
PROJETO E VALIDAÇÃO
4374
Retorna;
4375
}
4376
lineno = agora-> nval;
4377
switch (agora-> ntyp) {
4378
case CONST:
case '!':
case UMIN:
4379
caso '^':
case '/':
case '*':
4380
case '-':
case '+':
caso '%':
4381
case '<':
case '>':
case '&':
4382
case '|':
case LE:
caso GE:
4383
caso NE:
caso EQ:
caso OU:
4384
case AND:
case LSHIFT:
case RSHIFT:
4385
case TIMEOUT: case LEN:
caso NAME:
4386
case 'R':
putstmnt (tb, agora, m);
4387
quebrar;
4388
case RUN:
fprintf (tb, "delproc (0, agora._nr_pr-1)");
4389
quebrar;
4390
case 's':
if (m_loss)
4391
{
fprintf (tb, "if (m! = 2) / * msg foi perdida * / \n \t \t");
4392
fprintf (tb, "{\n \t \t");
4393
push_loss (tb, agora, 1);
4394

```

```

fprintf (tb, "goto R999; \ n \ t \ t");
4395
fprintf (tb, "} \ n \ t \ t");
4396
}
4397
fprintf (tb, "m = cancelar o envio");
4398
putname (tb, "(", agora-> lft, m, ")");
4399
quebrar;
4400
case 'r':
para (v = agora-> rgt, j = 0; v; v = v-> rgt)
4401
if (v-> lft-> ntyp! = CONST)
4402
j++;
4403
if (j> 0)
/* variáveis foram definidas */
4404
{
fprintf (tb, "sv_restor ()");
4405
quebrar;
4406
}
4407
para (v = agora-> rgt, i = 0; v; v = v-> rgt, i++)
4408
{
fprintf (tb, "unrecv");
4409
putname (tb, "(", agora-> lft, m, ", 0,");
4410
fprintf (tb, "% d", i);
4411
undostmnt (v-> lft, m);
4412
fprintf (tb, ",% d); \ n \ t \ t", (i == 0)? 1: 0);
4413
}
4414
quebrar;
4415
case '@':
fprintf (tb, "p_restor (II)");
4416
quebrar;
4417
caso ASGN:
nocast = 1; putstmtnt (tb, agora-> lft, m);
4418
nocast = 0; fprintf (tb, "= trpt-> oval");
4419
check_proc (agora-> rgt, m);
4420
quebrar;
4421
case 'c':
check_proc (agora-> lft, m);
4422
quebrar;
4423
case '.':
4424
case GOTO:
4425
case BREAK:
quebrar;
4426
case ASSERT:
4427
case PRINT:
check_proc (agora, m);

```

```

521
4428
quebrar;
4429
padrão:
printf ("spin: tipo de nó inválido% d (.b) \ n",
4430
agora-> ntyp);
4431
saída (1);
4432
}
4433}
4434
4435 any_undo (agora)
4436
Nó * agora;
4437 {/ * há algo para desfazer em um movimento de retorno? * /
4438
4439
if (! now) return 1;
4440
switch (agora-> ntyp) {
4441
case 'c':
return any_proc (now-> lft);
4442
case ASSERT:
4443
case PRINT:
retornar any_proc (agora);
4444
4445
case '.':
4446
case GOTO:
4447
case BREAK:
return 0;
4448
4449
padrão:
return 1;
4450
4451
4452 any_proc (agora)
4453
Nó * agora;
4454 {/ * verifique se uma expressão se refere a um processo * /
4455
if (! now) return 0;
4456
if (now-> ntyp == '@' || now-> ntyp == RUN)
4457
return 1;
4458
return (any_proc (now-> lft) || any_proc (now-> rgt));
4459}
4460
4461 vazio
4462 check_proc (agora, m)
4463
Nó * agora;
4464 {
4465
se (! agora)
4466
Retorna;
4467
if (now-> ntyp == '@' || now-> ntyp == RUN)
4468
{
fprintf (tb, "; \ n \ t \ t");
4469
undostmnt (agora, m);
4470
}
4471
check_proc (agora-> lft, m);
4472
check_proc (agora-> rgt, m);

```

```

4473}
4474
4475 vazio
4476 genunio ()
4477 {char * buf1;
4478
Fila * q; int i;
4479
4480
buf1 = (char *) emalloc (128);
4481
n vezes (tc, 0, 1, R13);

```

Página 533

```

522
APÊNDICE E
PROJETO E VALIDAÇÃO
4482
para (q = qtab; q; q = q-> nxt)
4483
{
sprintf (buf1, "((Q% d *) z) -> conteúdo [j] .fld", q-> qid);
4484
fprintf (tc, "caso% d: \ n", q-> qid);
4485
para (i = 0; i <q-> nflds; i++)
4486
fprintf (tc, "\ t \ t% s% d = 0; \ n", buf1, i);
4487
if (q-> nslots == 0)
4488
{
/* verificar se o encontro foi bem-sucedido, 1 nível abaixo */
4489
fprintf (tc, "\ t \ tm = (trpt + 1) -> o_m; \ n");
4490
fprintf (tc, "\ t \ tUnBlock; \ n");
4491
} outro
4492
fprintf (tc, "\ t \ tm = trpt-> o_m; \ n");
4493
fprintf (tc, "\ t \ tbreak; \ n");
4494
}
4495
n vezes (tc, 0, 1, R14);
4496
para (q = qtab; q; q = q-> nxt)
4497
{
sprintf (buf1, "((Q% d *) z) -> conteúdo", q-> qid);
4498
fprintf (tc, "caso% d: \ n", q-> qid);
4499
if (q-> nslots == 0)
4500
fprintf (tc, "\ t \ tif (strt) boq = de; \ n");
4501
else if (q-> nslots> 1) /* shift */
4502
{
fprintf (tc, "\ t \ tif (strt && slot <% d) \ n",
4503
q-> nslots-1);
4504
fprintf (tc, "\ t \ t {\ tfor (j--; j> = slot; j -) \ n");
4505
fprintf (tc, "\ t \ t \ t {\");
4506
para (i = 0; i <q-> nflds; i++)
4507
{
fprintf (tc, "\ t% s [j + 1] .fld% d = \ n \ t \ t \ t",
4508
buf1, i);
4509
fprintf (tc, "\ t% s [j] .fld% d; \ n \ t \ t \ t",
4510
buf1, i);

```

```

4511
}
4512
fprintf (tc, "} \ n \ t \ t} \ n");
4513
}
4514
strcat (buf1, "[slot] .fld");
4515
fprintf (tc, "\ t \ tif (strt) {\ \ n");
4516
para (i = 0; i <q-> nflds; i++)
4517
fprintf (tc, "\ t \ t \ t% s% d = 0; \ n", buf1, i);
4518
fprintf (tc, "\ t \ t} \ n");
4519
if (q-> nflds == 1)
/* set */
4520
fprintf (tc, "\ t \ tif (fld == 0)% s0 = fldvar; \ n",
4521
buf1);
4522
outro
4523
{
fprintf (tc, "\ t \ tswitch (fld) {\ \ n");
4524
para (i = 0; i <q-> nflds; i++)
4525
{
fprintf (tc, "\ t \ tcase% d: \ t% s", i, buf1);
4526
fprintf (tc, "% d = fldvar; break; \ n", i);
4527
}
4528
fprintf (tc, "\ t \ t} \ n");
4529
}
4530
fprintf (tc, "\ t \ tbreak; \ n");
4531
}
4532
n vezes (tc, 0, 1, R15);
4533}
4534
4535 / ***** spin: pangen5.c ***** /

```

Página 534

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

523
524
525
526
527
528
529
530
531
532
533
534
535
536
537 #include <stdio.h>
538 #include <sys / types.h>
539 #include <sys / stat.h>
540 #include "spin.h"
541 #include "y.tab.h"
542
543 extern int nproc, nstop, Tval, Rvous, Have_claim;
544 extern RunList
* executar, * X;
545 extern int verbose, lineno;
546 profundidade interna externa;
547
548 ARQUIVO * fd;
549
550 vazio
551 que proc (p)
552 {RunList * oX;
553
554 para (oX = executar; oX; oX = oX-> nxt)
555
556 if (oX-> pid == p)
557 {

```

```

printf ("%s", oX-> n-> nome);
4557
quebrar;
4558
}
4559}
4560
4561 int
4562 mais recente (f1, f2)
4563
char * f1, * f2;
4564 {
4565
struct stat x, y;
4566
4567
if (stat (f1, (struct stat *) & x) <0) retorna 0;
4568
if (stat (f2, (struct stat *) & y) <0) retorna 1;
4569
if (x.st_mtime <y.st_mtime) retorna 0;
4570
return 1;
4571}
4572
4573 vazio
4574 match_trail ()
4575 {int i, pno, nst, lv0 = 0, lv1 = 0;
4576
símbolo externo * Fname;
4577
4578
if (Fname-> nome [0] == '\\ ')
4579
{
i = strlen (Fname-> nome);
4580
Fname-> nome [i-1] = '\\ 0';
4581
Fname = lookup (& Fname-> nome [1]);
4582
}
4583
4584
if (mais recente (Fname-> nome, "pan.trail"))
4585
printf ("Aviso, arquivo %s modificado desde que a trilha foi escrita \\n",
4586
Fname-> nome);
4587
4588
if (! (fd = fopen ("pan.trail", "r")))
4589
{
printf ("spin -t: não foi possível encontrar 'pan.trail' \\n");

```

Página 535

524
APÊNDICE E
PROJETO E VALIDAÇÃO
4590
saída (1);
4591
}
4592
Tval = 1; /* timeouts podem fazer parte da trilha */
4593
while (fscanf (fd, "% d:% d:% d:% d \\n", & profundidade, & pno, & nst, & lv0)
4594
== 4)
4595
{
if (lv1 >= 0 && depth > 0 && (verbose & 32 || lv1 != lv0))
4596
talk (X-> pc, X-> syntab);
4597
lv1 = lv0;
/* não verboso em etapas intermediárias */
4598
if (profundidade == -1)

```

4599
{
if (verboso)
4600
printf ("<<<< INÍCIO DO CICLO >>>> \ n");
4601
continuar;
4602
}
4603
if (profundidade == -2)
4604
{
4605
start_claim (pno);
4606
continuar;
4607
}
4608
i = nproc - nstop;
4609
if (nst == 0)
4610
{
if (pno == i-1 && run-> pc-> n-> ntyp == '@')
4611
{
run = run-> nxt;
4612
nstop++;
4613
continuar;
4614
} outro
4615
{
printf ("etapa% d: erro de parada,% d% d% c \ n",
4616
profundidade, pno, i, run-> pc-> n-> ntyp);
4617
saída (1);
4618
}
}
4619
para (X = executar; X; X = X-> nxt)
4620
{
4621
if (--i == pno)
4622
quebrar;
4623
}
4624
if (! X)
4625
{
int k = 0;
4626
printf ("etapa% d: trilha perdida", profundidade); qualproc (pno);
4627
if (Have_claim)
4628
{
if (pno == 1)
4629
printf ("(estado% d) \ n", nst);
4630
outro
4631
{
se (pno> 1) k = 1;
4632
printf ("(proc% d estado% d) \ n", pno-k, nst);
4633
}
4634
} outro
4635
printf ("(proc% d estado% d) \ n", pno-k, nst);

```

```

4636
lost_trail ();
4637
embrulhar();
4638
saída (1);
4639
}
4640
lineno = X-> pc-> n-> nval;
4641
Faz
4642
{
X-> pc = d_eval_sub (X-> pc, pno, nst);
4643
} enquanto (X && X-> pc && X-> pc-> seqno! = nst);

```

Página 536

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN

```

525
4644
if (! X || ! X-> pc)
4645
{
int k = 0;
4646
printf ("etapa% d: trilha perdida", profundidade); qualproc (pno);
4647
if (Have_claim)
4648
{
if (pno == 1)
4649
printf ("(estado% d) \ n", nst);
4650
outro
4651
{
se (pno> 1) k = 1;
4652
printf ("(proc% d estado% d.) \ n", pno-k, nst);
4653
}
4654
} outro
4655
printf ("(proc% d estado% d.) \ n", pno, nst);
4656
lost_trail ();
4657
embrulhar();
4658
saída (1);
4659
}
4660
}
4661
talk (X-> pc, X-> symtab);
4662
printf ("rotação: a trilha termina após% d etapas \ n", profundidade);
4663
embrulhar();
4664}
4665
4666 vazio
4667 lost_trail ()
4668 {int d, p, n, l;
4669
4670
while (fscanf (fd, "% d:% d:% d:% d \ n", & d, & p, & n, & l) == 4)
4671
{
printf ("etapa% d: proc% d", d, p); qual proc (p);
4672
printf ("(estado% d) - d% d \ n", n, l);
4673
}

```

```

4674}
4675
4676 Int Profundidade = 0;
4677
4678 Element * 
4679 walk_sub (e, pno, nst)
4680
Elemento * e;
4681 {
4682
SeqList * z;
4683
Elemento * f;
4684
4685
if (Profundidade > 32) /* provavelmente circular */
4686
return (Element *) 0;
4687
Profundidade++;
4688
para (z = e-> sub; z; z = z-> nxt)
4689
{
4690
if (z-> this-> frst-> seqno == nst)
4691
{
Profundidade--; return z-> this-> primeiro; }
4692
if (! z-> this-> primeiro-> nxt)
4693
fatal ("não pode acontecer", "walk_sub");
4694
if (z-> this-> frst-> sub)
4695
{
f = walk_sub (z-> this-> primeiro, pno, nst);
4696
if (f) {Profundidade--; return f; }
4697
}

```

```

526
APÊNDICE E
PROJETO E VALIDAÇÃO
4698
f = huntele (z-> isto-> primeiro, z-> isto-> primeiro-> estado);
4699
if (f-> seqno == nst)
4700
{
Profundidade--; return f; }
4701
if (f-> seqno == X-> pc-> seqno)
/* looping */
4702
continuar;
/* falha */
4703
if (f-> sub && (f = walk_sub (f, pno, nst)))
4704
{
Profundidade--; return f; }
4705
if (f && f-> n-> ntyp == ATOMIC)
4706
{
f = f-> n-> seql-> this-> primeiro;
4707
if (f-> seqno == nst)
4708
{
Profundidade--; return f; }
4709
}
4710
}
4711

```

```

Profundidade--;
4712
return (Element *) 0;
4713}
4714
Elemento * 4715 *
4716 d_eval_sub (s, pno, nst)
4717
Elemento * s;
4718 {
4719
Elemento * e = s;
4720
4721
if (e-> n-> ntyp == GOTO)
4722
{
4723
return get_lab (e-> n-> nsym);
4724
}
4725
if (e-> sub)
4726
{
if (e = walk_sub (e, pno, nst))
4727
{
4728
return e;
4729
}
4730
} else if (e-> n && e-> n-> ntyp == ATOMIC)
4731
{
e-> n-> seql-> this-> last-> nxt = e-> nxt;
4732
if (e-> n-> seql-> this-> primeiro-> seqno == nst)
4733
return e-> n-> seql-> this-> primeiro;
4734
return d_eval_sub (e-> n-> seql-> this-> primeiro, pno, nst);
4735
} else if (eval (e-> n))
4736
{
4737
return e-> nxt;
4738
}
4739
if (e && (nst == e-> seqno))
4740
return e;
4741
if (s && (nst == s-> seqno))
4742
return s;
4743
printf ("etapa% d: trilha perdida", profundidade);
4744
if (Have_claim)
4745
{
int k = 0;
4746
if (pno == 1)
4747
printf ("(");
4748
outro
4749
{
se (pno> 1) k = 1;
4750
printf ("%d", pno-k);
4751
}

```

```

DE PROTOCOLOS DE COMPUTADOR
FONTE DO VALIDADOR DA VERSÃO 0 SPIN
527
4752
} outro
4753
printf ("(proc% d", pno);
4754
qualproc (pno);
4755
printf ("estado.% d) [preso em% d] \ n", nst, (e)? e-> seqno: -1);
4756
lost_trail ();
4757
embrulhar();
4758
saída (1);
4759}

```

PROTOCOLO F DE TRANSFERÊNCIA DE ARQUIVOS PROMELA

Aqui está uma lista completa do conjunto de modelos de validação de protocolo de transferência de arquivos que foram desenvolvidos no Capítulo 7, com as modificações discutidas no Capítulo 14. É um erro para recuperar menos parâmetros em uma entrada de mensagem de um canal do que definido em a declaração de canal correspondente. Os campos de parâmetros não utilizados são, portanto, definidos para zero em envia e recebe.

```

1 / *
2 * Modelo de Validação PROMELA - script de inicialização
3 * /
4
5 #include "define"
6 #incluir "usuário"
7 #include "presente"
8 #incluir "sessão"
9 #include "fserver"
10 #include "flow_cl"
11 #include "datalink"
12
13 init
14 {atomic {
15
execute userprc (0); execute userprc (1);
16
executar presente (0); executar presente (1);
17
sessão de execução (0); sessão de execução (1);
18
execute fserver (0); execute fserver (1);
19
execute fc (0);
execute fc (1);
20
execute data_link ()
21
}
22}
23
24 / *
25 * Definições Globais
26 * /
27
28 # define PERDA
0
/* perda de mensagem */
29 #define DUPS
0
/* msgs duplicadas */
30 # define QSZ

```

```
2
/* tamanho da fila
 */
31
528
```

Página 540

DE PROTOCOLOS DE COMPUTADOR
PROTÓCOLO DE TRANSFERÊNCIA DE ARQUIVOS PROMELA

```
529
32 mtype = {
33
vermelho branco azul,
34
abortar, aceitar, ack, sync_ack, fechar, conectar,
35
criar, dados, eof, abrir, rejeitar, sincronizar, transferir,
36
FATAL, NON_FATAL, COMPLETE
37
}
38
39 chan use_to_pres [2] = [QSZ] de {byte};
40 chan pres_to_use [2] = [QSZ] de {byte};
41 chan pres_to_ses [2] = [QSZ] de {byte};
42 chan ses_to_pres [2] = [QSZ] de {byte, byte};
43 canais_para_fluxo [2] = [QSZ] de {byte, byte};
44 canal flow_to_ses [2] = [QSZ] de {byte, byte};
45 canais dll_to_flow [2] = [QSZ] de {byte, byte};
46 canais flow_to_dll [2] = [QSZ] de {byte, byte};
47 chan ses_to_fsrv [2] = [0] de {byte};
48 chan fsrv_to_ses [2] = [0] de {byte};
49
50 / *
51 * Modelo de validação de camada de usuário
52 */
53
54 proctype userprc (bit n)
55 {
56
use_to_pres [n]! transferência;
57
E se
58
:: pres_to_use [n]? aceitar -> ir para Concluído
59
:: pres_to_use [n]? rejeitar -> ir para Concluído
60
:: use_to_pres [n]! abortar -> goto abortado
61
fi;
62 Abortado:
63
E se
64
:: pres_to_use [n]? aceitar -> ir para Concluído
65
:: pres_to_use [n]? rejeitar -> ir para Concluído
66
fi;
67 Feito:
68
pular
69}
70
71 / *
72 * Modelo de validação da camada de apresentação
73 */
74
75 proctipo presente (bit n)
76 {status do byte, uabort;
77
78 endIDLE:
79
Faz
80
:: use_to_pres [n]? transferir ->
81
uabort = 0;
82
```

```
quebrar
83
:: use_to_pres [n]? abortar ->
84
pular
85
od;
```

```
530
APÊNDICE F
PROJETO E VALIDAÇÃO
86
87 TRANSFERÊNCIA:
88
pres_to_ses [n]! transferência;
89
Faz
90
91 :: use_to_pres [n]? abortar ->
92
E se
93 :: (! uabort) ->
94
uabort = 1;
95
pres_to_ses [n]! abortar
96
97 :: (uabort) ->
afirmar (1 + 1! = 2)
98
fi
99
99 :: ses_to_pres [n]? aceitar, 0 ->
vá para FEITO
100
100 :: ses_to_pres [n]? rejeitar (status) ->
101
E se
102
102 :: (status == FATAL || uabort) ->
103
ir para FALHA
104
104 :: (status == NON_FATAL && ! uabort) ->
105 progresso:
ir para TRANSFERÊNCIA
106
106 fi
107
od;
108 FEITO:
109
109 pres_to_use [n]! aceitar;
110
goto endIDLE;
111 FALHA:
112
112 pres_to_use [n]! rejeitar;
113
ir para endIDLE
114}
115
115
116 / *
117 * Modelo de validação de camada de sessão
118 */
119
Sessão 120 proctype (bit n)
120 {alternar bit;
122
tipo de byte, status;
123
124 endIDLE:
125
Faz
126
126 :: pres_to_ses [n]? digite ->
```

```

127
E se
128
:: (tipo == transferência) ->
129
ir para DATA_OUT
130
:: (digite! = transferir) /* ignore */
131
fi
132
:: flow_to_ses [n]? tipo, 0 ->
133
E se
134
:: (digite == conectar) ->
135
ir para DATA_IN
136
:: (digite! = conectar)
/* ignore */
137
fi
138
od;
139

```

Página 542

DE PROTOCOLOS DE COMPUTADOR
PROTÓCOLO DE TRANSFERÊNCIA DE ARQUIVOS PROMELA

```

531
140 DATA_IN:
/* 1. prepare o arquivo local fsrver */
141
ses_to_fsrv [n]! criar;
142
Faz
143
:: fsrv_to_ses [n]? rejeitar ->
144
ses_to_flow [n]! rejeitar, 0;
145
ir para endIDLE
146
:: fsrv_to_ses [n]? aceitar ->
147
ses_to_flow [n]! aceitar, 0;
148
quebrar
149
od;
150
/* 2. Receba os dados, até o final de */
151
Faz
152
:: flow_to_ses [n]? dados, 0 ->
153
ses_to_fsrv [n]! dados
154
:: flow_to_ses [n]? eof, 0 ->
155
ses_to_fsrv [n]! eof;
156
quebrar
157
:: pres_to_ses [n]? transferir ->
158
ses_to_pres [n]! rejeitar (NON_FATAL)
159
:: flow_to_ses [n]? fechar, 0 ->
/* usuário remoto abortado */
160
ses_to_fsrv [n]! close;
161
quebrar
162
:: tempo limite ->
/* foi desconectado */
163

```

```

ses_to_fsrv [n]! close;
164
ir para endIDLE
165
od;
166
/* 3. Feche a conexão */
167
ses_to_flow [n]! fechar, 0;
168
goto endIDLE;
169
170 DATA_OUT:
/* 1. prepare o arquivo local fsrver */
171
ses_to_fsrv [n]! open;
172
E se
173
:: fsrv_to_ses [n]? rejeitar ->
174
ses_to_pres [n]! rejeitar (FATAL);
175
ir para endIDLE
176
:: fsrv_to_ses [n]? aceitar ->
177
pular
178
fi;
179
/* 2. inicializar o controle de fluxo */
180
ses_to_flow [n]! sincronizar, alternar;
181
Faz
182
:: atomic {
183
flow_to_ses [n]? sync_ack, digite ->
184
E se
185
:: (tipo! = alternar)
186
:: (tipo == alternar) -> pausa
187
fi
188
}
189
:: tempo limite ->
190
ses_to_fsrv [n]! close;
191
ses_to_pres [n]! rejeitar (FATAL);
192
ir para endIDLE
193
od;

```

Página 543

```

532
APÊNDICE F
PROJETO E VALIDAÇÃO
194
toggle = 1 - alternar;
195
/* 3. preparar arquivo remoto fsrver */
196
ses_to_flow [n]! conectar, 0;
197
E se
198
:: flow_to_ses [n]? rejeitar, 0 ->
199
ses_to_fsrv [n]! close;
200
ses_to_pres [n]! rejeitar (FATAL);
201

```

```

ir para endIDLE
202
:: flow_to_ses [n]? conectar, 0 ->
203
ses_to_fsrv [n]! close;
204
ses_to_pres [n]! rejeitar (NON_FATAL);
205
ir para endIDLE
206
:: flow_to_ses [n]? aceitar, 0 ->
207
pular
208
:: tempo limite ->
209
ses_to_fsrv [n]! close;
210
ses_to_pres [n]! rejeitar (FATAL);
211
ir para endIDLE
212
fi;
213
/* 4. Transmita os dados, até o final de */
214
Faz
215
:: fsrv_to_ses [n]? dados ->
216
ses_to_flow [n]! dados, 0
217
:: fsrv_to_ses [n]? eof ->
218
ses_to_flow [n]! eof, 0;
219
status = COMPLETO;
220
quebrar
221
:: pres_to_ses [n]? abortar ->
/* usuário local abortado */
222
ses_to_fsrv [n]! close;
223
ses_to_flow [n]! fechar, 0;
224
status = FATAL;
225
quebrar
226
od;
227
/* 5. Feche a conexão */
228
Faz
229
:: pres_to_ses [n]? abortar
/* ignore */
230
:: flow_to_ses [n]? fechar, 0 ->
231
E se
232
:: (status == COMPLETO) ->
233
ses_to_pres [n]! aceitar, 0
234
:: (status != COMPLETO) ->
235
ses_to_pres [n]! rejeitar (status)
236
fi;
237
quebrar
238
:: tempo limite ->
239
ses_to_pres [n]! rejeitar (FATAL);
240
quebrar
241

```

```

od;
242
ir para endIDLE
243}
244
245 / *
246 * Modelo de validação de servidor de arquivos
247 */

```

DE PROTOCOLOS DE COMPUTADOR
PROTÓCOLO DE TRANSFERÊNCIA DE ARQUIVOS PROMELA

```

533
248
249 proctype fserver (bit n)
250 {
Fim 251:
Faz
252
:: ses_to_fsrv [n]? criar ->
/* entrando */
253
E se
254
:: fsrv_to_ses [n]! rejeitar
255
:: fsrv_to_ses [n]! aceitar ->
256
Faz
257
:: ses_to_fsrv [n]? dados
258
:: ses_to_fsrv [n]? eof -> pausa
259
:: ses_to_fsrv [n]? fechar -> quebrar
260
od
261
fi
262
:: ses_to_fsrv [n]? abrir ->
/* extrovertido */
263
E se
264
:: fsrv_to_ses [n]! rejeitar
265
:: fsrv_to_ses [n]! aceitar ->
266
Faz
267
:: fsrv_to_ses [n]! dados -> progresso: pular
268
:: ses_to_fsrv [n]? fechar -> quebrar
269
:: fsrv_to_ses [n]! eof -> pausa
270
od
271
fi
272
od
273}
274
275 / *
276 * Modelo de validação de camada de controle de fluxo
277 */
278
279 #define true
1
280 #define false
0
281
282 #define M 4
/* números de sequência de intervalo */
283 # define W 2
/* tamanho da janela: M / 2
*/
284
285 proctype fc (bit n)

```

```

286 {bool
ocupado [M];
/* mensagens pendentes
*/
287
byte
q;
/* seq # msg mais antiga não confirmada */
288
byte
m;
/* seq # última mensagem recebida */
289
byte
s;
/* seq # próxima mensagem a enviar */
290
byte
byte
janela;
/* nº de mensagens pendentes */
291
byte
tipo;
/* tipo de mensagem
*/
292
mordeu
recebeu [M];
/* manutenção do receptor */
293
mordeu
x;
/* variável de rascunho
*/
294
byte
p;
/* seq # da última mensagem confirmada */
295
byte
I_buf [M], O_buf [M];
/* buffers de mensagem */
296
297
/* parte do remetente */
Fim 298:
Faz
299
:: atomic {
300
(janela <W && len (ses_to_flow [n])> 0
301
&& len (flow_to_dll [n]) <QSZ) ->

```

534
APÊNDICE F
PROJETO E VALIDAÇÃO
302
ses_to_flow [n]? tipo, x;
303
janela = janela + 1;
304
ocupado [s] = verdadeiro;
305
O_buf [s] = tipo;
306
flow_to_dll [n]! tipo, s;
307
E se
308
:: (digite! = sincronizar) ->
309
s = (s + 1)% M
310
:: (tipo == sync) ->
311
janela = 0;
312
s = M;

```

313
Faz
314
:: (s> 0) ->
315
s = s-1;
316
ocupado [s] = falso
317
:: (s == 0) ->
318
quebrar
319
od
320
fi
321
}
322
:: atomic {
323
(janela> 0 && ocupado [q] == false) ->
324
janela = janela - 1;
325
q = (q + 1)% M
326
}
327 #if DUPS
328
:: atomic {
329
(len (flow_to_dll [n]) <QSZ
330
&& window> 0 && busy [q] == true) ->
331
flow_to_dll [n]! O_buf [q], q
332
}
333 #endif
334
:: atomic {
335
(tempo limite && len (flow_to_dll [n]) <QSZ
336
&& window> 0 && busy [q] == true) ->
337
flow_to_dll [n]! O_buf [q], q
338
}
339
340
/* parte do receptor */
341 #if PERDA
342
:: dll_to_flow [n]? type, m / * perde qualquer mensagem */
343 #endif
344
:: dll_to_flow [n]? digite, m ->
345
E se
346
:: atomic {
347
(tipo == ack) ->
348
ocupado [m] = falso
349
}
350
:: atomic {
351
(tipo == sincronizaçao) ->
352
flow_to_dll [n]! sync_ack, m;
353
m = 0;
354
Faz
355
:: (m <M) ->

```

DE PROTOCOLOS DE COMPUTADOR
PROTÓCOLO DE TRANSFERÊNCIA DE ARQUIVOS PROMELA

```
535
356
recebido [m] = 0;
357
m = m + 1
358
:: (m == M) ->
359
quebrar
360
od
361
}
362
:: (tipo == sync_ack) ->
363
flow_to_ses [n]! sync_ack, m
364
:: (digite! = ack && type! = sync && type! = sync_ack) ->
365
E se
366
:: atomic {
367
(recebido [m] == verdadeiro) ->
368
x = ((0 <pm && pm <= W)
369
|| (0 <p-m + M && p-m + M <= W));
370
E se
371
:: (x) -> flow_to_dll [n]! ack, m
372
:: (! x) /* else ignorar */
373
fi
374
:: atomic {
375
(recebido [m] == falso) ->
376
I_buf [m] = tipo;
377
recebido [m] = verdadeiro;
378
recebido [(m-W + M)% M] = falso
379
}
380
fi
381
fi
382
:: (recebido [p] == verdadeiro && len (flow_to_ses [n]) <QSZ
383
&& len (flow_to_dll [n]) <QSZ) ->
384
flow_to_ses [n]! I_buf [p], 0;
385
flow_to_dll [n]! ack, p;
386
p = (p + 1)% M
387
od
388}
389
390 /* *
391 * Modelo de validação de camada de link de dados
392 */
393
394 proctype data_link ()
395 {tipo de byte, seq;
396
Fim 397:
Faz
```

```

398
:: flow_to_dll [0]? tipo, seq ->
399
E se
400
:: dll_to_flow [1]! tipo, seq
401
:: pular / * perder mensagem * /
402
fi
403
:: flow_to_dll [1]? tipo, seq ->
404
E se
405
:: dll_to_flow [0]! tipo, seq
406
:: pular / * perder mensagem * /
407
fi
408
od
409}

```

Página 547

536
APÉNDICE F
PROJETO E VALIDAÇÃO

Página 548

NOME INDEX

Ackermann
106
Adi, W.
65, 352
Ésquilo
1
Agamenon
1
Agerwala, TKM
186, 352
Aggarwal, S.
241-242, 352
Aho, AV
164, 185, 202,
240-241, 296, 352
Alderden, R.
354
Alexandre o grande
3
Ansart, JP
241, 363
Apt, KR
241, 352
Arthurs, E.
161, 352
Balkovic, MD
65, 353
Bardeen, J.
10
Bartlett, KA
12, 17, 41, 74,
88, 186, 353, 364
Baudot, JME
9, 370
Beeforth, TH
88, 353
Bell, AG
9
Bennet, WR
17, 65, 353, 379
Bentley, J.
xi, 41, 64
Berlekamp, ER
65, 353
Bertsekas, D.
40, 87, 353, 379
Bochmann, G. von
18, 186,
212, 240, 242, 353, 357, 361
Bolognesi, T.
18, 353
Bond, DJ
65, 353
Bose, RC
56, 353
Bourguet, A.
241, 353

Brand, D.
186, 212, 240, 242,
353, 366
Brattain, WH
10
Bredt, TH
110, 353
Brilhante, MB
65, 353
Brinksma, E.
18, 353-354
Brown, GM
xi, 41, 88, 240,
354
Browne, MC
127, 242, 354
Bruyn, NG de
110
Budkowski, S.
18, 354
Campbell-Kelly, M.
17, 354
Cardelli, L.
296, 354
Cargill, TA
XI
Cerf, VG
88, 354
Chappe, Claude
3, 16, 354
Chappe, Ignace
16, 354
Chaves, JA
xi, 41
Chesson, G.
xi, 41, 88, 161,
352, 354
Chiu, DM
88, 359
Choi, TY
354
Chow, T.
202, 354
Chu, PM
212, 241, 354, 360
Clark, D.
89, 355
Clarke, EM
127, 242, 354-
355, 357, 363, 365
Cole, R.
89, 355
Cooke, WF
5, 17, 355
Courcoubetis, C.
xi, 241, 352,
355
Cowan, DD
366
Cunha, PRF
241, 355
Dahbura, AT
xi, 201-202,
352, 355, 364-365
Dahl, OJ
355
Davey, JR
17, 65, 353, 379
deBruyn, NG
354
Decina, M.
65, 355
Dekker, T.
95, 126, 320
Dembinski, P.
18, 354
Diaz, M.
18, 356
Dijkstra, EW
41, 103, 109,
115, 161, 202, 355
Dill, DL
127, 242, 354
Duke, R.
240, 356
Edelcrantz, AN
4, 17, 356
Edmonds, J.
201, 356
Eijk, P. van
18, 41, 316, 356
Eisenberg, MA
110, 356
Emerson, EA
316, 355
Estrin, G.
240, 363
Euclides
102, 178
Euler
191

Faraday, M.
5
Fibonacci, L.
108, 295
Field, JA
161, 356
Finkel, A.
186, 356
Fleming, HC
65, 356
Fletcher, JG
63, 65, 356, 362
Floyd, RW
240, 356
Fourier, JBJ de
371
Fraser, AG
xi, 41, 356
Friedman, AD
201, 356
Friedman, HG Jr.
296, 364
Galileo
3
Gallager, R.
40, 87, 353, 379
Garey, MG
127, 356
Gerla, M.
88, 357
Gibbons, A.
202, 357
Gill, A.
357
Gobershtein, SM
201, 357
Godefroid, P.
241, 357
Gotzheim, R.
212, 357
Gouda, MG
xi, 41, 88, 212,
217, 240-241, 354, 357
Goudsblom, J.
XI
Gray, E.
9
Griffiths, G.
65, 357
Grimsdale, RL
353
Haahr, P.
xi, 109
Hajek, J.
240, 357
Halsall, F.
353
Hamming, R.
47, 65, 357
Han, JY
241, 357
HarEl, Zri
242, 357
Harbison, SP
296, 357
Harrison, MA
186, 357
Hartley, RVL
357
Hartmanis, J.
186, 357
Hayes, I.
240, 356, 358
Hennie, FC
201, 358
Henry, J.
5
Herbarth, D.
358
Herbert, AJ
41, 362
Hoare, CAR
109, 355, 358
Hocquenghem, A.
56, 358
Hodge, FW
17, 358
Holzmann, GJ
110, 186,
241, 317, 358
Hooke, R.
3, 17
Hopcroft, JE
185, 240-241,
352
Hubbard, G.
17, 40, 359
 Huffman, DA
185, 359
Hume, A.
XI

Hutchinson, D.
200, 366
Hutchinson, RM Jr.
65, 356
Huth, G.
3
Hyman, H.
359
Jacobson, V.
89, 359
Jain, R.
88-89, 359
Jard, C.
241, 362
Johnson, DS
127, 356
Johnson, EL
201, 356
Joyner, WH Jr.
240, 353
Julio, U. de
65, 355
Kahn, RE
88, 354
Kain, RY
185, 359
Kanellakis, PC
359
Karn, P.
88, 359
Kendall, DG
72, 359
Kernighan, BW
xi, 41, 109,
296, 360, 362, 390
537

Página 549

538 ÍNDICE DE NOME

King, P.
240, 356
Klancer, HW
353
Klare, SW
353
Klee, V.
201, 360
Kleinrock, L.
88, 357
Knudsen, HK
242, 360
Knuth, DE
110, 217, 240,
360
Kohavi, Z.
201, 360
Kozen, DZ
241, 352
Krogdahl, S.
217, 240, 360
Kruijer, HSM
41, 360
Kuan, MK.
202, 360
Kuo, FF
65, 360
Kurshan, RP
241-242, 352,
357, 363, 365
Lagemaat, J. van de
354
Lam, SS
186, 236, 241, 360
Lambert, M.
89, 355
Lamport, L.
41, 110, 127, 360,
362
Langerak, R.
354
Lee, D.
xi, 202, 352, 366
Lesk, M.
XI
Leung, T.
202, 365
Lin, FJ
241, 360
Lin, S.
186, 360
Lindgren, B.
42, 360
Linn, RJ
202, 360
Lint, van JH
49, 65, 360
Liu, MT

212, 241, 354, 360
Logrippo, L.
41, 365
Lombardi, L.
364
Lynch, WC
22, 40, 88, 106,
360
MacWilliams, FJ
65, 361
Maibaum, TSE
241, 355
Mallery, J.
17, 361
Malmgren, E.
17, 361
Mandelbrot, B.
45, 65, 130,
361
Manna, Z.
127, 361
Marconi, G.
9
Marland, EA
17, 40, 361
Marshall, WT
41, 356
Mase, A.
17, 364
Maxemchuck, NF
241, 361
McCarthy, J.
186, 364
McCoy, WH
202, 360
McCulloch, WS
185, 361
McGruther, WG
353
McGuire, MR
110, 356
McIlroy, MD
xi, 160, 241,
361
McKee, S.
XI
McLeod S.
365
McNamara, JE
361, 379
McQuillan, JM
89, 361
Mealy, GH
163, 185, 361
Menon, PR
356
Merlin, PM
40, 212, 361
Michaelis, AR
17, 361
Michel, T.
365
Miller, RE
88, 240, 354
Milner, R.
14, 18, 361
Mishra, B.
127, 242, 354
Mitchell, DP
xi, 60
Mockapetris, P.
365
Moitra, A.
361
Moore, EF
163, 185, 201,
362
Morris, R.
241, 362
Morse, S.
5, 370
Mowbray, M.
240, 358
Multari, N.
41, 362
Murray, G.
4, 17
Naito, S.
362
Nakassis, A.
65, 362
Needham, RM
41, 362
Nightingale, JS
200, 362
Nock, OS
9, 17, 362
Nyquist, H.
362, 376
Oestreicher, AB
XI

Owicki, S.
127, 362
Pageot, JM
241, 362
Partridge, C.
88, 359
Patti, J.
358
Pehrson, B.
XI
Perez, A.
65, 362
Peterson, J.
XI
Peterson, WW
65, 362
Petri, CA
162, 181, 186, 241,
362
Pike, R.
xi, 41, 296, 354, 359,
362
Pitts, W.
185, 361
Pnueli, A.
127, 361-362
Polybius
1, 9, 16, 38-39
Porizek, R.
40, 363
Pouzin, L.
17, 40, 88, 362
Prescott, GB
17, 363
Presotto, D.
XI
Preço, WL
202, 363
Prineth, R.
363
Probert, RL
202, 363, 365
Probst, DK
241, 363
Purusothaman, S.
xi, 185
Puzman, J.
40, 363
Queille, JP
127, 363
Rado, T.
165, 186, 360, 363
Rafiq, O.
241, 363
Ramakrishnan, KK
88, 359
Ramsey, N.
XI
Ray-Chaudhuri, DK
56, 353
Rayner, D.
200, 363
Razouk, BB
240, 363
Reed, R.
18, 364
Reeds, J.
XI
Reid, JD
17, 363
Reif, JH
241, 363
Rhodes, NW
60
Richier, JL
241, 363
Ritchie, DM
41, 109, 296,
360, 390
Ritchie, GR
65, 363
Rockstrom, A.
18, 41, 364,
380
Rodriguez, C.
363
Rolt, LTC
17, 364
Rose, GA
240, 356, 358
Rosier, L.
186, 356
Rubin, J.
240-241, 364
Rudie, K.
212, 364
Rudin, H.
366
Sabnani, KK
xi, 201-202,
241, 355, 361, 364

Saracco, R.
18, 41, 364, 380
Sarikaya, B.
201, 364
Scantlebury, RA
12, 17, 41,
74, 88, 186, 353, 364
Scheffler, PE
65, 363
Schneider, A.
17, 364
Schreiner, AT
296, 364
Schwabe, D.
240, 364
Sethi, R.
xi, 164, 185, 296, 352
Shankar, AU
186, 236, 241,
360
Shannon, C.
51, 65, 186, 364,
370, 377
Sharma, D.
242, 352
Shaw, RB
17, 364
Shen, YN.
364
Shih, T.
202, 364
Shockley, W.
10
Sidhu, D.
202, 364
Sifakis, J.
363
Sistla, AP
355
Slepian, D.
65, 365
Sloane, NJA
65, 361
Smallberg, DA
240, 365
Smith, JRW
18, 364
Smolka, SA
241, 359, 363
Snepscheut, JLA van den
127, 365
Stallings, W.
42, 65, 88-89,
365, 379
Stearns, RE
186, 357
Steele, GL Jr.
296, 357
Stenning, NV
88, 365
Ainda um.
17, 365
Stones, GC
65, 357
Preso, BW
161, 352
Sullivan, M.
XI
Sunshine, CA
186, 240, 353,
365
Tanenbaum, AS
42, 65, 88-
89, 161, 365, 379
Tarjan, RE
202, 365
Thompson, K.
XI
Tilanus, PAJ
18, 364
Tretmans, J.
354
Trickey, H.
XI
Tsunoyama, M.
362
Tugal, D. e O.
65, 365, 379
Tukey, JW
370
Turing, AM
165, 185, 365
Ullman, JD
164, 185, 240-
241, 296, 352
Ural, H.
202, 363, 365
Uyar, MU
202, 352, 365
Vail, T.

Valmari, A.
241, 365
Vardi, M.
241, 355
Vasilevskii, MP
202, 365
Vissers, CA
18, 41, 356,
365-366
Voiron, J.,
363
Walden, DC
89, 361
Wang, B.
200, 366
Weldon, EJ Jr.
65, 362
West, CH
xi, 226, 240-241,
364, 366
Wheatstone, C.
5, 17
Wilkinson, PT
41, 74, 88,
186, 353
Wirth, N.
41, 366
Wolper, P.
127, 241, 352, 355,
361, 366
Wonham, WM
212, 364
Woolons, DJ
353
Wu, C.
41, 354
Yannakakis, M.
xi, 202, 241,
355, 366
Yu, YT
241, 357
Zafropulo, P.
186, 212, 240,
242, 353, 366
Zhang, L.
88-89, 355, 366
Zimmerman, H.
17, 40, 363
17, 40, 363

Página 550

ÍNDICE DE ASSUNTO

->
93
abstração
20
rótulos de estado de aceitação
118
Função de Ackermann
106
reconhecimento
70, 87, 216
sequência adaptativa
194
endereçamento
89
protocolo de bit alternado
41, 74,
87, 205
Guerra Revolucionária Americana
3
sinais analógicos
368
poder analítico
184
ANSI
13, 59
sinal aperiódico
372
Argos
109
soma de verificação aritmética
63
ARPA
17
rede
11
ARQ
controle de fluxo
81
protocolo
217
matriz
92, 259
seta
93

Código ASCII
13, 33, 130
afirmar declaração
114, 254,
266-267, 321
afirmação
114, 203
tarefa
385
associatividade de operadores
384
assíncrono
comunicação
97
transmissão
368
AT&T
13
atômico
seqüência
96, 106, 389
declaração
254, 275, 279, 288
atenuação
51
validação de protocolo automatizada
218
intervalo de média
86
ciclos ruins
118
largura de banda
51, 130, 370, 377
limitação
43
Telefone
373
utilização
64
frequência base
373
cesta de telégrafo
3, 17
Baud
370
Códigos BCH
56, 65
definição de comportamento
112
Bell Labs
10, 41
Bentley J.
16
passeio aleatório tendencioso 219
binário
comunicação
100
dígito
370
canal simétrico 44, 50
Protocolo bsync
10, 33, 59
mordeu
370
tipo de dados
92
taxa de erro 43-44, 50, 129, 378
taxa de erro
44
campo
139
espaço de estado 238, 297, 305, 311,
315
espaço estadual
226
estofamento
33
protocolo orientado a bits
32
Diretiva do compilador BITSTATE
436
quadra
reconhecimento
82, 88
código
47
tipo de dados bool
92
fronteira
37, 237
bps
370
pesquisa ampla
221
declaração de quebra
101, 105, 254,
275, 279-280
difundir
129
Tipo de bolha
108

estourar
erro
55, 58, 130, 137-138
taxa
86
problema de castor ocupado
165
limites de byte
139
protocolo de contagem de bytes
34, 133
ligar
estabelecimento
132
terminação
132
Cambridge Ring
41
produto cartesiano
175
caso
seleção
388
seleção
100
CCITT
13, 17, 30, 41, 200, 354
CCS
14
CD
56
centros de certificação
200
declaração chan
96
canal
96, 387
suposições
21, 130
largura de banda
84, 372
capacidade
51
declaração
96
taxa de erro 45-47, 51
saturação
83, 134
Tamanho
97
recheio de personagem
34, 40, 133
protocolo orientado a caracteres
33
caracterizando seqüência
197
verificar bits
47, 52-53
soma de verificação
34, 47, 56, 138
colocação
134
polinomial
56
residual
57
Problema do carteiro chinês 193
Túnel Clayton
7, 16
agrupamento
45
coaxial
cabo
378
cabo
43
código
eficiência
60
intercalar
55
mapeamento
49
a sobrecarga
48
taxa
47-48
taxa
49, 51-52
palavra
47, 52
cara ou coroa
50
combinando máquinas
175
comentários SPIN
383
comunicando estado finito
máquina
166
discos compactos

56
máquina completamente especificada
190
completude
37
suposição
190
complexidade
111, 241
limites
111
gestão
219, 222, 236
composto
sinal
372
estado do sistema 220
declaração composta
273
simultaneidade
95
doença
91
execução condicional
163
conformidade
539

Página 551

540 ÍNDICE DE ASSUNTO
PROJETO E VALIDAÇÃO
testando
174, 187
algoritmo de teste 190
congestionamento
66
evasão
83
protocolo de gerenciamento de conexão
203
constantes
384
contínuo
ARQ
134
sinal
372
ao controle
código
40
código
3-4, 6, 16, 20
campo
23, 35
fluxo
100
mensagem
20
pesquisa parcial controlada 219,
222
código de convolução
47
fio de cobre
367
correção
critério
111-112
requisitos
220
máquina de estado finito de acoplamento
166
CRC
soma de verificação
48
código
56
tabela de pesquisa
60, 65
Código CRC-12
58
CRC-16
58
CRC-32
59
CRC-CCITT
58
soma de verificação
138
criando processos
94
crédito
76
controle de fluxo
70
crítico
seção
109

seção
95, 104
conversa cruzada
43, 130, 376
reconhecimento cumulativo
82
frequência de corte
373
cíclico
código
47, 54
verificação de redundância 48
verificação de redundância
138
verificação de redundância
56
dados
envelope
27, 31
camada de ligação
29
taxa
378
taxa
86
tipos
92, 386
datagrama
76
Datakit
41
Interface DCE
30
Protocolo DDCMP
34
impasse
22, 38, 69, 111, 117,
182, 203, 209-211, 220, 224,
241, 289, 297, 302, 311, 323,
326
DEZ
34
decibel
376
decidibilidade
111, 241
declarações
386
Algoritmo de Dekker
95, 320
erro de deleção
138
demodulador
368
pesquisa em profundidade
221
clareza descritiva
184
Projeto
por comitê
13
falha
26
problema
7, 13
requerimento
15
regras
38
determinismo
164
canal do dispositivo
295
Semáforo dijkstra
103, 109
canal discreto sem memória
44
sequência distintiva
194,
201
erro de distorção
138
divisão de polinômios
57
Código de controle DLE
33-34
fazer declaração
389
códigos de correção de erro duplo
56
Interface DTE
30
duplex
canal
368
transmissão
68
mensagem duplicada
76
duplicação
erro

74
erros
46
controle de fluxo dinâmico
84
eco
43
distorção
376
pesos de borda
224
telégrafo elétrico
5
onda eletromagnética
370
estado final
164
rótulos de estado final
117
de ponta a ponta
30, 83, 129
Computador ENIAC
10
entropia
51
máquina de estado finito de equivalência
174
erro
características
44, 64
código
47
ao controle
138, 325
ao controle
43
correção
48
correção
53
distribuição
49
padronizar
58
probabilidades
43
Estado
220
trilhas
314
código de correção de erros
46
código de detecção de erros
46
intervalo livre de erros
45
Estelle
14, 17
ETX
código de controle 40, 133
código de controle
33
Algoritmo de Euclides
102, 178
Euler tour
191
avaliando expressões
245
executabilidade
91, 97, 100, 114
executável
163
execução
ciclo
113
máquina de estados finitos
170
seqüência
113
exhaustivo
pesquisa
219
limites de pesquisa 221
validação
297
expansão
236
distribuição exponencial
72
expressões
384
máquina de estado finito estendida
176
fatorial
programa
244, 310
programa
104
progresso justo
224
cobertura de falhas

200, 202
Soma de verificação FDDI
59
FDT
14, 18, 366
controle de erro de feedback
46
fibra ótica
43
Fibonacci
número
108
sequência
295
FIFO net
162, 181, 183, 186
finito
máquina estadual 210
variável
176
máquina de estados finitos
162
sinal de fogo
1
fluxo
gráficos
380
validação de controle 325
teste de formato
188
controle de erro de encaminhamento
46, 48
Séries de Fourier
371
enquadramento
29-30
erro
369
FTSC
13
cheio
transmissão duplex
204
requisitos de memória de pesquisa
222
espaço de estado 305, 310, 322
pesquisa no espaço estadual 219
full-duplex
130, 368
teste funcional
188
generalização
112, 178, 236
polinômio gerador
57
variável global
91, 93, 386
voltar-N ARQ
81
vamos para
388
declaração
102, 105
gramática
245
literais
246
terminais
246
ícone
246
algoritmo de aumento de gráfico
193
guarda
100, 389
línguagem de comando protegida
109
guiado
pesquisa
224
simulação
298, 307-308
expressões orientadoras
224
pólvora
40
meio duplex
368
Hamming
código
53
distância
51
hardware CRC
65
freqüência harmônica
372
cerquilha
conflitos
228
fator

313, 315
função
227-228, 305
mesas
227, 259
hashing
226, 299, 305
Protocolo HDLC
30, 33
cabecalho
34, 161
Hertz
371

Página 552

DE PROTOCOLOS DE COMPUTADOR

ÍNDICE DE ASSUNTO 541

protocolo de alta velocidade
83
sequência de retorno
194, 201
salto a salto
30, 83
Hz
371
declarações i / o
389
IBM
10, 33, 56, 359
pesquisa
65
identificadores
383
declaração if
388
IFIP
13, 88, 359
impedância incompatível
376
implementação
158, 174
rescisão indevida
38
ruído de impulso
44-45
incluir arquivos
390
incremental
composição
236
Projeto
210
prova de indução
216
inevitabilidade
112
espaço de estado infinito 226
em formação
370
iniciar
processo
282
processo
93
inicializadores
386
entrada
seleção
168
sinais
163
erro de inserção
87
tipo de dados int
92
tempos entre chegadas
72
intermodulação
376
máquina de estado finito de interação
166
protocolo de interface
31
Rede de internet
11
interrupções
109
Protocolo IP
30
ISO
13-14, 28, 33, 200, 359
ITU
13
IUT
187
palavras-chave
383
rótulos

388
operador len
98, 269
lexical
análise
247, 255, 266
convenções
383
Jarra de Leyden
5
protocolo leve
37, 41,
161
motorista de linha
368
código linear
47
protocolo de nível de link
30
livelock
38, 118
variáveis locais
386
consistência lógica
214
verificação de redundância longitudinal
49
olhe adiante token 248
tabela de pesquisa
227, 257
erro de trilha perdida
310
Lotos
14, 17
Código LRC
49
Protocolo de lynch
22, 106
Fila M / M / 1
72
estado da máquina
220
macros
95, 390
makefile
253
Codificação Manchester
369
prova manual
214
gráfico marcado
182
marcação
182
máximo
probabilidade de decodificação 51
progresso
224
velocidade de sinalização 376-377
velocidade de sinalização
52
tamanho da janela
80
atraso médio de ida e volta 72
prova mecânica
297
megafone telégrafo
3
Diretiva do compilador MEMCNT
436
supertrace de requisitos de memória
pesquisa
229
mensagem
canal
368
delimitador
133
erro de duplicação 327
erro de duplicação
76
formato
134
perda
87, 123, 327
perda
72
parâmetro
97
passagem
96, 268
atraso de propagação 130
atraso de propagação
70
fila
167, 169
erro de reordenamento
76, 88
tipo
282
tipo

104
campo de tipo
134
problema de fluxo máximo de custo mínimo
193, 202
mínimo
Projeto
38
suite de teste 193
máquina de estado finito de minimização
171, 173
tamanho mínimo de janela 85
ambiente modelo
163
modelagem
poder
184
retransmissão
167
modem
29, 368
tours de transição modificados 196
modulação
374
modulador
368
soma módulo-2
47-48
Código Morse
9, 370
Protocolo MSP
41
multiponto
129
hashing múltiplo
230
Lei de Murphy
225
espaço de nomes
89
NBS
13, 200
vizinho mais próximo decodificando 51
telégrafo de agulha
5, 17
reconhecimento negativo
5,
20, 22-23, 80
rede
Projeto
36, 41, 89
problema de fluxo 193
NIST
13, 200
barulho
43, 370
espiões
130, 376
distorção não linear
43, 376
sinal não periódico
371
ciclos de não progresso
118
não determinismo
100, 164,
171, 388
definiram
164
Complexidade NP
201
NPL
88
Taxa de Nyquist
378
carga oferecida
85
número de sequência de um bit
135
operadores
384
óptico
fibra
370, 378
telégrafo
3-4, 17
tamanho de dados ideal 135
Modelo de referencia OSI
28, 30
mensagem fora de seqüência
74
superespecificação
37
erros de saturação
69
pacote
30
preenchimento
139
transmissão paralela
368

código de verificação de paridade 48
árvore de análise
246
analisador
260, 266, 282
análise de expressões
250
parcial
ordem
224
algoritmo de busca
223
heurística de pesquisa
223
objetivos de busca
223
buscas
316
pesquisa no espaço estatal 219
estados de partição
195
variáveis de peek
385
protocolo de pares
31
código perfeito
53, 65
sinal periódico
371
Petri
internet
186
internet
162, 181-182
restrições de rede
183
distorção de fase
375
mudança de fase
371
camada física
29
pid
124
Protocolo de ping-pong
69, 72, 74,
81
telégrafo de bola de fogo
5
sincronização de lugar
210
ponto a ponto
129
Distribuição de veneno
45, 72,
130
polinomial
complexidade
193
grau
56
reconhecimento positivo
22-23, 134
precedência
de operadores
384
regras
247
regras
261
pré-processador
390
declaração printf
244, 267
busca probabilística
224
procedimento
104
regras
21, 90-91, 107, 203
regras
140
processo
91, 387
álgebra
14
comportamento
162
corpo
93
eliminação
94
instanciação
93
número de instanciação 124
parâmetro
94
tipo
282
regras de produção
245

rótulos de estado de progresso
118
PROMELA
91
simulador
243

542 ÍNDICE DE ASSUNTO
PROJETO E VALIDAÇÃO

validador
297
prova por construção
203
protocolo
completude
90
conformidade
187
consistência
90
conversão
89
definição
20-21
derivação
203
eficiência
136
elementos
21, 90
Motores
88
formato
21
gramática
40
gramática
21
hierarquia
20, 37
implementação
111, 158
incompletude
25
interface
203
camadas
37
camadas
26, 325
modelagem
236
fases
132
serviço
21, 26, 203
esqueleto
203, 210
especificação
91
padrão
13
sintaxe
21
síntese
203
suite de teste 187
validação
214, 297
vocabulário
21, 170
vocabulário
21
protocolos introduzidos
12
pseudo-transições
196
Protocolo PSP
41
Complexidade PSPACE
111, 127,
196-197, 231, 241
estado de fila
220
condição de corrida
99
rádio telégrafo
9
estrada de ferro
7
semáforo
7
aleatória
algoritmo de busca 226
simulação
219, 226, 238, 241
intervalo número de sequência 80

taxa
ao controle
5, 135
ao controle
37-38, 85, 89
acessibilidade
análise
214, 218
algoritmo de análise 221
estado alcançável
214, 239
protocolo de leitor / escritor
211
receber
389
declaração
97
recursão
104
técnicas de redução
112,
178, 236
redundância
34, 37, 46
Códigos Reed-Solomon
56, 65
regular
expressão
210
língua
165
velocidades relativas
167
referenciamento remoto
385
encontro
99, 103
erros de reordenamento
46
repetição
101, 389
redefinir propriedade
190
residual
taxa de erro 56
taxa de erro
47-48, 50, 56
retransmissão
76
intervalo
72
cronômetro
88
robustez
38
atraso de ida e volta
64
roteamento
89
Interface RS232
13
operador de corrida
94, 124
problema do carteiro rural chinês
196
Rede SABRE
11
teorema de amostragem
376
escalares
259
pesquisa dispersa
224
Agendador
285
SDL
14, 17, 41, 364
Protocolo SDLC
33
pesquisa
cobertura
219, 221, 229-230,
315
qualidade
219, 221
métodos de redução
236
seletivo
repetir
217
repetir ARQ
81
protocolo de auto-adaptação
37
auto-estabilização
41, 369
protocolo de auto-estabilização
37
semáforo
100, 115
óptico

3
ponto e vírgula
93
enviar
389
declaração
97
sequência
campo numérico
134
números
215
números
34, 74
transmissão serial
368
serviço
ponto de acesso
210
ponto de acesso
31
primitivo
210
especificação
22, 26
validação de camada de sessão 337
definir propriedade
190
Limite de Shannon
52
Lei Shannon-Hartley
377
variável compartilhada
95
registrator de deslocamento
65
tipo de dados curto
92
o mais curto
distância
193
algoritmo de caminho
202
telégrafo de veneziana
4, 17
sinal
amplitude
371
atenuação
375, 377
distorção
368, 375
elementos
370
energia
376
frequência
371
poder
376
propagação
370
amostragem
376
estroboscópio
368
transmissão
368
a relação sinal-ruido
51, 130,
376
velocidade de sinalização
370
dados assinados e não assinados
92
sequência de I / O simples
201
simplex
canal
368
transmissão
68
curva senoidal
370
pular
388
declaração
94, 102, 105
deslizamento
protocolo de janela 74, 217
protocolo de janela
87
protocolo de início lento
85, 89
sinal de fumaça
17
abrangendo
arborescência
192
árvore
192

tubo de fala
3
GIRAR
avaliador
263, 267
macros
292
opções
293
visão global
244
corrida de amostra 244
afirmações
265
Estrutura SPIN
246
GIRAR
matriz de transição
302
validador
298
variáveis
255, 289
transição espontânea
167
tupla de estado estável 210
ordem de empilhamento
221
Estado
invariante
214
rótulos
113
números
333
propriedades
113
assinatura
201
explosão espacial 214
explosão espacial
222
diagrama de transição 74, 147,
164
declaração
388
separador
93
estados
113
controle de fluxo estático
84
propriedade de status
190
pare e espere
ARQ
81
protocolo
69
gráfico fortemente conectado 189
teste estrutural
189, 201
programação estruturada
41
método de estruturação
26, 31, 34
STX
código de controle 40, 133
código de controle
33
supertrace
implementação
311
pesquisa
226, 322
mesa de símbolos
257
simétrico
aumento
196
gráfico
191
sincronização
66, 83
máquina de estados finitos
166
síncrono
comunicação
99
acoplamento
169
transmissão
368
invariante do sistema
114-115, 216
código sistemático
47, 54
símbolo de fita
165
Protocolo TCP
88

Protocolo TCP / IP
30
tele-máquina de escrever
9

DE PROTOCOLOS DE COMPUTADOR

ÍNDICE DE ASSUNTO 543

código telegráfico
370
Telefone
3, 9, 43, 52, 55
telescópio
3
código telex
9, 12-13
Satélite telstar
10
temporal
reivindicações
114, 119
lógica
126-127
problema de teste e ajuste
95
ruído térmico
376
aperto de mão de três vias
161
diagrama de seqüência de tempo
25,
73, 76
tempo esgotado
72, 75, 87-88, 125,
135, 267
declaração
105
símbolo
383
atributos
248
tocha
código
16
telégrafo
2, 39
Protocolo TP4
63
reboque
34, 134
protocolo
161
colisão de trem
7, 17
transistor
10
transição
diagrama
182
matriz
306
probabilidade
224
regra
163, 170
tabela
162
Tour
191
algoritmo de passeio 192
transparência
27
Troy
1
TTCN
200
Máquina de Turing
165
tempo de resposta
72
torcido
par
367, 378
par
43
tipos de erros de transmissão
43
UIO
seqüência
194, 201
derivação de seqüência 195
seqüência única de entrada / saída
194-195
estado inacessível
220
código não alcançado
333
não especificado

recepção
210
recepção
37, 117
testador superior
326
Protocolo URP
41
uucp checksum
62
validação
núcleo
299
modelo
90
verificação de modelo 214
gerador validador
297, 304,
316
validador-gerador
303
transferência de valor
389
variável
386
declaração
92
verificação
Lotos
14
SDL
14
verificação de redundância vertical 49
canal virtual
27
Código VRC
49
Watt
376
protocolo bem formado
37
Wheatstone automático
9
ruído branco
51, 376
janela
protocolo
70
protocolo invariante 217
Tamanho
217
Tamanho
76, 83
cálculo do tamanho
134
Protocolo X.21
13, 30
Protocolo X.25
13, 30, 41
Protocolo XTP
85-86, 89
Linguagem Z
240