Teste de programas concorrentes com
memória compartilhada

Felipe Santos Sarmanho

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Teste de programas concorrentes com memória compartilhada

Felipe Santos Sarmanho

Orientador: *Prof. Dr. Paulo Sérgio Lopes de Souza*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC/USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional.

**USP – São Carlos
Fevereiro de 2009**

Aos meus pais, José Carlos e Gisélia, aos meus irmãos, à minha esposa Joanna e ao meu filho Pedro.

Agradecimentos

Em primeiro lugar a Deus, pelo dom da vida, e por me dar saúde, paz e todas as condições para conduzir este trabalho de mestrado.

Especialmente aos meus pais, o Sr. José Carlos e a Sra. Gisélia, pelo incentivo constante e apoio incondicional nessa caminhada. Agradeço especialmente também à Joanna, minha amada esposa, que me apoiou sempre incentivando e que foi muito paciente e compreensiva nos momentos em que precisei ficar ausente. E aos meus irmãos, Tatiana, Junior e Gabriel.

Ao professor Paulo Sérgio que com sabedoria, amizade, paciência e muita dedicação me orientou na concepção e desenvolvimento deste trabalho. À professora Simone e ao professor Adenilso que com o conhecimento na disciplina de Engenharia e Teste de software contribuíram diretamente para o desenvolvimento do trabalho. Suas sugestões e críticas foram de extrema importância para atingirmos o resultado esperado.

Ao grupo de pesquisa LaSDPC que regularmente prove discussões em torno dos trabalhos dos membros do grupo, inclusive o meu, buscando conduzir o aluno a atingir a excelência no desenvolvimento dos trabalhos. A todos os professores e funcionários, da Pós (ICMC/USP) e da graduação (UNICAP) que contribuíram com a minha formação.

Aos demais familiares, tios, primos, cunhados, que de alguma forma participaram dessa minha conquista. Aos amigos, Saulo, Marcelo, Newton, João Alves, Júlio, Thiago, Bruno, Jonathan, Maycon, André, Mário, e todos os outros que não citei (não há espaço).

Por fim, agradeço ao CNPq pelo apoio financeiro.

Resumo

Este trabalho propõe um modelo de teste para programas concorrentes que utilizam memória compartilhada. O modelo é inovador em três aspectos principais: (1) tratar a sincronização e a comunicação de *threads* de forma separada, (2) considerar a sincronização decorrente da inicialização/finalização de *threads*, e (3) apresenta um método baseado em *timestamps* para determinar as comunicações exercitadas em uma dada execução do programa. Os critérios de cobertura existentes para programas concorrentes foram adaptados ao contexto de programas baseados no paradigma de memória compartilhada. A ferramenta chamada ValiPThread foi implementada neste trabalho para apoiar a aplicação do modelo e dos critérios definidos. Com essa ferramenta é possível criar sessões de teste que podem ser salvas, interrompidas e retomadas a qualquer momento. Também é possível adicionar e executar casos de teste, avaliando a cobertura do código fonte em relação aos critérios de teste. A implementação da ferramenta mostra que é possível instanciar o modelo proposto em um software que auxilie a atividade de teste no contexto de programas com memória compartilhada. O trabalho apresenta soluções significativas para os principais desafios impostos pela programação concorrente para a atividade de teste, destacando-se dentre eles: (1) desenvolvimento de novas técnicas de análise estática para analisar programas concorrentes no contexto de memória compartilhada; (2) testar aspectos cruciais à programação concorrente como: sincronização, comunicação e fluxo de dados; (3) reproduzir uma execução de maneira controlada; (4) mapeamento de critérios de teste já existentes para programas concorrentes com passagem de mensagem para o contexto de memória compartilhada; (5) projetar critérios de fluxo de dados para programas concorrentes, considerando variáveis compartilhadas e (6) desenvolvimento de uma ferramenta de apoio a essas atividades.

Abstract

This work presents a novel test model for shared memory concurrent programs. Some important new features in this model are: (1) analysis the communication and synchronization in an isolated manner, (2) examines the synchronization due the start and the finish of threads, and (3) employs a method based on timestamps to check the communication exercised for an execution of the program. The coverage criteria set defined for concurrent programs was adjusted to the shared memory programs. In this work, the tool *ValiPThread* was implemented to support the application of the test model and of the coverage criteria. This tool allows to create a test session that can be saved, stopped and resumed at any time. In this tool is also possible to add and to execute test cases, analyzing the source code coverage with respect to the coverage criteria. The new tool shows that is possible instance the proposed model in a software that supports the test activity in context of shared memory. This work presents solutions for the major challenges related to task of to test concurrent programs, such as: (1) develop new techniques to do static analyzes of shared memory programs; (2) test important aspects to concurrent programs such as like: synchronization, communication and data flow; (3) replay and deterministic re-execution; (4) adjust coverage criteria from the context of message passing concurrent programs to shared memory context; (5) design data flow criteria based on shared variables and (6) build a tool to support these activities.

Sumário

1. Introdução.....	1
1.1. Contextualização.....	1
1.2. Motivação e objetivos.....	3
1.3. Estrutura desta dissertação.....	5
2. Programação concorrente.....	7
2.1. Considerações iniciais.....	7
2.2. Plataformas paralelas usadas na programação concorrente	8
2.3. Suporte à programação concorrente.....	13
2.3.1. Primitivas para inicializar/finalizar processos.....	17
2.3.2. Primitivas para interação.....	19
2.4. PThreads: a API POSIX para threads.....	26
2.4.1. PThreads: Criação e finalização de threads.....	27
2.4.2. PThreads: Sincronização com semáforos.....	29
2.4.3. PThreads: Sincronização com variável de condição.....	32
2.5. Considerações finais.....	33
3. Teste de software aplicado a programas concorrentes.....	35
3.1. Considerações iniciais.....	35
3.2. Teste de software.....	36
3.2.1. Técnica funcional.....	37
3.2.2. Técnica baseada em erros.....	38
3.2.3. Técnica estrutural.....	38
3.2.4. Ferramentas de apoio ao teste de software.....	43
3.3. Teste de programas concorrentes.....	45
3.3.1. Execução determinística de programas concorrentes.....	50
3.3.2. Ferramentas de teste para programas concorrentes.....	52
3.4. PCFG: Um modelo de teste para ambientes de passagem de mensagem.....	56
3.4.1. O modelo de teste PCFG.....	56
3.4.2. ValiPar: Ferramenta de apoio ao teste de programas concorrente.....	63
3.5. Considerações finais.....	69
4. Critérios de teste estrutural aplicados a programas multithreads.....	71
4.1. Considerações iniciais.....	71
4.2. PCFGsm.....	72
4.2.1. Representação por grafo (PCFGsm) do programa multithread.....	74
4.2.2. Fluxo de controle e de sincronização.....	75
4.2.3. Fluxo de dados, comunicação e caminhos.....	80
4.2.4. Caminhos e associações.....	82
4.3. Extrair dinamicamente informações de fluxo de dados.....	84
4.3.1. Atribuição de timestamps.....	87
4.3.2. Algoritmo para obtenção de pares de comunicação.....	95
4.4. Critérios de cobertura	98
4.5. Execução determinística.....	101
4.6. Considerações finais.....	104

5. ValiPThread: uma ferramenta de apoio ao teste de programas multithreads.....	107
5.1. Considerações iniciais.....	107
5.2. Arquitetura da ferramenta.....	108
5.2.1. O Módulo valipthread_inst: análise estática e instrumentação.....	109
5.2.2. O Módulo valipthread_elem: obtenção dos elementos requeridos.....	114
5.2.3. O Módulo valipthread_exec: execução de casos de teste.....	119
5.2.4. O Módulo valipthread_eval: avaliação da cobertura do código.....	128
5.3. Colocando a ValiPThread para executar.....	132
5.4. Considerações finais.....	134
6. Resultados.....	135
6.1. Considerações finais deste trabalho.....	135
6.2. Contribuições.....	136
6.3. Trabalhos futuros.....	139
A. Arquivos de configuração usados para a IDeL	149
B. Programa prodcons.c instrumentado	177

Lista de Figuras

Figura 2.1: Taxonomia das arquiteturas paralelas (STALLINGS, 2002).....	10
Figura 2.2: As várias camadas que podem implementar a memória compartilhada. (a) implementada pelo hardware. (b) implementada pelo sistema operacional. (c) linguagens de tempo de execução do sistema (TANENBAUM; STEEN, 2002).....	12
Figura 2.3: Exemplo de utilização das primitivas para inicializar/finalizar processos concorrentes.....	18
Figura 2.4: Implementação do problema do produtor-consumidor com buffer limitado usando semáforos (TANENBAUM, 2003).....	24
Figura 2.5: Implementação do problema do produtor-consumidor com buffer limitado usando monitor (TANENBAUM, 2003).....	25
Figura 2.6: Produtor-consumidor implementado com PThreads e semaphore.h.....	27
Figura 3.7: Exemplo dos Grafos CFG e Def-Use. (a) programa que calcula o máximo divisor comum (GCD – Greatest Common Divisor) entre dois valores; (b) CFG do programa da Figura 3.7(a); e (c) grafo Def-Use do programa da Figura 3.7(a).....	40
Figura 3.8: Implementação do problema GCD utilizando o ambiente de passagem de mensagem PVM. (a) Programa referente ao processo Mestre; e (b) programa referente a cada um dos três processos Escravos.....	58
Figura 3.9: GFCP referente ao programa GCD da Figura 3.8 (SOUZA et al., 2008).....	59
Figura 3.10: Arquitetura da ferramenta ValiPar (SOUZA et al., 2008).....	64
Figura 3.11: Representação textual do PCFG, gerada pela ferramenta ValiPar, referente ao programa do processo mestre pertencente ao programa paralelo GCD (SOUZA et al., 2008).	66
Figura 3.12: Elementos requeridos pelo critério Todas-Arestas-S para o programa GCD.....	67
Figura 3.13: Traces (rastros) de execução do programa GCD (SOUZA et al., 2008).....	68
Figura 3.14: Análise de cobertura feita pela ferramenta ValiPar para o critério Todas-Arestas-S para o programa GCD (SOUZA et al., 2008).....	69
Figura 3.15: Casos de Teste executados na ferramenta ValiPar (SOUZA et al., 2008).....	69
Figura 4.16: Problema do produtor / consumidor implementado com PThreads / ANSI C.....	78
Figura 4.17: Grafo PCFGsm que representa o programa mostrado na Figura 4.16.....	80
Figura 4.18: Traces produzidos pelas threads do programa apresentado na Figura 2.2 considerando uma execução hipotética.....	89
Figura 4.19: Algoritmo para atribuir timestamps aos eventos de um trace de execução.....	92
Figura 4.20: Exemplo do diagrama de espaço-tempo lógico, considerando o método de atribuição de timestamps apresentado nessa seção aplicado ao programa multithread do produtor / consumidor (Figura 2.2).....	94
Figura 4.21: Algoritmo para encontrar pares de comunicação do tipo comm-c-use.....	97
Figura 4.22: Algoritmo para encontrar pares de comunicação do tipo comm-p-use.....	98
Figura 4.23: Exemplo de não-determinismo.....	102
Figura 5.24: Arquitetura da ferramenta ValiPThread.....	109
Figura 5.25: Conteúdo dos arquivos Todas-Arestas-S.req (a), e Todas-Arestas-S.aut (b) para o critério Todas-Arestas-S, considerando o programa da Figura 2.6.....	118
Figura 5.26: Como ler a descrição textual de um autômato finito.....	119
Figura 5.27: Representação gráfica do autômato finito que corresponde ao elemento requerido descrito na linha 9 dos arquivos Todas-Arestas-S.req e Todas-Arestas-S.aut (Figura 5.25)....	119
Figura 5.28: Trace de cada thread para uma das possíveis execuções do programa prodcons.c	

(Figura 2.6).....	126
Figura 5.29: Seqüência de eventos de sincronização para cada um dos semáforos do programa prodcons.c (Figura 2.6) considerando um dos possíveis traces de execução.....	127
Figura 5.30: Resultado da avaliação da cobertura para o critério Todas-Arestas-S, considerando o caso de teste 1 (exemplo de uso do valipthread_exec).....	130
Figura 5.31: Resultado da avaliação da cobertura para o critério Todas-Arestas-S, considerando o caso de teste 1 (exemplo de uso do valipthread_exec).....	131
Figura 5.32: Timestamps atribuídos aos eventos do trace da Figura 2.5.....	131
Figura 5.33: Exemplo da estrutura de diretório para a instalação da ValiPThread.....	133

Lista de Tabelas

Tabela 2.1: Comparação entre as vantagens e desvantagens do uso de plataformas MIMD com memória compartilhada e com memória distribuída.....	13
Tabela 3.2: Ferramentas que apóiam o teste de software aplicado a programas concorrentes.	54
Tabela 3.3: Conjuntos do PCFG referente ao programa GCD (SOUZA et al., 2008).....	60
Tabela 3.4: Alguns elementos requeridos pelos critérios estruturais para o programa GCD (SOUZA et al., 2008).....	63
Tabela 4.5: Correspondência entre as primitivas e as formas de sincronização.....	76
Tabela 4.6: Conjuntos obtidos pela aplicação do modelo de teste definido ao programa da Figura 4.16.....	79
Tabela 4.7: Valores dos conjuntos referentes a fluxo de dados e comunicação para o programa da Figura 2.1.....	82
Tabela 4.8: Alguns dos elementos requeridos para os critérios de cobertura apresentados....	100

Introdução

1.1 Contextualização

O surgimento da programação concorrente ocorreu de forma natural devido à evolução dos sistemas operacionais, quando estes passaram a implementar multiprogramação (TANENBAUM, 2003). A multiprogramação permite que diversos processos executem em uma única arquitetura de hardware concorrendo ao uso de seus recursos. Para que isso funcione adequadamente o escalonador de processos do sistema operacional deve ser capaz de bloquear e desbloquear processos, a fim de realizar a troca de contexto (TANENBAUM, 2003). A multiprogramação permitiu reduzir a latência de memória e de entrada/saída, além de tornar os sistemas interativos do ponto de vista do usuário. Porém nesse cenário os processos não trabalham de forma colaborativa, interagindo entre si com o objetivo de produzir um resultado único, pelo menos não diretamente. Tais processos trabalham de forma concorrente, tentando obter para si os recursos necessários à sua execução em particular.

Em seguida viu-se a possibilidade de utilizar a programação concorrente para reduzir o tempo de execução das aplicações, quando estas são executadas em máquinas com diversos elementos de processamento. Os programas então podem ser projetados para terem diversos processos em paralelo que interagem entre si com o principal objetivo (porém não único) de terminarem mais rapidamente aquilo que se propõe a fazer. Para autores como (ALMASI;

GOTTlieb, 1994) isso é o que define a Programação Paralela.

É nesse cenário que surgiu a especialização da programação concorrente: a programação paralela. A programação paralela também é composta por diversos processos que interagem entre si, porém esses devem, de fato, estar executando em paralelo em um dado instante de tempo (DONGARRA *et al.*, 2002, GRAMA *et al.*, 2003). Neste trabalho, ambas, a programação concorrente e a programação paralela, podem ser tratadas como sinônimos, pois o trabalho aborda tais programas no nível da aplicação, não influenciando os aspectos de hardware citados.

A construção de processos concorrentes requer o uso de algumas primitivas fundamentais, não necessárias na programação seqüencial (ALMASI; GOTTlieb, 1994). São necessárias primitivas que: (1) definam quais processos são executados concorrentemente; (2) iniciem e finalizem processos concorrentes e; (3) coordenem e especifiquem a interação entre os processos concorrentes enquanto estes estiverem executando.

O projeto e o desenvolvimento de programas concorrentes é por natureza mais complexo que o de programas seqüenciais tradicionais. O projetista de aplicações concorrentes deve considerar tudo que se aplica a programas seqüenciais, além de precisar considerar os aspectos específicos da programação concorrente. Entre os principais desafios da programação concorrente é possível citar (FOSTER, 1995, ALMASI; GOTTlieb, 1994; GRAMA *et al.*, 2003; YANG; POLLOCK, 1997; CARVER; TAI, 2006): (1) definir como o programa será dividido (decomposição funcional e decomposição de domínio) e o quanto será dividido (granularidade); (2) planejar e implementar a interação entre os processos; (3) definir plataforma de hardware; (4) definir os aspectos de softwares a serem utilizados para o desenvolvimento e a execução; (5) traduzir todos os elementos acima em *SpeedUp*¹. É possível que um programa concorrente mal projetado apresente desempenho pior que sua versão seqüencial. (6) Testar e depurar esse tipo de programa. Este trabalho não aborda os conceitos de projeto de aplicações concorrentes.

Assim como o projeto e desenvolvimento de programas concorrentes apresenta desafios adicionais, a atividade de teste para esse tipo de programa também se torna muito

¹ *SpeedUp* é o cálculo que diz o benefício relativo de construir um programa concorrente/paralelo para resolver um problema. É obtido pela razão entre o tempo que o programa seqüencial leva para resolver um problema pelo tempo que a versão concorrente desse programa leva (Grama *et al.*, 2003; Dongarra *et al.*, 2002).

complexa. No entanto, é desejável que a qualidade desses programas também seja garantida. A engenharia de software é uma disciplina que busca a aplicação de técnicas de engenharia a fim de produzir software de alta qualidade a baixo custo (PRESSMAN, 2005, TIAN, 2005). A atividade de teste é uma das atividades de garantia de qualidade e tem por objetivo revelar a presença de defeitos no software. Sabe-se que o teste é uma atividade bastante custosa e por isso é constante o empenho por parte da academia para definir modelos de testes cada vez mais eficientes.

Os principais desafios adicionais ao teste de programas concorrentes são (YANG; POLLOCK, 1997): (1) desenvolver técnicas de análise estática para analisar programas concorrentes; (2) detectar situações não desejadas como: erros de sincronização, comunicação, de fluxo de dados e de *deadlock*; (3) reproduzir uma execução com a mesma entrada de teste e forçar a execução de um caminho na presença de não-determinismo; (4) gerar uma representação do programa concorrente que capture as informações pertinentes para a atividade de teste; (5) investigar a aplicação de critérios de teste sequenciais para programas concorrentes; e (6) projetar critérios de fluxo de dados para programas concorrentes, considerando troca de mensagens e variáveis compartilhadas.

1.2 Motivação e objetivos

Os dois principais ambientes para desenvolvimento de programas concorrentes são o ambiente de passagem de mensagem, ou memória distribuída, e o ambiente de memória compartilhada. No ambiente de passagem de mensagem cada processo possui seu próprio espaço de endereçamento de memória. A interação entre os processos é feita por meio da troca de mensagens que pode ser implementada através das primitivas *send/receive* ou então encapsuladas em estruturas como RPC (*Remote Procedure Calls*), RMI (*Remote Method Invocation*), entre outras (GRAMA *et al.*, 2003; TANENBAUM, 2003; DONGARRA *et al.*, 2002).

No ambiente de memória compartilhada, os processos compartilham um mesmo espaço de endereçamento de memória, espaço este utilizado para a interação entre os processos que acessam (lêem/escrevem) dados em variáveis compartilhadas. A sincronização nesse ambiente ocorre de forma explícita, usando-se recursos de software como: semáforos, variáveis de condição e monitores. Por outro lado, a comunicação ocorre de forma implícita,

por meio da definição e uso de variáveis compartilhadas.

Atualmente as grandes empresas produtoras de hardware e software, tais como Intel, IBM, Microsoft, Sun, entre outras, estão investindo bastante em arquiteturas multiprocessadas (KAHLE *et al.*, 2005). Essas arquiteturas são ideais para o contexto de programas concorrentes com memória compartilhada. A popularização dessas arquiteturas torna necessário o investimento em técnicas de engenharia de software e teste de software que busquem aumentar a produtividade e garantir que a qualidade dos programas produzidos. No âmbito de teste de software, se comparado com os trabalhos desenvolvidos para programas seqüências, ainda há muito a se fazer para o contexto de programas concorrentes.

Em outra frente, essas mesmas empresas junto com grandes centros de pesquisas, têm investido cada vez mais no desenvolvimento de sistemas computacionais *petascale*. Sistemas *petascale* são caracterizados por serem capazes de alcançar desempenho superior a 1 *petaFLOPS* (ou seja, um quadrilhão de operações de ponto flutuante por segundo). Esses sistemas são geralmente construídos por dezenas de milhares de elementos de processamento (*cores*) interconectados por infra-estruturas de rede proprietárias de alto desempenho. O suporte de software necessário à execução das arquiteturas *petascale* abrange diferentes conceitos, incluindo aqui o desenvolvimento de aplicações baseadas no paradigma de memória compartilhada. Isso ocorre devido ao alto grau de interação existente (granularidade fina) entre os processos (ou *threads* de execução). As principais aplicações para sistemas *petascale* são: previsão do tempo e do clima, química quântica, simulação nuclear, astronomia, etc. (BELL; GRAY; SZALAY, 2006).

Nesse contexto, o teste de programas concorrentes se torna crucial para que esses programas possam ser utilizados na prática. Alguns trabalhos nessa área já estão sendo desenvolvidos. Yang e Pollock (2003) propõem a adaptação do critério All-uses para programas concorrentes com memória compartilhada. Virgilio *et al.* (2005), por sua vez, propõem o modelo de teste **PCFG** (*Parallel Control Flow Graph*, ou Grafo de Fluxo de Controle Paralelo) para apoiar aplicação de critérios estruturais a programas concorrentes com passagem de mensagem. Carver e Lei (2006) propõem o teste de alcançabilidade para tratar programas concorrentes de um modo geral.

Dada essa motivação, os objetivos deste trabalho é adaptar e propor critérios de cobertura de teste para o contexto de programas concorrentes com memória compartilhada.

Esses critérios devem explorar os aspectos estruturais desses programas, focando principalmente a sincronização e a comunicação. Para propor esses critérios, um modelo de teste é definido, tal que as informações necessárias à atividade de teste sejam evidenciadas.

O objetivo secundário é estender a ferramenta ValiPar para apoiar a aplicação do modelo e dos critérios de teste definidos. Esta ferramenta deve ser instanciada para o contexto de programas implementados em C ANSI com o uso da biblioteca de *thread* PThreads.

1.3 Estrutura desta dissertação

Além deste capítulo introdutório, a dissertação apresenta, no Capítulo 2, um estudo da área de programação concorrente, contemplando aspectos de hardware e software com uma visão *top-down*, até convergir para as arquiteturas multiprocessadas com memória compartilhada e para as bibliotecas de *threads*.

O Capítulo 3 apresenta uma visão geral dos conceitos de teste de software dando maior atenção aos aspectos relativos à técnica estrutural. Em seguida apresenta trabalhos relacionados ao teste de software aplicado a programas concorrentes e, finalizando este capítulo, o modelo **PCFG** para ambientes de passagem de mensagem, e também a ferramenta ValiPar são apresentados.

No Capítulo 4 é apresentado o modelo **PCFGsm** proposto neste trabalho, contemplando a discussão a cerca das abordagens e decisões tomadas para elaborar essa proposta. O modelo considera a relação de causalidade (causa-e-efeito) para determinar sincronizações exercitadas, considerando a disciplina LIFO para a obtenção dos pares de sincronização, e também generaliza a inicialização e a finalização de *threads* como sincronizações comuns. São propostos critérios estruturais de cobertura adaptados e propostos para o contexto de programas *multithreads*. É proposto um método para revelar comunicações nesses programas que também utilizam a relação de causalidade e uma técnica de atribuição de *timestamps* adaptada de trabalhos anteriores (CARVER; LEI, 2006). Nesse capítulo 4 a aplicação do modelo é ilustrada com um programa de exemplo – o problema do produtor-consumidor com *buffer* limitado.

O Capítulo 5 apresenta a ferramenta ValiPThread implementada para apoiar a aplicação do modelo **PCFGsm**. Nesse capítulo 5 são apresentadas as funcionalidades da

ferramenta e como utilizá-la, mais uma vez ilustrando com um programa de exemplo. Os detalhes e decisões de implementação são também discutidos.

Este trabalho finaliza apresentando no Capítulo 6 as considerações finais relacionando as principais contribuições e sugerindo alguns trabalhos futuros.

Há no final desta dissertação duas seções de apêndices. O Apêndice A mostra os arquivos (C.l e C.y) que definem a gramática livre de contexto para a versão resumida da linguagem C ANSI que a IDeL trata. O Apêndice B mostra a versão instrumentada do programa *prodcons.c* (Problema do produtor-consumidor com *buffer* limitado) produzido pelo módulo *valipthread_inst* da ferramenta ValiPThread.

Programação concorrente

2.1 Considerações iniciais

As aplicações computacionais historicamente têm requerido maior poder de processamento aliado a menor tempo de resposta. Esses requisitos se dão principalmente devido a crescente quantidade de informação a ser tratada e a necessidade dos negócios das empresas, respectivamente. A programação concorrente e paralela é uma alternativa para atender tal demanda (ALMASI; GOTTLIEB, 1994; DONGARRA *et al.*, 2002; GRAMA *et al.*, 2003).

A idéia de se utilizar paralelismo no processo computacional é quase tão antiga quanto os computadores eletrônicos digitais. Em seus trabalhos, von Neumann já sugeria formas paralelas para resolver equações diferenciais (BARNES, *et al.*, 1968). Porém, o passo inicial do processamento paralelo é considerado o surgimento do computador ILLIAC IV, construído no início da década de 1970, composto por 64 unidades de processamento (NAVAUX, 1989).

A programação paralela consiste em dividir uma aplicação em partes menores para serem executadas por diversos elementos de processamento, no entanto, com a condição de interagir entre si para obter o resultado correto. Um programa paralelo é composto por

processos² ou *threads* conforme o contexto no qual está inserido. Em ambientes de passagem de mensagens (multicomputadores) os programas são compostos por processos. Em ambientes de memória compartilhada (multiprocessadores) são compostos por *threads*.

A programação concorrente é uma generalização da programação paralela, pois relaxa a necessidade de haver diversos elementos de processamento. Um programa concorrente pode, inclusive, executar em uma plataforma sequencial (ex. arquitetura de von Neuman) desde que esta suporte multiprogramação (ALMASI; GOTTLIEB, 1994; GRAMA *et al.*, 2003; DONGARRA *et al.*, 2002).

Neste capítulo são apresentados alguns conceitos a respeito de plataformas que suportam a programação concorrente do ponto de vista da implementação de hardware e software. Na sequência são apresentados aspectos de software para suporte à programação concorrente destacando as principais formas para desenvolver um programa concorrente. No entanto, são apresentados mais detalhes quando tratadas as bibliotecas de *threads*, uma vez que estão diretamente relacionadas ao escopo deste trabalho. O capítulo é finalizado com a apresentação da biblioteca de *threads* PThreads, pois esta é usada nos demais capítulos da dissertação para exemplificar a aplicação dos conceitos definidos.

2.2 Plataformas paralelas usadas na programação concorrente

A plataforma utilizada para executar programas concorrentes deve ser considerada durante o desenvolvimento destes programas. Assim é possível adotar um modelo de programação que seja adequado ao hardware e ao software básico (sistemas operacionais e compiladores, neste caso) disponíveis à execução. Segundo Flynn e Rudd (1996), as plataformas paralelas podem ser classificadas com base nos diferentes fluxos (*streams*) existentes em um sistema computacional. Nesse contexto, um fluxo pode ser uma sequência de objetos (por ex. dados) ou de ações (por ex. instruções). Com base nisso, Flynn e Rudd (1996) propõem quatro classes de arquiteturas paralelas. A Figura 2.1 ilustra a organização geral dessa taxonomia.

Na categoria **SISD** (*Single Instrution, Single Data stream*) estão caracterizadas as

2 No contexto deste trabalho uma aplicação é composta por um ou mais processos. Um programa é uma sequência de instruções. Um processo é um programa em execução, composto pelo programa, um conjunto de dados e um registro descritor (área da memória que armazena o conteúdo dos registradores quando o processo não está executando). E uma *thread* (ou processo leve) é uma linha de controle independente dentro do processo (TANENBAUM, 2006; TOSCANI; OLIVEIRA; CARISSIMI, 2003).

arquiteturas uniprocessadas tradicionais. As arquiteturas **SISD** também são conhecidas como arquiteturas de von Neumann e, conceitualmente, essa classe não apresenta paralelismo algum para a aplicação do usuário sendo executada, visto que apenas uma instrução e um dado são executados por vez.

No entanto, as atuais tecnologias de hardware permitem implementar nas arquiteturas **SISD** a técnica de *pipeline*, a qual divide a execução de uma instrução em diferentes estágios e os executa seqüencialmente. O paralelismo no *pipeline* ocorre quando há diferentes instruções executando em diferentes estágios do *pipeline*. As organizações de processadores com *pipeline*, múltiplos *pipelines* (arquiteturas superescalares), execução desordenada, entre outras, são vistas como técnicas que permitem melhorar a eficiência das arquiteturas **SISD** (NULL; LOBUR, 2003; STALLINGS, 2002; TANENBAUM, 2007).

Muitas dessas técnicas utilizadas na sua otimização empregam estratégias de execução paralela no nível de micro-arquitetura. Por esse motivo, tais estratégias são denominadas como paralelismo implícito, pois não são observadas explicitamente no nível de aplicação. O estudo do paralelismo implícito existente na arquitetura de von Neumann não pertence ao escopo desse trabalho e, portanto, não será discutido nesta dissertação.

Na categoria **SIMD** (*Single Instruction, Multiple Data stream*) estão presentes as arquiteturas de processadores vetoriais e processadores matriciais, os quais possuem uma única Unidade de Controle responsável por determinar o fluxo de instrução a ser executado pelas unidades funcionais. As arquiteturas dessa classe são uma resposta natural ao processamento de estruturas de dados regulares, tais como vetores e matrizes. Um processador vetorial é composto por uma Unidade de Controle e por um elemento de processamento que divide as operações de ponto-flutuante em diversos estágios e, dessa forma, é capaz de operar em seqüência sobre um conjunto de dados. Processadores matriciais, por sua vez, são formados por uma única Unidade de Controle e por vários elementos de processamento, todos, realizando a mesma operação sobre dados diferentes. Mais detalhes sobre as arquiteturas **SIMD** podem ser encontrados em Tanenbaum (2007) e Stallings (2002).

A categoria **MISD** (*Multiple Instruction, Single Data stream*), considerada por alguns autores como uma classe de arquiteturas sem produtos comerciais que a representem, é conceitualmente vista como um *pipeline* de diversas unidades funcionais executando independentemente, recebendo a saída da unidade anterior e passando o resultado para a unidade seguinte. Há pouco interesse comercial por esse tipo de arquitetura paralela,

principalmente pela ausência de métodos de programação desenvolvidos que facilmente façam o mapeamento para essa arquitetura (FLYNN; RUDD, 1996; DUCAN, 1990; TANENBAUM, 2007; NULL; LOBUR, 2003).

Figura 2.1: Taxonomia das arquiteturas paralelas (STALLINGS, 2002).

Máquinas **MIMD** (*Multiple Instrution, Multiple Data stream*) representam as arquiteturas mais utilizadas atualmente e de maior importância neste trabalho de pesquisa. Nessa classe se encontram as arquiteturas paralelas formadas por múltiplas Unidades de Controle e com múltiplos elementos de processamento independentes (embora possam executar o mesmo programa). Máquinas **MIMD** são classificadas em função da sua organização de memória, fato que afeta diretamente a maneira como as aplicações concorrentes realizam a interação durante as suas execuções. As máquinas com memória compartilhada apresentam um único espaço de endereçamento da memória e são conhecidas como Multiprocessadores. Máquinas **MIMD** multiprocessadoras usam variáveis compartilhadas para a interação entre os programas concorrentes. Por outro lado, máquinas **MIMD** com memória distribuída apresentam espaços de endereçamento distintos para cada porção de memória associada a um processador. Assim, para que a interação ocorra, são necessárias trocas de mensagens explícitas, normalmente usando-se primitivas de comunicação *send* e *receive*. São conhecidas como multicomputadores.

Máquinas **MIMD** com memória compartilhada (Multiprocessadores) são classificadas em função da sua organização de memória. Quando todos os processadores têm acesso a todas

as posições de memória com um mesmo custo, então têm-se uma máquina **UMA** (*Uniform Memory Access*). Quando os processadores têm igual acesso aos módulos de memória, aos dispositivos de E/S e também são tratados de maneira uniforme pelo Sistema Operacional, então tal máquina é conhecida como **SMP** (*Symmetric MultiProcessor*). Máquinas **UMA** facilitam o desenvolvimento de aplicações concorrentes, pois o desenvolvedor não precisa se preocupar com a localização dos dados. Todos os dados serão acessados com um mesmo custo. No entanto, máquinas **UMA** apresentam um grande problema de escalabilidade, pois poucos processadores e/ou módulos de memória podem ser usados para se garantir bons e uniformes tempos de acesso.

Máquinas **MIMD** com memória compartilhada, porém, com tempo de acesso não uniforme às diferentes posições de memória compartilhadas, são conhecidas como máquinas **NUMA** (*NonUniform Access Memory*). Embora permitam o acesso compartilhado à memória, máquinas **NUMA** repassam ao programador a preocupação com a localidade dos dados na plataforma. Por outro lado, máquinas **NUMA** não têm o mesmo problema de escalabilidade encontrado nas máquinas **UMA**. Há ainda outras classificações para máquinas multiprocessadas, como por exemplo: **COMA** (*Cache Only Memory Access*), **CC-NUMA** (*Cache Coherent NUMA*) e **NC-NUMA** (*NonCoherent NUMA*). A explicação dessas classes de arquiteturas vai além do escopo deste trabalho. O leitor pode encontrar um rico material sobre as mesmas em Tanenbaum (2007) e em Stallings (2002).

Máquinas **MIMD** com memória distribuída (Multicomputadores) podem ser representadas por dois tipos de arquiteturas: *clusters* (também chamados de **COW** – *Clusters of Workstations*) e **MPP** (*Massively Parallel Processors*). Os *clusters* são formados por dezenas, centenas ou milhares de **PCs** (*Personal Computer*, ou estações de trabalho), conectados por uma rede de conexão comum (LASTOVETSKY, 2003). Os **MPPs**, apesar de possuírem uma arquitetura semelhante à arquitetura dos *clusters*, apresentam um desempenho muito superior, vinculado a um custo também muito superior, pois são construídos com tecnologia proprietária e de propósito específico. Segundo Tanenbaum (2007), “a principal diferença entre um **MPP** e um *cluster* é análoga à diferença entre um mainframe e um PC”. Ambos têm CPU, memória RAM, discos e sistema operacional. No entanto, os mainframes têm estes recursos mais eficientes e são gerenciados e executados de maneira diferenciada.

Deve-se destacar também que os sistemas computacionais modernos não são monolíticos, mas sim construídos em diversas camadas, conforme pode ser observado na

Figura 2.2 (TANENBAUM; STEEN, 2002). Dessa forma, pode-se concluir que características de hardware podem ser simuladas em software e, assim, disponibilizadas às aplicações que executam no topo dessa hierarquia de camadas. Por outro lado, características de software também podem ser implementadas em hardware (TANENBAUM, 2007). A Figura 2.2 exemplifica a implementação de uma memória compartilhada em uma máquina **MIMD**.

Figura 2.2: As várias camadas que podem implementar a memória compartilhada. (a) implementada pelo hardware. (b) implementada pelo sistema operacional. (c) linguagens de tempo de execução do sistema (TANENBAUM; STEEN, 2002).

Na Figura 2.2(a) tem-se a implementação em hardware, sendo que o sistema operacional oferece uma visão única do espaço de endereçamento compartilhado. Na Figura 2.2(b) tem-se também uma plataforma com memória compartilhada, esta implementada agora no nível de software básico, pelo Sistema Operacional. Esta máquina também é conhecida como **DSM** (*Distributed Shared Memory*) (TANENBAUM, 2007). Neste caso, o sistema operacional, quando intercepta acessos a posições de memória remotas, monta uma mensagem que será encaminhada pelo mesmo ao processador de destino. Após realizar o acesso solicitado, outra mensagem é encaminhada de volta ao processador de origem. Todo esse processo é transparente à aplicação do usuário final.

Na Figura 2.2(c) a implementação da memória compartilhada é garantida no nível de usuário, por uma linguagem de programação e/ou ambiente de execução que ofereça esse tipo

de suporte. Neste caso, nem o sistema operacional nem o hardware tem consciência de que a aplicação está utilizando o paradigma de memória compartilhada.

Observa-se com este exemplo que os recursos da plataforma computacional podem estar implementados em hardware ou em software. Essa diferença de implementação é, muitas vezes, transparente à aplicação concorrente. Neste trabalho de pesquisa em particular, essa transparência é relevante na implementação da memória compartilhada, pois este trabalho insere-se no contexto de aplicações concorrentes desenvolvidas para plataformas **MIMD** com espaço de endereçamento compartilhado, independentemente se este compartilhamento é viabilizado por hardware ou por software.

Na Tabela 2.1 é apresentada uma comparação entre as vantagens e desvantagens do uso de plataformas **MIMD** com memória distribuída e com memória compartilhada (ALMASI; GOTTLIEB, 1994; GRAMA *et al.*, 2003; QUINN, 2003; DONGARRA *et al.*, 2002). Considerando as vantagens e as desvantagens de cada organização, é cada vez mais comum que projetistas de sistemas computacionais de grande porte optem por mesclas dessas duas opções, a fim de explorar o que há de melhor em cada uma delas (QUINN, 2003; STALLINGS, 2002; GRAMA *et al.*, 2003).

Tabela 2.1: Comparação entre as vantagens e desvantagens do uso de plataformas **MIMD** com memória compartilhada e com memória distribuída.

Crítérios	Memória Compartilhada	Memória Distribuída
Interação (comunicação/sincronização)	Por meio da própria memória compartilhada	Por meio de mecanismos de passagem de mensagem
Eficiência na interação	Maior	Menor
Implementação de hardware para sistemas de grande porte	Difícil / caro	Fácil / barato
Escalabilidade	Baixa	Alta
Projeto e desenvolvimento de programas concorrentes	Fácil	Difícil

2.3 Suporte à programação concorrente

A programação sequencial utilizada nas arquiteturas de von Neumann possui basicamente: instruções executadas sequencialmente, desvios condicionais e incondicionais, estruturas de repetição e chamadas a procedimentos (estes equivalem a desvios

incondicionais).

A programação concorrente permite otimizar o desempenho dessas aplicações, explorando a concorrência em arquiteturas com um ou vários elementos de processamento, permitindo assim uma melhor utilização dos recursos disponíveis.

A idéia básica da programação concorrente, portanto, é que determinadas aplicações sejam divididas em partes menores e que cada parcela resolva uma porção do problema. Para essa divisão são necessários recursos de software adicionais, estes responsáveis, por exemplo, pela ativação e finalização dos processos concorrentes, e que não estão disponíveis na programação seqüencial.

Segundo Almasi e Gottlieb (1994), três fatores são essenciais para a execução paralela: (1) definir um conjunto de subtarefas (ou processos) para serem executadas em paralelo; (2) capacidade de iniciar e finalizar a execução das subtarefas; e (3) capacidade de coordenar e especificar a interação entre as subtarefas enquanto estiverem executando.

Para que os processos concorrentes sejam ativados, fazem-se necessárias primitivas que determinem porções do código a serem executadas em paralelo e porções a serem executadas seqüencialmente. Além da ativação de diferentes linhas de execução concorrentes, fazem-se necessárias primitivas que permitam a sincronização da execução do programa, após estas linhas terem sido finalizadas. Isso é essencial para garantir a ordem em que os blocos devem ser executados.

Após terem sido iniciados os processos necessários à execução do programa, faz-se necessário definir os meios de interação entre os mesmos. A interação entre processos abrange duas necessidades básicas: comunicar e sincronizar. A comunicação realiza a troca de dados entre os processos e assim contribui para o avanço da computação a ser realizada. A sincronização, por sua vez, tem por objetivo coordenar as operações realizadas pelos processos e assim garantir que o programa produza um resultado correto. A sincronização pode ser de dois tipos: sincronização de *controle de seqüência* e sincronização de *controle de acesso* (ALMASI; GOTTLIEB, 1994; HWANG; XU, 1998).

Quando em um programa concorrente há restrições temporais entre as ações (instruções) dos processos, faz-se necessário atrasar ou bloquear ações a fim de respeitar a restrição temporal. Nesse caso há sincronização de controle de seqüência. Um exemplo simples desse tipo de restrição é quando um processo aguarda que outro defina o valor de uma variável para então fazer a leitura dela.

Outro caso que requer sincronização é quando há o acesso a recursos compartilhados, como: variáveis compartilhadas ou *spooling* de impressão. Nesse caso é desejável que apenas um número específico de processos (geralmente apenas um), obtenha acesso ao recurso. Nesse caso há sincronização de controle de acesso. O trecho do programa que requer sincronização de controle de acesso é chamado **região crítica** (TANENBAUM, 2007; ALMASI; GOTTLIEB, 1994).

As interações podem ser feitas usando-se o paradigma de variáveis compartilhadas (no caso de plataforma com memória compartilhada), ou usando-se o paradigma de passagem de mensagens (no caso de plataforma com memória distribuída).

Para ativar tais funcionalidades que iniciam/paralisam e também coordenam processos concorrentes, são necessárias ferramentas de software que disponibilizem primitivas para este fim. Essas ferramentas podem ser implementadas de maneiras distintas. Usualmente têm-se as seguintes opções (LASTOVETSKY, 2003): (1) compiladores para linguagens tradicionais que geram código concorrente de maneira automática, (2) linguagens específicas para a programação concorrente ou (3) bibliotecas de programação associadas a linguagens sequenciais, como por exemplo: PThreads (*POSIX – Threads*) (POSIX, 2003), OpenMP (*Open Multi-Processing*) (CHAPMAN *et al.*, 2007), PVM (*Parallel Virtual Machine*) (GEIST *et al.*, 1994) e MPI (*Message Passing Interface*) (SNIR *et al.*, 1996).

Os compiladores geradores de código concorrente permitem a criação de aplicações concorrentes de maneira transparente ao usuário final. Estes compiladores têm a função de gerar um código binário final diferente da execução definida pelo código fonte do programa sequencial. No entanto, apesar das alterações, a semântica permanece a mesma, sendo que apenas a ordem e a forma que as instruções ocorrem são modificadas, a fim de melhor utilizar os recursos computacionais e/ou reduzir o tempo de execução do programa. Tais compiladores normalmente atuam na paralelização de estruturas de repetição e de operações sobre vetores (LASTOVETSKY, 2003). O maior desafio dos compiladores otimizantes, no entanto, é determinar com eficiência quais laços e operações sobre vetores podem ser paralelizados. Para responder a essa questão, técnicas como a análise de dependência de dados são aplicadas no processo de otimização dos códigos. Exemplos desses compiladores são: Parafrase e **PFC** (*Parallel Fortran Converter*) (ALMASI; GOTTLIEB, 1994).

As linguagens de programação concorrente contêm os comandos específicos que os algoritmos concorrentes necessitam. Estas linguagens podem ser adaptações de linguagens

seqüenciais, como por exemplo as linguagens C e Fortran; ou completamente distintas, como as linguagens ADA e OCCAM (LASTOVETSKY, 2003). Maiores informações sobre linguagens de programação concorrente podem ser encontrados em (ALMASI; GOTTLIEB, 1994; QUINN, 2003; HWANG; XU, 1998; GRAMA *et al.*, 2003, LASTOVETSKY, 2003).

As bibliotecas de programação inserem em linguagens seqüenciais tradicionais as funcionalidades necessárias à programação concorrente. Com isso, podem ser utilizados recursos já conhecidos para o desenvolvimento das aplicações, tais como: as próprias linguagens de programação seqüenciais, seus compiladores e editores de texto. Exemplos dessas bibliotecas que estendem as funcionalidades das linguagens seqüenciais são: o PVM, as diferentes implementações do padrão – MPI, OpenMP e PThreads.

O PVM e o MPI permitem o desenvolvimento de aplicações concorrentes voltadas ao paradigma de passagem de mensagens. Maiores detalhes sobre os mesmos podem ser encontrados em (GEIST *et al.*, 1994; SNIR *et al.*, 1996).

O padrão OpenMP é uma alternativa ao uso de bibliotecas bem aceita pelos desenvolvedores, quando é necessária a programação concorrente usando o paradigma de memória compartilhada (QUINN, 2003; GRAMA *et al.*, 2003; LASTOVETSKY, 2003). O OpenMP foi definido por um grupo de empresas de hardware e/ou software, como: Intel, HP, SGI, IBM, SUN, Compaq, entre outras. Ele é composto por um conjunto de diretivas de compilação, uma biblioteca de funções e algumas variáveis de ambiente. Suas extensões para as linguagens C/C++ e Fortran são semanticamente equivalentes (GRAMA *et al.*, 2003; LASTOVETSKY, 2003; QUINN, 2003).

O uso de *threads* pode ser feito através de diferentes bibliotecas, as quais têm sido desenvolvidas por empresas de hardware e/ou software, vinculadas ao desenvolvimento de aplicações concorrentes no paradigma de memória compartilhada. Alguns exemplos de implementações são: PThreads, *Threads* em Java, Solaris, NT, SO/2, entre outras (LASTOVETSKY, 2003). Elas apresentam um comportamento semelhante, pois são baseadas no mesmo modelo de *multithreading*, apenas com pequenas variações particulares. Porém, dentre todas, a implementação PThreads merece uma atenção maior, pois foi tomada como padrão pelo IEEE Std 1003.1 (POSIX, 2003). Considerando o contexto deste trabalho, as bibliotecas de *threads* serão discutidas em maiores detalhes mais adiante no texto.

2.3.1 Primitivas para inicializar/finalizar processos

A mais básica das necessidades para a programação concorrente é a capacidade para inicializar e finalizar processos. Para atender essa necessidade alguns modelos foram propostos, cada um trazendo características específicas. Almasi e Gottlieb (1994) relatam três abordagens:

- **Fork/join:** é o mecanismo mais primitivo para inicializar/finalizar processos. A primitiva *fork* é capaz de inicializar processo enquanto a primitiva *join* tem a função de sincronizar a sua finalização. O *fork* é conceitualmente muito simples, porém complexo e caro para implementar, visto que são necessárias muitas instruções para criar um novo processo. A primitiva *join* também é conceitualmente simples, porém seu uso é um pouco mais complicado. Quando um processo (pai) executa o *fork*, passando como parâmetro um endereço de memória, outro processo (filho) é criado e sua execução inicia a partir do endereço passado. Ambos os processos (pai e filho) seguem suas execuções concorrentemente. Para executar a primitiva *join* o processo precisa fornecer três valores: *var*, *end1* e *end2*. Quando executa a primitiva *join* o valor de *var* é decrementado de uma unidade e, se o novo valor for igual a 0 (zero) a execução é desviada para *end1*, senão é desviada para *end2*. A Figura 2.3(a) exemplifica o uso dessas primitivas. Esse mecanismo é bastante versátil e poderoso, mas deixa todo o controle sob a responsabilidade do usuário. Isso tornam elevadas as chances de se obter programas com escrita desestruturada, suscetível a erros e difícil de compreender (HWANG; XU, 1998; ALMASI; GOTTLIEB, 1994; ANDREWS; SHINEIDER, 1983).
- **Parbegin/Parend:** é um mecanismo de mais alto nível, sendo considerado menos versátil que o fork/join. Parbegin e Parend são usados como palavras reservadas e definem um bloco de execução paralela. Todas as instruções e/ou blocos de instruções encapsulados pelo Parbegin/Parend são executados paralelamente, sendo, para isso, iniciado um novo processo ou *thread* para cada elemento do bloco. Ao fim da execução do bloco paralelo todos os processos criados são sincronizados, o que poupa o programador dos detalhes da sincronizações desses blocos concorrentes (HWANG; XU, 1998; ALMASI; GOTTLIEB, 1994; ANDREWS; SHINEIDER, 1983). A Figura 2.3(b) exemplifica seu uso. Quando um processo executa o Parbegin são criados tantos processos quanto forem necessários para que cada um execute concorrentemente cada

instrução dentro do bloco Parbegin/Parend. Ao final do bloco (Parend) todos os processos criados são sincronizados (como uma espécie de barreira) e finalizados. Observe que é possível aninhar blocos Parbegin/Parend.

Figura 2.3: Exemplo de utilização das primitivas para inicializar/finalizar processos concorrentes.

- Doall:** é usado para paralelizar estruturas de repetição, sendo que as iterações destas são divididas entre os processos/*threads* criados sob demanda. Algumas características encontradas normalmente nas implementações Doall são: requer que os processos sejam sincronizados automaticamente ao final da estrutura de repetição e a possibilidade de gerar uma quantidade dinâmica de processos, dada a quantidade de iterações. Funciona de maneira semelhante ao Parbegin/Parend, quando um processo

executa o Doall tantos processos quanto necessário são criados e cada um fica responsável por executar uma iteração do *loop*. Quando o *loop* acaba todos os processos são sincronizados e finalizados. É muito útil para manipular matrizes e vetores, sendo bastante empregado em plataformas **SIMD** (HWANG; XU, 1998; ALMASI; GOTTLIEB, 1994; ANDREWS; SHINEIDER, 1983). A Figura 2.3(c) exemplifica o uso dessa primitiva.

2.3.2 Primitivas para interação

Para cada paradigma de programação há formas particulares de realizar a comunicação e a sincronização. Aqui são apresentados aspectos relativos ao paradigma de passagem de mensagem e ao paradigma de memória compartilhada, sendo este último abordado em maiores detalhes, dado o contexto deste projeto.

Usando Passagem de Mensagens

Quando a passagem de mensagens é usada para comunicação e sincronização, os processos enviam e recebem informações ao invés de compartilhar variáveis. A troca de mensagens entre dois processos (comunicação ponto-a-ponto) é feita basicamente através de primitivas *send/receive*.

Há também alternativas disponibilizadas aos programadores, as quais são abstrações construídas sobre as primitivas *send/receive*. Exemplos dessas alternativas são: chamadas de procedimentos remotos (**RPC**, do inglês *Remote Procedure Calls*), Invocação Remota de Métodos (**RMI** – do inglês *Remote Method Invocation*) e primitivas coletivas de comunicação, estas envolvendo geralmente mais que dois processos. (HWANG; XU, 1998; ALMASI; GOTTLIEB, 1994; QUINN, 2003).

A sintaxe das primitivas *send/receive* é:

send lista de expressões *to* destino

receive lista de expressões *from* origem

As primitivas *send/receive* podem apresentar variações na sua semântica, dependendo da implementação das mesmas. Essas variações estão relacionadas principalmente aos conceitos de bloqueante e não bloqueante, síncrono ou assíncrono e se há ou não *buffers* intermediários para armazenar temporariamente a mensagem (GRAMA *et al.*, 2003; DONGARRA *et al.*, 2002).

Primitivas *send/receive* bloqueantes são liberadas da execução (permitem a execução da próxima instrução) apenas quando a mensagem está segura. Isto implica que a mensagem tenha sido efetivamente enviada, no caso de um *send* sem *buffers*, ou que a mensagem tenha sido copiada para um *buffer* intermediário, no caso de um *send* com *buffer*. No caso de um *receive* bloqueante, este só é liberado quando a mensagem é recebida, ou do *buffer* ou diretamente do *send* (caso não haja *buffers*) (GRAMA *et al.*, 2003; DONGARRA *et al.*, 2002).

Primitivas *send/receive* não bloqueantes não aguardam que a mensagem esteja segura, isto é, tenha sido copiada para um *buffer* ou tenha sido de fato enviada ao *receive*. Fica a cargo do programador não alterar o conteúdo da mensagem na memória local até que a mesma esteja segura. Apesar de agregarem uma complexidade a mais à programação concorrente, mensagens não bloqueantes permitem que comunicação e computação sejam sobrepostas, reduzindo assim a sobrecarga da comunicação (GRAMA *et al.*, 2003).

As primitivas *send/receive* síncronas, no nível da aplicação, como o próprio nome indica, determinam que haja um sincronismo entre a execução do *send* e do *receive*. As primitivas assíncronas não garantem que haverá essa situação durante a execução (GRAMA *et al.*, 2003; ALMASI; GOTTLIEB, 1994).

É importante observar que os conceitos de bloqueante/não bloqueante e síncrono/assíncrono são distintos, no nível da aplicação. Um par *send/receive* bloqueantes com *buffers* não são necessariamente síncronos, pois as primitivas são bloqueadas apenas enquanto a mensagem não está segura nos seus respectivos *buffers* de envio e recebimento.

O mecanismo de comunicação **ponto-a-ponto** permite que dois processos concorrentes, através das primitivas *send/receive*, realizem uma comunicação unidirecional. No mecanismo *rendezvous* dois processos concorrentes executam dois pares de primitivas *send/receive*, apresentando com isso uma comunicação bidirecional (ALMASI; GOTTLIEB, 1994).

Uma comunicação coletiva permite que mais que dois processos participem diretamente da comunicação, através de uma única primitiva de comunicação. Esta primitiva, na verdade, encapsula diversas chamadas *send/receive* básicas, o que simplifica a codificação da aplicação. As primitivas coletivas podem viabilizar diversas funcionalidades às aplicações, tais como: *broadcast*, *gather*, *scatter*, barreira de sincronização e operações de redução como soma global ou outras (GRAMA *et al.*, 2003; HWANG; XU, 1998). O uso de operações coletivas de comunicação é encorajado por muitos autores, visto que, além de reduzir o tamanho do código fonte e a probabilidade de erros, as comunicações coletivas são otimizadas pelos desenvolvedores das ferramentas (como o **MPI**) para as plataformas de execução. Assim, tais primitivas coletivas tendem a garantir portabilidade com eficiência ao código fonte desenvolvido (GRAMA *et al.*, 2003; FOSTER, 1995; QUINN, 2003).

Memória Compartilhada

Nesse paradigma as *threads* utilizam variáveis compartilhadas para interagirem entre si, permitindo assim a comunicação e sincronização. Diferente do paradigma de passagem de mensagem, em que a comunicação entre os processos é explícita, no paradigma de memória compartilhada a comunicação é implícita. No entanto a sincronização continua sendo explícita e, portanto desassociada da comunicação (DONGARRA *et al.*, 2002; GRAMA *et al.*, 2003).

A comunicação feita por variável compartilhada é algo simples, bastando para isso que a *thread* transmissora escreva o dado na variável compartilhada e que a *thread* receptora leia o dado dessa variável. No entanto, para que a semântica seja garantida, a comunicação deve ocorrer da forma esperada. Assim, faz-se necessário estabelecer uma restrição temporal entre a escrita e a leitura da variável. A solução para estabelecer essa restrição é feita através do uso de sincronizações de controle de seqüência.

O acesso exclusivo às variáveis compartilhadas também é importante para a comunicação neste contexto, principalmente quando há escritas em tais variáveis. Isso evita que dados sejam corrompidos devido à sobreposição da escrita e/ou utilização de dados antigos (já não mais válidos) por *threads* simultâneas. Para resolver esse problema, usa-se a sincronização de controle de acesso, visto que a variável compartilhada também é considerado um recurso compartilhado.

Apesar de a comunicação ser implícita, a sincronização ser explícita e ambas serem

independentes, a comunicação apresenta uma grande dependência da sincronização. Um programa concorrente com memória compartilhada só irá comunicar corretamente se as sincronizações estiverem corretas.

Para isso diversos mecanismos de sincronização, tais como semáforos e variáveis de condição, permitem garantir a ordem de execução das *threads* e também a ordem de leitura/escrita das variáveis compartilhadas.

Os mecanismos de sincronização tiveram seu desenvolvimento iniciado devido à evolução dos sistemas operacionais, quando esses passaram a implementar o conceito de multiprogramação (TANENBAUM, 2007; TOSCANI; OLIVEIRA; CARISSIMI, 2003). As sincronizações em sistemas operacionais tinham por principal objetivo a implementação de exclusão mútua para proteger dados quando ocorresse preempção de processos.

As primeiras alternativas para realizar essa operação foram: (1) desabilitar interrupções; (2) utilização da variável de bloqueio; (3) alternância estrita; (4) solução de Peterson; e (5) a instrução TSL (*Test and Set Lock*) (TANENBAUM; STEEN, 2002). Porém, todos esses mecanismos para implementação da exclusão mútua apresentam o problema da espera ocupada. Na espera ocupada há a execução indefinida de um laço que testa a possibilidade de acesso à região crítica. A espera ocupada é indesejável, pois acarreta desperdício de ciclos de CPU.

Para sanar o problema da espera ocupada as chamadas de sistema³ *Sleep* e *Wakeup* foram sugeridas. A primitiva *Sleep* bloqueia o processo até que outro chame a primitiva *Wakeup*. Com a utilização dessas primitivas evita-se a espera ocupada. No entanto, é possível que o sinal de *Wakeup* seja perdido, pois caso esta primitiva seja executada antes do respectivo *Sleep*, o sinal gerado não será armazenado pelo Sistema Operacional para uso futuro (TANENBAUM; STEEN, 2002).

Em 1965, E. W. Dijkstra, propôs o mecanismo para sincronização de processos chamado semáforo (TANENBAUM, 2003). O semáforo é um tipo de dado abstrato composto por uma variável inteira, normalmente iniciada com o valor 1, e uma lista de processos, inicialmente vazia. O semáforo admite duas operações: (1) **P** (do holandês, *proberen*, testar) comumente referenciada como *down*; e (2) **V** (do holandês, *verhogen*, incrementar) comumente referenciada como *up*. Essas duas operações são atômicas.

O semáforo é operado da seguinte forma: a operação *down* verifica se valor da variável

³ Chamadas de sistema são funções (instruções) que possibilitam a comunicação entre o sistema operacional e os programas do usuário (TANENBAUM, 2003).

é maior que 0. Nesse caso o valor é decrementado e o processo segue sua execução. Caso o valor seja 0 o processo é bloqueado e colocado para dormir em uma lista de espera. A operação *up* sobre o semáforo incrementa de uma unidade seu valor e libera um dos processos da lista de espera (caso exista algum), este escolhido pelo sistema operacional. Caso uma ou mais operações *up* sejam executadas antes de uma operação *down*, não haverá a perda dessa chamada, como ocorre com *sleep/wakeup*. Isso é viabilizado pela variável associada ao semáforo, a qual é incrementada a cada operação *up*.

Um caso especial de semáforo é o semáforo binário, onde se espera que o mesmo assuma apenas dois valores: ou 0 ou 1. O semáforo binário é também conhecido por *mutex* (**MUTual EXclusion**) e recebe este nome justamente porque é destinado a garantir a exclusão mútua (*mutual exclusion*) no acesso a regiões críticas (TANENBAUM, 2003).

Na Figura 2.4 é apresentada uma forma de implementar o problema do produtor-consumidor com *buffer* limitado usando apenas semáforos (TANENBAUM, 2003). Nesse problema o processo produtor deve produzir itens e colocá-los no *buffer* para que, em algum momento seguinte, o processo consumidor possa retirá-los do *buffer* e consumi-los. Nessa aplicação concorrente não podem ser produzidos mais itens que a capacidade do *buffer* e não podem ser consumidos itens de um *buffer* vazio.

Na implementação mostrada na Figura 2.4 três semáforos são utilizados: *mutex*, *empty* e *full*. O semáforo *mutex* é binário e utilizado para sincronização de controle de acesso, para que a inserção e remoção de itens no *buffer* ocorram de forma serial, mantendo-se a integridade do *buffer*. O trecho de código que faz acesso ao *buffer* é neste caso uma região crítica. Já os semáforos *empty* e *full* são utilizados para sincronização de controle de sequência. Sempre que o *buffer* estiver vazio, o semáforo *full* estará com o valor 0, o que fará o processo consumidor bloquear. De forma análoga, quando o *buffer* estiver cheio, o semáforo *empty* conterá o valor 0 e assim o processo produtor será bloqueado.

Na intenção de eliminar vários detalhes presentes na programação de aplicações concorrentes com memória compartilhada e com semáforos, Hoare, em 1974, e Brinch e Hansen, em 1975, em trabalhos independentes, propuseram um mecanismo de sincronização de mais alto nível chamado monitor (TANENBAUM, 2003). O monitor é um tipo abstrato semelhante a um pacote contendo procedimentos, variáveis e estruturas de dados. Ele encapsula vários detalhes de implementação existentes nos semáforos. Assim, nenhum processo tem acesso à estrutura interna dos monitores e qualquer operação com monitores

deve ser feita através dos procedimentos de alto nível por ele definido.

```
1 #include "prototypes.h"
2
3 #define N 100          /*número de espaços no buffer*/
4
5 typedef int semaphore; /*semáforos são um tipo especial de inteiro*/
6
7 semaphore mutex = 1;    /*controla o acesso á região crítica*/
8 semaphore empty = N;    /*conta os espaços vazios no buffer*/
9 semaphore full = 0;     /*conta os espaços ocupados no buffer*/
10
11 void producer(void)
12 {
13     int item;
14
15     while (TRUE) {      /*TRUE é a constante 1*/
16         produce_item(&item); /*gera alguma coisa para colocar no buffer*/
17         down(&empty);        /*decrementa o contador empty*/
18         down(&mutex);        /*entra na região crítica*/
19         enter_item(item);    /*coloca o novo item no buffer*/
20         up(&mutex);          /*deixa a região crítica*/
21         up(&full);           /*incrementa o contador full*/
22     }
23 }
24
25 void consumer(void)
26 {
27     int item;
28
29     while (TRUE) {      /*TRUE é a constante 1*/
30         down(&full);        /*decrementa o contador full*/
31         down(&mutex);        /*entra na região crítica*/
32         remove_item(item);  /*retira um item do buffer*/
33         up(&mutex);          /*deixa a região crítica*/
34         up(&empty);          /*incrementa o contador empty*/
35         consume_item(&item); /*faz alguma com o item */
36     }
37 }
```

Figura 2.4: Implementação do problema do produtor-consumidor com buffer limitado usando semáforos (TANENBAUM, 2003).

Apesar de qualquer processo, em qualquer momento, poder solicitar operações do monitor, apenas um processo terá permissão de execução em um determinado momento. Na Figura 2.5 é mostrado um esboço da implementação do problema produtor/consumidor utilizando monitor em uma linguagem de programação imaginária (TANENBAUM, 2003).

A utilização de monitores requer a definição de um novo tipo de variável, chamada variável de condição, para tornar possível a sincronização de controle de seqüência. A variável de condição recebe esse nome porque tem a função de controlar (sincronizar) a execução das *threads* concorrentes até que uma condição da lógica do programa esteja satisfeita. Essa variável de condição admite duas operações: *Wait* e *Signal*.

Essas operações funcionam de forma semelhante às primitivas *Sleep* e *Wakeup*,

respectivamente, sendo associada a uma condição da lógica do programa. Variáveis de condição apresentam o mesmo problema de possível perda de sinal encontrado com *Sleep* e *Wakeup*. Na Figura 2.5 observa-se que a própria definição do monitor é responsável pela sincronização de controle de acesso (no caso, acesso ao *buffer* de itens) enquanto as variáveis de condição *full* e *empty* são responsáveis pela sincronização de controle de sequência.

```
1 monitor ProducerConsumer
2     condition full, empty
3     integer count;
4
5     procedure enter;
6     begin
7         if count = N then wait(full);
8         enter_item;
9         count := count + 1;
10        if count = 1 then signal(empty);
11    end;
12
13    procedure remove;
14    begin
15        if count = 0 then wait(empty);
16        remove_item;
17        count := count - 1;
18        if count = N - 1 then signal(full);
19    end;
20
21    count := 0;
22 end monitor;
23
24 procedure producer;
25 begin
26     while true do
27     begin
28         produce_item;
29         ProducerConsumer.enter;
30     end
31 end;
32
33 produce consumer;
34 begin
35     while true do
36     begin
37         ProducerConsumer.remove;
38         consume_item;
39     end
40 end;
```

Figura 2.5: Implementação do problema do produtor-consumidor com buffer limitado usando monitor (TANENBAUM, 2003).

Existem outros mecanismos de mais alto nível para fazer a sincronização de *threads*. Por exemplo: barreira e semáforos leitores e escritores. Em geral, esses mecanismos são construídos a partir de mecanismos básicos como semáforos e monitores (LEWIS; BERG, 1996; DONGARRA *et al.*, 2002; GRAMA *et al.*, 2003).

Esses mecanismos de sincronização, bem como as primitivas para inicializar/finalizar processos, podem estar disponíveis na própria linguagem de programação paralela, como o que ocorre em Ada, Occam e OpenMP; ou por meio de bibliotecas compartilhadas, como o que ocorre em PThreads.

2.4 PThreads: a API POSIX para threads

O padrão PThreads é uma das normas definidas pelo IEEE em seu conjunto de normas POSIX (*Portable Operating Systems Interface*). O padrão PThreads está definido pela norma IEEE std POSIX 1003.1c-1995 (POSIX, 2003b) que por sua vez define uma interface para programação de *threads* contemplando um conjunto de funções e tipos de dados para a linguagem C (LEWIS; BERG, 1996). A definição dessa interface permite que programas concorrentes implementados com PThreads sejam portáteis para qualquer plataforma que implemente esse padrão, bastando para isso compilar o programa na plataforma de destino.

As funções definidas na **API** (*Application Programming Interface*) satisfazem as necessidades básicas para a programação concorrente fornecendo funções para criar, inicializar e finalizar *threads*; e funções que permitem a sincronização das *threads*. A comunicação nesse contexto é feita por meio de variáveis compartilhadas e, por isso, não há funções específicas para realizar as comunicações.

Na **API** PThreads o nome de todas as funções e tipos de dados pré-definidos são prefixados com *pthread_* para facilitar a identificação e evitar conflito de nomes com outra eventual biblioteca qualquer. A sincronização definida é basicamente feita por semáforos binários (*mutex*) e variáveis de condição. Além do uso desses mecanismos de sincronização é muito comum em programas implementados com PThreads o uso do semáforo definido pela norma POSIX IEEE Std 1003.1b-1993 (POSIX, 2003a). Esta API para semáforos tem suas funções e tipos de dados prefixados por *sem_*.

A combinação dessas duas normas que definem APIs para *threads* permite a implementação eficiente de programas *multithreads*. O programa apresentado na Figura 2.6 implementa uma versão do problema do produtor-consumidor com *buffer* limitado da Figura 2.4, agora usando essas duas APIs para *threads*. Esse programa será usado para exemplificar o uso de PThreads e semáforos quando forem apresentadas a seguir algumas de suas principais funções.

<pre> 1 #include <stdlib.h> 2 #include <stdio.h> 3 #include <pthread.h> 4 #include <semaphore.h> 5 6 sem_t mutex, empty, full; 7 int queue[2], avail; 8 9 void *producer(void); 10 void *consumer(void); 11 12 /* Thread T0 (main) */ 13 int main(void) { 14 pthread_t prod_h, cons_h; 15 16 avail = 0; 17 sem_init (&mutex, 0, 1); 18 sem_init (&empty, 0, 2); 19 sem_init (&full, 0, 0); 20 pthread_create(&prod_h, 0, 21 producer, NULL); 22 pthread_create(&cons_h, 0, 23 consumer, NULL); 24 pthread_join(prod_h, 0); 25 pthread_join(cons_h, 0); 26 exit(0); 27 } 28 29 30 31 </pre>	<pre> 32 /* Thread T1 (producer) */ 33 void *producer(void) { 34 int prod = 0, item; 35 while (prod < 2) { 36 item = rand()%1000; 37 sem_wait(&empty); 38 sem_wait(&mutex); 39 queue[avail] = item; 40 avail++; 41 prod++; 42 sem_post(&mutex); 43 sem_post(&full); 44 } 45 pthread_exit(0); 46 } 47 /* Thread T2 (consumer) */ 48 void *consumer(void) { 49 int cons= 0, my_item; 50 while (cons < 2) { 51 sem_wait(&full); 52 sem_wait(&mutex); 53 cons++; 54 avail--; 55 my_item = queue[avail]; 56 sem_post(&mutex); 57 sem_post(&empty); 58 printf('consumed: %d\n', 59 my_item); 60 } 61 pthread_exit(0); 62 } </pre>
--	---

Figura 2.6: Produtor-consumidor implementado com PThreads e semaphore.h.

2.4.1 PThreads: Criação e finalização de *threads*

O modelo adotado pelo PThreads permite a criação de *threads* dinamicamente, ou seja, sob demanda, uma vez que ele implementa as primitivas do modelo *fork/join* para isso. A função para criar uma nova *thread* é:

pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine (void *), void *arg).

A chamada a essa função cria uma nova *thread* para o programa e a coloca na fila de “prontos para executar” do escalonador. A nova *thread* terá um PC (*Program Counter*) próprio e que no momento da criação conterá o endereço de início da sub-rotina especificada pelo parâmetro *start_routine*. Essa rotina desempenha o papel da função “*main*” da nova *thread*. É a partir dela que a *thread* inicia sua execução. Note que essa rotina é uma função como qualquer outra em C, porém, por padrão, requer que o tipo de retorno seja um ponteiro

para um tipo de dados indefinido (*void **) e que apenas um parâmetro de entrada seja possível, este também sendo um tipo de dado indefinido. Isso fornece flexibilidade ao programador, para que ele defina diferentes tipos de dados simples ou estruturados como parâmetro de entrada e de saída, usando para tanto os devidos *type casts*, quando necessário.

O parâmetro **attr* permite configurar atributos para a *thread*. Entre esses atributos estão: *DetachState*, *SchedulingScope*, *StackSize* e *StackAddress* (LEWIS; BERG, 1996; OPENGROUP, 2009). Quando o parâmetro **attr* tiver o valor 0 (zero) são assumidos os atributos padrão. Por fim, o parâmetro **thread* retorna ao “pai” (*thread* criadora) um ponteiro para a nova *thread* (“filho”) para que o pai possa ter controle sobre a execução da *thread* filha. O padrão também permite que uma *thread* criada possa criar outras a partir dela.

O programa da Figura 2.6 implementa três *threads*: *main* (primeira *thread*, obrigatória), *producer* e *consumer*. O programa inicia a execução a partir da *main*, que por sua vez cria duas novas *threads* (linhas 20 e 22): uma *producer* e outra *consumer*. Para ambas as *threads* criadas não há parâmetros de entrada, nem atributos configurados, sendo neste caso usados os atributos padrão.

A segunda primitiva básica para a programação concorrente estabelece maneiras de finalizar uma *thread*. No padrão PThreads uma *thread* é finalizada sempre que seu PC atinge uma instrução finalizadora. Instruções tais com *exit* e *pthread_exit* são capazes de finalizar uma *thread* em qualquer momento quando executadas. Outro caso em que a *thread* é finalizada é quando atinge o final da sua função “*main*” (função indicada como ponto de partida no momento da criação da *thread*) ou quando nesta função é executada a instrução *return*.

A função ***pthread_join(pthread th, void **thread_return)*** é usada no contexto de finalização de *threads* e tem dois objetivos: (1) sincronizar a *thread* que chamou a função *pthread_join* com a finalização da *thread* apontada pela variável *th*; e (2) fornecer para a *thread* chamadora o valor retornado pela *thread* finalizada, caso haja algum.

No programa da Figura 2.6 ambas as *threads* criadas (*producer* e *consumer*) finalizam sua execução quando executam o *pthread_exit* (linha 45 e 61). A *thread* principal (*main*) sincroniza com a finalização destas, primeiro com a *thread producer* (linha 24) e em seguida com a finalização da *thread consumer* (linha 25). No entanto, a *thread main* despreza o valor de retorno das *threads* filhas (segundo argumento de *pthread_join*).

2.4.2 PThreads: Sincronização com semáforos

O semáforo especificado pela norma IEEE Std 1003.1b-1993 (POSIX, 2003a) utiliza uma variável declarada pelo tipo predefinido *sem_t*. Essa variável tem seu valor iniciado pela função **sem_init(sem_t *sem, int pshared, unsigned int value)**. O parâmetro **sem* é um ponteiro para a variável semáforo, *pshared* indica se o semáforo será ou não compartilhado entre diferentes processos (além de ser compartilhado entre *threads* no mesmo espaço de endereçamento) e *value* indica o valor inicial do semáforo.

No programa da Figura 2.6 três semáforos foram declarados: *mutex*, *empty* e *full* (linha 6). O semáforo *mutex* tem a função de garantir exclusão mútua no acesso ao *buffer*. O *empty* e *full* mantêm, respectivamente, a quantidade de posições vazias e cheias no *buffer*. Os três são inicializados respectivamente nas linhas 17 (*mutex*, com o valor 1), 18 (*empty*, com o valor 2) e 19 (*full*, com o valor 0).

Esta API define as seguintes operações (funções) para manipular os semáforos:

- **int sem_post(sem_t *sem):** incrementa o valor do semáforo, cuja semântica é análoga ao *up*. Caso haja alguma *thread* bloqueada na fila de espera do semáforo um sinal é emitido para desbloquear uma delas selecionada a critério do sistema operacional. O parâmetro *sem* é um ponteiro para a variável semáforo a qual a operação *post* será executada. No programa da Figura 2.6 essa função é usada nas linhas 42 e 56 para desbloquear o *mutex* e conseqüentemente liberar o acesso à região crítica. Na linha 43 para incrementar o semáforo *full*, indicando que mais uma posição do *buffer* foi preenchida. E na linha 57 para incrementar o semáforo *empty* para indicar que há mais uma posição no *buffer* vazia;
- **int sem_wait(sem_t *sem):** bloqueia a *thread* caso o seu valor seja zero ou, caso contrário, decrementa o valor do semáforo e retorna imediatamente. Sua semântica é análoga ao *down*. O parâmetro *sem* é um ponteiro para a variável semáforo que sofrerá a operação *wait*. No programa da Figura 2.6 essa função é usada nas linhas 38 e 52 para bloquear o semáforo *mutex* e obter acesso exclusivo à região crítica. Na linha 37 para decrementar o semáforo *empty* indicando que uma posição do *buffer* está prestes a ser preenchida. E na linha 51 para decrementar o semáforo *full* indicando que uma posição do *buffer* está prestes a ser liberada;
- **int sem_trywait(sem_t *sem):** decrementa o valor do semáforo caso seja maior que zero. Caso contrário retorna um valor indicando o fato, porém não bloqueia a *thread*.

O parâmetro *sem* corresponde ao ponteiro para a variável semáforo a qual a operação *wait* será executada;

- **int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout):** decrementa o valor do semáforo caso seja maior que zero. Caso contrário é colocado na fila de *threads* bloqueadas pelo tempo máximo especificado. Quando o tempo máximo é extrapolado sem que a *thread* tenha sido liberada para prosseguir, ela é desbloqueada e um valor de retorno indica este caso. O parâmetro *sem* corresponde ao ponteiro para a variável semáforo a qual a operação *wait* será executada e o parâmetro *abs_timeout* corresponde ao tempo máximo de bloqueio da *thread*.

Todas essas funções retornam 0 em caso de sucesso, caso contrário retornam o valor inteiro -1 e a variável global *errno* é definida para indicar o código do erro.

O padrão de semáforo especificado pela norma IEEE std POSIX 1003.1c define o tipo especial de semáforo *mutex*, o qual assume apenas dois valores: 1 ou 0 (respectivamente desbloqueado ou bloqueado). Este semáforo é declarado com o tipo predefinido *pthread_mutex_t* e inicializado pela função *pthread_mutex_init* ou pela macro *PTHREAD_MUTEX_INITIALIZER*.

A função **int pthread_mutex_init(pthread_mutex_t *mut, const pthread_mutex_attr_t *attr)** recebe como parâmetro um ponteiro para o *mutex* (*mut*) a ser iniciado e outro ponteiro (*attr*) para uma estrutura de dados que configura atributos para o *mutex*. Entre os atributos possíveis estão: *Cross-Process*, *Priority Inheriting* e *Ceiling Priority* (LEWIS; BERG, 1996). Quando para o parâmetro *attr* é passado a constante NULL ou o valor 0 os atributos padrões são atribuídos ao *mutex*. Sempre que o *mutex* é corretamente inicializado ele estará desbloqueado (valor igual a 1).

A macro *PTHREAD_MUTEX_INITIALIZER* deve ser chamada no momento da declaração do *mutex* (por exemplo, *pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER*) e equivale à chamada da função *pthread_mutex_init* com o segundo parâmetro NULL.

Para reverter o processo de inicialização do *mutex*, ou seja, liberar todo recurso alocado para ele, o padrão oferece a função **int pthread_mutex_destroy(pthread_mutex_t *mutex);**

As três principais funções que manipulam o *mutex* são:

- **int pthread_mutex_lock(pthread_mutex_t *mutex):** decrementa o valor do semáforo bloqueando-o, ou bloqueia a *thread* e a coloca na fila de espera do *mutex* caso este já esteja bloqueado. É semelhante à *sem_wait* (para valores binários). O parâmetro *mutex* corresponde ao ponteiro para o *mutex* a ser manipulado;
- **int pthread_mutex_unlock(pthread_mutex_t *mutex):** incrementa o valor do semáforo e desbloqueia-o (sinalizando para desbloqueia alguma *thread* da fila de espera do semáforo caso haja alguma) caso seu valor seja zero. Caso contrário, nada é feito. Essa função apresenta um comportamento semelhante à *sem_post* (para valores binários). O parâmetro *mutex* corresponde ao ponteiro para o *mutex* a ser manipulado;
- **int pthread_mutex_trylock(pthread_mutex_t *mutex):** decrementa o valor do semáforo caso o valor seja igual a um e bloqueia-o. Caso contrário retorna um valor indicando o fato, porém não bloqueia a *thread*. Apresenta um comportamento semelhante à *sem_trywait*. O parâmetro *mutex* corresponde ao ponteiro para o *mutex* a ser manipulado;
- **int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime):** decrementa o valor do semáforo caso seja 1 e bloqueia-o. Caso contrário a *thread* é colocada na fila de espera do *mutex* pelo tempo máximo especificado. Quando o tempo máximo é extrapolado sem que a *thread* tenha sido liberada para prosseguir, ela é desbloqueada e um valor de retorno indica este caso. Apresenta um comportamento análogo à **sem_timedwait**. O parâmetro *mutex* corresponde ao ponteiro para o *mutex* a ser manipulado e o parâmetro *abstime* corresponde ao tempo máximo de bloqueio da *thread*;

Todas essas funções retornam 0 em caso de sucesso, caso contrário retornam um código do erro específico.

O programa da Figura 2.6 poderia fazer uso do *pthread_mutex* substituindo a declaração *sem_t mutex* (linha 6) por *pthread_mutex_t mutex*. Substituindo as funções *sem_init(&mutex, 0, 1)* (linha 17) por *pthread_mutex_init(&mutex, NULL)*. E substituindo as funções *sem_wait(&mutex)* (linhas 38 e 52) por *pthread_mutex_lock(&mutex)* e *sem_post(&mutex)* (linhas 42 e 56) por *pthread_mutex_unlock(&mutex)*.

2.4.3 PThreads: Sincronização com variável de condição

Variáveis de Condição usando PThreads aceitam duas operações: *Signal* e *Wait*. Uma variável de condição é declarada pelo tipo predefinido *pthread_cond_t* e inicializada pela função *pthread_cond_init* ou pela macro *PTHREAD_COND_INITIALIZER*.

A função **int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)** recebe como parâmetro um ponteiro (*cond*) para a variável de condição a ser inicializada e um segundo parâmetro (*attr*), também ponteiro que especificam os atributos da variável de condição a ser inicializada. O único atributo disponível é: *Cross-Process* (LEWIS; BERG, 1996). Se for especificado como segundo parâmetro para essa função a constante NULL ou o valor 0, o atributo padrão será assumido para a variável de condição.

A macro *PTHREAD_COND_INITIALIZER* deve ser usada no momento da declaração da variável compartilhada (por exemplo, *pthread_cond_t cond = PTHREAD_COND_INITIALIZER*) e equivale à chamada função *pthread_cond_init* recebendo NULL como segundo parâmetro.

Para reverter a inicialização, ou seja, liberar os recursos alocados com a inicialização de uma variável de condição (*cond*) o padrão disponibiliza a função **int pthread_cond_destroy(pthread_cond_t *cond)**.

O padrão PThreads requer que variáveis de condição sejam utilizadas em conjunto com *mutex*, isto é, a condição analisada deve estar protegida por um *mutex*. As funções definidas para manipular variáveis de condição são:

- **int pthread_cond_signal(pthread_cond_t *cond):** sinaliza a uma *thread* que já esteja bloqueada na fila de espera da variável de condição especificada pelo parâmetro *cond*. Caso não haja nenhuma o sinal é perdido não ficando acumulado;
- **int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex):** atômicamente libera o *mutex* associado à variável de condição (argumento *cond*), e bloqueia a *thread*, colocando-a na fila de espera da variável de condição passada como argumento *cond*;
- **int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime):** atômicamente libera o *mutex* associado à variável de condição (argumento *cond*), e bloqueia a *thread* colocando-a na fila de espera da variável de condição pelo tempo máximo especificado pelo argumento *abstime*;

- **int pthread_cond_broadcast(pthread_cond_t *cond):** sinaliza todas as *threads* que já estejam bloqueadas na fila de espera da variável de condição *cond* para que sejam todas desbloqueadas.

Todas essas funções retornam 0 em caso de sucesso, caso contrário retornam um código do erro específico.

No programa da Figura 2.6 seria possível substituir os semáforos *full* e *empty* por variáveis de condição, no entanto mudanças substanciais teriam que ser feitas na lógica do programa.

2.5 Considerações finais

A programação concorrente é uma alternativa interessante para atender a demanda por maior poder e capacidade de processamento aliado a menor tempo de respostas das aplicações. Para contemplá-la diversas são as abordagens de software e hardware propostas, bem como metodologias e ambientes de programação. Especificamente no contexto de programas concorrentes com memória compartilhada a popularização das arquiteturas *multicores* e também o crescente e constante foco em arquiteturas *petascales* tem requerido atenção nesta área.

Neste capítulo foram discutidos aspectos de hardware e software para composição de plataformas paralelas, sob uma ótica *top-down* até chegar às arquiteturas multiprocessadas com memória compartilhada que é o foco deste trabalho. Depois foram discutidas alternativas disponíveis para a construção de programas concorrentes, novamente adotando uma ótica *top-down* até chegar às bibliotecas de *threads*. Por fim a biblioteca de *threads* PThreads foi apresentada sempre exemplificada com a aplicação do problema do produtor/consumidor com *buffer* limitado.

Finalizando, é importante que programas concorrentes apresentem qualidade para que possam ser efetivamente usados na prática. Para isso metodologias de desenvolvimento, padrões de projeto e modelos de teste precisam ser definidos especificamente para eles. No próximo capítulo são discutidos aspectos relacionados ao teste de software aplicado a programas concorrentes apresentando o estado da arte desta área e alguns trabalhos relevantes para o desfecho desta dissertação de mestrado.

Teste de software aplicado a programas concorrentes

3.1 Considerações iniciais

Dado o custo da atividade de teste, cada vez mais se busca definir técnicas e critérios que apresentem maior capacidade de detecção de defeitos a um menor custo, favorecendo assim todo o ciclo de desenvolvimento de software.

De forma resumida, a atividade de teste consiste em: (1) selecionar um conjunto de casos de teste, em que cada caso de teste é formado por valores de entrada para o programa (selecionados a partir do domínio de entrada) e a respectiva saída esperada; (2) executar cada caso de teste; e (3) comparar o resultado obtido com o resultado esperado. Assim o testador pode avaliar se o software está ou não correto em relação ao conjunto de casos de teste.

Assim, os seguintes itens são relacionados à atividade de teste (MYERS, 2004):

- a atividade de teste é o processo de execução de um software a fim de revelar defeitos;
- um bom caso de teste deve possuir alta probabilidade de revelar defeitos;
- um teste bem-sucedido é aquele que revela algum defeito ainda não descoberto.

Um dos desafios da atividade de teste é definir um conjunto de casos de teste que seja suficiente para revelar todos os defeitos de um software. O ideal seria obter um conjunto de

casos de teste igual a todas as entradas possíveis para um software, porém é impraticável devido ao custo de se executar o software para cada caso de teste. Não existe um algoritmo de baixo custo que seja capaz de, para um dado software, gerar um conjunto de casos de teste eficaz para revelar defeitos (PRESSMAN, 2005; MYERS, 2004; TIAN, 2005). Sendo assim, técnicas e critérios são definidos a fim de sistematizar a seleção de casos de teste.

Dada a importância da programação concorrente também é importante (e necessário) que esses programas apresentem boa qualidade e que estejam livre de defeitos. Para isso técnicas e critérios de teste precisam ser desenvolvidas especificamente para o universos dos programas concorrentes uma vez que estes apresentam dificuldades adicionais em relação ao teste de software convencional.

Este capítulo traz uma revisão dos principais conceitos relacionados ao teste de software de um modo geral, apresentando os jargões, as principais técnicas e critérios de teste, dando maior atenção à técnica estrutural. Também são apresentadas algumas ferramentas que apóiam a atividade de teste. Em seguida é apresentado o estado da arte para o teste de software aplicado a programas concorrentes, aqui sendo apresentado em detalhes o modelo de teste *PCFG* (*Parallel Control Flow Graph*, ou Grafo de Fluxo de Controle Paralelo). Algumas ferramentas que apóiam o teste de programas concorrentes também são apresentadas.

3.2 Teste de software

No processo de teste de software diversas técnicas estão disponíveis sendo possível optar por uma combinação delas, pois, essas técnicas se complementam. São algumas delas: funcional, estrutural e baseada em erros (PRESSMAN, 2005; TIAN, 2005; MALDONADO *et al.*, 2004). Cada uma dessas técnicas está relacionada a uma fase do ciclo de desenvolvimento do software e sendo assim cada uma tem objetos de teste diferentes, em que os objetos de teste são as fontes de informação para cada técnica. Por exemplo, as especificações funcionais são os objetos de teste da técnica funcional.

Outro desafio é determinar quando um software já foi suficientemente testado a fim de garantir a qualidade desejada. Então, além da técnica, é importante ter definido o critério de teste a ser utilizado pela técnica escolhida. Para cada técnica há diversos critérios de teste

propostos. Pode-se escolher aplicar um ou mais desses critérios. O critério de teste pode ser definido como um predicativo que determinará quais elementos devem ser testados pela técnica utilizada (PRESSMAN, 2005; TIAN, 2005). Esses elementos são denominados elementos requeridos.

As subseções seguintes abordam a técnica funcional e a baseada em erros. Maior atenção é dispensada a técnica estrutural uma vez que essa técnica é explorada neste trabalho.

3.2.1 Técnica funcional

Essa técnica é também conhecida com teste caixa-preta, pois analisa o software como uma caixa cujo conteúdo é desconhecido. Em outras palavras, apenas observa-se o conjunto de entrada e o resultado produzido pelo software, podendo ser aplicada a partir da fase de especificação do software. Essa técnica é denominada funcional, pois se baseia nos requisitos funcionais do software. Ela busca encontrar erros das seguintes categorias (PRESSMAN, 2005): (1) funções incorretas ou inexistentes, (2) erros de interface, (3) erros na estrutura de dados ou no acesso a base de dados externas, (4) erros de comportamento ou desempenho, e (5) erros de inicialização ou de finalização do software.

Alguns critérios de teste funcionais podem ser destacados (PRESSMAN, 2005; TIAN, 2005; MALDONADO *et al.*, 2004):

Particionamento em classes de equivalência: esse critério de teste divide o domínio de entrada do software em classes de equivalência (válidas e inválidas) a partir das possíveis entradas. Então seleciona-se o menor número possível de casos de teste levando em consideração que um caso de teste de uma classe de equivalência seja representativo a toda classe.

Análise do valor limite: esse critério de teste complementa o critério particionamento em classes de equivalência, buscando selecionar casos de testes para cobrir os valores limites de cada classe, pois são nesses pontos que se encontram um maior número de erros (TIAN, 2005). Tanto para o caso de particionamento em classes de equivalência com para o caso de análise de valor limite o domínio de saída também pode ser levado em consideração (PRESSMAN, 2005).

Dado que a técnica funcional se baseia nas especificações funcionais do software, que

são geralmente elaboradas de modo informal, os requisitos de teste também tendem a ser informal e isso inviabiliza a automação do processo de testes. Dessa forma, esse tipo de teste normalmente é aplicado manualmente (MALDONADO *et al.*, 2004; PRESSMAN, 2005; TIAN, 2005). A principal vantagem dessa técnica é a possibilidade de gerar testes ainda na fase de especificação. Como desvantagem, os testes gerados por essa técnica não garantem a cobertura de partes essenciais do código.

3.2.2 Técnica baseada em erros

Essa técnica de teste utiliza informações referentes aos erros mais comuns cometidos no processo de desenvolvimento do software para estabelecer os requisitos de teste (PRESSMAN, 2005; TIAN, 2005). Entre os critérios mais utilizados dessa técnica destaca-se o critério análise de mutantes.

O critério análise de mutantes procura demonstrar que um conjunto de defeitos pré-estabelecidos não está presente no programa. Para isso a partir do programa P a ser testado gera-se um conjunto de programas ligeiramente modificados (mutantes). Os mutantes são gerados por meio de pequenas mudanças sintáticas, tentando representar possíveis defeitos. Deve-se então encontrar um conjunto de casos de teste que seja capaz de revelar as mudanças de comportamento existentes entre o programa e seus mutantes (DEMILLO *et al.*, 1978; PRESSMAN, 2005). É um critério com alta taxa de detecção de defeitos além de ser aplicável a diversos contextos, porém apresenta um alto custo computacional (MALDONADO *et al.*, 2004).

3.2.3 Técnica estrutural

A técnica de teste estrutural é também conhecida como teste caixa-branca, pois nesse caso a lógica interna do programa é levada em consideração durante o teste.

Essa técnica utiliza, geralmente, uma representação em grafo dirigido do programa a ser testado. Esse grafo é denominado **CFG** (*Control Flow Graph*, ou Grafo de Fluxo de Controle), definido por $G=(N, E, s)$, em que N representa o conjunto de nós do grafo, E representa o conjunto de arestas dirigidas que ligam os nós, e s representa o nó inicial do

grafo, tal que $s \in N$ (RAPPS; WEYUKER, 1985). Nesse grafo cada nó representa um bloco de instruções do programa que são executadas seqüencialmente e cada aresta dirigida representa uma transferência de controle entre os nós. A Figura 3.7(b) mostra o **CFG** referente ao programa mostrado na Figura 3.7(a). O programa na Figura 3.7(a), denominado *GCD* (*Greatest Common Divisor*), realiza o cálculo do máximo divisor comum entre dois valores inteiros e positivos. Observe como no **CFG** cada nó representa um bloco de comandos seqüenciais, por exemplo, o nó 1 (Figura 3.7(b)) representa as linhas 5 e 6 do programa (Figura 3.7(a)).

A partir do **CFG** são definidos os caminhos que podem ser encontrados no programa. Seguem as definições:

caminho: é definido por uma seqüência finita de nós $(n_1, n_2, \dots, n_k)$, $k \geq 2$, tal que existe uma aresta de n_i para n_{i+1} , $i = 1, 2, \dots, k-1$. Por exemplo, os nós 4, 6, 2 e 7 na Figura 3.7(b) formam um caminho.

caminho simples: é definido por um caminho em que todos os nós são diferentes, exceto possivelmente o primeiro e o último nó. Por exemplo, os nós 2, 3, 5, 6 e 2 na Figura 3.7(b) formam um caminho simples.

caminho livre de laço: ocorre quando todos os nós do caminho, sem exceção, são diferentes. Os nós 1, 2, 3, 5 e 6, na Figura 3.7(b), formam um caminho livre de laços.

caminho completo: é um caminho em que o primeiro nó é o inicial e o último nó é o final do grafo. Exemplo: os nós 1, 2, 3, 5, 6, 2 e 7, na Figura 3.7(b), formam um caminho completo.

Os critérios da técnica de teste estrutural podem ser classificados em três grupos: (1) critérios baseados em fluxo de controle; (2) critérios baseados em fluxo de dados; e (3) critérios baseados na complexidade (MALDONADO *et al.*, 2004). Apenas os critérios baseados em fluxo de controle e em fluxo de dados são descritos em detalhes nas subseções a seguir, pois são de interesse nesse trabalho.

Crítérios baseados em fluxo de controle

Os critérios deste grupo utilizam apenas informações de controle de execução do programa, tais como comandos e desvios, para determinar as estruturas que precisam ser testadas. São alguns dos critérios propostos para este grupo:

Critério Todos-Nós: este critério requer que todos os nós do *CFG* sejam executados ao menos uma vez. Em outras palavras, cada comando do programa deve ser exercitado pelos casos de teste.

Critério Todas-Arestas: para satisfazer este critério é necessário que cada aresta do *CFG*, ou seja, cada transferência de fluxo de controle seja exercitada pelo menos uma vez pelo conjunto de casos de teste.

Figura 3.7: Exemplo dos Grafos *CFG* e *Def-Use*. (a) programa que calcula o

máximo divisor comum (GCD – Greatest Common Divisor) entre dois valores; (b) CFG do programa da Figura 3.7(a); e (c) grafo Def-Use do programa da Figura 3.7(a).

Critério Todos-Caminhos: requer que todos os caminhos possíveis do **CFG** sejam exercitado ao menos uma vez. Este critério é na maioria das vezes impossível de ser alcançado, pois pela definição de caminho, um laço infinito teria infinitos caminhos a serem exercitados.

Crítérios baseados em fluxo de dados

Os critérios deste grupo utilizam informações de fluxo de dados para definir os requisitos de teste. Esse tipo de critério requer que sejam testadas as interações entre a definição e o subsequente uso das variáveis do programa. A proposta desses critérios de teste foi motivada pelo fato de que os critérios baseado em fluxo de controle não são capazes de revelar a maioria dos defeitos, até mesmo, os mais simples (RAPPS; WEYUKER, 1985).

Rapps e Weyuker (1985) estendem o **CFG**, definindo o grafo *Def-Use* (Definição e Uso). O grafo *Def-Use* é construído de forma semelhante ao **CFG** agregando informações referentes à definição e ao uso de variáveis do programa. A Figura 3.7(c) mostra o grafo *Def-Use* para o programa mostrado na Figura 3.7(a).

A definição de uma variável ocorre quando um valor é atribuído a ela, seja por uma instrução de atribuição ou pela passagem por referência da variável para uma função. Sempre que o valor da variável é alterado na memória ocorre a definição da variável. No grafo é representado pelo nome da variável colocada entre parênteses e precedida da palavra “**def**” e posicionada junto ao nó onde ocorre a definição. Por exemplo, na Figura 3.7(c), no nó 4 há definição da variável *y*.

O uso de uma variável é caracterizado quando ela é referenciada, sem que esteja sendo redefinida (MALDONADO *et al.*, 2004). O uso de uma variável pode ser de dois tipos:

Uso computacional (c-uso): ocorre quando a referência (uso) à variável afeta uma computação sendo realizada. No grafo é representado pelo nome da variável colocada entre

parênteses e precedida da palavra “**c-uso**” e posicionada próximo ao nó onde ocorre o uso. Por exemplo, na Figura 3.7(c), no nó 4 há **c-uso** das variáveis x e y .

Uso predicativo (p-uso): ocorre quando a referência (uso) à variável afeta uma instrução de decisão determinando o fluxo de execução. No grafo é representado pelo nome da variável colocada entre parênteses e precedida da palavra “**p-uso**” e posicionada junto à aresta que faz a transferência de controle relacionada ao uso predicativo. Na Figura 3.7(c), nas arestas (3,4) e (3,5) ocorrem **p-uso** das variáveis x e y .

No contexto de fluxo de dados uma nova definição de caminho possível é chamado de caminho livre de definição:

Caminho livre de definição: um caminho livre de definição com relação à variável x é um caminho em que não há redefinição de x . Um exemplo de caminho livre de definição em relação à variável x no grafo *Def-Use* da Figura 3.7(c) é formado pelos nós 5, 6 e 2, pois o nó 5 contém uma definição da variável x e nos nós 6 e 2 não ocorre a redefinição dessa variável.

Para este grupo de critérios as seguintes associações podem ser definidas:

Associação c-uso: a associação c-uso é definida, formalmente, pela tripla, (n_i, n_j, x) tal que a variável x é definida no nó n_i , o nó n_j contém um c-uso de x e há ao menos um caminho de n_i para n_j livre de definição com relação a x . Um exemplo, com base na Figura 3.7(c), de associação c-uso é: $(1, 5, x)$, pois o nó 1 contém a definição de x , o nó 5 faz c-uso de x e há ao menos um caminho livre de definição (ex. 1, 2, 3 e 5) com relação a x do nó 1 ao nó 5.

Associação p-uso: a associação p-uso é formalmente definida pela tripla $(n_i, (n_j, n_k), x)$, tal que a variável x é definida no nó n_i , a aresta (n_j, n_k) faz p-uso de x e há um caminho de n_i para (n_j, n_k) livre de definição com relação a x . A associação $(5, (3, 5), x)$ é um exemplo de p-uso, com base no grafo *Def-Use* da Figura 3.7(c), pois o nó 5 contém a definição de x , a aresta (3, 5) contém o p-uso de x e há ao menos um caminho livre de definição (ex. 5, 6, 2, 3 e 5) com relação a x .

Alguns dos critérios que compõem este grupo de critérios são (MALDONADO *et al.*, 2004):

Critério Todas-Definições: requer que cada definição de variável seja exercitada pelo conjunto de teste, pelo menos uma vez, não importa se por um c-uso ou por um p-uso.

Critério Todos-Usos: requer que todas as associações entre uma definição de variável e seus subseqüentes usos (c-usos e p-usos) sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição.

Critério Todos-c-Usos: requer que todas as associações entre uma definição de variável e seus subseqüentes c-usos sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição.

Critério Todos-p-Usos: requer que todas as associações entre uma definição de variável e seus subseqüentes p-usos sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição.

3.2.4 Ferramentas de apoio ao teste de software

Depois de escolher as técnicas e critérios adequados para realizar a atividade de teste é de grande importância que também se utilize uma ferramenta de apoio. A utilização de ferramentas de apoio não se justifica somente porque alguns dos procedimentos realizados durante a atividade de teste são automatizáveis, mas porque se a atividade de teste for realizada manualmente ela estará factível a erro (humano) que poderá comprometer o resultado do teste. Seria completamente indesejável que um defeito do programa deixe de ser detectado porque o procedimento foi erroneamente aplicado.

Também é importante observar que as ferramentas de apoio propiciam a transferência do conhecimento científico na atividade de teste para a indústria.

A seguir são descritas algumas ferramentas existentes e que apóiam o teste de programas seqüenciais.

Poketool

A Poketool (*P*OTential *U*ses *C*Riteria *T*OOL for program testing) é uma ferramenta interativa que apóia a aplicação de critérios estruturais baseados em fluxo de controle e em fluxo de dados (CHAIM, 1991). Ela implementa os seguintes critérios: Todos-Nós, Todas-Arestas e a família de critérios Potenciais-Usos definidas por Maldonado (1991). É uma ferramenta multi-linguagem e, atualmente, possui versões para testar programas em C,

FORTRAN e Cobol.

Sua arquitetura é composta por quatro módulos e sua execução, em linhas gerais, se dá em quatro etapas: (1) a análise estática e a instrumentação do código fonte do programa; (2) a compilação do programa instrumentado; (3) a execução do programa instrumentado com os casos de teste fornecidos pelo usuário; e (4) avaliação da cobertura dos casos de teste com base nos critérios de teste selecionados (BUENO *et al.*, 1994).

JaBUTi

Essa ferramenta foi proposta por Vincenzi *et al.* (2003) para apoiar a aplicação de critérios de teste estrutural para programas em Java. Os critérios cobrem aspectos de teste de unidade e de teste de integração no contexto de orientação a objetos. Ela opera o programa Java a ser testado em nível de *bytecode*⁴ e isso torna desnecessário que se possua o código fonte Java para testar o programa.

A implementação da JaBUTi é feita em Java e é composta de onze pacotes divididos em duas partes. A primeira parte contempla seis pacotes e é responsável por realizar os procedimentos de teste propriamente ditos. São eles: análise estática, computação de elementos requeridos, execução e avaliação dos casos de teste. A segunda parte, composta por cinco pacotes, implementa a interface gráfica com o usuário (GUI - *Graphical User Interface*) o que torna a ferramenta mais amigável e simples de se operar. Através da GUI é possível visualizar as classes de três formas: o *bytecode* em modo texto, o código fonte, se estiver disponível, em texto, o grafo DUG (*Def-Use Graph*). Isso permite que o usuário iterativamente acompanhe a cobertura do critério escolhido à medida que executa cada caso de teste (VINCENZI *et al.*, 2006).

IDeL

IDeL (*Instrumentation Description Language*) é uma meta-linguagem para descrição de instrumentações (SIMÃO *et al.*, 2003). A instrumentação é uma técnica amplamente empregada na engenharia de software com diversos propósitos. No contexto deste trabalho é

⁴ *Bytecode* é a versão compilada do programa Java a qual será interpretada pela máquina virtual Java (JVM - Java Virtual Machine) (LEWIS; BERG, 1999).

utilizada para avaliação de cobertura de teste. A IDeL tem por principal característica abstrair e capturar os conceitos mais importantes no processo de instrumentação. Nesse processo é preciso obter o grafo do programa (*CFG*), coletar informações importantes sobre o programa e inserir instruções que registrarão a execução (*trace*) do programa.

Pelo fato da IDeL ser uma meta-linguagem é possível fazer instrumentação de códigos em diversas linguagens, apenas definindo uma gramática livre de contexto para a linguagem específica e fornecendo-a como entrada para o programa IDeLgen. O IDeLgen é uma espécie de compilador que criará um instrumentador para a linguagem específica.

O trabalho realizado pela IDeL/IDeLgen é fundamental para ferramentas de teste, pois o trabalho de instrumentação feito manualmente é um processo lento e suscetível a erros. Além disso, a maioria dos instrumentadores está disponível para linguagens específicas o que torna difícil o reúso dos mesmos.

3.3 Teste de programas concorrentes

Os primeiros trabalhos que tratam de teste de programas concorrentes são datados das décadas de 1970 e 1980 e tratam principalmente dos aspectos de depuração e reexecução de programas concorrentes (HANSEN, 1973; HANSEN 1978; TAI 1985; CARVER; TAI, 1986; TAI, 1989). Não são tratados nesses trabalhos aspectos relacionados à cobertura de código. No entanto, esses trabalhos apresentam conceitos básicos aplicados ao teste de programas concorrentes que ainda são utilizados nas propostas atuais.

Taylor *et al.* (1992) apresentam uma das primeiras propostas para a adaptação de técnicas de teste estrutural para aplicação em programas concorrentes. Nessa proposta, é utilizada o grafo de concorrência (ou grafo de alcançabilidade - *Reachability Graph (RG)*) obtido a partir da análise estática do programa concorrente, no caso, escritos na linguagem Ada e CSP. O grafo de concorrência é utilizado para representar os possíveis estados de concorrência, considerando a sincronização/comunicação do programa concorrente. Então, a partir desse grafo, critérios de cobertura são propostos. Os critérios propostos apresentam grande semelhança a critérios de fluxo de dados, apresentados para programas sequenciais. Um problema dessa proposta é que o grafo de concorrência apresenta explosão do número de estados de concorrência, que ocorre quando há um grande número de possíveis

sincronizações/comunicações entre as tarefas (processos). Para o **RG** o número de estados possíveis cresce exponencialmente com o número de processos e sincronizações (TAYLOR *et al.*, 1992).

Buscando contornar o problema de explosão do número de estados de concorrência, Koppol e Tai (1996) apresentam uma abordagem incremental para o teste estrutural de programas concorrentes. A proposta é integrar processos que estão no mesmo nível de uma maneira *bottom-to-top*. A abordagem é feita da seguinte forma: aplica-se o teste no nível mais baixo do grafo de concorrência e, em seguida, uma redução é feita no grafo de concorrência, ocultando transações (sincronização/comunicação) entre processos do mesmo nível. Então sobe-se um nível e aplica-se novamente o teste nesse nível levando em consideração agora apenas transações entre processos do nível atual com processos do nível inferior. Em seguida a redução é aplicada ao nível em questão e sobe-se mais um nível. Isso é repetido até se atingir o nível mais alto do grafo. Com essa abordagem a cada incremento do nível, transações entre processos de níveis mais baixos deixam de ser considerados, o que reduz consideravelmente a cardinalidade do grafo em níveis superiores.

Além de aliviar a explosão do número de estados de concorrência essa abordagem apresenta outras vantagens, tais como: (1) suporte ao desenvolvimento e teste de forma incremental, ou seja, funções/programas mais simples vão sendo desenvolvidos, testados e então integrados a outras funções/programas, que também já foram testadas; (2) detecção precoce de falhas durante o ciclo de desenvolvimento de softwares (KOPPOL; TAI, 1996).

Em Yang e Pollock (1997) foi iniciada uma discussão de como adaptar a aplicação de critérios de fluxo de dados em programas concorrentes com memória compartilhada. Eles mostram que a principal dificuldade é detectar a comunicação, feita através de variáveis compartilhadas, e a sincronização realizada por chamadas de funções *wait* (semelhante à *Sleep*) e *post* (semelhante à *Wakeup*). Abordando essas dificuldades adicionais é adaptado o conceito de definição e uso de variáveis para programas concorrentes. Em seguida, é proposto um algoritmo híbrido para encontrar caminhos, e conseqüentemente casos de teste, que cubram todos os elementos requeridos, considerando definição e uso de variáveis em *threads* (processos) diferentes.

O algoritmo é híbrido porque combina e estende outros dois algoritmos propostos para determinar caminhos em programas sequenciais: **DFS** (*Depth-First Search*) e **DT-IT**

(*Dominator Tree and Implied Tree*) (YANG; POLLOCK, 1997).

Observou-se que a aplicação do algoritmo DFS, como proposto para programas sequenciais, em programas concorrentes é capaz de encontrar caminhos aceitáveis⁵, porém não é garantido que esses caminhos sejam *w-runnable* (caminho que contém para cada *wait* um *post* associado tornando-o livre de espera infinita).

O algoritmo **DT-IT** é utilizado, em programas sequenciais, para encontrar todos os possíveis caminhos para estruturas de decisão. Porém, apresenta o mesmo problema que o **DFS** quando aplicado sem adaptações a programas concorrentes: não garante que os caminhos encontrados sejam *w-runnable*. Além disso, também não garante que os caminhos sejam aceitáveis pois não garante que definição de variáveis precederá seu uso. A proposta então é combinar ambos os algoritmos, devidamente adaptados, para determinar caminhos considerando definição e uso de variáveis em programas concorrentes com memória compartilhada.

Yang *et al.* (1998) propõem uma adaptação do critério Todos-Usos, para programas concorrentes. Nesse trabalho os autores abordam o paradigma de memória compartilhada e citam que o modelo pode ser aplicado a programas concorrentes que empregam passagem de mensagem. Nesse modelo um grafo de fluxo de programa paralelo (**PPFG** – *Parallel Program Flow Graph*) é obtido através da análise estática do programa concorrente.

O **PPFG** é constituído de um grafo *Def-Use* para cada *thread* (processo) incluindo arestas que representam instruções de criação, sincronização e comunicação das *threads*. A partir do **PPFG** são extraídas as informações pertinentes à atividade de teste, entre elas os caminhos que caracterizam as associações Todos-Usos, que são os elementos requeridos de teste. Por fim, esse trabalho apresenta um procedimento para obtenção de Todos-Usos a partir do **PPFG** e um ferramenta, *della pasta*, para apoiar a aplicação desse modelo no teste de programas concorrentes (YANG *et al.*, 1998).

Considerando o critério Todos-Usos, Yang e Pollock (2003) propõem um modelo de teste para aplicação desse critério em programas concorrentes. Nessa proposta eles tratam especificamente de programas concorrentes com memória compartilhada. No modelo proposto utilizam a definição de du-caminho para programas concorrentes feita por Yang e

5 Um caminho em programas paralelos (composto pelo caminho de cada processo) é dito *aceitável* se não há caminhos não-executáveis para nenhum dos processo (YANG; POLLOCK, 1997).

Pollock (1997) e fazem algumas restrições (YANG; POLLOCK, 2003):

- o número total de *threads* (processos) é fixo e conhecido no momento da construção do *PPFG*;
- assume o modelo de programação **MIMD** (*Multiple Instruction and Mutiple Data*); e
- as possíveis combinações de *posts* e *waits* são estaticamente conhecidas no momento da construção do *PPFG*.

Nesse trabalho é considerada a utilização de casos de teste temporais que, além do conjunto de entrada para o programa, contém um parâmetro que altera o tempo de execução das instruções de comunicação/sincronização das diversas *threads*. Com alteração do tempo de execução das instruções de comunicação/sincronização das *threads* espera-se que várias seqüências de comunicação/sincronização sejam exercitadas. Com essa abordagem eles defendem a execução de casos de teste temporais com diferentes valores para o parâmetro temporal e então a comparação do estado da memória do programa no fim de cada execução. Caso todas as execuções apresentem o mesmo estado de memória e esse esteja correto então os elementos requeridos, exercitados pelo caso de teste, estarão cobertos e um novo caso de teste pode ser exercitado. Caso contrário um defeito foi detectado e deve ser corrigido.

Essa proposta faz uso de execução determinística de programas concorrente (YANG; POLLOCK, 2003). A execução determinística é um assunto abordado por muitos trabalhos e será tratado na próxima subseção. Uma limitação da proposta é que apenas programas concorrentes com memória compartilhada que utilizam primitivas básicas de sincronização (*wait* e *post*) são tratados. Em trabalhos futuros os autores pretendem estender a proposta para que passe a tratar outros mecanismos de comunicação e sincronização.

Outra abordagem para o teste de programas concorrentes é a semeadura de instruções de *delay* (instruções que causam o atraso da execução de outras instruções). Essa técnica é também conhecida por “*noise maker*”. Consiste em incluir (instrumentar) no programa concorrente instruções de *delay* (como por exemplo, *sleep*⁶ e *yield*⁷ do Java) para atrasar a execução de outras instruções (principalmente as de sincronização) com o objetivo de

6 A função *sleep* do Java faz com que a *thread* seja bloqueada até que outra a desbloqueie. Sua variante *sleep(time)* faz a *thread* bloquear até um tempo limite (*time*) quando a *thread* é desbloqueada pela sistema operacional, se nenhuma outra *thread* o fez antes (LEWIS; BERG, 1999).

7 A função *yield* do Java faz com que a *thread* disponibilize o processador pra outra e volte imediatamente para fila de “pronto para executar” do escalonador (LEWIS; BERG, 1999).

executar diferentes pares de sincronização e conseqüentemente produzir diferentes seqüências de sincronização. Edelstein et al. (2003) propõem uma metodologia, baseada em heurística, para determinar a melhor forma de semear os *delays* em programas Java. No trabalho também é apresentada uma ferramenta chamada ConTest que apóia a aplicação dessa proposta. A ferramenta conta com módulos para: (1) instrumentação (semeadura de *delays*) feita em nível de *bytecode*; (2) detecção de disputa de dados; (3) analisador de cobertura de código; e (4) reexecução.

Vergilio *et al.* (2005) apresentam uma família de critérios para teste de programas concorrentes que utilizam o paradigma de passagem de mensagem. Eles consideram aspectos de sincronização e comunicação do programa concorrente que são independentes do ambiente (linguagem de programação ou API). O modelo de teste proposto é baseado no trabalho apresentado por Yang *et al.*, (1998) que captura informações relativas ao fluxo de controle e fluxo de dados. Esse trabalho é descrito em maiores detalhes na Seção 3.4.

Uma abordagem para o teste de programas concorrentes que faz uma mescla da abordagem de execução determinística e não-determinística é apresentada por Lei e Carver (2006) e é denominada *teste de alcançabilidade*. Nessa abordagem o programa é executado deterministicamente até certo ponto, com base em seqüências de sincronizações possíveis, e a partir desse ponto a execução é feita de forma não-determinística. Nesse trabalho é apresentado um método para obtenção de variantes de disputas (seqüências de sincronizações diferentes e válidas) com base no *trace* coletado de execuções anteriores. As variantes de disputas são obtidas por meio da atribuição de *timestamp* a eventos concorrentes. Essa proposta, assim como a proposta do RG, busca exercitar todos os possíveis estados de concorrência do programa. No entanto, o teste de alcançabilidade elimina todos os estados que não podem ser alcançáveis. Ainda nesse trabalho os autores mostram por meio de propriedades da teoria dos grafos que para um dado programa o teste de alcançabilidade exercita das seqüências de sincronização válidas. Porém é possível que uma mesma seqüência seja exercitada mais que uma vez, pois os algoritmos propostos são *on-the-fly* (sob demanda) e sendo assim não guardam informações de execuções anteriores (LEI; CARVER, 2004, 2006).

Outro problema do teste de alcançabilidade, semelhante ao problema do RG, é a explosão de estados de concorrência, pois uma vez que o número de estados cresce

exponencialmente com o número de *threads* e o número de sincronizações a serem exercitadas, mesmo para programas pequenos, pode ser absurdamente grande. Para reduzir esse problema, em trabalhos mais recentes, os autores propõem a adaptação do teste *combinatorial*, para programas seqüenciais, ao teste de alcançabilidade para programas concorrentes (LEI *et al.*, 2007). No contexto de programas seqüenciais, o teste *combinatorial t-way* parte do pressuposto que nem todos os parâmetros do programa contribuem para todas as falhas, e muitas falhas podem ser reveladas considerando apenas a interação entre um número pequeno de parâmetros. A adaptação está em tratar eventos de sincronização/comunicação tal como parâmetros do programa concorrente e combiná-los assim como é feito para programas seqüenciais.

Essa adaptação foi chamada de *teste de alcançabilidade t-way*, em que *t* é o parâmetro, geralmente um valor baixo, que define o nível de combinações feitas com as seqüências de sincronizações. Resumidamente essa técnica busca combinar (de acordo com o valor de *t*) eventos de sincronizações, estabelecendo uma dependência entre eles. O teste configurado para *2-way* (*t* = 2) diz que os eventos precisam ser combinados dois-a-dois. O teste configurado para *1-way* corresponde ao teste de alcançabilidade original. Estudos mostram que com essa abordagem é possível reduzir consideravelmente o conjunto de seqüências de sincronizações a serem executadas e ainda assim eficientemente encontrar defeitos em programas concorrentes (LEI *et al.*, 2007).

3.3.1 Execução determinística de programas concorrentes

A execução determinística é uma característica inerente dos programas seqüenciais, pois para diversas execuções desse tipo de programa com a mesma entrada produzirá sempre o mesmo resultado. No entanto, essa característica não se apresenta em programas concorrentes devido ao não-determinismo. Diversas execuções do programa concorrente com os mesmos valores de entrada podem produzir diferentes seqüências de sincronizações e conseqüentemente, diferentes resultados (ALMASI; GOTTLIEB, 1994; DONGARRA *et al.*, 2002). Nesse sentido, durante a atividade de teste é necessário observar se os diferentes resultados produzidos estão corretos e garantir que todas as seqüências de sincronização foram testadas.

A execução determinística é aplicada no contexto de teste para forçar determinadas seqüências de sincronização a fim de aumentar a taxa de cobertura para um dado critério de teste. No contexto de teste de regressão, ela é importante para que, após revelar e corrigir o defeito, o caso de teste e sua respectiva seqüência de sincronização sejam executados para certificar-se que o defeito foi de fato corrigido. Além disso, todos os casos de teste e suas respectivas seqüências de sincronização precisam ser executadas para certificar-se que a modificação/correção feita no programa não inseriu novos defeitos (LEI; CARVER, 2006; CARVER; TAI, 1991, MCDOWELL; HELMBOLD, 1989).

Alguns autores apresentam propostas para execução determinística de programas concorrentes (HANSEN, 1978; CARVER; TAI, 1986; LEBLANC; MELLOR-CRUMMEY, 1987; RUSSINOVICH; COGSWELL, 1996; CARVER; TAI, 1991, MCDOWELL; HELMBOLD, 1989).

Carver e Tai (1991) definem três fases para obter a execução determinística: (1) definir o formato das seqüências de sincronizações do programa em termos das primitivas de sincronizações disponíveis pela linguagem. É preciso definir cuidadosamente esse formato para fornecer informações suficientes para a execução determinística; (2) Desenvolver uma ferramenta para coletar a seqüências de sincronização durante a execução do programa. Esse processo pode ser feito por instrumentação para registrar o rastro (*trace*) de execução; e (3) Desenvolver uma ferramenta de reexecução. Também é uma opção usar a instrumentação para controlar a execução das primitivas de sincronização desejada.

Em seus trabalhos Carver e Tai (1991) propõem um modelo para reexecução de programas concorrentes com memória compartilhada que utilizam semáforos e/ou monitores. Para isso, após coletar a uma seqüência de sincronização a ser re-executada, o programa é instrumentado especificamente para a execução controlada. Nessa instrumentação as instruções de sincronização são antecedidas e precedidas, respectivamente, pelas funções REQUEST_PERMIT e RELEASE_PERMIT. Quando uma *thread* chega ao ponto de executar um evento de sincronização (*post* ou *wait*, por exemplo) antes ela irá executar a chamada REQUEST_PERMIT. Essa função manipula uma estrutura global que dirá se o evento é o da vez a ser executado. Em caso negativo a *thread* é bloqueada até que outra *thread* a libere. Em caso positivo a *thread* retorna da função REQUEST_PERMIT e em seguida executa o evento de sincronização determinado.

Após executar o evento de sincronização em questão a *thread* executará a chamada `RELEASE_PERMIT` para acordar todas as *threads* bloqueadas, então elas voltam a testar se o evento de sincronização que desejam executar é o evento da vez, bloqueando caso negativo ou seguindo com sua execução caso positivo.

Ambas as funções são *thread-safe*⁸ a fim de proteger – contra acesso concorrente – a estrutura de dados global que apóia a execução controlada. Carver e Tai (1991) mostram ainda que do ponto de vista do resultado da execução controlada é irrelevante a utilização de seqüências de sincronizações total⁹ ou parcialmente¹⁰ ordenadas, apontando vantagens pela utilização de seqüências de sincronização parcialmente ordenadas, devido a simplicidade e a economia de recursos.

3.3.2 Ferramentas de teste para programas concorrentes

Assim como acontece em teste de programas seqüenciais, o teste de programas concorrentes requer que ferramentas automatizem a aplicação de técnicas e critérios. Isso ocorre porque esse tipo de atividade realiza procedimentos complexos que são muitos custosos e, em geral, impraticáveis de serem feitos manualmente.

Existem diversas ferramentas que apóiam, de alguma forma, o teste de programas concorrente de um modo geral. Cada uma trata de uma ou mais atividades relacionadas ao teste de software. Algumas dessas ferramentas são brevemente descritas a seguir e a Tabela 3.2 relaciona suas principais características.

Xab (*X-window Analysis and deBugging*) permitem o monitoramento de programas PVM. Faz isso por meio de instrumentação. Ela possui uma interface gráfica “amigável” para o usuário (BEGUELIN, 1993).

XPVM adiciona uma interface gráfica ao console de comandos do PVM, permitindo o usuário interagir com a execução do programa. Ela atua na tarefa de depuração e análise de

8 Uma função é dita *thread-safe* se ela manipula dados globais e ela pode ser executada concorrente por diferentes *threads*. Porém ela implemente exclusão mútua em toda ou parte dela para que os dados não sejam corrompidos (LEWIS; BERG, 1996).

9 Uma seqüência de sincronização Totalmente Ordenada de um programa é uma seqüência linear e única de eventos de sincronização. Neste caso é possível estabelecer uma relação de ordem entre qualquer dois eventos da seqüência de sincronização (FIDGE, 1996; CARVER; TAI, 2006).

10 Uma seqüência de sincronização Parcialmente Ordenada de um programa é uma coleção de seqüências – uma para cada *thread* ou objeto de sincronização. Assim nem todos os eventos concorrentes do programa são ordenados (FIDGE, 1996; CARVER; TAI, 2006).

desempenho (KOHL; GEIST, 1996).

VisIt (*Visualize it!*) também permite a depuração e análise de desempenho de programas PVM (ILMBERGER; THÜRMEL; WIEDEMANN, 1993).

MDB (*Message-passing DeBugger*) realiza a execução controlada e a depuração de programas PVM. Porém, seu principal objetivo é detectar condição de disputa nesse tipo de programa (DAMODARAN-KAMAL; FRANCIONI, 1994).

ParaGraph permite visualizar programas MPI a fim de observar problemas de desempenho (HEATH, ETHERIDGE, 1991).

XMPI é uma API que fornece funcionalidade para registrar a execução de programas MPI (*trace*), implementado especificamente para a distribuição LA/MPI. Permite a visualização e monitoramento durante, ou após, a execução desses programas (OHIO, 1997).

Umpire realiza a depuração e o monitoramento de programas MPI (VETTER; SUPINSKI, 2000).

TDC Ada apóia a depuração de programas na linguagem Ada, utilizando a execução determinística (TAI; CARVER; OBAID, 1991).

ConAn (*Concurrency Analyser*) gera *drivers* para o teste de unidade em classes de programas concorrentes escritos na linguagem Java. Ela utiliza um relógio externo para a execução determinística (LONG, HOFFMAN, STROOPER, 2003).

CHESS é uma ferramenta para revelar e reproduzir defeitos do tipo Heisenbugs em programas concorrentes (MUSUVATHI; QADEER; BALL, 2007). O termo Heisenbugs surgiu do “Princípio da Incerteza de Heisenberg” (Física Quântica). Esses *bugs* são aqueles que desaparecem ou mudam de comportamento quando se tenta analisá-los. A ferramenta repetitivamente executa o programa garantindo que cada execução terá uma seqüência de sincronização diferente. Para isso faz uso de técnicas de verificação de modelos. Quando revela um defeito, a ferramenta pode reproduzir a seqüência de sincronização para ajudar na depuração.

Vyrd+ faz o teste de unidade de programas Java buscando revelar *bugs* relacionados à concorrência. Para isso a ferramenta produz programas que testam as possíveis sobreposições de chamadas de métodos públicos da unidade (classe) em questão. Ela possui módulos para: instrumentação, *annotation* e monitor em tempo de execução (ELMAS;

TASIRAN; QADEER, 2005).

Bandera é um conjunto de ferramentas para analisar, transformar e visualizar programas Java. Ela recebe como entrada o código fonte Java e um requisito de software formalizado em uma linguagem de especificação temporal própria. Como saída produz uma especificação e um modelo do programa no padrão de outras ferramentas (Spin, SMV e JPF) para executar a verificação de modelo (CORBETT *et al.*, 2000).

Algumas das ferramentas que focam a aplicação de critérios de cobertura para o teste estrutural são:

Astral (Ambiente de Simulação e Teste de pRogramas parALelos) permite o monitoramento da simulação de execução, feita pela análise simbólica de programas PVM. São implementados critérios baseados em fluxo de controle. A ferramenta supõe que o programa é determinístico. É possível, ainda, identificar problemas relacionados ao desempenho (MOREIRA *et al.*, 2004).

Tabela 3.2: Ferramentas que apóiam o teste de software aplicado a programas concorrentes.

Ferramenta	Linguagem / Ambiente	Monitoramento / Visualização	Análise de desempenho	Execução determinística	Depuração	Crítérios estru. de cobertura
<i>Xab</i>	PVM	✓				
<i>XPVM</i>	PVM	✓	✓			
<i>VisIt</i>	PVM	✓	✓			
<i>MDB</i>	PVM	✓		✓	✓	
<i>ParaGraph</i>	MPI	✓	✓			
<i>XMPI</i>	MPI	✓				
<i>Umpire</i>	MPI	✓		✓	✓	
<i>TDC Ada</i>	Ada			✓		
<i>ConAn</i>	Java			✓		
<i>CHESS</i>	Mem. Comp.			✓	✓	
<i>Vyrd+</i>	Java	✓				
<i>Bandera</i>	Java	✓				
<i>Astral</i>	PVM					✓
<i>della pasta</i>	Mem. Comp.			✓		✓
<i>STEPS</i>	PVM	✓		✓		✓
<i>ConTest</i>	Java			✓	✓	
<i>RichTest</i>	Java			✓		✓
<i>ValiPar</i>	PVM/MPI			✓		✓

della pasta (*DELaware PARallel Software Testing Aid*) foi uma das primeiras

ferramentas de teste para análise de caminhos em programas concorrentes com memória compartilhada. Ela possui um analisador estático que gera os possíveis caminhos para cobrir associações entre as variáveis envolvidas na sincronização de tarefas (YANG; SOUTER; POLLOCK, 1998). As principais deficiências dessa ferramenta são o fato dela disponibilizar apenas um critério de cobertura – o Todos-du-caminhos – e esse critério ser adaptado do contexto de programas seqüenciais para o contexto de programas concorrentes. Esse critério engloba em um único critério o conceito de sincronização e comunicação. Além disso, não uma versão dessa ferramenta disponível à comunidade.

STEPS permite o monitoramento através da análise simbólica de cenários de teste e execução controlada em programas paralelos/concorrentes implementados em PVM. Ela também dá suporte a um critério de teste estrutural baseado em fluxo de controle com base nas sincronizações, semelhante ao critério Todas-Arestas-S (KRAWCZYK *et al.*, 1998).

ConTest é uma ferramenta de teste para programas concorrentes implementados na linguagem Java utilizando o recurso de *threads*. O principal objetivo é revelar defeitos relativos à concorrência, além de dar suporte a critérios baseados em fluxo de controle e fluxo de dados (Todo-Nós, Todos-Usos, etc.). Suporta execução determinística parcial ou total. Todo processo de análise estática e instrumentação do programa é feito em nível de *bytecode*, portanto, não é necessário possuir o código fonte para testar um programa. Por fim, a ferramenta é capaz de gerar alguns relatórios, durante e após o teste, permitindo um acompanhamento mais próximo por parte do testador (EDELSTEIN *et al.*, 2003).

RichTest é a ferramenta que implementa a proposta de teste de alcançabilidade proposto por Lei e Carver (2006). Foi implementada em Java e não requer nenhuma modificação na JVM¹¹ para funcionar. O objetivo é cobrir todos os possíveis, e somente os possíveis, caminhos e seqüências de sincronização de um programa *multithread*.

ValiPar é uma ferramenta de teste para programas concorrentes com passagem de mensagem que suporta a aplicação dos critérios de teste definidos por Vergilio *et al.*, (2005). Dada a importância dessa ferramenta para esse trabalho ela será abordada em detalhes na subseção seguinte (SOUZA *et al.*, 2005).

11 JVM – Java Virtual Machine

3.4 PCFG: Um modelo de teste para ambientes de passagem de mensagem

3.4.1 O modelo de teste PCFG

Para aplicar os critérios de teste no contexto de programas paralelos/concorrentes com passagem de mensagem, um modelo de teste foi definido por Vergilio *et al.* (2005). Esse modelo captura as informações necessárias para testar o programa paralelo/concorrente.

O modelo de teste faz algumas considerações:

1. O programa concorrente $PC = \{p^0, p^1, \dots, p^{n-1}\}$ é um conjunto de n processos que executam concorrentemente.
2. o número n de processos é fixo e conhecido em tempo de compilação. Ou seja, para construir o modelo é necessário conhecer o número de processos que o programa executará;
3. todos os processos são criados no momento da inicialização do programa concorrente. Isso significa que não é tratada a criação dinâmica (sobre demanda) de processo. Todos os processos do programa são criados, de uma única vez, no início da execução do programa;
4. os processos se comunicam/sincronizam através de primitivas *send* e *receive*. A comunicação pode ocorrer de duas formas: (1) ponto-a-ponto: em que um processo envia uma mensagem para outro específico; ou (2) coletivamente: em que um processo envia uma mensagem para todos, ou um grupo, de processos. Nesse segundo caso, o modelo considera a comunicação coletiva como diversos *send/receive* básicos.

O modelo consiste em construir um **PCFG** (*Parallel Control Flow Graph*, ou Grafo de Fluxo de Controle Paralelo) a partir do **CFG'** correspondente a cada processo p^i que compõe o programa concorrente, interconectando-os por arestas que representam a comunicação/sincronização entre os processos. Na representação gráfica do **PCFG** as arestas que representam comunicação/sincronização são mostradas por setas tracejadas. O **CFG'** é um grafo de fluxo de controle referente ao processo p^i , baseado no **CFG** definido para programas sequenciais (SOUZA et al., 2005).

Após a construção do **PCFG** as seguintes informações podem ser obtidas:

- N : Conjunto de nós do **PCFG**;
- E : Conjunto de arestas do **PCFG**;
- N_s : Conjunto de nós que possuem a função *send*;
- N_r : Conjunto de nós que possuem a função *receive*;
- E_i : Conjunto de arestas intra-processos, ou seja, arestas que conectam nós pertencentes ao mesmo processo p^i ;
- E_s : Conjunto de arestas inter-processos, ou seja, arestas que representam comunicação/sincronização entre dois processos;
- R_i^p : Conjunto de nós que podem receber a mensagem enviada pelo nó $n_i^p \in N_s$, em que n_i^p representa o i -ésimo nó do processo p . Esse conjunto é importante para representar os nós que podem receber uma dada mensagem.

A título de ilustração considere o programa GCD (*Greatest Common Divisor*), mostrado na Figura 3.8, que calcula o máximo divisor comum entre três valores assim como o programa seqüencial GCD mostrado na Figura 2.1, sendo agora implementando no ambiente de passagem de mensagem PVM.

Ao iniciar a execução desse programa o processo mestre, representado por p^m (Figura 3.8(a)), cria três processos escravos (p^0 , p^1 e p^2) (linha 9 na Figura 3.8(a)) que executam o programa **gcd** (Figura 3.8(b)). Ao iniciar sua execução, cada processo escravo aguarda através da função **pvm_recv** (linha 7 na Figura 3.8(b)), que implementa a primitiva *receive* de modo bloqueante, o envio (**pvm_send**, linhas 14, 18 e 28 na Figura 3.8(a)) de dois valores pelo processo mestre e então calcula o máximo divisor comum desses dois valores. Por fim os processos escravos enviam (**pvm_send**, linha 19 na Figura 3.8(b)) o resultado para o processo mestre, que o recebe de forma não-determinística através das funções **pvm_recv** (linhas 19, 21 e 29 na Figura 3.8(a)). É possível, ainda, que o processo mestre utilize apenas dois dos três processos escravos para obter o máximo divisor comum entre os três valores, dependendo dos valores e da condição presente na linha 23 (Figura 3.8(a)).

Na Figura 3.9 é mostrado o **PCFG** correspondente ao programa da Figura 3.8. A Tabela 3.3 apresenta as informações obtidas a partir da análise estática do **PCFG** desse

programa.

<pre> 1 /*Master program GCD - mgcd.c */ 2 #include <stdio.h> 3 #include <pvm3.h> 4 int main() { 5 int x,y,z; 6 int S[3]; 7 printf("Enter x, y and z:"); /*1*/ 8 scanf("%d %d %d", &x,&y,&z); /*1*/ 9 pvm_spawn("gcd", 10 (char**)0,0,"",3,S); /*1*/ 11 pvm_init(PvmDataDefault); /*2*/ 12 pvm_pkint(&x,1,1); /*2*/ 13 pvm_pkint(&y,1,1); /*2*/ 14 pvm_send(S[0],1); /*2*/ 15 pvm_init(PvmDataDefault); /*3*/ 16 pvm_pkint(&y,1,1); /*3*/ 17 pvm_pkint(&z,1,1); /*3*/ 18 pvm_send(S[1],1); /*3*/ 19 pvm_recv(-1,2); /*4*/ 20 pvm_upkint(&x,1,1); /*4*/ 21 pvm_recv(-1,2); /*5*/ 22 pvm_upkint(&y,1,1); /*5*/ 23 if ((x>1)&&(y>1)) { /*6*/ 24 pvm_init(PvmDataDefault); /*7*/ 25 pvm_pkint(&x,1,1); /*7*/ 26 pvm_pkint(&y,1,1); /*7*/ 27 pvm_send(S[2],1); /*7*/ 28 pvm_recv(-1,2); /*8*/ 29 pvm_upkint(&z,1,1); /*8*/ 30 } else { /*9*/ 31 pvm_kill(S[2]); /*9*/ 32 z = 1; /*9*/ 33 } 34 printf("%d",z); /*10*/ 35 pvm_exit(); /*10*/ </pre> <p style="text-align: center;">(a)</p>	<pre> 1 /*Slave program GCD - gcd.c */ 2 #include <stdio.h> 3 #include "pvm3.h" 4 int main() { 5 int tid,x,y,numt,aux; 6 tid = pvm_parent(); /*1*/ 7 pvm_recv(tid,-1); /*2*/ 8 pvm_upkint(&x,1,1); /*2*/ 9 pvm_upkint(&y,1,1); /*2*/ 10 while (x != y) { /*3*/ 11 if (x<y) /*4*/ 12 y = y-x; /*5*/ 13 else 14 x = x-y; /*6*/ 15 } /*7*/ 16 pvm_init(PvmDataDefault); /*8*/ 17 pvm_pkint(&x,1,1); /*8*/ 18 pvm_send(tid,2); /*8*/ 19 pvm_exit(); /*9*/ 20 } </pre> <p style="text-align: center;">(b)</p>
---	--

Figura 3.8: Implementação do problema GCD utilizando o ambiente de passagem de mensagem PVM. (a) Programa referente ao processo Mestre; e (b) programa referente a cada um dos três processos Escravos.

Assim como o grafo *Def-Use*, definido para programas sequenciais, o *PCFG* pode agregar informações de fluxo de dados para viabilizar a aplicação de critérios de teste baseados em fluxo de dados. Além do *c-use* (uso computacional) e *p-use* (uso predicativo), Vergilio *et al.* (2005) definem outro possível uso para uma variável: o *s-use* (uso comunicacional).

Figura 3.9: *GFCP* referente ao programa *GCD* da Figura 3.8 (SOUZA et al., 2008).

Uso comunicacional (*s-use*): ocorre quando uma variável é referenciada em funções de comunicação/sincronização, tais como *send* e *receive*. O *s-use* está especificamente relacionado a programas concorrentes. No grafo *PCFG* o uso comunicacional de uma variável é representado pelo nome da variável colocada entre parênteses, precedida da palavra “*s-use*” e posicionada junto à aresta onde o *s-use* ocorre.

Tabela 3.3: Conjuntos do **PCFG** referente ao programa GCD (SOUZA et al., 2008).

$n = 4$	
$Prog = \{p^m, p^0, p^l, p^2\}$	
$N = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, 10^m, 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 1^l, 2^l, 3^l, 4^l, 5^l, 6^l, 7^l, 8^l, 9^l\}$	
$N_s = \{2^m, 3^m, 7^m, 8^0, 8^l, 8^2\}$ (nós que representam <code>pvm_send()</code>)	
$N_r = \{4^m, 5^m, 8^m, 2^0, 2^l, 2^2\}$ (nós que representam <code>pvm_recv()</code>)	
$R_2^m = \{2^0, 2^l, 2^2\}$	$R_8^0 = \{4^m, 5^m, 8^m\}$
$R_3^m = \{2^0, 2^l, 2^2\}$	$R_8^l = \{4^m, 5^m, 8^m\}$
$R_7^m = \{2^0, 2^l, 2^2\}$	$R_8^2 = \{4^m, 5^m, 8^m\}$
$E_i^m = \{(1^m, 2^m), (2^m, 3^m), (3^m, 4^m), (4^m, 5^m), (5^m, 6^m), (6^m, 7^m), (7^m, 8^m), (8^m, 10^m), (6^m, 9^m), (9^m, 10^m)\}$	
$E_i^0 = \{(1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (4^0, 6^0), (5^0, 7^0), (6^0, 7^0), (7^0, 3^0), (3^0, 8^0), (8^0, 9^0)\}$	
$E_i^l = \{(1^l, 2^l), (2^l, 3^l), (3^l, 4^l), (4^l, 5^l), (4^l, 6^l), (5^l, 7^l), (6^l, 7^l), (7^l, 3^l), (3^l, 8^l), (8^l, 9^l)\}$	
$E_i^2 = \{(1^2, 2^2), (2^2, 3^2), (3^2, 4^2), (4^2, 5^2), (4^2, 6^2), (5^2, 7^2), (6^2, 7^2), (7^2, 3^2), (3^2, 8^2), (8^2, 9^2)\}$	
$E = \{(2^m, 2^0), (2^m, 2^l), (2^m, 2^2), (3^m, 2^0), (3^m, 2^l), (3^m, 2^2), (7^m, 2^0), (7^m, 2^l), (7^m, 2^2), (2^m, 2^0), (2^m, 2^l), (8^0, 4^m), (8^0, 5^m), (8^0, 8^m), (8^l, 4^m), (8^l, 5^m), (8^l, 8^m), (8^2, 4^m), (8^2, 5^m), (8^2, 8^m)\}$	
$E = E_i^0 \cup E_i^l \cup E_i^2 \cup E_s$	

Estendendo as associações definidas para programas sequenciais (associação c-uso e associação p-uso) o modelo propõe mais três possíveis associações.

Associação s-use: é definida pela tripla $(n_i^{p_f}, (n_j^{p_f}, n_k^{p_s}), x)$, tal que a variável x é definida no nó $n_i^{p_f}$, a aresta $(n_j^{p_f}, n_k^{p_s})$ faz um *s-use* de x , e o caminho de $n_i^{p_f}$ até $(n_j^{p_f}, n_k^{p_s})$ é livre de definição em relação à variável x . Observe que a aresta que caracteriza s-uso $(n_j^{p_f}, n_k^{p_s})$ refere-se a aresta intra-processos $(p_f \text{ e } p_g)$ de *PC*.

Associação s-c-use: é definida pela quintupla $(n_i^{p_f}, (n_j^{p_f}, n_k^{p_s}), n_l^{p_s}, x^{p_f}, x^{p_s})$, tal que haja uma associação *s-use* $(n_i^{p_f}, (n_j^{p_f}, n_k^{p_s}), x^{p_f})$ e uma associação c-uso $(n_k^{p_s}, n_l^{p_s}, x^{p_s})$. A variável x aparece duas vezes na associação $(x^{p_f} \text{ e } x^{p_s})$ porque ela está sendo definida no processo p_f e usada (*c-use*) no processo p_g e por isso é possível que a variável receba nomes diferentes nesses processos.

Associação s-p-use: é dada pela quintupla $(n_i^{p_f}, (n_j^{p_f}, n_k^{p_s}), (n_l^{p_s}, n_m^{p_s}), x^{p_f}, x^{p_s})$, tal que haja uma associação s-uso $(n_i^{p_f}, (n_j^{p_f}, n_k^{p_s}), x^{p_f})$ e uma associação *p-use* $(n_k^{p_s}, (n_l^{p_s}, n_m^{p_s}), x^{p_s})$. A variável x aparece duas vezes na associação $(x^{p_f} \text{ e } x^{p_s})$ pelo mesmo motivo descrito na associação anterior.

As duas últimas associações foram definidas dada a seguinte motivação (SOUZA *et al.*, 2008): a associação c-uso e p-uso ocorrem no mesmo processo, ou seja, a definição e o uso (*c-use* ou *p-use*) ocorrem em nós distintos de um mesmo processo. Dessa forma, quando aplicados em programas concorrentes critérios de teste definidos para essas associações nenhuma comunicação/sincronização é exercitada e assim não é possível detectar defeitos com relação à comunicação/sincronização. Com isso a proposta é observar aspectos de comunicação/sincronização nas associações do tipo *c-use* e *p-use*.

Quanto aos critérios de testes aplicáveis a programas concorrentes em ambientes de passagem de mensagem é possível aplicar aqueles definidos para programas seqüenciais, desde que aplicados separadamente a cada *CFG'*. Porém, dado que o objetivo desse modelo é testar a comunicação/sincronização presentes no *PCFG*, a partir das considerações e extensões apresentadas, os seguintes critérios de teste são propostos (VERGILIO *et al.*, 2005; SOUZA *et al.*, 2008):

Critérios baseados em fluxo de controle e comunicação

Critério All-nodes-r: requer que todos os nós $n_i^p \in N_r$ sejam exercitados ao menos uma vez. Ou seja, requer que toda instrução que executa o recebimento (*receive*) de uma mensagem seja exercitada.

Critério All-nodes-s: requer que todos os nós $n_i^p \in N_s$ sejam exercitados ao menos uma vez. Ou seja, requer que toda instrução que executa o envio (*send*) de uma mensagem seja exercitada.

Critério All-nodes: requer que todos os nós $n_i^p \in N$ sejam exercitados. Ou seja, todos os nós do *PCFG* devem ser testados ao menos uma vez.

Critério All-edges-s: requer a que todas as arestas $(n_i^f, n_j^g) \in E_s$ sejam executadas. Ou seja, toda aresta referente à comunicação/sincronização entre processos deve ser exercitada.

Critério All-edges: requer a que todas as arestas $(n_i, n_j) \in E$ sejam executadas. Em outras palavras, requer que todas, sem exceção, as arestas do grafo *PCFG* sejam exercitadas.

Critérios baseados em fluxo de dados e passagem de mensagem

Critério All-defs-s: para cada nó n_i^p que possua definição de uma ou mais variáveis uma associação inter-processos (*s-c-uso* ou *s-p-uso*) deve ser exercitada em relação a cada uma das variáveis. Caso tais associações não existirem, outra associação (*c-use*, *p-use* ou *s-use*) deve ser utilizada para exercitar a definição da variável.

Critério All-s-uses: requer que todas as associações s-uso sejam exercitadas.

Critério All-s-c-uses: requer que todas as associações s-c-uso sejam exercitadas.

Critério All-s-p-uses: requer que todas as associações s-p-uso sejam exercitadas.

Selecionando-se o(s) critério(s) de teste a ser(em) utilizado(s) é necessário determinar os elementos requeridos para tal(tais) critério(s). Na Tabela 3.4 são mostrados alguns elementos requeridos para todos os critérios de teste definidos.

É interessante notar que apenas um caso de teste pode cobrir um ou mais elementos requeridos para um ou mais critérios. Para exemplificar, imagine a execução do caso de teste

$t_1 = (\{x=1, y=2, z=1\}, \{saida=1\})$ para o programa da Figura 3.8. Suponha que o caminho executado Π por esse caso de teste é dado por $\Pi = (\pi^m, \pi^0, \pi^1)$:

$\pi^m = \{1, 2, 3, 4, 5, 6, 9, 10\}$, $\pi^0 = \{1, 2, 3, 4, 5, 7, 3, 8, 9\}$ e $\pi^1 = \{1, 2, 3, 4, 6, 7, 3, 8, 9\}$. O processo p_2 não chega a ser executado, pois o resultado já é obtido por p_0 e p_1 .

Considerando o critério All-s-uses, o caso de teste t_1 cobriria os seguintes elementos requeridos: $(1^m, (2^m, 2^0), x, y)$ e $(1^m, (3^m, 2^1), y, z)$ e, dependendo do “casamento” entre os nós 4^m e 5^m com os nós 8^0 e 8^1 , cobriria os elementos requeridos $(2^0, (8^0, 4^m), x)$ e $(2^0, (8^1, 5^m), x)$, ou $(6^1, (8^1, 4^m), x)$ e $(6^1, (8^0, 5^m), x)$. Se o critério Todos-S-c-Usos estivesse em consideração o mesmo caso de teste t_1 cobriria os seguintes elementos requeridos: $(1^m, (2^m, 2^0), 5^0, x, x)$, $(1^m, (2^m, 2^0), 5^0, y, y)$, $(1^m, (3^m, 2^1), 6^1, y, x)$, entre outros.

Tabela 3.4: Alguns elementos requeridos pelos critérios estruturais para o programa GCD (SOUZA et al., 2008).

Cr�terios	Elementos requeridos
<i>All-nodes-s</i>	$2^m, 3^m, 7^m, 8^0, 8^1, 8^2$
<i>All-nodes-r</i>	$4^m, 5^m, 8^m, 2^0, 2^1, 2^2$
<i>All-nodes</i>	$1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, 10^m, 1^0, 2^0, 3^0, \dots, 1^1, 2^1, 3^1, \dots, 9^2$
<i>All-edges-s</i>	$(2^m, 2^0), (2^m, 2^1), (2^m, 2^2), (3^m, 2^0), (3^m, 2^1), (3^m, 2^2), (7^m, 2^0), (7^m, 2^1), (7^m, 2^2), (2^m, 2^0), (2^m, 2^1), (8^0, 4^m), (8^0, 5^m), (8^0, 8^m), (8^1, 4^m), (8^1, 5^m), (8^1, 8^m), (8^2, 4^m), (8^2, 5^m), (8^2, 8^m)$
<i>All-edges</i>	$(1^m, 2^m), (2^m, 3^m), (3^m, 4^m), \dots, (1^0, 2^0), (2^0, 3^0), \dots, (1^1, 2^1), (2^1, 3^1), \dots, (2^m, 2^0), (2^m, 2^1), \dots$
<i>All-defs</i>	$(8^m, 10^m, z), (2^0, 5^0, x), (2^0, 6^0, x), (2^0, (3^0, 4^0), x), (2^0, 6^0, y), \dots$
<i>All-defs-S</i>	$(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, y, y), (1^m, (2^m, 2^0), (4^0, 5^0), y, y), (1^m, (3^m, 2^0), 5^0, z, y), \dots$
<i>All-c-uses</i>	$(1^m, 10^m, z), (8^m, 10^m, z), (2^0, 8^0, x), \dots$
<i>All-p-uses</i>	$(4^m, (6^m, 7^m), x), (4^m, (6^m, 9^m), x), (5^m, (6^m, 7^m), y), (5^m, (6^m, 9^m), y), (2^0, (3^0, 4^0), x), (2^0, (3^0, 8^0), y), \dots$
<i>All-s-uses</i>	$(1^m, (2^m, 2^0), x, y), (1^m, (2^m, 2^1), x, y), (1^m, (3^m, 2^0), y, z), (4^m, (7^m, 2^2), x), (5^m, (7^m, 2^0), y), (5^m, (7^m, 2^1), y), \dots$
<i>All-s-c-uses</i>	$(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, x, x), (1^m, (2^m, 2^0), 5^0, y, y), (1^m, (2^m, 2^0), 6^0, y, y), (1^m, (2^m, 2^1), 6^1, x, x), (1^m, (3^m, 2^1), 6^1, x, x), (2^0, (8^0, 8^m), 10^m, x, z), \dots$
<i>All-s-p-uses</i>	$(1^m, (2^m, 2^0), (3^0, 4^0), x, x), (1^m, (2^m, 2^0), (3^m, 8^0), x, x), (1^m, (2^m, 2^0), (4^0, 5^0), x, x), (1^m, (3^m, 2^0), (3^0, 4^0), z, y), (5^m, (7^m, 2^0), (3^0, 4^0), y, x), (2^0, (8^0, 4^m), (6^m, 7^m), x, y), \dots$

3.4.2 ValiPar: Ferramenta de apoio ao teste de programas concorrente

A ValiPar uma ferramenta orientada a sesso de teste, o que permite ao usurio iniciar uma atividade de teste, par-la e retom-la quando desejar (SOUZA et al., 2008). De modo geral, a ValiPar fornece trs funcionalidades: (1) criar sesso de teste; (2) salvar e executar casos de testes; e (3) avaliar a cobertura de teste de acordo com um critrio de teste. A arquitetura da ValiPar apresentada na Figura 3.10. Quatro mdulos compem a ValiPar: *ValiInst*, *ValiElem*, *ValiExec* e *ValiEval*. A comunicao entre esses mdulos se d atravs de arquivos textos.

Figura 3.10: Arquitetura da ferramenta ValiPar (SOUZA et al., 2008).

A proposta da ValiPar é apoiar diversos ambientes de passagem de mensagem, porém atualmente há duas instâncias da ferramenta implementadas: (1) a ValiPVM que trata programas paralelos escritos na linguagem C ANSI que utilizam a API PVM (SOUZA *et al.*, 2005); e (2) a ValiMPI (HAUSEN, 2005) que trata os programas paralelos escritos na linguagem C ANSI que utilizam a API MPI. Ambas as ferramentas foram implementadas no modo texto para o sistema Linux/Unix. Uma interface gráfica foi construída e atualmente está instanciada para a ferramenta ValiPVM (SOUZA; SAWABE *et al.*, 2008). A seguir cada um dos módulos que compõem a ValiPar é detalhado. Para exemplificar o funcionamento de cada módulo são ilustradas as informações geradas para a ferramenta ValiPVM, considerando programa *GCD* mostrado na Figura 2.2.

O módulo *ValiInst* é responsável por executar a análise estática do código fonte do programa concorrente e assim extrair informações de fluxo de controle e fluxo de dados, que serão pertinentes à atividade de teste. Entre as informações obtidas estão o *PCFG* para o programa paralelo com informações a respeito de nós, arestas, definição e uso de variáveis e arestas que correspondem a comandos *send* e *receive*. Esse módulo também é responsável por instrumentar o código com instruções para gerar *traces* (ou registros) da execução no

momento que o código instrumentado for compilado e executado. Para realizar a tarefa de instrumentação o módulo *ValiInst* faz uso da IDeL (descrita na Seção 3.3).

Na Figura 3.11 é mostrada a representação em texto do **CFG** para o processo mestre. No Apêndice B é apresentado o programa instrumentado produzido pela execução do módulo *ValiInst*.

O módulo *ValiElem* gera os elementos requeridos para os critérios de cobertura definidos por Vergilio *et al.*, (2005). Esses elementos requeridos são gerados a partir do **PCFG** e das informações de fluxo de dados, gerados pelo módulo *ValiInst*. Para esse propósito, dois outros grafos são usados: o grafo reduzido de herdeiros (CHUSHO, 1987) e o grafo(*i*). Esses grafos foram definidos com base na ferramenta de teste PokeTool (CHAIM, 1991).

No grafo reduzido de herdeiros, todas as arestas são primitivas. O grafo é construído com base no fato que há arestas pertencentes ao **CFG** que são sempre executadas quando outras são executadas. Se cada caminho completo que inclui a aresta *a* também inclui a aresta *b*, então *b* é chamada herdeira de *a* e, *a* é chamada ancestral de *b*, pois *b* encapsula informações a respeito da execução de *a*. Em outras palavras, uma aresta que é sempre executada quando alguma outra é executada é chamada de aresta herdeira. Uma aresta é chamada primitiva, se não é herdeira de nenhuma outra. A ValiPar adaptou o grafo para o contexto de programas paralelos (SOUZA *et al.*, 2005).

1 digraph gfc {	30 /* pusage of x at 6 */
2 node [shape = circle] 1;	31 /* pusage of y at 6 */
3 /* definition of x at 1 */	32 node [shape = circle] 7;
4 /* definition of y at 1 */	33 /* m_pkint of &x,1,1 at 7 */
5 /* definition of z at 1 */	34 /* m_pkint of &y,1,1 at 7 */
6 /* defref of S at 1 */	35 /* m_send of S[2],1 at 7 */
7 node [shape = circle] 2;	36 /* susage of x at 7 */
8 /* m_pkint of &x,1,1 at 2 */	37 /* susage of y at 7 */
9 /* m_pkint of &y,1,1 at 2 */	38 /* defref of S[2] at 7 */
10 /* m_send of S[0],1 at 2 */	39 node [shape = circle] 8;
11 /* susage of x at 2 */	40 /* m_rcv of -1,2 at 8 */
12 /* susage of y at 2 */	41 /* m_upkint of &z,1,1 at 8 */
13 /* defref of S[0] at 2 */	42 /* definition of z at 8 */
14 node [shape = circle] 3;	43 node [shape = circle] 9;
15 /* m_pkint of &y,1,1 at 3 */	44 /* defref of S[2] at 9 */
16 /* m_pkint of &z,1,1 at 3 */	45 /* definition of z at 9 */
17 /* m_send of S[1],1 at 3 */	46 node [shape = circle] 10;
18 /* susage of y at 3 */	47 /* csage of z at 10 */
19 /* susage of z at 3 */	48 1 -> 2;
20 /* defref of S[1] at 3 */	49 2 -> 3;
21 node [shape = circle] 4;	50 3 -> 4;
22 /* m_rcv of -1,2 at 4 */	51 4 -> 5;
23 /* m_upkint of &x,1,1 at 4 */	52 5 -> 6;
24 /* definition of x at 4 */	53 6 -> 7;
25 node [shape = circle] 5;	54 6 -> 9;
26 /* m_rcv of -1,2 at 5 */	55 7 -> 8;
27 /* m_upkint of &y,1,1 at 5 */	56 8 -> 10;
28 /* definition of y at 5 */	57 9 -> 10;
29 node [shape = circle] 6;	58 }

Figura 3.11: Representação textual do *PCFG*, gerada pela ferramenta *ValiPar*, referente ao programa do processo mestre pertencente ao programa paralelo GCD (SOUZA et al., 2008).

O conceito de arestas de sincronização foi incluído ao conceito de arestas primitivas. Usando ambos os conceitos é possível minimizar o número de arestas requeridas pela *ValiPar*.

Um grafo(*i*) é construído para cada nó que contém definição de variável. A proposta dele é obter todos os caminhos livres de definição com relação a cada variável definida em um dado nó. Então, um dado nó *k* irá pertencer a um grafo(*i*) se existe ao menos um caminho de *i* para *k* em que não haja redefinição da variável *x*, definida no nó *i*.

Para cada elemento requerido, o *ValiElem* também produz um descritor. Um descritor é uma expressão regular que descreve um caminho que exercita (ou cobre) um elemento requerido. Por exemplo, o descritor para os elementos requeridos pelo critério Todos-Nós é dado pela expressão:

$$N * n_i^p N *$$

em que *N* é o conjunto de nós do GFC^p . Um nó requerido n_i^p será exercitado (ou coberto) pelo caminho π^p , se π^p inclui n_i^p . De maneira similar, uma expressão regular é definida para cada elemento requerido pelo critério de teste. Os descritores

descrevem todos os caminhos no grafo que cobrem o elemento requerido correspondente e é usado posteriormente pelo módulo *ValiEval*.

Na Figura 3.12 são mostrados os elementos requeridos gerados para o critério Todas-Arestas-S, considerando o programa GCD da Listagem 3.1.

1	1) 2-0 2-1	9	9) 7-0 2-3	17	17) 8-2 8-0
2	2) 2-0 2-2	10	10) 8-1 4-0	18	18) 8-2 2-1
3	3) 2-0 2-3	11	11) 8-1 5-0	19	19) 8-2 2-3
4	4) 3-0 2-1	12	12) 8-1 8-0	20	20) 8-3 4-0
5	5) 3-0 2-2	13	13) 8-1 2-2	21	21) 8-3 5-0
6	6) 3-0 2-3	14	14) 8-1 2-3	22	22) 8-3 8-0
7	7) 7-0 2-1	15	15) 8-2 4-0	23	23) 8-3 2-1
8	8) 7-0 2-2	16	16) 8-2 5-0	24	24) 8-3 2-2

Figura 3.12: Elementos requeridos pelo critério Todas-Arestas-S para o programa GCD.

O módulo *ValiExec* executa o programa instrumentado com o caso de teste fornecido pelo usuário. Um *script*¹² é usado para inicializar o ambiente de passagem de mensagem, no caso do exemplo o PVM, antes da execução do programa paralelo. O *ValiExec* armazena a entrada e a saída do caso de teste, os parâmetros de execução e os respectivos *traces* (registros) de execução. O *trace* de execução inclui o caminho executado por cada processo paralelo, bem como as seqüências de sincronização. Ele será usado pelo módulo *ValiEval* para determinar quais elementos foram cobertos.

Depois da execução, o testador pode visualizar as saídas e trace de execução para determinar se a saída obtida é a esperada. Em caso negativo, um defeito foi detectado e deve ser corrigido antes de continuar com a atividade de teste.

Na Figura 3.13 é mostrado os *traces* (*rastros*) gerados para o programa GCD com o seguinte caso de teste: $\langle (1,3,5), 1 \rangle$. Para este caso de teste o processo 3 não foi executado.

O módulo *ValiExec* também possibilita a execução controlada do programa paralelo em teste. Este recurso é útil para repetir a execução de um caso de teste, garantindo que a mesma seqüência de sincronizações seja exercitada. Para isso funções foram implementadas as quais garante que as seqüências de sincronizações se repitam para o mesmo caso de teste.

O módulo *ValiEval* avalia a cobertura obtida pelo caso de teste em relação a um dado

¹² Script: linguagem de programação interpretadas e por isso depende de um interpretador para ser executado. É usada principalmente para controlar a execução de uma ou várias outras aplicações.

critério. *ValiEval* usa os descritores e os elementos requeridos gerados pelo *ValiElem* e os caminhos executados (*traces*) pelos casos de teste para verificar quais elementos requeridos – para um dado critério de teste – foram exercitados. O módulo implementa um autômato associado aos descritores. Então, um elemento requerido está coberto se um caminho Π executado é reconhecido pelo seu autômato correspondente. A taxa de cobertura (porcentagem de elementos requeridos cobertos) e uma lista de elementos requeridos não cobertos em relação ao critério de teste selecionado é mostrado ao fim da execução desse módulo.

1	traceP0:										
2	1-0	2-0	3-0	4-0	8-1	4-0	5-0	8-2	5-0	6-0	9-0
3	10-0										
4											
5	traceP1:										
6	1-1	2-1	2-0	2-1	3-1	4-1	5-1	3-1			
7	4-1	5-1	3-1	7-1	8-1	9-1					
8											
9	traceP2:										
10	1-2	2-2	3-0	2-2	3-2	4-2	5-2	3-2			
11	4-2	6-2	3-2	4-2	5-2	3-2	7-2	8-2	9-2		
12											
13	traceP3:										

Figura 3.13: *Traces (rastros) de execução do programa GCD (SOUZA et al., 2008).*

Nesse momento é importante observar que alguns elementos requeridos podem ser não-executáveis, ou seja, não há nenhum fluxo que exercite um caminho que cubra o elemento requerido. A ferramenta não é capaz de revelar quais elementos requeridos são não-executáveis, ficando sobre a responsabilidade do testador identificá-los.

Na Figura 3.14 são mostradas essas informações considerando o critério Todas-Arestas-S para o programa GCD. Esses resultados foram gerados após a execução dos quatro casos de testes mostrados na Figura 3.15.

1	Required elements not covered - criterion all_edgesS:
2	
3	Required element 2 not covered: 2) 2-0 2-2
4	Required element 3 not covered: 3) 2-0 2-3
5	Required element 4 not covered: 4) 3-0 2-1
6	Required element 6 not covered: 6) 3-0 2-3
7	Required element 8 not covered: 8) 7-0 2-2
8	Required element 9 not covered: 9) 7-0 2-3
9	Required element 11 not covered: 11) 8-1 4-0
10	Required element 13 not covered: 13) 8-1 2-2
11	Required element 14 not covered: 14) 8-1 2-3
12	Required element 15 not covered: 15) 8-2 4-0
13	Required element 17 not covered: 17) 8-2 8-0
14	Required element 18 not covered: 18) 8-2 2-1
15	Required element 19 not covered: 19) 8-2 2-3
16	Required element 20 not covered: 20) 8-3 5-0
17	Required element 21 not covered: 21) 8-3 8-0
18	Required element 23 not covered: 23) 8-3 2-1
19	Required element 24 not covered: 24) 8-3 2-2
20	
21	Coverage: 29.17%

Figura 3.14: Análise de cobertura feita pela ferramenta ValiPar para o critério Todas-Arestas-S para o programa GCD (SOUZA et al., 2008).

1	input1.tes:	9	input3.tes:
2	1 3 5	10	5 1 2
3	output1.tes:	11	output3.tes:
4	1	12	1
5	input2.tes:	13	input4.tes:
6	2 8 4	14	4 4 4
7	output2.tes:	15	output4.tes:
8	2	16	4

Figura 3.15: Casos de Teste executados na ferramenta ValiPar (SOUZA et al., 2008).

3.5 Considerações finais

Produzir software de qualidade é necessário e, por que não dizer, sempre louvável. O teste de software busca guiar e quantificar essa qualidade. Porém, quando se trata do teste de software aplicado a programas concorrentes, há desafios adicionais que tornam a atividade ainda mais difícil e custosa. Os principais desafios apontados neste capítulo são: desenvolver técnicas de análise estática, tratar as sincronizações, comunicações e fluxo de dados nesses programas, gerar uma representação do programa concorrente que capture as informações pertinentes para a atividade de teste, projetar critérios de cobertura para programas concorrentes e ser capaz de reproduzir uma execução com a mesma entrada de teste.

Este capítulo apresentou alguns conceitos da área de teste software de um modo geral, descrevendo as principais técnicas e detalhando a técnica de teste estrutural. Em seguida,

foram apresentados os conceitos e os trabalhos relacionados ao tema teste de software aplicado a programas concorrentes. Nessa seção algumas ferramentas foram descritas de forma superficial.

Por fim o capítulo apresentou em detalhes o modelo de teste *PCFG* e a ferramenta ValiPar aplicados o teste de programas concorrentes em ambiente de passagem de mensagem. Os conceitos e as idéias mostrados aqui, além dos outros trabalhos relacionados, foram importantes para definir o modelo de teste *PCFGsm* proposto para o contexto de programas concorrentes com memória compartilhada.

O Capítulo 4 apresenta a proposta do modelo *PCFGsm*, enquanto o Capítulo 5 apresenta a ferramenta ValiPThread concebida para automatizar a aplicação do modelo *PCFGsm*.

Critérios de teste estrutural aplicados a programas multithreads

4.1 Considerações iniciais

A necessidade de produzir programas com qualidade exige constante aperfeiçoamento das técnicas de engenharia de software e conseqüentemente das técnicas de teste. Com esse objetivo, modelos de teste têm sido propostos com certa regularidade (TAYLOR *et al.*, 1992; YANG; POLLOCK, 1997; YANG; POLLOCK, 2003; EDELSTEIN *et al.*, 2003; VERGILIO *et al.*, 2005; LEI; CARVER, 2006). Este capítulo apresenta um novo modelo de teste estrutural e critérios de cobertura aplicáveis a programas concorrentes que utilizam o paradigma de memória compartilhada. O modelo de teste definido neste trabalho compreende: (1) a análise estática do programa em teste evidenciando elementos estruturais, (2) o levantamento de elementos requeridos de acordo com critérios de testes adaptados para o contexto de programas *multithreads* e (3) um método para determinar a comunicação realizada pelas *threads* a partir das sincronizações efetuadas.

A proposta desse novo modelo é baseada principalmente no modelo **PCFG** (*Parallel Control Flow Graph*) proposto em (VERGILIO *et al.*, 2005) para programas com passagem

de mensagens, além do modelo **PPFG** (*Parallel Program Flow Graph*) proposto em (YANG, 2003) e do modelo de teste de alcançabilidade proposto em (LIE; CARVER, 2006). Em comparação com o modelo **PCFG**, as principais contribuições, além do diferente contexto de aplicação, são: (1) a abordagem que considera a sincronização realizada pela inicialização e finalização de *threads*; (2) a utilização de *timestamps* (rótulos de tempo) para determinar comunicação entre *threads*; e (3) a separação dos critérios de cobertura referentes à sincronização e à comunicação.

Uma vez que no contexto de programas concorrentes com memória compartilhada a comunicação se dá de forma implícita foi preciso investigar uma maneira de determinar a comunicação feita pelas *threads*. Para isso é proposta uma abordagem que considera o *trace* (rastro) de execução do programa atribuindo *timestamps* a eventos concorrentes. Isso permite estabelecer uma ordem entre os eventos concorrentes e em seguida obter os pares de comunicação exercitados.

Outro aspecto abordado nesse capítulo é a adaptação de um mecanismo para permitir a execução determinística desses programas, aspecto muito importante para a condução dos testes. A execução determinística permite a reprodução da execução de um caso de teste incluindo dado de teste e a sequência de sincronização executada. A execução determinística tem três propósitos: (1) permitir a reprodução fiel de um caso de teste, auxiliando a atividade de depuração; (2) permitir a realização de testes de regressão¹³; e (3) forçar a execução de seqüências de sincronizações específicas, a fim de aumentar a cobertura para o critério selecionado.

No decorrer do capítulo, essas definições serão ilustradas com um programa exemplo – um programa que implementa o problema do produtor-consumidor com *buffer* limitado.

4.2 PCFGsm

A metodologia de teste proposta neste trabalho faz uso da técnica estrutural de teste,

¹³ Teste de regressão: Quando uma nova versão do programa é construída (seja devido à correção de defeitos, alteração ou inclusão de funcionalidades) é possível que alterações não intencionais afetem outras partes do programa. Isso pode levar à inclusão de novos defeitos. Assim, sempre que é construída uma nova versão de um programa já testado é preciso que a nova versão seja novamente confrontada com o procedimento de teste realizado anteriormente, caracterizando assim um teste de regressão (PEZZE; YOUNG, 2007).

ou seja, a própria estrutura interna (código-fonte) do programa é a principal fonte de informação para o desenvolvimento da atividade de teste. Um modelo de teste, chamado **PCFGsm** (*Parallel Control Flow Graph for Shared Memory*, ou Grafo de Fluxo de Controle Paralelo para Memória Compartilhada), é definido com o objetivo de capturar as informações necessárias a atividade de teste a partir da estrutura do programa. Essa metodologia é composta por quatro etapas:

Realizar a análise estática: consiste em capturar informações pertinentes à atividade de teste fazendo uso do modelo de teste **PCFGsm**, definido a seguir. Essa etapa é realizada em tempo de compilação do programa, ou seja, sem conhecimento de informações geradas em tempo de execução programa. As informações obtidas nesta etapa serão usadas no decorrer da atividade de teste.

Determinar os elementos requeridos: nessa etapa são obtidos os elementos requeridos de acordo com critérios de cobertura, estes definidos a partir do modelo de teste **PCFGsm**. Após selecionar o(s) critério(s) de cobertura desejado(s), os elementos estruturais (obtidos a partir da análise estática) que os satisfazem são relacionados.

Determinar e executar os casos de teste: nessa fase o programa é executado com casos de teste estrategicamente selecionados a fim de exercitar um ou mais dos elementos requeridos para o critério de cobertura selecionado. Com a execução de cada caso de teste observa-se o resultado produzido pelo programa e compara-se com o resultado esperado. Se forem compatíveis nenhum erro foi revelado e pode-se seguir com a execução do próximo caso de teste. Se não são compatíveis significa que um erro foi revelado e deve ser corrigido e toda a atividade de teste deve ser reiniciada.

Avaliar a cobertura atingida: após a execução de um ou mais casos de teste é preciso analisar a taxa de cobertura atingida para o critério de cobertura selecionado. Se a taxa de cobertura desejada foi atingida significa que a atividade de teste chegou ao final. Caso contrário, novos casos de teste precisam ser executados.

Diferentes modelos de teste são intimamente ligados a diferentes técnicas de teste. Detalhes do modelo podem ser descritos apenas associando-os à descrição da técnica correspondente (TIAN, 2005). Porém, algumas das etapas e atividades genéricas, que estão envolvidas na construção de um modelo de teste, são: (1) identificar a fonte de informações e coletar dados, (2) analisar e construir o modelo inicial, e (3) validar o modelo e realizar um

melhoramento incremental. A subseção a seguir descreve o modelo de teste *PCFGsm* e alguns aspectos de seu desenvolvimento.

4.2.1 Representação por grafo (*PCFGsm*) do programa *multithread*

Seja $MT = (t^0, t^1, \dots, t^{n-1})$ um programa *multithread* composto por n *threads* cada qual denotada por t^i . No contexto desse trabalho *threads* podem executar diferentes funcionalidades (rotinas) porém todas elas compartilham o mesmo espaço de endereçamento de memória. Elas podem também usar uma memória privativa adicional para armazenar dados locais e informações acerca de sua execução. Cada *thread* t tem seu próprio CFG^t (*Control Flow Graph*, ou Grafo de Fluxo de Controle) que é construído usando os mesmos conceitos aplicados a programas seqüenciais tradicionais (RAPPS; WEYUKER, 1985).

O CFG da *thread* t é composto por um conjunto de nós N^t e um conjunto de arestas dirigidas E^t . Essas arestas que ligam nós pertencentes a mesma *thread* são chamadas de arestas *intra-thread*. Cada nó n na *thread* t é representado pela notação $n_i^t \in N^t$.

No CFG cada nó corresponde a uma instrução ou a um bloco de instruções que são executados seqüencialmente. Um bloco de instruções seqüenciais é caracterizado pelo fato de que a execução da primeira instrução do bloco sempre acarretará a execução de todas as outras na ordem dada. No contexto de programas *multithreads* um nó do CFG^t pode também estar associado a uma primitiva de sincronização (*post* ou *wait*).

Para construir o modelo, um programa *multithread* MT composto por n *threads* é associado com um Grafo de Fluxo de Controle Paralelo com Memória Compartilhada (*PCFGsm*), que é composto pelos $CFG^t[0 \leq t < n]$ de todas as *threads* e as arestas que representam a sincronização entre essas *threads* (*descritas a seguir*). Nesse modelo N e E representam, respectivamente, o conjunto de nós e arestas do *PCFGsm*. Para a construção do *PCFGsm* é assumido que:

- n (número de *threads* do programa) é fixo e conhecido em tempo de compilação;
- existe comunicação feita de forma implícita por meio de variáveis compartilhadas;

- há sincronização explícita usando semáforos; e
- a inicialização e a finalização de *threads* atuam como uma sincronização sobre semáforos “virtuais”. A concepção e utilidade desse tipo de semáforo estão explicadas na subseção seguinte.

4.2.2 Fluxo de controle e de sincronização

No contexto de programas *multithreads* o conceito de **CFG** foi estendido para representar as sincronizações possíveis entre as *threads*. Para isso foi considerada a relação de **causa-e-efeito** presente entre as primitivas de sincronização *post* e *wait* (CARVER; TAI, 2006). Nessa relação, o *post* é o agente ativo (a causa) e o *wait* assume a vez de passivo (o efeito), pois o *wait* sempre depende do *post* para concluir sua execução, o que não é verdade para o *post*. A título de exemplificação, pode-se fazer uma analogia com ambientes de passagem de mensagens, onde tem-se que as primitivas *post* e *wait* equivalem às primitivas *send* e *receive*, respectivamente. Dessa forma, as sincronizações são representadas no grafo por arestas dirigidas partindo do nó que representa a instrução *post* até o nó que denota o *wait*.

As arestas no **PCFG_{sm}** podem ser de dois tipos. As *intra-threads* representam o fluxo de controle entre os nós de uma *thread*. As *inter-threads* representam interação (sincronização) entre diferentes *threads*. No entanto há uma ressalva, pois pode haver arestas *intra-thread* que representem sincronizações. Isso ocorre quando em uma mesma *thread* há primitivas *post* e *wait* fazendo referências à mesma variável semáforo.

Uma particularidade está na instrução que inicializa o valor do semáforo. No modelo essa ação é representada no grafo por $n + 1$ nós correspondentes ao evento de inicialização em si mais n eventos *post* consecutivo, em que n corresponde ao valor inicial que se deseja atribuir ao semáforo.

Quando uma *thread* cria outra *thread* por meio de uma primitiva específica (**create**) o modelo define uma sincronização sobre uma variável semáforo “virtual”. Esse semáforo “virtual” só existe no contexto do modelo de teste e serve para ajudar na observação das sincronizações decorrentes da inicialização e finalização de *threads*. Essas sincronizações observadas ajudam na troca de informação de *timestamps* permitindo que *timestamps* atualizados sejam atribuídos aos eventos. Mais detalhes a respeito da atribuição de

timestamps e a interferência das sincronizações referentes à inicialização e finalização de *threads* nesse processo serão apresentadas mais adiante ainda neste capítulo. Com esse objetivo cada função que corresponde a “*main*” de *threads* tem um semáforo “virtual” associado que recebe o nome “*c_<nome_função>*”.

Para evidenciar as sincronizações referentes à inicialização de *threads* o nó que representa a primitiva de criação equivale à primitiva *post* enquanto que o primeiro nó na *thread* criada equivale à primitiva *wait*. Isso estende o modelo de teste proposto de maneira simples e objetiva, permitindo que o mesmo trate a sincronização obtida pela criação/inicialização de uma nova *thread*.

Outra facilidade comumente oferecida por bibliotecas de *threads* é que uma *thread* pode sincronizar com o término de execução (“morte”) de outra. Essa facilidade é implementada pela primitiva *join*. O modelo de teste *PCFGsm* também contempla essa possibilidade de sincronização. Nesse caso o nó que representa a operação *join* é associado à primitiva *wait* enquanto que o nó que representa o fim da execução da *thread* (último nó do *CFG*, nó que representam *exit*, *return*, etc) é associado à primitiva *post*.

Para resumir considere a Tabela 4.5 que lista a correspondência entre as primitivas de sincronizações.

Tabela 4.5: Correspondência entre as primitivas e as formas de sincronização.

Tipos de sincronização	Causa (ativo)	Efeito (passivo)
Por semáforos	<i>post</i>	<i>wait</i>
criação / inicialização	primitiva <i>create</i>	1º nó do <i>CFG</i> da <i>thread</i> criada
finalização	último comando da <i>thread</i> (<i>exit</i> , <i>return</i>)	primitiva <i>join</i>

Para o *PCFGsm* três subconjuntos de nós N são definidos: N^t composto pelos nós pertencentes a *thread* t ; N_p composto pelos nós que representam primitivas *post* e suas equivalentes; e N_w composto pelos nós que representam primitivas *wait* e suas equivalentes.

Para cada $n_i^t \in N_p$ (onde n_i^t possui um *post* para o semáforo *sem*), um conjunto

$M_w(n_i^t)$ é associado, contendo um conjunto de nós $n_j^q \in N_w$, tal que existe a *thread* $q \in [0..n-1]$ e uma primitiva *wait* com relação à (c.r.a.) variável semáforo *sem* em n_j^q . De outra maneira:

$$M_w(n_i^t) = \{n_j^q \mid \exists \text{ post}(\text{sem}) \text{ em } n_i^t \text{ e } \exists \text{ wait}(\text{sem}) \text{ em } n_j^q, \forall t, q \in [0..n-1]\}, \forall n_i^t \in N_p.$$

De maneira similar, para cada $n_i^t \in N_w$ (onde n_i^t possui um *wait* para o semáforo *sem*), um conjunto $M_p(n_i^t)$ é associado, contendo um conjunto de nós $n_j^q \in N_p$, tal que existe a *thread* $q \in [0..n-1]$ e uma primitiva *post* com relação à (c.r.a.) variável semáforo *sem* em n_j^q . De outra maneira:

$$M_p(n_i^t) = \{n_j^q \mid \exists \text{ post}(\text{sem}) \text{ em } n_j^q \text{ e } \exists \text{ wait}(\text{sem}) \text{ em } n_i^t, \forall t, q \in [0..n-1]\}, \forall n_i^t \in N_w.$$

Em outras palavras, $M_w(n_i^t)$ contém todos os possíveis nós *wait* que podem ser afetados por um *post* e $M_p(n_i^t)$ contém todos os possíveis nós *post* que podem afetar um comando *wait*.

Para complementar, também foi definido o conjunto $E_S \subset E$ que contém todas as arestas que representam sincronização (*edge-s*) entre duas *threads*, tal que:

$$E_S = \{(n_j^t, n_k^q) \mid n_j^t \in M_p(n_k^q) \wedge n_k^q \in M_w(n_j^t)\}.$$

Ou seja, o conjunto E_S representa todas as combinações (Todos-para-Todos) que formam pares de sincronização – *post* e *wait* sobre uma mesma variável semáforo.

O programa concorrente mostrado na Figura 4.16 é usado para ilustrar essas definições. Esse programa implementa o problema do produtor consumidor com *buffer* limitado, usando a biblioteca PThreads em ANSI C. Ele é composto por três *threads*: (1) uma

mestre, que inicializa as variáveis e cria as *threads* produtora e consumidora; (2) uma **produtora**, que coloca itens no *buffer*; e (3) uma **consumidora**, que remove itens do *buffer* para processamento futuro.

<pre> #include <stdlib.h> #include <stdio.h> #include <pthread.h> #include <semaphore.h> sem_t mutex, empty, full; int queue[2], avail; void *producer(void); void *consumer(void); /*Thread T0*/ int main(void) { pthread_t prod_h, cons_h; avail = 0; sem_init(&mutex, 0, 1); sem_init(&empty, 0, 2); sem_init(&full, 0, 0); pthread_create(&prod_h, 0, producer, NULL); pthread_create(&cons_h, 0, consumer, NULL); pthread_join(prod_h, 0); pthread_join(cons_h, 0); exit(0); } </pre>	<pre> /*Thread T1*/ void *producer(void) { int prod = 0, item; while (prod < 2) { item = rand()%1000; sem_wait(&empty); sem_wait(&mutex); queue[avail] = item; avail++; prod++; sem_post(&mutex); sem_post(&full); } pthread_exit(0); } /*Thread T2*/ void *consumer(void) { int cons = 0, my_item; while (cons < 2) { sem_wait(&full); sem_wait(&mutex); cons++; avail--; my_item = queue[avail]; sem_post(&mutex); sem_post(&empty); printf("consumed: %d\n", my_item); } pthread_exit(0); } </pre>
--	--

Figura 4.16: Problema do produtor / consumidor implementado com PThreads / ANSI C.

Na Figura 4.16 os comentários (*/*...*/*) ao lado das instruções foram colocados para relacionar as instruções do programa com o nó correspondente no grafo *PCFGsm* (Figura 4.17). Por exemplo, a primeira instrução de atribuição da função *main* instrução *avail = 0;* é representada pelo nó 1 no grafo *PCFGsm*. O modelo associa para instrução no programa um nó no grafo. Isso simplifica a concepção do modelo, no entanto a opção por representar blocos de instruções sequenciais como nós único é possível e sua implementação é direta (RAPPS; WEYUKER, 1985).

Observe a particularidade dos nós 2, 3, 4, 5, 6 e 7 da Thread T0 (comentados na Figura 4.16) referentes às instruções de inicialização dos semáforos (*mutex*, *empty* e *full*). Essas funções são desmembradas no modelo e a inicialização passa a ser representada por diversos *post*. Os nós 2, 4 e 7 representam a inicialização (*sem_init*) em si. Os nós 3, 5 e 6

representam os post associados ao valor de inicialização do semáforo – o nó 3 ao valor 1 do *mutex* e os nós 5 e 6 ao valor 2 do *empty*.

A Tabela 4.6 contém os valores de todos os conjuntos introduzidos até então.

Tabela 4.6: Conjuntos obtidos pela aplicação do modelo de teste definido ao programa da Figura 4.16.

$n=3$	$M_w(3^0) = \{6^1, 5^2\}$	$M_p(10^0) = \{12^1, 12^2\}$
$MT = \{t^0, t^1, t^2\}$	$M_w(5^0) = \{5^1\}$	$M_p(11^0) = \{12^1, 12^2\}$
$N_0 = \{1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 12^0\}$	$M_w(6^0) = \{5^1\}$	$M_p(1^1) = \{8^0\}$
$N_1 = \{1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1, 11^1, 12^1\}$	$M_w(8^0) = \{1^1\}$	$M_p(5^1) = \{5^0, 6^0, 10^2\}$
$N_2 = \{1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2, 11^2, 12^2\}$	$M_w(9^0) = \{1^2\}$	$M_p(6^1) = \{3^0, 10^1, 9^2\}$
$N = N_0 \cup N_1 \cup N_2$	$M_w(10^1) = \{6^1, 5^2\}$	$M_p(1^2) = \{9^0\}$
$N_p = \{3^0, 5^0, 6^0, 8^0, 9^0, 10^1, 11^1, 12^1, 9^2, 10^2, 12^2\}$	$M_w(11^1) = \{4^2\}$	$M_p(4^2) = \{11^1\}$
$N_w = \{10^0, 11^0, 1^1, 5^1, 6^1, 1^2, 4^2, 5^2\}$	$M_w(12^1) = \{10^0, 11^0\}$	$M_p(5^2) = \{3^0, 10^1, 9^2\}$
$E_l^0 = \{(1^0, 2^0), (2^0, 3^0), \dots, (10^0, 11^0), (11^0, 12^0)\}$	$M_w(9^2) = \{5^2, 6^1\}$	
$E_l^1 = \{(1^1, 2^1), (2^1, 3^1), \dots, (11^1, 3^1), (3^1, 12^1)\}$	$M_w(10^2) = \{5^1\}$	
$E_l^2 = \{(1^2, 2^2), (2^2, 3^2), \dots, (11^2, 3^2), (3^2, 12^2)\}$	$M_w(12^2) = \{10^0, 11^0\}$	
$E_s = \{(3^0, 6^1), (3^0, 5^2), (5^0, 5^1), (6^0, 5^1), (8^0, 1^1), (9^0, 1^2), (10^1, 5^2), (10^1, 6^1), (11^1, 4^2), (12^1, 10^0), (12^1, 11^0), (9^2, 6^1), (9^2, 5^2), (10^2, 5^1), (12^2, 10^0), (12^2, 11^0)\}$		
$E = E_l \cup E_l^1 \cup E_l^2 \cup E_s$		

Na Figura 4.17 é mostrado o **PCFGsm** para o programa da Figura 4.16. As *threads* mestre, produtora e consumidora são representadas, respectivamente, por: t^0, t^1 e t^2 . Nessa figura as arestas tracejadas representam arestas de sincronizações. Alguns exemplos de arestas de sincronização são: $(9^2, 6^1)$ é uma sincronização sobre semáforos, (t^1, t^1) é uma aresta de sincronização referente à inicialização da *thread* consumidora (t^2) e a sincronização $(12^1, 10^0)$ é uma aresta de sincronização referente à finalização da *thread* produtora (t^1). Note também que existem arestas de sincronização *intra-thread*, tais como $(10^1, 6^1)$ e $(9^2, 6^2)$.

Figura 4.17: Grafo *PCFGsm* que representa o programa mostrado na Figura 4.16.

4.2.3 Fluxo de dados, comunicação e caminhos

Um aspecto importante a ser observado no teste estrutural diz respeito ao fluxo de dados, o qual refere-se ao caminho estabelecido entre definições e usos de variáveis (RAPPS; WEYUKER, 1985; MYERS, 2004). A definição de uma variável ocorre quando um valor é atribuído a ela, como por exemplo, quando há uma instrução de atribuição ou pela passagem por referência da variável para uma função.

O uso de uma variável ocorre pela utilização de seu conteúdo em uma instrução. O

uso pode ocorrer de duas formas: uso computacional, quando a variável é usada em uma instrução para computar um novo valor; e uso predicativo, quando uma variável é usada para decidir o fluxo de execução do programa em instruções de decisão.

No contexto de programas *multithreads* a variável assume um papel adicional. Nesse caso a variável também é usada para realizar a interação entre as diversas *threads*. As variáveis usadas para realizar essa tarefa são consideradas variáveis compartilhadas. Tais variáveis possuem escopo de acesso global em relação às *threads* do programa, ou seja, todas as *threads* têm igual direito de acesso à variável (TANENBAUM, 2003). A comunicação entre *threads* é feita por meio de definição e uso de uma variável compartilhada (posição de memória em espaço de endereçamento compartilhado). A *thread* que faz a operação de definição (escrita) é interpretada como a *thread* locutora, enquanto que as *threads* que fazem o uso (leitura) da variável compartilhada são interpretadas como as *threads* interlocutoras (ALMASI; GOTTLIEB, 1994). A sincronização entre *threads* também requer variáveis compartilhadas, porém essas são de propósito específico. As variáveis para sincronização precisam ser compartilhadas (globais) pois é preciso que todas as *threads* tenham acesso a elas para poderem executar as primitivas de sincronização.

Conforme detalhado na seção anterior, as variáveis compartilhadas do tipo semáforo são usadas para realizar a sincronização entre as *threads*. Considerando o fluxo de dados e a comunicação em programas *multithreads*, o modelo de teste **PCFGsm** também é capaz de capturar essas informações. Para isso foram definidos quatro conjuntos que representam os possíveis tipos de variáveis nesse contexto. O conjunto V denota todas as variáveis definidas para o programa, além das variáveis “virtuais” que são usadas para representar a sincronização referente à inicialização e finalização das *threads*. $V_L^t \subset V$ contém as variáveis locais (privadas) da *thread* t . Essas variáveis possuem escopo de acesso restrito à *thread* t . $V_C \subset V$ contém as variáveis compartilhadas, ou seja, as variáveis que possuem escopo de acesso global, excluindo-se as variáveis de sincronização (semáforos). Ou seja, V_C representa as variáveis usadas para comunicação entre as *threads*. E $V_S \subset V$ contém as variáveis de sincronização, incluindo-se as variáveis “virtuais” de inicialização e finalização de *threads*.

Além desses conjuntos, foi definido um conjunto que reúne todas as variáveis com definição (escrita) em um dado nó. Esse conjunto é dado por:

$$\text{def}(n_i^t) = \{x \mid x \text{ é uma variável definida em } n_i^t\}.$$

A Tabela 4.7 contém os valores dos conjuntos correspondente à análise de fluxo de dados do programa mostrado na Figura 2.1.

Tabela 4.7: Valores dos conjuntos referentes a fluxo de dados e comunicação para o programa da Figura 2.1.

$V_L^0 = \{prod_h, cons_h\}$	$\text{def}(1^0) = \{avail\}$	$\text{def}(9^1) = \{prod\}$
$V_L^1 = \{prod, item\}$	$\text{def}(2^1) = \{prod\}$	$\text{def}(2^2) = \{cons\}$
$V_L^2 = \{cons, my_item\}$	$\text{def}(4^1) = \{item\}$	$\text{def}(6^2) = \{cons\}$
$V_c = \{queue, avail\}$	$\text{def}(7^1) = \{queue\}$	$\text{def}(7^2) = \{avail\}$
$V_s = \{mutex, empty, full\}$	$\text{def}(8^1) = \{avail\}$	$\text{def}(8^2) = \{my_item\}$

4.2.4 Caminhos e associações

Com a obtenção do grafo *PCFGsm* e considerando o fluxo de dados e de comunicação é possível definir caminhos e associações para o programa em teste. O levantamento dessas informações possibilita estabelecer critérios de avaliação de cobertura, tornando possível quantificar e qualificar a atividade de teste.

Um caminho $\pi^t = \{n_1^t, n_2^t, \dots, n_j^t\}$, em que $(n_i^t, n_{i+1}^t) \in E_I^t$, é denominado **caminho intra-thread** se e somente se não possui aresta de sincronização. O caminho $\pi^2 = (10^2, 11^2, 3^2, 4^2, 5^2, 6^2)$ observado no grafo da Figura 2.2 é um caminho intra-thread pertencente à thread t^2 .

Também é possível classificar o caminho como **completo**: quando o primeiro e último nó são, respectivamente, um nó de entrada e um nó de saída no grafo. Um caminho é **simples** quando todos os nós que o compõem são distintos, exceto possivelmente o primeiro e último. Um caminho é **livre de laços** quando todos os nós, sem exceção, são distintos. No grafo da Figura 2.2 o caminho $(1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 12^0)$ é simples, completo e livre de laços, enquanto que o caminho $(3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1, 11^1, 3^1)$ é dito apenas simples.

Um caminho que inclui ao menos uma aresta de sincronização é chamado de **caminho inter-thread** e é denotado por $\Pi = (PATHS, SYNCs)$, em que

$PATHS = \{\pi^1, \pi^2, \dots, \pi^n\}$ e $SYNCs = \{(p_i^t, w_j^q) \mid (p_i^t, w_j^q) \in E_s\}$ (SOUZA *et al.*, 2008).

Aqui p_i^t denota um nó *post* p_i na *thread* t e w_j^q denota um nó *wait* w_j na *thread* q . O caminho

$$\begin{aligned} PATHS = & \{\pi^0 = (1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 9^0, 10^0, 11^0, 12^0), \} \\ & \{\pi^1 = (1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 9^1, 10^1, 11^1, 3^1, 12^1), \} \\ & \{\pi^2 = (1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 9^2, 10^2, 11^2, 3^2, 12^2)\} \end{aligned}$$

$$SYNCs = \{(8^0, 1^1), (9^0, 1^2), (6^0, 5^1), (3^0, 6^1), (12^1, 10^0), (11^1, 4^2), (10^1, 5^2), (12^2, 11^0)\}$$

no grafo apresentado na Figura 2.2 é um caminho *inter-thread*.

Um caminho $\pi^t = \{n_1, n_2, \dots, n_j, n_k\}$ é dito livre de definição c.r.a variável local $c \in V_L^t$ se e somente se partindo do nó n_1 ao nó n_k ou à aresta (n_j, n_k) , $x \in \text{def}(n_1)$ e $x \notin \text{def}(n_i)$, para $i \in (2..j)$. Ou seja, um caminho é livre de definição se o primeiro nó possui uma definição para a variável x e último nó ou aresta faz uso da variável sem que nenhum nó intermediário a defina novamente.

A noção de caminho livre de definição não é aplicável às variáveis compartilhadas porque a comunicação (definição e uso de variáveis compartilhadas) em *threads* é implícita e instanciada em tempo de execução. Por isso é impraticável estabelecer estaticamente um caminho de definição e uso para variáveis compartilhadas. Na Seção 4.3, é apresentado um método para contornar essa dificuldade determinando caminhos livre de definição para variáveis compartilhadas com base na execução, usando-se uma metodologia de análise *post-mortem*.

O uso de variáveis compartilhadas em programas *multithreads* pode ser de cinco tipos:

uso computacional (*c-use*): quando ocorre em instruções de computação e é relacionada à variáveis locais $x \in V_L^t$;

uso predicativo (*p-use*): quando ocorre em estruturas condicionais, as quais modificam o fluxo de controle da *thread*, sendo este uso relacionado às variáveis locais $x \in V_L^t$;

uso de sincronização (*sync-use*): relacionado a instruções de sincronização, estas vinculadas a variáveis semáforo $x \in V_S$;

c-use comunicacional (*comm-c-use*): quando ocorre um uso computacional de variável compartilhada.

p-use comunicacional (*comm-p-use*): quando ocorre uso predicativo de variável compartilhada.

Baseando-se nessas definições, foram estabelecidas associações entre definição e uso das variáveis de programas *multithread*. Cinco tipos de associações foram definidas:

- **Associação c-use:** é definida pela tripla (n_i^t, n_j^t, x) se e somente se $x \in V_L^t$, $x \in \text{def}(n_i^t)$, n_j^t tem um **c-use** de x e há ao menos um caminho livre de definição c.r.a. x de n_i^t até n_j^t .
- **Associação p-use:** é definida pela tripla $(n_i^t, (n_j^t, n_k^t), x)$ se e somente se $x \in V_L^t$, $x \in \text{def}(n_i^t)$, (n_j^t, n_k^t) tem um **p-use** de x e há ao menos um caminho livre de definição c.r.a. x do nó n_i^t até a aresta (n_j^t, n_k^t) .
- **Associação sync-use:** é definida pela tripla $(n_i^t, (n_j^t, n_k^q), s)$ se e somente se $s \in V_S$, (n_j^t, n_k^q) tem um **sync-use** de s e há ao menos um caminho livre de definição c.r.a. s do nó n_i^t até a aresta (n_j^t, n_k^q) .
- **Associação comm-c-use:** é definida pela tripla (n_i^t, n_j^q, x) se e somente se $x \in V_C$, $x \in \text{def}(n_i^t)$ e n_j^q tem um **c-use** da variável compartilhada x .
- **Associação comm-p-use:** é definida pela tripla $(n_i^t, (n_j^q, n_k^q), x)$ se e somente se $x \in V_C$, $x \in \text{def}(n_i^t)$ e (n_j^q, n_k^q) tem um **p-use** da variável compartilhada x .

4.3 Extraíndo dinamicamente informações de fluxo de dados

Determinar as comunicações possíveis para um programa *multithread* durante a fase

de análise estática é uma tarefa extremamente difícil e em muitos casos impraticável. A comunicação nesse contexto é implícita e se dá por meio de escrita e leitura de variáveis compartilhadas. Quando uma *thread* locutora deseja se comunicar com outra, ela deve escrever a mensagem (dados) na variável em questão e em seguida a *thread* interlocutora deve ler o conteúdo da variável. Observe que o processo de comunicação em programas *multithreads* requer uma ordem entre os eventos de definição e uso. Se em uma comunicação a *thread* que deveria receber a mensagem faz a leitura da variável antes que a outra *thread* faça a escrita da mensagem na variável compartilhada a comunicação não ocorre com sucesso. Nesse caso a *thread* interlocutora irá, possivelmente, obter uma informação errada (“sujeira de memória”).

A correta comunicação em programas *multithreads* depende da correta sincronização das diversas *threads*, pois a comunicação requer a sincronização de controle de sequência entre o evento de definição e uso da variável compartilhada. Mas observar apenas se as sincronizações estão corretas em um programa *multithread* não garante a corretude da comunicação. É preciso observar especificamente também a comunicação. No entanto é difícil estabelecer uma relação direta entre uma comunicação e determinada(s) sincronização(ões).

No contexto de teste de software, deseja-se testar toda possível comunicação entre as *threads*. Para isso, é necessário estabelecer quando essa comunicação de fato ocorre para que o teste ocorra com efetividade.

Para determinar as comunicações possíveis, bem como as comunicações que foram executadas, para programas *multithreads* é preciso observar a ordem de execução dos eventos de definição e uso de variáveis compartilhadas.

A relação de causalidade (*Happens-before*) definida por Lamport (1978) pode ser usada como ferramenta para estabelecer a ordem entre eventos concorrentes. Essa relação estabelece que se os eventos *A* e *B* são eventos locais na mesma *thread* e *A* é executado antes que *B*, então o *timestamp* lógico de *A* irá indicar que *A* aconteceu antes de *B*. De maneira similar, se *P* é um evento de sincronização *post* em uma *thread* e *W* é o evento *wait* correspondente em outra *thread*, o *timestamp* lógico de *P* irá indicar que *P* ocorreu antes de *W* (mais precisamente, *P* ocorreu antes da finalização da operação *W*). Nesse caso é a execução das operações do programa que governa a evolução do tempo lógico, não o pulso do relógio

de tempo-real (*clock* do processador).

É possível também estabelecer uma relação de causa-e-efeito entre esses eventos, pois ordenando o *post* antes do *wait* coloca-se a causa antes do efeito. A causalidade ou relação de *happens-before* ($\rightarrow$) para uma execução de um programa com sincronizações assíncronas é definida como segue (LAMPORT, 1978):

- Se a e b são eventos na mesma *thread* e a ocorre antes que b , então $a \rightarrow b$.
- Se há uma sincronização de a para b , então $a \rightarrow b$.
- Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$. (Isto é, $\rightarrow$ é uma relação transitiva.)

Essa relação não é aplicável somente a eventos na mesma *thread* ou eventos de sincronização, mas a qualquer tipo de evento, pois se o evento C pode potencialmente afetar (causar) outro evento E , diz-se que C ocorre antes de E em ordem causal. Porém para ordenar eventos quaisquer (aqueles que não formam pares de sincronização) é necessário recorrer a uma técnica de atribuição de *timestamps* a esses eventos. Na sequência será apresentada um método adaptado para atribuir *timestamps*.

Para dois eventos quaisquer a e b , se nem $a \rightarrow b$ nem $b \rightarrow a$, então os eventos a e b são ditos concorrentes, denotado por $a \parallel b$. Uma vez que $a \parallel b$ e $b \parallel a$ são equivalentes, a relação $\parallel$ é dita simétrica. Contudo a relação $\parallel$ não é transitiva, pois se $a \parallel b$ e $b \parallel c$ **não** se pode afirmar que $a \parallel c$.

Uma vez a relação de causalidade admite que dois eventos concorrentes quaisquer não possam ser ordenados (isto é, eles podem ocorrer em qualquer ordem) a relação apenas ordena **parcialmente** os eventos de um *trace* de execução. No entanto uma ordenação parcial é ainda útil, pois dado um programa, uma entrada e um *trace* de execução de eventos de sincronização parcialmente ordenado que seja baseado na relação de causalidade, há apenas uma saída possível.

A ordenação parcial de eventos é um subconjunto da ordenação total de eventos para um dado programa, pois é considerado que duas ou mais execuções que têm a mesma entrada e a mesma sequência de sincronização parcialmente ordenada são execuções equivalentes. Um *trace* de execução totalmente ordenado pode equivaler a diversos *traces* de execução parcialmente ordenados (CARVER; LEI, 2006).

Fidge (1996) estudou e comparou as seguintes técnicas para atribuir *timestamps* a eventos concorrentes que objetivam ordenar esses eventos: ordem de chegada, relógio local de tempo-real, relógio global de tempo-real, relógio lógico totalmente ordenado, e relógio lógico parcialmente ordenado. Ele observou que a maioria das técnicas apresenta alguns problemas. Entre esses problemas estão: *ordenação incorreta* e *ordenação arbitrária*. Mais detalhes a respeito dessas técnicas e desses problemas podem ser encontrados em (FIDGE, 1996; CARVER; LEI, 2006).

Fidge (1996) chegou a conclusão que a técnica mais interessante para obter ordenação de eventos concorrentes é aquela que usa relógio lógico parcialmente ordenado. Essa técnica atribui *timestamps*, baseados em um vetor de relógio lógico, a eventos de interesse para posteriormente comparar esses *timestamps* e então estabelecer uma ordem entre os eventos (CARVER; LEI, 2006).

Nesse técnica cada *thread* mantém um vetor de valores de relógio lógico. Cada valor desse vetor corresponde ao relógio lógico da uma *thread* do sistema e os valores são atualizados sempre que uma *thread* executa um evento qualquer ou sempre que executa uma sincronização, nesse caso há a troca de informações referente ao vetor relógio da outra *thread*.

Mais detalhes são apresentado na próxima seção quando será mostrado esse método adaptado para determinar pares de comunicação (pares de definição e uso de variáveis compartilhadas) em programas *multithreads*.

4.3.1 Atribuição de *timestamps*

Para determinar os pares de comunicação exercitados para uma dada execução, optou-se por uma abordagem de instrumentação para obtenção do *trace* de execução e análise *post-mortem* do *trace* obtido. O método proposto é uma adaptação daquele utiliza vetor relógio lógico parcialmente ordenado (LEI; CARVER, 2006). As principais mudanças são: (1) considerar o critério **LIFO** (*Last In, First Out*) quando há não-determinismo no “casamento” de *posts* e *waits*; (2) incluir os eventos de inicialização e finalização de *threads* tais como sincronizações comuns; e (3) considerar eventos de definição e uso de variáveis compartilhadas na atribuição de *timestamps*, além dos eventos de sincronização.

A idéia é que, após a execução do programa *multithread*, por meio de instrumentação,

para cada *thread* seja produzido um *trace* de execução. Esse *trace* deve corresponder ao caminho do **PCFGsm** que foi exercitado. É importante lembrar que os *traces* mantêm informações a respeito das sincronizações realizadas considerando o critério **LIFO** para o casamento de *posts* e *waits* não-determinísticos. Então, com a execução do programa finalizada, em um momento *post-mortem*, o *trace* é processado e analisado para determinar os pares de comunicações exercitadas para aquela execução.

Após a obtenção dos *traces* de execução para cada *thread* do programa dois passos são necessários para se obter os pares de comunicação exercitados: (1) atribuir *timestamps* lógicos a eventos de interesse (neste caso, eventos de comunicação e sincronização); e (2) comparar os *timestamps* dos eventos de comunicação (definição e uso de variáveis compartilhadas) para estabelecer a relação de *happens-before* entre eles.

O método para atribuir *timestamps* consiste em, inicialmente, atribuir um vetor relógio local para cada *thread* t^i , denotado por $t^i.cv$. Esse vetor é ordenado e tem dimensão n , em que n é o número total de *threads* e cada posição $i \in [0..n-1]$ no vetor relógio é associada à *thread* t^i . Assim cada *thread* possui a informação relativa ao seu relógio local e o relógio local de todas as outras *threads* que compõem o programa. Inicialmente, todas as posições do vetor relógio recebem o valor 0 (zero). Sempre que um evento de interesse ocorre em uma *thread*, seu vetor relógio é atualizado. Essa atualização obedece as seguintes regras:

1. Quando a thread t^i executa um evento de sincronização *post* p ou seus equivalentes, as seguintes operações devem ser realizadas:

$$(a) \quad t^i.cv[i] = t^i.cv[i] + 1 ; e$$

(b) $p.ts = t^i.cv$, e a sincronização, quando estabelecida, carrega o timestamps $p.ts$.

2. Quando a *thread* t^j executa um evento de sincronização *wait* w , ou seus equivalentes, que “casa” com o *post* p , as seguintes operações devem ser realizadas:

$$(a) \quad t^j.cv[j] = t^j.cv[j] + 1 ;$$

$$(b) \quad t^j.cv[j] = \text{“max”}(t^j.cv, p.ts) ; e$$

$$(c) \quad w.ts = t^j.cv \quad .$$

3. Quando a thread t^k executa um evento e de comunicação (definição d ou uso u de variáveis compartilhadas) a seguinte operação deve ser realizada:

$$(a) \quad t^k.cv[k] = t^k.cv[k] + 1 \quad ; e$$

$$(b) \quad e.ts = t^k.cv \quad .$$

Em outras palavras, quando uma *thread* executa um evento de comunicação ou um evento de sincronização *post* ela atualiza seu vetor relógio local em sua posição correspondente, registrando que adiantou uma unidade de tempo e o evento recebe o *timestamp* para registrar o exato momento lógico que o evento executou. Quando um evento de sincronização *wait* ocorre, o vetor relógio da *thread* é atualizado com a informação do *timestamp* do evento *post* com o qual casou, permitindo que a *thread* tome conhecimento do relógio das outras *threads* do programa. Ou seja, os eventos *waits* são os maiores responsáveis por atualizar as *threads* quanto ao andamento das outras e permitem que os outros eventos recebam *timestamps* atualizados, permitindo que posteriormente sejam ordenados. Esse método permite que eventos concorrentes sejam parcialmente ordenados.

Para entender melhor a aplicação desse método de atribuição de *timestamps* é útil representar graficamente os eventos de interesses, as sincronizações ocorridas e os *timestamps* atribuídos a esses eventos. Considere que em uma execução hipotética do programa produtor/consumidor (Figura 2.2), o *trace* produzido pelas *threads* é aquele apresentado na Figura 2.2.

$$\begin{aligned} \pi^0 &= \{1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 12^0\} \\ \pi^1 &= \{1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1, 11^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1, 11^1, 12^1\} \\ \pi^2 &= \{1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2, 11^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2, 11^2, 12^2\} \\ SYNC &= \{(8^0, 1^1), (6^0, 5^1), (3^0, 6^1), (9^0, 1^2), (11^1, 4^2), (10^1, 5^2), (5^0, 5^1), (9^2, 6^1), \\ &\quad (12^1, 10^0), (11^1, 4^2), (10^1, 5^2), (12^2, 11^0)\} \end{aligned}$$

Figura 4.18: Traces produzidos pelas threads do programa apresentado na Figura 2.2 considerando uma execução hipotética.

Esse trace corresponde ao caminho Π executado pelo programa mostrado na Figura 2.2. O caminho Π é composto pelo caminho π^t que cada *thread* executou e o conjunto *SYNCS* de arestas de sincronização executadas.

A execução do produtor/consumidor que produz esse *trace* é o caso em que a ordem de acesso ao *buffer* (variável compartilhada *queue*) é feita da seguinte forma: a *thread* produtora coloca um item no *buffer*, logo em seguida a *thread* consumidora o retira, então a *thread* produtora insere outro item e em seguida a *thread* consumidora o retira.

Mais detalhadamente, observando o *PCFGsm* (Figura 2.2) e o *trace* (Figura 4.18) obtido pela execução do programa da Figura 2.1, a ordem de operações que produz esse *trace*, considerando um sistema uniprocessado é: o programa inicia sua execução a partir da *thread* **mestre** (*main*) que faz a inicialização das variáveis compartilhadas (globais) e dos semáforos (produção dos *tokens* iniciais: 1 para o *mutex* (nó 3^0), 2 para o *empty* (nós 5^0 e 6^0) e 0 para o *full*), e em seguida cria a *thread* produtora (produzindo 1 *token* para o semáforo virtual *c_producer*). A *thread* **produtora** é então escalonada pelo Sistema Operacional para executar e nesse momento concretiza a sincronização referente a sua inicialização $(8^0, 1^1)$ e em seguida obtém acesso ao *buffer* por meio das sincronizações $(6^0, 5^1)$, $(3^0, 6^1)$. Observe que nesse momento o evento *wait(empty)* na *thread* produtora tem dois *tokens* (produzidos pelos nós 5^0 e 6^0) disponíveis para serem consumidos. No entanto, o último *token* produzido foi o escolhido (6^0) observada a disciplina **LIFO** para o consumo de *tokens*.

Para deixar a região crítica a *thread* produtora realiza dois eventos *post* produzindo dois novos *tokens*: 10^1 para *mutex* e 11^1 para *full*. Nesse instante a *thread* produtora é bloqueada e a *thread* mestre é novamente escalonada, então ela cria a *thread* consumidora (produzindo o *token* 9^0 para o semáforo virtual *c_consumer*) e em seguida bloqueia na instrução *join(prod_h)* aguardando a finalização da *thread* produtora. A *thread* consumidora é então escalonada e inicia sua execução concretizando a sincronização referente a sua inicialização $(9^0, 1^2)$. A seguir esta *thread* acessa a região crítica graças às sincronizações $(11^1, 4^2)$ e $(10^1, 5^2)$. A *thread* consumidora acessa o *buffer* e retira um item deixando a região crítica com a produção de dois novos *tokens* (9^2 para o semáforo *mutex* e 10^2 para o semáforo *empty*). A *thread* consumidora então é bloqueada e a *thread* produtora é novamente escalonada.

A *thread* produtora faz seu segundo acesso à região crítica realizando as sincronizações $(5^0, 5^1)$ e $(9^2, 6^1)$, insere um novo item no *buffer*, deixa a região crítica produzindo dois novos *tokens* $(10^1$ e $11^1)$ e finaliza sua execução produzindo o primeiro *token* (12^1) para o semáforo virtual f_thread . Na sequência, a *thread* mestre é escalonada novamente e realiza a sincronização $(12^1, 10^0)$ referente à finalização (*join*) da *thread* produtora e em seguida volta a ser bloqueada, cedendo o processador para a *thread* consumidora. Então a *thread* consumidora volta a acessar a região crítica por meio das sincronizações $(11^1, 4^2)$ e $(10^1, 5^2)$, retira o segundo item do *buffer*, deixa a região crítica produzindo dois novos *tokens* $(9^2$ e $10^2)$ e finaliza sua execução produzindo o segundo *token* (12^2) para o semáforo virtual f_thread .

A *thread* mestre volta a ser escalonada, concretiza a sincronização $(12^2, 11^0)$ referente à finalização da *thread* consumidora e finaliza sua execução e, por consequência, a execução do programa concorrente.

De posse do *trace* de execução da Figura 4.18 é possível atribuir os *timestamps* aos eventos de interesse (sincronização e comunicação) e em seguida confrontar esses *timestamps* para ordenar os eventos.

Um algoritmo simples para atribuir *timestamps* aos eventos é apresentado em pseudo-código na Figura 4.19. Esse algoritmo tem como entrada o número de *threads* e o *trace* executado pelo programa (*trace* de cada *thread* mais o conjunto de sincronizações exercitadas). Após receber os parâmetros de entrada (3 primeiras linhas do algoritmo) o algoritmo inicializa como 0 (zero) cada uma das n posições do vetor relógio de cada *thread* ($t.vc[i]=0$, $i \in [0..n-1]$). Em seguida o algoritmo executa um laço (Faça) repetitivamente até que o conjunto REP chegue ao final do laço vazio.

O conjunto *REP* é um repositório de eventos que foram visitados mas que tiveram *timestamps* atribuídos. Os eventos *wait* são os únicos que podem sofrer esse problema, pois dada a sequência de leitura dos *traces* das *threads* e dos eventos em cada *trace*, é possível que um evento *wait* seja visitado antes que se visite o evento *post* com o qual ele sincronizou. Pela regra de atribuição de *timestamps* apresentada anteriormente, o evento *wait* deve ter seu *timestamp* atualizado com máximo entre o vetor relógio da *thread* que executou o evento *wait* e o *timestamps* do evento *post* ($t.vc \leftarrow \max(t.vc, p.ts)$). Se o evento *post* em questão ainda

não teve o *timestamp* atribuído fica impossível, momentaneamente, atribuir o *timestamp* ao evento *wait*. Nesse caso, o evento *wait* é colocado no conjunto *REP* para que em uma próxima passagem tente-se atribuir seu *timestamp*.

```

 $n \leftarrow$  número de threads do programa
 $TRACES \leftarrow \{\pi^0, \pi^1, \dots, \pi^{n-1}\}$ 
 $SYNCS \leftarrow$  arestas de sincronização exercitadas

Para cada  $t \in TRACES$  faça
  Para  $i = 0$  até  $i = n - 1$  faça
     $t.vc[i] = 0$ 
  Fim Para
Fim Para cada

Faça
   $REP \leftarrow \emptyset$ 
  P1: Para cada  $t \in TRACES$  faça

    P2: Para cada  $e \in t$  faça
      Se (  $\exists e.ts$  )
        Vá para P2
      Fim Se
       $t.vc[t.tid] \leftarrow t.vc[t.tid] + 1$ 
      Se (  $e$  é WAIT) faça
        Se (  $\exists p.ts \mid \langle p, e \rangle \in SYNCS$  )
           $t.vc \leftarrow \max(t.vc, p.ts)$ 
           $e.ts \leftarrow t.vc$ 
        Senão
           $REP \leftarrow REP \cup \{e\}$ 
        Vá para P1
      Fim Se
    Senão
       $e.ts \leftarrow t.vc$ 
    Fim Se
  Fim Para cada

  Fim Para cada
  Enquanto (  $REP \neq \emptyset$  )

```

Figura 4.19: Algoritmo para atribuir timestamps aos eventos de um trace de execução.

Então o algoritmo propriamente dito, inicia esvaziando o “repositório de eventos visitados sem *timestamp* atribuído” ($REP \leftarrow \emptyset$). Em seguida seleciona-se o *trace* π^0 da primeira *thread* (*main*). E para cada evento e do *trace*, partindo do primeiro para o último, tenta-se atribuir o *timestamps* correspondente. A primeira condição (Se) apenas tem sentido quando o laço (Faça) é executado pela segunda em diante, pois nesse caso já haverá eventos com *timestamp* atribuído e nesse caso deve buscar o próximo evento (Vá para **P2**).

Se o evento ainda não tem *timestamps* atribuído então o vetor relógio da *thread* é incrementado ($t.vc[t.tid] \leftarrow t.vc[t.tid] + 1$) indicando a execução de mais um evento e a segunda condição (Se) verifica se o evento em questão é um WAIT.

Caso afirmativo deve-se observar se o *post* com o qual o *wait* sincronizou já teve o *timestamps* atribuído (terceira condição, Se $\exists p.ts \mid \langle p, e \rangle \in SYNCs$). Nesse caso pode-se proceder com a atualização do vetor relógio da *thread* fazendo o máximo (componente-a-componente) entre o vetor relógio da *thread* e o *timestamp* do evento *post* ($t.vc \leftarrow \max(t.vc, p.ts)$) e em seguida atribuindo o vetor relógio da *thread* com *timestamp* do evento *wait*. Porém, se o *post* ainda não tem *timestamp* (primeiro Senão) deve-se colocar o evento *wait* no “repositório de eventos visitados sem *timestamp* atribuído” ($REP \leftarrow REP \cup \{e\}$) deixando para uma próxima passagem do laço (Faça). Em seguida deve-se selecionar o *trace* da *thread* seguinte (Vá para **P1**) selecionando outro evento e repetindo todo o processo de atribuição de *timestamp*.

Caso negativo (segundo Senão), o evento corresponde a eventos sem restrição (*post* ou eventos de comunicação) e a atribuição de *timestamps* é direta e sem problemas.

Após passar por todos os traces o final do laço (Para) verifica se há algum evento ficou aguardando para ter o *timestamp* atribuído ($REP \neq \emptyset$). Caso afirmativo, o laço (Para) é repetido. Em outro caso o algoritmo termina com todos os eventos tendo recebido *timestamps*.

O resultado da atribuição de *timestamps* aos eventos do *trace* da Figura 4.18 usando esse algoritmo pode ser observado na Figura 4.20. Nesse caso o algoritmo iniciou atribuindo *timestamps* ao evento 1^0 e seguiu sequencialmente até o evento 10^0 , pois o *post* (12^1) com qual “casou” ainda não foi visitado e conseqüentemente não teve *timestamp* atribuído. Então o evento 10^0 foi colocado no “repositório de eventos visitados, porém sem

timestamp atribuído”, o trace seguinte foi selecionado (t^1) e a atribuição continuou com o primeiro evento da *thread* produtora (t^1) seguindo sequencialmente até o evento 6^1 , quando este é colocado no repositório, pois o evento *post* 9^2 ainda não possui *timestamps*. Em seguida o algoritmo continua com *thread* consumidora (t^2) até a situação se repetir.

Figura 4.20: Exemplo do diagrama de espaço-tempo lógico, considerando o método de atribuição de timestamps apresentado nessa seção aplicado ao programa multithread do produtor / consumidor (Figura 2.2).

Após percorrer o *trace* de todas as *threads* o algoritmo verifica se resta no repositório algum evento que não recebeu *timestamps* e encontra três eventos (10^0 , 6^1 e 4^2).

Então repete a tentativa de atribuir o *timestamp* ao evento 10^0 , porém o *post* 12^1 ainda não recebeu *timestamps*, dessa forma o evento 10^0 é recolocado no repositório e tenta-se atribuir o *timestamp* ao próximo evento (6^1). Nesse caso o evento 6^1 recebe o *timestamp* com sucesso pois o *post* (9^2) já o recebeu. Observe que o evento 6^1 recebeu o *timestamp* atualizado de acordo com máximo entre o vetor relógio da *thread* e o *timestamps* do evento *post* (9^2). Nesse caso o vetor relógio da *thread* t^1 possuía $[8,14,0]$ que pela passagem do evento *wait* (6^1) foi incrementado para $[8,15,0]$, e o *timestamp* do evento *post* (9^2) era $[9,11,9]$. Por fim o $\max([8,15,0],[9,11,9])=[9,15,9]$ foi o valor atribuído ao vetor relógio da *thread* t^1 e em seguida atribuído como *timestamps* do evento *wait* (6^1).

Então a atribuição de *timestamp* segue sequencialmente pelos eventos que sucedem o evento 6^1 na *thread* t^1 . Esse processo se repete até que o repositório encontre-se vazio e conseqüentemente não reste nenhum evento sem *timestamp* atribuído.

4.3.2 Algoritmo para obtenção de pares de comunicação

Após ter o *timestamp* atribuído a todos os eventos é hora de confrontá-los a fim de estabelecer uma ordem entre os eventos.

É possível dizer que $e_1 \rightarrow e_2$, se e somente se (LAMPORT, 1978, CARVER; LEI, 2006):

1. (e_1, e_2) é um par de sincronização; ou
2. $e_1.ts[e_1.tid] \leq e_2.ts[e_1.tid]$ e $e_1.ts[e_2.tid] < e_2.ts[e_2.tid]$

em que $e_i.ts$ denota o vetor *timestamps* do evento e_i , e $e_i.tid$ denota o identificador da *thread* (*Thread IDentification*) a qual o evento e_i pertence.

Para entender melhor essas desigualdades considere o diagrama mostrado na Figura 2.5. Nessa figura os eventos 1^0 e 4^1 , aqui denotados respectivamente por v e p ,

em que $v.ts = [1, 0, 0]$ e $p.ts = [8, 4, 0]$. Uma vez que $v.ts[0] < p.ts[0]$ e $v.ts[1] < p.ts[1]$, é possível concluir que $1^0 \rightarrow 4^1$.

A primeira desigualdade ($e_1.ts[e_1.tid] \leq e_2.ts[e_1.tid]$) da regra permite que, no pior dos casos, eventos imediatamente após uma sincronização sejam considerados. Por exemplo, os eventos $9^2[9, 11, 9]$ e $7^1[9, 16, 9]$ podem tornar verdade que $9^2 \rightarrow 7^1$, pois é verdade que $9 \leq 9$ (terceira posição de cada vetor de *timestamps*).

A segunda desigualdade ($e_1.ts[e_2.tid] < e_2.ts[e_2.tid]$) da regra garante a exata ordem dos eventos. Nesse caso conclui-se que realmente $9^2 \rightarrow 7^1$, pois $11 < 16$.

De posse dos *traces* de eventos com *timestamps* atribuídos e usando a regra de *happens-before* é possível ordenar qualquer dois eventos. Porém é preciso observar que a relação de *happens-before* é parcial, ou seja, se não é verdade que $e_1 \rightarrow e_2$, então não se pode dizer que $e_2 \rightarrow e_1$. Para obter os pares de comunicação (definição e uso de variável compartilhada) é preciso buscar para cada uso a última definição feita para a variável em questão. Na Figura 4.21 é mostrado um algoritmo para obter pares de comunicação observando as associações *comm-c-use* (*uso computacional de uma variável compartilhada*).

Nesse algoritmo três situações são possíveis quando se encontra uma definição que precede um uso (*comm-c-use*): (1) é a primeira definição candidata a compor o par de comunicação (Caso 1); (2) não é mais a primeira definição candidata porém é uma definição mais recente que a(s) candidata(s) anterior(es) (Caso 2, Se); ou (3) não é mais a primeira definição candidata e é uma definição concorrente com as demais candidatas (Caso 2, Senão Se).

O último condicional (Se) do algoritmo confirma que para cada uso deve haver apenas uma definição para que a comunicação ocorra corretamente. Caso contrário é muito provável que problemas ocorram no processo de comunicação das *threads*. Aliás, esse foi um resultado não previsto alcançado com o desenvolvimento deste trabalho. O método para determinar pares de comunicação pode também detectar definições concorrentes de variável, o que é no mínimo indesejado e considerado uma má prática de projeto/desenvolvimento do programa concorrente.

```

Paresdef_use ← ∅
Para cada s ∈ Vc faça
    Nc-use(s) ← Nós de c-use da variável compartilhada s
    Ndef(s) ← {nk | s ∈ def(nk)}
    Para cada ni ∈ Nc-use(s) faça
        Ndef_cand(s) ← ∅
        Para cada nj ∈ Ndef(s) faça
            Caso 1: ( nj → ni && Paresdef_use(s) = ∅ )
                Ndef_cand(s) ← {nj}

            Caso 2: ( nj → ni && Paresdef_use(s) ≠ ∅ )
                Se ( ∃ nk ∈ Ndef_cand(s) | nk → nj )
                    Ndef_cand(s) ← {nj}

                Senão
                    Se ( nj → nk && nk → nj, ∀ nk ∈ Ndef_cand(s) )
                        Ndef_cand(s) ← Ndef_cand(s) ∪ {nj}

        Fim para cada nj
    Fim para cada ni
    Se ( # Ndef_cand(s) = 1 )
        Paresdef_use ← Paresdef_use ∪ {(nj, ni, s)}
Fim para cada s

```

Figura 4.21: Algoritmo para encontrar pares de comunicação do tipo comm-c-use.

Similar ao algoritmo para determinar pares de comunicação do tipo *comm-c-use*, na Figura 4.22 é apresentada uma versão para determinar associações de comunicação *comm-p-use*.

```

Paresdef_use ← ∅
Para cada  $s \in V_C$  faça
     $E_{p\text{-use}}(s) \leftarrow$  Arestas de  $p\text{-use}$  da variável compartilhada  $s$ 
     $E_{\text{def}}(s) \leftarrow \{n_k \mid s \in \text{def}(n_k)\}$ 
    Para cada  $(n_i, n_k) \in E_{p\text{-use}}(s)$  faça
         $N_{\text{def\_cand}}(s) \leftarrow \emptyset$ 
        Para cada  $n_j \in N_{\text{def}}(s)$  faça
            Caso 1: (  $n_j \rightarrow n_i$  && Paresdef_use( $s$ ) = ∅ )
                 $N_{\text{def\_cand}}(s) \leftarrow \{n_j\}$ 

            Caso 2: (  $n_j \rightarrow n_i$  && Paresdef_use( $s$ ) ≠ ∅ )
                Se (  $\exists n_p \in N_{\text{def\_cand}}(s) \mid n_p \rightarrow n_j$  )
                     $N_{\text{def\_cand}}(s) \leftarrow \{n_j\}$ 
                Senão
                    Se (  $n_j \rightrightarrows n_p$  &&  $n_p \rightrightarrows n_j, \forall n_p \in N_{\text{def\_cand}}(s)$  )
                         $N_{\text{def\_cand}}(s) \leftarrow N_{\text{def\_cand}}(s) \cup \{n_j\}$ 
        Fim para cada  $n_j$ 
    Fim para cada  $(n_i, n_k)$ 
    Se (  $\# N_{\text{def\_cand}}(s) = 1$  )
        Paresdef_use ← Paresdef_use ∪ {( $n_j, (n_i, n_k), s$ )}
    Fim para cada  $s$ 
Fim para cada

```

Figura 4.22: Algoritmo para encontrar pares de comunicação do tipo comm-p-use.

4.4 Critérios de cobertura

Considerando o modelo de fluxo de controle, de dados e de comunicação e as definições apresentadas até aqui, foram propostos dois conjuntos de critérios de teste estrutural para programas paralelos baseados em memória compartilhada. Esses critérios permitem o teste de aspectos sequenciais e paralelos desses programas.

Critérios baseados em fluxo de controle e de sincronização

Critério All-p-nodes: requer que o conjunto de casos de teste execute todos os nós $post \ n_i' \in N_p$.

Critério All-w-nodes: requer que o conjunto de casos de teste execute todos os nós

wait $n_i^t \in N_w$.

Critério *All-nodes*: requer que o conjunto de casos de teste execute todos os nós $n_i^t \in N$. Este critério já é utilizado em programas seqüenciais e paralelos baseados em passagem de mensagens.

Critério *All-s-edges*: requer que o conjunto de casos de teste execute todas as arestas de sincronização $(n_i^t, n_j^q) \in E_s$. Este critério é utilizado em programas paralelos com passagem de mensagens.

Critério *All-edges*: requer que o conjunto de casos de teste execute todas as arestas $(n_i^t, n_j^q) \in E$. Este critério já é utilizado em programas seqüenciais e paralelos baseados em passagem de mensagens.

Critérios baseados em fluxo de dados e de comunicação

Critério *All-defs-comm*: o conjunto de casos de teste deve executar caminhos que exercitem ao menos uma a associação *comm-c-use* ou *comm-p-use* para cada ponto de definição da variável $x \in V_c$.

Critério *All-defs*: o conjunto de casos de teste devem executar caminhos que exercitem ao menos uma associação *c-use*, *p-use*, *comm-c-use* ou *comm-p-use* para cada ponto de definição da variável $x \in \text{def}(n_i^t)$. Este critério já é utilizado em programas paralelos baseados em passagem de mensagens.

Critério *All-comm-c-uses*: o conjunto de casos de teste deve executar caminhos que exercitem todas as associações *comm-c-use*.

Critério *All-comm-p-uses*: o conjunto de casos de teste deve executar caminhos que exercitem todas as associações *comm-p-use*.

Critério *All-c-uses*: o conjunto de casos de teste deve executar caminhos que exercitem todas as associações *c-use*. Este critério já é utilizado em programas seqüenciais e paralelos baseados em passagem de mensagens.

Critério *All-p-uses*: o conjunto de casos de teste deve executar caminhos que

exercitem todas as associações *p-use*. Este critério já é utilizado em programas sequenciais e paralelos baseados em passagem de mensagens.

Critério *All-sync-uses*: o conjunto de casos de teste deve executar caminhos que exercitem todas as associações *sync-use*. Este critério já é utilizado em programas paralelos baseados em passagem de mensagens.

Na Tabela 4.8 é mostrado alguns dos elementos requeridos para os critérios definidos nessa seção. Esses elementos requeridos são obtidos a partir da análise estática do programa.

Tabela 4.8: Alguns dos elementos requeridos para os critérios de cobertura apresentados.

Critério	Elementos requeridos	Total
All-p-nodes	$3^0, 5^0, 6^0, 8^0, 9^0, 10^1, 11^1, 12^1, 9^2, 10^2, 12^2$	11
All-w-nodes	$10^0, 11^0, 1^1, 5^1, 6^1, 1^2, 4^2, 5^2$	8
All-nodes	$1^0, 2^0, 3^0, \dots, 12^0, 1^1, 2^1, 3^1, \dots, 12^1, 1^2, 2^2, 3^2, \dots, 12^2$	36
All-s-edges	$(3^0, 6^1), (3^0, 5^2), (5^0, 5^1), (6^0, 5^1), (8^0, 1^1), (9^0, 1^2), (10^1, 5^2), (10^1, 6^1), (11^1, 4^2), (12^1, 10^0), (12^1, 11^0), (9^2, 6^1)$	16
All-edges	$(1^0, 2^0), (2^0, 3^0), \dots, (11^0, 12^0), (1^1, 2^1), (2^1, 3^1), \dots, (11^1, 3^1), (3^1, 12^1), (1^2, 2^2), \dots, (11^2, 3^2), (3^2, 12^2), (3^0, 6^1), (3^0, 5^2), \dots, (9^0, 1^2), (10^1, 6^1), \dots$	51
All-defs-comm	$(1^0, 7^1, \text{avail}), (8^1, 7^1, \text{avail}), (7^2, 8^2, \text{avail}), (7^1, 8^2, \text{queue})$	4
All-defs	$(1^0, 8^2, \text{avail}), (2^1, (3^1, 4^1), \text{prod}), (4^1, 7^1, \text{item}), (7^1, 8^2, \text{queue}), (8^1, 7^1, \text{avail}), (6^2, (3^2, 12^2), \text{cons})$	10
All-comm-c-uses	$(1^0, 7^1, \text{avail}), (1^0, 7^2, \text{avail}), (7^2, 7^2, \text{avail}), (7^2, 8^1, \text{avail}), (7^1, 8^2, \text{queue}), \dots$	13
All-comm-p-uses	$\emptyset$	0
All-c-uses	$(8^1, 7^2, \text{avail}), (7^2, 8^2, \text{avail}), (2^1, 9^1, \text{prod}), (9^1, 9^1, \text{prod}), (4^1, 7^1, \text{item}), (7^1, 8^2, \text{queue}), (8^2, 11^2, \text{my_item}), \dots$	16
All-p-uses	$(2^1, (3^1, 4^1), \text{prod}), (2^1, (3^1, 12^1), \text{prod}), (6^2, (3^2, 4^2), \text{cons}), (6^2, (3^2, 12^2), \text{cons}), \dots$	8
All-sync-uses	$(2^0, (3^0, 6^1), \text{mutex}), (2^0, (3^0, 5^1), \text{mutex}), (5^0, (6^0, 5^1), \text{empty}), (6^1, (10^1, 6^1), \text{mutex}), (5^2, (9^2, 6^1), \text{mutex}), \dots$	14

4.5 Execução determinística

Quando se está desenvolvendo a atividade de teste é de grande utilidade poder repetir a execução de um caso de teste, a fim de observar com mais atenção um erro encontrado. Em programas seqüenciais isso é uma característica inerente desse tipo de programa, pois com as mesmas entradas garante-se a repetição de uma execução anterior. Isso se dá porque programas seqüenciais são determinísticos.

Já para programas concorrentes, não basta apenas fornecer as mesmas entradas para repetir uma execução, pois esse tipo de programa é por natureza não-determinístico – duas execuções de um programa concorrente com a mesma entrada, podem produzir resultados diferentes e ainda sim serem corretos. Uma maneira para conseguir a execução determinística de programas concorrentes é fornecer meios para controlar a execução desses programas.

Na Figura 4.23 é mostrado o exemplo em que os nós 8^1 e 9^1 na *thread* t^1 têm operações *wait* não-determinísticas, e os nós 2^0 (t^0) e 2^2 (t^2) são nós com a operações *post* para t^1 . Todas essas operações são sobre a mesma variável semáforo. Nesse caso as duas seqüências de sincronizações possíveis são: $(2^0, 8^1)$ e $(2^2, 9^1)$; ou $(2^2, 8^1)$ e $(2^0, 9^1)$.

A execução determinística é útil para realizar o teste de regressão e também para forçar uma seqüência de sincronização específica. Duas outras características úteis para o teste estrutural abordado neste trabalho.

Nesta seção é apresentado um método adaptado daquele apresentado por Carver e Tai (1991). O método proposto por Carver e Tai foi escolhido porque é específico para tratar programas nos contexto de memória compartilhada. Outro fator que colaborou para a escolha deste método é o conhecimento prévio do grupo de pesquisadores nesse método, quando o mesmo foi adaptado para o contexto de passagem de mensagem. Neste trabalho, o método foi adaptado para contemplar as sincronizações obtidas pela inicialização e finalização de *threads*.

Na proposta original, Carver e Tai (1991) descrevem que o projeto de uma ferramenta para execução determinística deve contemplar 3 passos: (1) definir um formato para representar a seqüência de sincronização de acordo com as primitivas de sincronização disponíveis na linguagem; (2) desenvolver um mecanismo para coletar as seqüências de

sincronização durante a execução do programa; e (3) desenvolver um mecanismo para repetir uma execução forçando a seqüência de sincronização coletada.

Figura 4.23: Exemplo de não-determinismo.

No contexto de semáforos, a seqüência de sincronização é representada com três operações possíveis: (1) o início da execução de uma operação *wait* (pode bloquear a *thread* ou não); (2) a conclusão da execução de uma operação *wait*; e (3) a execução de uma operação *post* (a qual nunca bloqueia). Com isso a execução de um programa pode ser caracterizado por uma seqüência de tais eventos de sincronização, denotada por:

$$((t^1, h^1, s^1), (t^2, h^2, s^2), \dots)$$

em que (t^i, h^i, s^i) representam a i -ésima sincronização, t^i representa a *thread*, h^i o tipo de sincronização (*post* ou *wait* nesse caso) e s^i a variável semáforo. Assim de posse da seqüência de sincronização exercitada e dos valores de entrada para o programa é possível repetir deterministicamente a execução desse programa. No entanto, eles constataram que uma versão simplificada dessa seqüência de sincronização necessita apenas conter o número da *thread* que completa a execução da operação *wait*. Essa versão simplificada é denotada por $(t^1, t^2, \dots)$ em que t^i corresponde a i -ésima operação *wait* completada.

Neste trabalho o modelo utilizado para representar as seqüências de sincronização é o simplificado. Porém foi incluído nessa seqüência a informação do evento *post* com qual o evento *wait* “casou”, visto que essa informação é importante para atribuir *timestamps* (método

discutido anteriormente). Também é produzido uma seqüência simplificada para cada objeto de sincronização. Isso permite ordenar os eventos apenas parcialmente, porém já é suficiente para atingir o objetivo de repetir uma execução do programa concorrente.

O projeto de uma ferramenta para coletar as seqüências de sincronização podem seguir duas linhas principais: (1) instrumentar internamente as primitivas de sincronização *post* e *wait*; (2) instrumentar o programa para interceptar as chamadas às primitivas de sincronização. Neste trabalho de mestrado a opção foi instrumentar o programa. Mais detalhes para essa escolha e como ela foi implementada são apresentados no Capítulo 5.

Por fim, para o projeto da ferramenta de reexecução Carver e Tai (1991) propõem que se utilize as mesmas primitivas de sincronização do programa original. Em seguida, propõem que o programa seja instrumentado incluindo chamadas a rotinas de controle. Basicamente uma rotina de controle é incluída, chamada de *permit* e precede as primitivas de sincronização. Essa rotina é *thread-safe*, ou seja, apenas uma *thread* é permitida executá-la por vez. Isso faz com que o controle seja distribuído entre as *threads*, sendo elas responsáveis por decidirem quem poderá executar uma sincronização.

Para programas baseados em semáforos e que utilizam a seqüência de sincronização na forma simplificada, a instrumentação necessária é modificar as operações *waits* para que pareçam com o seguinte padrão:

```
Request_S_Permit(TID);  
wait(...);  
Release_S_Permit();
```

A rotina *Request_S_Permit* é *thread-safe* e somente permitirá a *thread* retornar de sua chamada se a sincronização a ser executada for a da vez segundo a seqüência de sincronização. Caso contrário, a *thread* é bloqueada, liberando antes o acesso à rotina *Request_S_Permit* para que outra *thread* possa executá-la. Após uma *thread* concluir a execução da rotina *Request_S_Permit* com sucesso, a rotina faz a sincronização desejada e então deve executar a rotina *Release_S_Permit*. Essa rotina libera o acesso à rotina *Request_S_Permit* e libera as *threads* que foram bloqueadas na tentativa de executar a rotina *Request_S_Permit*.

Nesse ponto mais uma adaptação foi feita para este trabalho, estendendo a aplicação

das rotinas *Request_S_Permit* e *Release_S_Permit* à primitiva *post*, bem como para as primitivas referentes à inicialização e à finalização de *threads*. Estas são tratadas como *post* e *wait* de acordo com a Tabela 2.1. O controle da execução da primitiva *post* se fez necessário uma vez que na proposta de atribuição de *timestamps* é considerado o critério **LIFO** para o casamento entre *waits* e *posts* não-determinísticos.

4.6 Considerações finais

O teste de programas concorrentes é uma tarefa muito difícil devido a aspectos tais como: sincronização, comunicação e não-determinismo. No entanto, é fundamental para garantir a qualidade de software. No teste estrutural é desejável possuir critérios de cobertura de teste que possibilitem ao engenheiro de teste projetar testes que cubram características estruturais do programa em teste.

Este capítulo apresenta um modelo de teste e critérios de cobertura para programas concorrentes baseados em memória compartilhada. Esse modelo foi baseado no modelo *PCFG* para o contexto de passagem de mensagem além de outros trabalhos correlatos. Ele contempla basicamente a análise estática do programa, o levantamento de elementos requeridos para os critérios propostos e uma metodologia para analisar a cobertura da comunicação.

O maior desafio na definição desse modelo foi tratar a comunicação que se dá de forma implícita. A solução adotada foi tratar da ordenação de eventos concorrentes de definição e uso de variáveis compartilhadas com uso de *timestamps*. O uso de timestamps com essa finalidade é original e, para isso, algumas adaptações precisaram ser realizadas.

As principais contribuições do modelo proposto são: (1) tratar isoladamente a comunicação no contexto de memória compartilhada fornecendo critérios de cobertura específicos; (2) abordar as primitivas de inicialização e finalização de *threads* como primitivas básicas de sincronização permitindo obter mais detalhes e precisão na análise de cobertura dos critérios; e (3) a adaptação de um método de repetição de execução que pode ser usado para realizar testes de regressão ou para forçar a execução de seqüências de sincronizações específicas e, conseqüentemente, garantir a cobertura de elementos requeridos.

O próximo capítulo apresenta uma descrição da ferramenta implementada que apóia a

aplicação do modelo e dos critérios de teste definidos neste capítulo. A ferramenta ValiPThread foi implementada com o objetivo de mostrar a viabilidade e a aplicabilidade do modelo proposto.

ValiPThread: uma ferramenta de apoio ao teste de programas multithreads

5.1 Considerações iniciais

Este capítulo apresenta a concepção de uma ferramenta, chamada ValiPThread, para apoiar o teste de programas concorrentes com memória compartilha. Essa ferramenta implementa o modelo de teste *PCFGsm* (*Parallel Control Flow Graph for Shared Memory*) proposto neste trabalho.

A atividade de teste executada manualmente é uma tarefa demorada e suscetível a erros. Por esse motivo é necessário usar uma ferramenta que apóie todo ou parte do processo de teste. Com essa ferramenta é possível verificar a aplicabilidade do modelo de teste proposto nesta dissertação de mestrado. Além disso, é uma alternativa para a validação e verificação de programas *multithreads* implementados em C ANSI utilizando a biblioteca PThreads.

O capítulo é iniciado com a apresentação da arquitetura da ferramenta, que é composta por quatro módulos: *valipthread_inst*, *valipthread_elem*, *valipthread_exec* e *valipthread_eval*. Os módulos são então apresentados em detalhes e para cada um deles são

apresentadas suas limitações, usando para tanto um exemplo de uso. Os principais desafios enfrentados no desenvolvimento desses módulos foram: (1) limitar o escopo da ferramenta e (2) utilizar com outras ferramentas de apoio, também limitadas pelo fato de serem herdadas, muitas delas, da programação seqüencial. Na seqüência é apresentada, para efeitos de documentação, como se obter, instalar e configurar a ferramenta. Por fim, a última seção apresenta as considerações finais do capítulo.

5.2 Arquitetura da ferramenta

A ferramenta ValiPThread possui as seguintes funcionalidades: (1) criar sessões de testes; (2) obter e executar casos de testes; e (3) avaliar a cobertura dos critérios de teste definidos pelo modelo de teste *PCFGsm*. A arquitetura da ferramenta foi baseada na arquitetura ValiPar, esta proposta por Souza *et al.* (2005) para programas paralelos em ambientes de passagem de mensagem, com as devidas adaptações ao contexto de programas baseados em memória compartilhada.

A arquitetura da ValiPThread é mostrada na Figura 5.24 e é composta por quatro módulos principais: (1) **valipthread_inst**, responsável por fazer a análise estática do programa em teste e gerar uma versão desse programa devidamente instrumentado; (2) **valipthread_elem**, responsável por gerar os elementos requeridos para cada um dos critérios definidos no modelo e implementados na ferramenta; (3) **valipthread_exec**, responsável por compilar o programa instrumentado e auxiliar a condução da execução dos casos de testes fornecidos pelo usuário. Além disso, auxilia a execução determinística do programa; e (4) **valipthread_eval**, responsável por atribuir *timestamps* aos eventos executados pelo programa e avaliar a cobertura obtida para um ou mais critérios desejados.

Figura 5.24: Arquitetura da ferramenta ValiPThread.

As entradas e saídas dos módulos são feitas por meio de arquivos textos. A vantagem dessa abordagem é que, se desejar, o usuário pode ler e interpretar os resultados parciais do processo de teste. Cada módulo é apresentado em detalhes nas sub-seções seguintes.

5.2.1 O Módulo valipthread_inst: análise estática e instrumentação

O processo de teste inicia a partir desse módulo. Ele é responsável por realizar a análise estática do programa concorrente de acordo com modelo *PCFGsm* e também instrumentá-lo. Esse módulo recebe como entrada o programa concorrente escrito em C ANSI em conjunto com a biblioteca PThreads. São produzidos três arquivos como saída deste módulo: (1) uma versão instrumentada do programa original; (2) para cada função definida no programa, incluindo a função *main*, produz: (i) um arquivo *C.nome_da_função.dot* que descreve o *CFG* correspondente em modo texto e (ii) informações de fluxo de dados, ou seja, para cada nó do grafo são relacionadas quais variáveis são definidas e/ou usadas; e (3) produz um arquivo que relaciona as variáveis compartilhadas do programa, classificando-as em “de comunicação” e “de sincronização”.

Uma grande parte do trabalho destinado a esse módulo é feito pela ferramenta de apoio IDeL (SIMÃO *et al.*, 2003). A IDeL (*Intrumentation Description Language*) é capaz de produzir instrumentadores para diversas linguagens de programação e fazer a análise estática de programas seqüenciais. Para configurar a IDeL basta definir a gramática livre de contexto que descreve a linguagem desejada e a partir dessa descrição produzir (compilar) um instrumentador específico. Em seguida, um arquivo com a extensão *.idel* precisa ser produzido definindo as regras de instrumentação e de análise sintática. Mais detalhes sobre a IDeL são apresentados na Seção 3.2.4. No Apêndice A está listado os arquivos *C.l* e *C.y*, os quais descrevem a versão implementada da gramáticas da linguagem C ANSI e o arquivo *pthread.idel* produzido especificamente para o contexto da ferramenta ValiPThread, o qual descreve as regras de instrumentação a ser aplicada pela IDeL.

Inicialmente, o módulo produz uma nova versão do programa removendo os comentários e espaços em branco, pois é assim que a IDeL espera receber o programa a ser instrumentado/analísado. Em seguida é executado o *script preIDeL.pl*, escrito na linguagem Perl. Esse *script* tem basicamente a função de extrair do código fonte a relação de variáveis compartilhadas, classificando-as em comunicação e sincronização. As informações relativas as variáveis compartilhadas são registradas no arquivo *shared.var*. Outra informação produzida por esse *script* é obter o nome das funções que correspondem às *threads* (“ponto-de-entrada” de cada *thread*). O nome dessas funções são registradas no arquivo *thread.names*.

Outra função do *preIDeL.pl* é transformar, provisoriamente, a chamada da função *sem_init* em *n* chamadas *sem_post*, em que *n* corresponde ao valor de inicialização do semáforo. Além disso, faz adaptações, inserindo funções para que “marcas” sejam inseridas no início e no final do escopo das funções que correspondem a *threads* para representar, respectivamente, sincronizações de inicialização e finalização de *threads*. Esses tratamentos permitem que a IDeL trate de forma generalizada o *sem_init*, a inicialização e a finalização de *threads*.

É também função do *preIDeL.pl* ajustar alguns aspectos do código fonte, estes necessários no contexto de programas concorrentes e da biblioteca PThreads, os quais não são compatíveis com a IDeL. A IDeL, por ser uma ferramenta acadêmica e de escopo reduzido, não está, na sua versão atual, preparada para lidar com esses aspectos. Um exemplo desses aspectos é: a definição de funções que retornam ponteiros para tipos indefinidos (*void **). No

padrão PThreads é definido que as funções que definem as *threads* sejam declaradas retornando tais tipos indefinidos (*void **). No entanto, a IDeL só permite definir funções que retornam tipos primitivos simples.

Outro aspecto que precisou ser tratado pelo *preIDeL.pl* foi a inclusão da palavra chave *struct* na declaração de variáveis do tipo *sem_t* e *pthread_t*, essenciais para a criação de programas concorrentes em PThreads. A IDeL consegue interpretar esses elementos com estruturas compostas, apenas se explicitamente assim definidos.

Em suma, o *script preIDeL.pl* obtém algumas informações preliminares a respeito do contexto do programa concorrente e prepara o código fonte para que a IDeL execute sem problemas.

Após a execução do *script preIDeL.pl*, o módulo *valipthread_inst* faz a chamada da IDeL passando os devidos parâmetros. O resultado dessa etapa será a produção do grafo **CFG** para cada função do programa e a produção da versão instrumentada do programa concorrente.

O programa instrumentado produzido por esse módulo pode apresentar dois comportamentos. Primeiro, permitir apenas a geração do *trace* (rastro) de execução do programa. Segundo, permitir a execução controlada do programa em teste. A execução controlada pode ainda ser usada para repetir uma execução previamente feita, ou para forçar outra sincronização possível. O modo de execução, determinística ou não-determinística, é definido por meio de parâmetros, estes passados ao módulo *valipthread_exec*. Com isso, o programa instrumentado é único para as duas abordagens.

Após a execução da IDeL, o módulo *valipthread_inst* faz a chamada a um segundo *script* Perl, o *posIDeL.pl*, que reverterá as modificações provisórias feitas pelo *preIDeL.pl* e fará os ajustes finais no programa instrumentado e nos arquivos do grafo **CFG** (*C.nome_da_função.dot*). Entre essas mudanças estão: remover a palavra reservada *struct* inseridas na declaração de variáveis do tipo *sem_t* e *pthread_t*; inserir a declaração das variáveis virtuais semáforo propostas no modelo **PCFGsm** para tratar a inicialização e finalização de *threads*; inserir o *#include* da *header* *valipthread.h* e *hvalipthread.h*.

O *valipthread.h* possui o protótipo das funções que permitem a produção do *trace* de execução e a execução controlada. Ele é único para qualquer programa em teste. O *hvalipthread.h* possui declarações de variáveis (aqueles semáforos virtuais para inicialização e

finalização de *threads*, entre outras) e funções específicas para o contexto de um programa.

Exemplo de uso

Para exemplificar o uso do módulo *valipthread_inst* considere o programa da Figura 2.6 que implementa o problema do produtor consumidor com *buffer* limitado usando a linguagem C ANSI e a biblioteca PThreads. Para executar o módulo basta fornecer um único parâmetro, o código fonte do programa a ser testado, como o exemplo abaixo.

```
$ valipthread_inst.sh prodcons.c
```

Esta execução irá produzir os seguintes arquivos no diretório local:

- prodcons.c_09_instrumentado_final.c: versão instrumentada do programa original *prodcons.c*;
- hvalipthread.h: *header* com algumas variáveis usadas pelo programa instrumentado que são específicas para o contexto do programa original *prodcons.c*;
- C.main.dot: descrição textual do grafo **CFG** da função *main* do programa *prodcons.c* que por sua vez corresponde à **thread principal** do programa;
- C.producer.dot: descrição textual do grafo **CFG** da função *producer* do programa *prodcons.c*. Corresponde à **thread produtora** do programa;
- C.consumer.dot: descrição textual do grafo **CFG** da função *consumer* do programa *prodcons.c*. Corresponde à **thread consumidora** do programa;
- shared.var: listas de variáveis compartilhadas do programa, classificadas por tipo comunicação ou sincronização; e
- thread.names: contém o nome de todas as funções que correspondem às *threads*. No caso do programa *prodcons.c* três nomes estão listados neste arquivo (*main*, *producer*, *consumer*).

É interessante observar que os arquivos de descrição textual dos grafos **CFG** de cada função gerada pelo módulo *valipthread_inst* (mais especificamente, pela IDeL) são compatíveis com o formato *.dot*. A partir de arquivos *.dot* a ferramenta *Graphviz* (AT&T,

2002) é capaz de produzir uma versão gráfica do mesmo, o que ajuda quando se deseja visualizar o grafo da função/programa. Porém a ferramenta gera a uma versão gráfica para cada arquivo .dot, o que não permite visualizar o programa concorrente como um todo. Em trabalhos recentes o grupo de pesquisa implementou uma interface gráfica para a ferramenta ValiPVM que é capaz de unir os arquivos .dot de cada função em único novo arquivo .dot que representa o programa concorrente completamente, incluindo as arestas de sincronização (SOUZA; SAWABE *et al.*, 2008). Em trabalhos futuros essa interface gráfica deve ser adaptada para a ferramenta ValiPThread.

Limitações do Módulo

Pelo fato desse módulo ser altamente dependente do trabalho realizado pela IDeL e ela, por sua vez, possuir apenas uma descrição resumida da gramática da linguagem, as seguintes limitações e problemas são apresentados:

- A IDeL não processa ou ignora comentários e linhas em branco, sendo assim, estes comentários são suprimidos em um pré-processamento.
- A declaração de tipos (aqueles declarados com **typedef**) não são suportados pela IDeL. Nesse caso, esses tipos não primitivos precisam ser declarados em um arquivo .h à parte e em seguida fazer o *include* dele no programa principal antes da execução do módulo. Esse arquivo à parte não será instrumentado e não oferecerá problema para a atividade de teste uma vez que as informações a serem incluídas nele (declaração de tipos compostos) não são de interesse do modelo/ferramenta (HAUSEN, 2005).
- As declarações de funções não podem retornar ponteiros para tipo algum de dados, caso contrário a IDeL não os interpreta como funções e, conseqüentemente, não produz o **CFG** para elas. No entanto, o padrão PThreads define que funções que correspondem às *threads* devem retornar ponteiros para tipo indefinido (*void **). Para contornar esse problema, em um pré-processamento os “(void *)” da declaração das funções são substituídos por uma marcação que não trará problema para a IDeL porém que será facilmente identificável para sua remoção

posteriormente.

- A declaração de variáveis precisam estar no escopo mais externo da função e de preferência no início dela, caso contrário a IDeL conta os nós erradamente (HAUSEN, 2005).
- A IDeL não identifica definição de variável quando esta ocorre junto à declaração.
- Por tratar apenas de gramáticas livres de contexto, a IDeL não identifica automaticamente definições de variável por dereferenciação, para o caso de variáveis do tipo vetor ou apontador. Dessa forma ficou decidido tratar toda referenciação de variável como uma possível definição da mesma (MALDONADO, 1991; HAUSEN, 2005).

A grande maioria desses problemas podem ser corrigidos com a reescrita da gramática da linguagem C ANSI descrita pela IDeL, buscando representar a versão completa da linguagem.

5.2.2 O Módulo *valipthread_elem*: obtenção dos elementos requeridos

O segundo passo na condução das atividades de teste com a ferramenta ValiPThread é gerar os elementos requeridos que devem ser cobertos para os critérios de teste implementados. Essa tarefa é feita pelo módulo *valipthread_elem*. Esse módulo recebe como entrada o *PCFGsm* e as informações de fluxo de dados, ambas as informações produzidas pelo módulo *valipthread_inst*. Ainda recebe como parâmetro a informação do número total de *threads* que compõem o programa e a lista de LTID (*Logical Thread Identification*) que cada função (“tipo de *thread*”) possuirá. Por exemplo, para o programa do Produtor/Consumidor mostrado na Figura 2.6, esse módulo seria executado da seguinte maneira:

```
$ ./valipthread_elem 3 "main(0)" "producer(1)" "consumer(2)"
```

Como resultado esse módulo produz os elementos requeridos para todos os critérios baseados em fluxo de controle e sincronização, e os baseados em fluxo de dados e comunicação definidos no Capítulo 4. Além disso, produz descritores para cada elemento requerido. Para atingir esses resultados são gerados os grafos(i) e grafos reduzidos de

herdeiros, ambos explicados a seguir.

Para construir o grafo reduzido de herdeiros, parte-se do conceito de arestas primitivas e de arestas herdeiras. Se todo caminho completo que inclui uma aresta x sempre inclui uma aresta y , então x é chamada de aresta primitiva e y de aresta herdeira de x , pois y encapsula informação a respeito da execução de x (CHUSHO, 1987). Em outras palavras uma aresta é dita herdeira se ela é sempre executada após a execução de outra. Assim aresta primitiva é aquela que não é herdeira de nenhuma outra.

O conceito de aresta de sincronização foi acrescentado ao conceito de aresta primitiva no contexto de programas concorrentes. Assim, o grafo reduzido de herdeiros deve apenas possuir arestas de sincronização e de comunicação. Com a eliminação das arestas herdeiras o número de arestas requeridas pela ValiPThread cai consideravelmente.

Vale salientar que a atual versão do módulo *valipthread_inst* não implementam o conceito de grafo reduzido de herdeiro para produzir o programa instrumentado e o grafo *PCFGsm*, ficando para o módulo *valipthread_elem* a responsabilidade de reduzir o *PCFGsm*. No entanto, seria de grande valia a redução do *PCFGsm* e do programa instrumentado já no módulo *valipthread_inst*, principalmente para poupar processamento posterior de outros módulos e também para reduzir o tamanho do *PCFGsm* e do programa instrumentado, favorecendo sua visualização e leitura por parte do usuário. Tal implementação está relacionada como trabalhos futuros no final desta dissertação.

Após produzir o grafo reduzido de herdeiros, um grafo(i) é elaborado a partir de cada nó i que possua definição de variável. Um dado nó k pertencerá ao grafo(i) se existir pelo menos um caminho de i para k livre de definição com relação a pelo menos uma das variáveis

V_1 definida em i . Sendo $C(i, k)$ o conjunto de todos os possíveis caminhos do nó i para o nó k no *PCFGsm*, todos os nós de $C_1 \in C(i, k)$, inclusive o nó k , serão incluídos no grafo(i) se e somente se $V_1 \in \text{def}(i)$, tal que C_1 é livre de definição c.r.a V_1 (HAUSEN, 2005). O grafo(i) é obtido com o objetivo de permitir associações entre a definição e o uso de variáveis para, em seguida, determinar os elementos requeridos para critérios baseados em fluxo de dados.

O *valipthread_elem* gera, para cada critério de cobertura proposto no Capítulo 4, um arquivo listando os elementos requeridos (por exemplo, *Todas-Arestas-S.req*) e um segundo

arquivo que lista o descritor (por exemplo, *Todas-Arestas-S.aut*) de cada elemento requerido. Um descritor é uma expressão regular que representa um possível caminho para cobrir o elemento requerido. Por exemplo, para o critério Todas-Arestas-S, a aresta “ $ni - p1 \quad nj - p2$ ” (descrição textual da aresta (n_i^{p1}, n_j^{p2})) requerida será coberta se o programa concorrente exercitar um caminho $\Pi = (PATHS, SYNCs)$ tal que $(n_i^{p1}, n_j^{p2}) \in SYNCs$. As expressões regulares são estruturalmente diferentes para cada critério. A seguir as expressões regulares que descrevem os descritores são apresentadas.

All-nodes $\rightarrow N^* ni - p N^*$, em que N é o conjunto de nós do CFG^p .

All-nodes-W $\rightarrow N^* ni - p N^*$, em que N é o conjunto de nós do CFC^p .

All-nodes-P $\rightarrow N^* ni - p N^*$, em que N é o conjunto de nós do CFG^p .

All-edges-S $\rightarrow N^* ni - p1 \quad nj - p2 N^*$, em que N é o conjunto de nós do CFG^{p2} .

All-edges $\rightarrow N^* ni - p \quad nj - p N^*$, em que N é o conjunto de nós do CFG^p , ou $N^* ni - p1 \quad nj - p2 N^*$, em que N é o conjunto de nós do CFG^{p2} .

All-c-uses $\rightarrow N^* ni - p Nnv^* nj - p$, em que Nnv é o conjunto de nós do CFG^p que não redefinem a variável x em questão.

All-p-uses $\rightarrow N^* ni - p Nnv^* nj - p [Nnv^* nj - p]^* nk - p$, em que Nnv é o conjunto de nós do CFG^p que não redefinem a variável x em questão.

All-comm-c-uses $\rightarrow$ Não há expressão regular, logo não há descritor.

All-comm-p-uses $\rightarrow$ Não há expressão regular, logo não há descritor.

Para os critérios de cobertura que possuem descritor, a estratégia é que o módulo *valipthread_eval* use-o para verificar se determinado elemento requerido foi coberto dado um caminho Π executado. Mais detalhes de como isso funciona serão apresentados a seguir. Contudo, para os critérios que envolvem comunicação (*All-comm-c-uses* e *All-comm-p-uses*) não foi possível obter descritores, assim como é possível fazer para os outros critérios. Isso devido à dificuldade de, durante a etapa de análise estática, determinar caminhos livres de definição com relação às variáveis compartilhadas. Dada essa dificuldade, decidiu-se obter os elementos requeridos para esses critérios fazendo todas as combinações possíveis entre a definição e o uso de variáveis, ignorando o fato de existir ou não caminho livre de definição e

até o fato de existir um caminho explícito entre a definição e o uso. Para avaliar a cobertura desses critérios foi utilizada a técnica de atribuição de *timestamps* a eventos concorrentes. Mais detalhes sobre o uso dos *timestamps* também são apresentados neste capítulo.

Quanto aos parâmetros passados ao módulo *valipthread_elem*, eles são basicamente dois: (1) o número total de *threads* que o programa poderá executar; e (2) uma lista com o nome dos “tipos das *threads*” (função onde a *thread* inicia sua execução) e o respectivo LTID das *threads* entre parênteses. O exemplo de execução do módulo *valipthread_elem* apresentado anteriormente, estabelece que o programa executará três *threads*, sendo uma “*main*” com LTID = 0, uma “*producer*” com LTID = 1 e uma “*consumer*” com LTID = 2. Observe que o LTID é lógico e pertence apenas ao contexto da ferramenta ValiPThread, pois quando o programa instrumentado for executado cada *thread* receberá o LTID lógico para a condução das atividades de teste, mas ainda sim o sistema operacional atribuirá um TID (*Thread Identification*) “real” para seu controle da *thread*.

Para criar mais *threads* de um “tipo” basta listar todos os LTIDs disponíveis para ela dentre os parênteses correspondentes. Por exemplo, para criar três *threads* produtoras, basta especificar algo como: “*producer(3,0,2)*”. Observe que não é necessário que haja seqüenciamento dos valores de LTIDs nem que estejam ordenados. Inclusive não há a restrição de que a *thread main* (principal) seja sempre a *thread* 0. A única restrição é que os valores de LTID sejam únicos no contexto de toda a aplicação e que permaneçam fixos durante toda a atividade de teste para todos os módulos da ValiPThread.

Exemplo de uso

Mais uma vez considerando o programa da Figura 2.6, a execução do módulo *valipthread_elem*, logicamente, precedida da execução do módulo *valipthread_inst*, obtém os elementos requeridos e os descritores para todos os critérios. Aqui, para exemplificar será mostrado o resultado apenas para o critério Todas-Arestas-S. A Figura 5.25(a) mostra o conteúdo do arquivo *Todas-Arestas-S.req* enquanto que a Figura 5.25(b) mostra o conteúdo do arquivo *Todas-Arestas-S.aut*. As listas seguem a mesma ordem nesses arquivos, ou seja, o quinto elemento requerido, por exemplo, corresponde ao quinto descritor.

1) 4-0 7-1	1) 1 3 3 1 1:Np1 2:4-0 0 2 1:Np1/7-1 3:7-1 0 3 3:N 0
2) 4-0 6-2	2) 1 3 3 1 1:Np2 2:4-0 0 2 1:Np2/6-2 3:6-2 0 3 3:N 0
3) 6-0 6-1	3) 1 3 3 1 1:Np1 2:6-0 0 2 1:Np1/6-1 3:6-1 0 3 3:N 0
4) 7-0 6-1	4) 1 3 3 1 1:Np1 2:7-0 0 2 1:Np1/6-1 3:6-1 0 3 3:N 0
5) 9-0 1-1	5) 1 3 3 1 1:Np1 2:9-0 0 2 1:Np1/1-1 3:1-1 0 3 3:N 0
6) 10-0 1-2	6) 1 3 3 1 1:Np2 2:10-0 0 2 1:Np2/1-2 3:1-2 0 3 3:N 0
7) 11-1 7-1	7) 1 3 3 1 1:Np1 2:11-1 0 2 1:Np1/7-1 3:7-1 0 3 3:N 0
8) 11-1 6-2	8) 1 3 3 1 1:Np2 2:11-1 0 2 1:Np2/6-2 3:6-2 0 3 3:N 0
9) 12-1 5-2	9) 1 3 3 1 1:Np2 2:12-1 0 2 1:Np2/5-2 3:5-2 0 3 3:N 0
10) 14-1 11-0	10) 1 3 3 1 1:Np0 2:14-1 0 2 1:Np0/11-0 3:11-0 0 3 3:N 0
11) 14-1 12-0	11) 1 3 3 1 1:Np0 2:14-1 0 2 1:Np0/12-0 3:12-0 0 3 3:N 0
12) 15-1 11-0	12) 1 3 3 1 1:Np0 2:15-1 0 2 1:Np0/11-0 3:11-0 0 3 3:N 0
13) 15-1 12-0	13) 1 3 3 1 1:Np0 2:15-1 0 2 1:Np0/12-0 3:12-0 0 3 3:N 0
14) 10-2 7-1	14) 1 3 3 1 1:Np1 2:10-2 0 2 1:Np1/7-1 3:7-1 0 3 3:N 0
15) 10-2 6-2	15) 1 3 3 1 1:Np2 2:10-2 0 2 1:Np2/6-2 3:6-2 0 3 3:N 0
16) 11-2 6-1	16) 1 3 3 1 1:Np1 2:11-2 0 2 1:Np1/6-1 3:6-1 0 3 3:N 0
17) 14-2 11-0	17) 1 3 3 1 1:Np0 2:14-2 0 2 1:Np0/11-0 3:11-0 0 3 3:N 0
18) 14-2 12-0	18) 1 3 3 1 1:Np0 2:14-2 0 2 1:Np0/12-0 3:12-0 0 3 3:N 0
19) 15-2 11-0	19) 1 3 3 1 1:Np0 2:15-2 0 2 1:Np0/11-0 3:11-0 0 3 3:N 0
20) 15-2 12-0	20) 1 3 3 1 1:Np0 2:15-2 0 2 1:Np0/12-0 3:12-0 0 3 3:N 0
(a)	(b)
Elementos Requeridos	Descritores

Figura 5.25: Conteúdo dos arquivos *Todas-Arestas-S.req* (a), e *Todas-Arestas-S.aut* (b) para o critério *Todas-Arestas-S*, considerando o programa da Figura 2.6.

Os descritores são expressos por uma notação textual que descreve o autômatos finitos que valida um elemento requerido, ou seja, se o caminho Π executado for aceito pelo autômato então o elemento requerido foi coberto.

Considerando o 9º elemento requerido para o critério *Todas-Arestas-S* (Figura 5.25), a Figura 5.26 mostra como fazer a interpretação dos descritores dos elementos requeridos. O primeiro elemento do descritor é o número do elemento requerido a que ele se refere, **9** no caso. Os três valores seguintes, **1 3 3**, especificam que o elemento requerido é descrito por **1** autômato composto por **3** estados e que tem como estado final o **3º** estado. Em seguida cada estado é descrito, como neste exemplo: **1 1:Np2 2:12-1 0**. Este estado é o de número **1**, a condição de transição do estado 1 para ele próprio é passar por qualquer nó do processo p2 (**1:Np2**). A condição de transição do estado 1 para o estado 2 é passar pelo nó 12 do processo 1 (**2:12-1**). Por fim, o **0** indica o fim da descrição do estado. Esse esquema é seguido para descrever o restante dos estados.

Figura 5.26: Como ler a descrição textual de um autômato finito.

A Figura 5.27 mostra a representação gráfica do autômato descrito na Figura 5.26.

Figura 5.27: Representação gráfica do autômato finito que corresponde ao elemento requerido descrito na linha 9 dos arquivos *Todas-Arestas-S.req* e *Todas-Arestas-S.aut* (Figura 5.25).

5.2.3 O Módulo *valipthread_exec*: execução de casos de teste

O módulo *valipthread_exec* é responsável por auxiliar a execução dos casos de testes. Porém, antes de iniciar a execução dos casos de teste é necessário compilar o programa instrumentado. Para isso esse módulo oferece um *shell script* que faz a referência à biblioteca dinâmica *libvalipthread.so* (*-lvalipthread*) de forma transparente para o usuário. Essa biblioteca implementa as funções que o código fonte instrumentado utiliza para produzir o

trace de execução e repetir uma execução (seqüência de sincronização). Todas as funções implementadas pela *libvalipthread.so* são prefixadas por **valipthread_** para evitar conflitos de nomes com funções de outras bibliotecas. Esse *script* também já inclui a referência à biblioteca *libpthread.so* (*-lpthread*), desobrigando o usuário de referenciá-la.

Para compilar o programa, o usuário deverá executar o *script valipthread_cc* especificando o código fonte instrumentado e os demais parâmetros como se fosse compilar o programa original com o *gcc*. Por exemplo, para compilar o programa instrumentado referente ao programa da Figura 2.6, deve-se digitar:

```
$ ./valipthread_cc prodcons.c_09_instrumentado_final.c -o prodcons_inst
```

Nesse caso nenhum parâmetro de *include* (*-I*) ou biblioteca dinâmica (*-l*) foi necessário. Porém, quando for o caso do programa original necessitar, eles devem ser especificados.

Após ter o programa instrumentado compilado, devem ser executados os casos de testes desejados. Para executar cada caso de teste individualmente deve-se proceder da seguinte forma:

```
$ ./valipthread_exec <num_caso_teste> run <num_threads> <prog_inst> ["<args_prog>"]
```

O parâmetro *num_caso_teste* é um valor numérico usado para identificar o caso de teste. O parâmetro *run* especifica que para esse caso de teste deseja-se obter o *trace* de sua execução não-determinística. O parâmetro *num_threads* deve informar quantas *threads* poderão ser criadas, no máximo. Esse parâmetro deve ser igual aquele passado para o módulo *valipthread_elem*. O parâmetro *prog_inst* corresponde ao binário do programa instrumentado. Por fim, em *args_prog* devem estar listadas, em uma única *string* (“...”), todos os parâmetros específicos para o programa em teste. Caso o programa não necessite de parâmetros nada precisa ser especificado. Os argumentos presentes em *<args_prog>* devem estar separados por espaços, do mesmo jeito que estariam se estivesse sendo executado programa sem a

ferramenta ValiPThread.

Esse módulo produz como saída o um sub-diretório para o caso de teste contendo: (1) os *traces* de execução das *threads* (*trace.<nome_thread>.t<num_thread>*), incluindo as seqüências de sincronizações exercitadas; (2) a ordem de eventos de sincronização executados para cada semáforo (*seq_sync.<nome_semaforo>.dat*); (3) um arquivo (*args*) que registra os parâmetros de entrada fornecidos para o programa; (4) outro arquivo (*stdout*) com o registro da saída produzida pelo programa; (5) um arquivo (*stdin*) com o registro do que foi fornecido pelo usuário por meio da entrada padrão; e (6) o arquivo *nthreads* que registra o número de *threads* que o programa criou/executou.

Execução não-determinística: opção *run*

O programa instrumentado inicia sua execução fazendo a chamada das funções *valipthread_alloc_init_vars* e *valipthread_init*. Isto pode ser acompanhado pelo código fonte do programa *prodcons.c* instrumentado e listado no Apêndice B. As funções *valipthread_alloc_init_vars* e *valipthread_init* farão a alocação de variáveis e a preparação do ambiente para acompanhar e produzir o *trace* de execução do programa. Em seguida, por meio da chamada à função *valipthread_init_reporitory*, um repositório (lista encadeada dinâmica) é inicializado para cada semáforo “virtual” que irá representar a inicialização e finalização das *threads*. Na seqüência, a chamada à função *valipthread_start* indica que uma *thread* (em primeiro lugar a *thread* “*main*”) está sendo iniciada e precisa que suas ações sejam acompanhadas.

As outras funções implementadas pela biblioteca compartilhada *libvalipthread.so* que aparecem no programa instrumentado são:

valipthread_check: registra no *trace* que a *thread* (*ltid*) executará a instrução referente ao nó (node) do *PCFGsm*;

valipthread_init_reporitory: essa função sempre é invocada quando o programa original inicializa um semáforo (*sem_init*), além de aparecer no início da função *main* para inicializar um repositório para cada semáforo “virtual” referente à inicialização e finalização de *threads*. Por consequência, nestes casos ela é acompanhada por uma estrutura de repetição

(for) para registrar a passagem (*valipthread_check*) pelos nós correspondentes ao desmembramento do *sem_init* em *n* execuções do *sem_post*. Para cada um desses *sem_posts* um “token” é adicionado no repositório do semáforo em questão, com a informação do nó e da *thread* à qual corresponde.

valipthread_add_post: essa função aparece sempre que uma chamada *sem_post* é feita ou sua equivalente para inicialização e finalização da *thread*. Ela irá incluir no repositório do semáforo em questão um “token” informando a *thread* e o nó do ***PCFGsm*** correspondente ao *post* executado. Ela também aparece na estrutura de repetição referente à função *valipthread_init_repository* para incluir no repositório os “tokens” referentes ao desmembramento do *sem_init* explicado anteriormente. No contexto de inicialização de *threads* ela aparece junto ao *pthread_create*, enquanto que no contexto de finalização de *threads* aparece junto às instruções de finalização de *threads* (*return*, *exit*, *pthread_exit*, etc).

valipthread_add_wait: essa função aparece sempre que uma chamada *sem_wait* é feita ou então a sua equivalente para inicialização e finalização de *thread*. Ela irá retirar do topo do repositório o “token” (critério ***LIFO***) que contém a informação do *post* com o qual o *wait* em questão deve ser pareado (“casou”). Caso o repositório esteja vazio, a *thread* é bloqueada até que um *post* seja executado e um “token” seja inserido no repositório. Após retirar o “token”, a *thread* registra no arquivo de *trace* a sincronização exercitada no seguinte formato: ***WAIT POST WAIT***. No contexto de inicialização de *threads* essa função aparece antes da execução da primeira instrução da função “*main*” da *thread*. Já no contexto de finalização de *threads* aparece junto à função *pthread_join*.

valipthread_new_thread: após criar uma *thread* por meio da função *pthread_create*, o programa instrumentado faz uma chamada à função *valipthread_new_thread* para registrar tal feito e adicionar um descritor da *thread* criada para controle da ValiPThread. Essa função é sempre seguida da função *valipthread_add_post* que registra o “*post*” referente à inicialização da *thread*.

valipthread_exit_thread: essa função é a contra-partida à função *valipthread_new_thread*. Ela aparece junto ao *pthread_join* para excluir o descritor da *thread* finalizada e registrar a sincronização referente à finalização da *thread*.

valipthread_finish: essa função é a contra-partida à função *valipthread_start*. Sempre que uma *thread* inicia sua execução, antes de qualquer coisa, ela deve executar a função

valipthread_start no programa instrumentado para registrar suas próprias informações no descritor dela, este criado pela função *valipthread_new_thread*. A função *valipthread_exit*, por sua vez, irá atualizar o descritor da *thread* incluindo informações referentes a sua finalização.

Ao finalizar a execução da *thread* principal (*main*) são executadas as funções *valipthread_alloc_dispose_vars* e *valipthread_finalize* que liberam a memória alocada pela ValiPThread além de tomar outras providências de liberação de recursos alocados e fechamentos de identificadores.

Observando atentamente o programa instrumentado é possível notar que além das chamadas às funções *sem_post* e *sem_wait* feitas pelo programa original, a instrumentação insere chamadas às funções *sem_post* e *sem_wait*, no entanto, fazendo referência a semáforos declarados pela ValiPThread especificamente para o contexto do programa em teste. Para cada semáforo no programa original a ValiPThread cria outros dois semáforos chamados *valipthread_sem_<nome_do_semáforo_original>* e *valipthread_sem_<nome_do_semáforo_original>*.

Essas chamadas incluídas pela instrumentação têm a função de garantir o acesso exclusivo (*valipthread_sem_x*) às estruturas de dados manipuladas pelas diversas funções “*valipthread_*”, além de evitar o *deadlock* (*valipthread_mut_x*). Isso ocorre porque a maioria das funções “*valipthread_*”, propositalmente, não foram implementadas de forma *thread-safe*. Elas manipulam estruturas de dados globais e compartilhadas sem que as *threads* tenham acesso exclusivo a elas.

Em outras palavras, no nível da biblioteca *libvalipthread.so* as funções não são *thread-safe*, ficando, ao nível da aplicação, a responsabilidade de torná-las *thread-safe* por meio de semáforos adicionais, fato este que acarreta na chamada a essas funções.

Essa abordagem foi tomada com o intuito de minimizar o impacto no não-determinismo da execução do programa concorrente. Pois se as funções da *libvalipthread.so* fossem implementadas de forma *thread-safe* tornaria a execução do programa mais rígida sendo as funções gargalos para a execução das *threads*. Outro fator é a portabilidade da ferramenta desenvolvida. Ao trazer a responsabilidade pela exclusão mútua para o nível de usuário, não são necessárias alterações em compiladores e/ou no sistema operacional utilizado.

Execução determinística: opção *rerun*

Além da execução com o intuito de apenas produzir o trace de execução, o módulo *valipthread_exec* permite fazer a execução controlada de um caso de teste anteriormente executado. Para isso o módulo deve ser invocado da seguinte forma:

```
$ ./valipthread_exec <num_caso_teste> rerun
```

sendo que *num_caso_teste* determina qual o caso de teste deve ter a execução repetida.

Para a execução determinística é usado o mesmo programa instrumentado para a execução não-determinística. No entanto, as funções implementadas pela *libvalipthread.so* sofrem uma ligeira modificação no seu comportamento quando o modo de execução é configurado para *rerun*.

Basicamente, o que muda no comportamento da execução do programa instrumentado neste caso é que a função *valipthread_init* irá ler e carregar para uma estrutura de pilha a sequência de eventos para cada semáforo. Essas informações estarão contidas nos arquivos *seq_sync.<nome_semaforo>.dat*. Então o programa instrumentado continua sua execução. No entanto, quando as funções *valipthread_add_wait* e *valipthread_add_post* forem executadas elas iram observar se o evento a que correspondem é exatamente o evento que está no topo da pilha aguardando para ser executado. Caso afirmativo o evento é executado, o elemento é retirado do topo da pilha e ocorre uma notificação para outras *threads* que estejam bloqueadas esperando por eventos dessa pilha. Caso contrário, a *thread* é bloqueada até que outra *thread* modifique a pilha e notifique todas as que estão bloqueadas. Esse controle também é feito para a inicialização e finalização das *threads*, uma vez que são eventos tratados como sincronizações comuns.

O restante das funcionalidades das funções são mantidas fazendo com que o módulo *valipthread_exec* produza novamente todas as informações de *traces*, seqüências de sincronizações, etc, em um novo diretório. Por fim essas novas informações podem ser comparadas com as da primeira execução para determinar se realmente as execuções foram

iguais.

O modo *rerun* permite ainda, com base em um caso de teste anteriormente executado, que o usuário manipule os arquivos de sincronização para forçar uma sincronização específica. Porém nesse caso é responsabilidade do usuário garantir que a nova sequência de sincronização sugerida é executável.

Exemplo de uso

A execução do primeiro caso de teste para o programa *prodcons.c* (Figura 2.6) é feito da seguinte forma:

```
$ ./valipthread_exec 1 run 3 ./prodcons_inst
```

Esse programa não recebe parâmetros de entrada. Um dos possíveis *traces* de execução está listado na Figura 5.28 e a ordem dos eventos de sincronização para cada semáforo está listada na Figura 5.29.

Os nós pelos quais o fluxo de execução passou no *trace* de execução são representados por dois valores (por exemplo, 1-0). Estes valores denotam, respectivamente, o nó e a *thread* correspondente no *PCFGsm*. No entanto, para os nós que representam eventos *post* um terceiro valor foi incluído para determinar a exata ordem entre esse tipo de evento. Esse terceiro valor corresponde ao contador de eventos do repositório do semáforo em questão no momento em que o evento *post* ocorreu. Esse terceiro valor permite determinar exatamente qual o *post* foi pareado (“casou”) com qual *wait*, quando um mesmo nó *post* é executado diversas vezes, como por exemplo em uma estrutura de repetição.

trace.main.t0									
1-0	0-0-0	1-0	2-0	3-0	4-0-0	5-0	6-0-0	7-0-1	8-0
9-0-0	10-0-0	11-0	14-1-0	11-0	12-0	14-2-0	12-0	13-0	
trace.producer.t1									
1-1	9-0-0	1-1	2-1	3-1	4-1	5-1	6-1	7-0-1	6-1
7-1	4-0-0	7-1	8-1	9-1	10-1	11-1-1	12-1-0	13-1	4-1
5-1	6-1	6-0-0	6-1	7-1	10-2-2	7-1	8-1	9-1	10-1
11-1-3	12-1-1	13-1	4-1	14-1-0					
trace.consumer.t2									
1-2	10-0-0	1-2	2-2	3-2	4-2	5-2	12-1-0	5-2	6-2
11-1-1	6-2	7-2	8-2	9-2	10-2-2	11-2-2	12-2	13-2	4-2
5-2	12-1-1	5-2	6-2	11-1-3	6-2	7-2	8-2	9-2	10-2-4
11-2-3	12-2	13-2	4-2	14-2-0					

Figura 5.28: Trace de cada thread para uma das possíveis execuções do programa *prodcons.c* (Figura 2.6).

Também é possível notar que quando ocorrem as sincronizações há a repetição do nó *wait* no *trace*, ficando a notação das sincronizações da seguinte forma: WAIT POST WAIT. Isso é feito para tornar mais natural a leitura, por parte do módulo *valipthread_eval*, das sincronizações exercitadas, pois é natural lê-se uma sincronização na ordem POST WAIT. No entanto, o primeiro WAIT é mantido nesta notação, pois nos casos de defeitos por falta de execução de um POST que libere um WAIT, fica registrado que tal WAIT foi executado.

No caso da *thread main*, que não é criada por nenhuma outra, e sim pelo usuário quando inicia o programa, a sincronização que representa sua inicialização é representada pela marca 0-0-0. Com essa marca, o módulo *valipthread_eval* entende que essa aresta de sincronização deve ser ignorada.

Nos arquivos que listam os eventos de sincronização (Figura 5.29), na ordem em que ocorreram para cada semáforo, os eventos *post* são representados por três valores (nó, *thread* e ordem), enquanto que os eventos *wait* são representados por cinco valores, dois referentes ao próprio *wait* (nó e *thread*) e os três últimos (nó, *thread* e ordem) referentes ao *post* com o qual foi pareado (“casou”).

É interessante observar aqui que, apesar do semáforo do tipo *sem_t* serem do tipo contador, é possível sugerir qual semáforo o programa faz comportar-se como *mutex* (semáforo binário) ou como contador mesmo. Por exemplo, na Figura 5.28 é possível notar que, independente no nome da variável ser *mutex* essa variável tem o comportamento de um semáforo binário, pois os eventos *post* e *wait* se intercalam nunca havendo dois do mesmo

tipo juntos. Enquanto isso, o semáforo *empty* pode-se afirmar que se comporta como um semáforo contador, pois há dois eventos do mesmo tipo (*wait*) juntos. Quanto ao semáforo *full* fica sugerido que ele se comporta como um semáforo binário, no entanto sabe-se que dada a lógica do programa ele é um semáforo contador e que essa sequência de eventos foi uma casualidade. Esse tipo de análise, junto com o conhecimento do programa por parte do testador, podem ser combinadas para indicar um mau comportamento do programa.

seq_sync.mutex.dat post:4:0:0 wait:7:1:4:0:0 post:11:1:1 wait:6:2:11:1:1 post:10:2:2 wait:7:1:10:2:2 post:11:1:3 wait:6:2:11:1:3 post:10:2:4	seq_sync.empty.dat post:6:0:0 post:7:0:1 wait:6:1:7:0:1 wait:6:1:6:0:0 post:11:2:2 post:11:2:3	seq_sync.full.dat post:12:1:0 wait:5:2:12:1:0 post:12:1:1 wait:5:2:12:1:1
seq_sync.consumer.dat post:10:0:0 wait:1:2:10:0:0	seq_sync.producer.dat post:9:0:0 wait:1:1:9:0:0	seq_sync.valipthread_f_thread.dat post:14:1:0 post:14:2:0 wait:11:0:14:1:0 wait:12:0:14:2:0

Figura 5.29: Seqüência de eventos de sincronização para cada um dos semáforos do programa *prodcons.c* (Figura 2.6) considerando um dos possíveis traces de execução.

Quanto aos semáforos “virtuais” *consumer* e *producer*, apenas um par de sincronização está listado porque apenas uma *thread* foi criada para cada “tipo” de *thread*. Se forem criadas, por exemplo, três *threads producer*, devem haver seis eventos de sincronização, três *posts* e três *waits*, para o semáforo “virtual” *producer*. É fácil perceber que para os semáforos “virtuais” referentes à inicialização de *threads* seu comportamento é o mesmo de um semáforo contador. Por exemplo, considere três *threads* criadas em um primeiro momento, causando a produção de três *posts* referentes a essa criação. Em um segundo momento as *threads* são iniciadas e então consomem cada um dos *posts* criados.

O semáforo “virtual” *valipthread_f_thread* corresponde às sincronizações de finalização de todas as *threads* do programa, exceto a *thread main* que não sincroniza sua

finalização com nenhuma outra. A finalização de todas as *threads* fazem referência a um mesmo semáforo (*valipthread_f_thread*). Isso ocorre porque durante a análise estática é muito difícil relacionar a qual *thread* específica um *pthread_join* específico faz referência, uma vez que a *thread* que terá sua finalização sincronizada com o *pthread_join* é definida por uma variável do tipo *pthread_t*, cujo valor é atribuído em tempo de execução. Por esse motivo, na construção do grafo *PCFGsm* é feita a combinação todos-para-todos entre as finalizações de *threads* e os *pthread_join*.

5.2.4 O Módulo *valipthread_eval*: avaliação da cobertura do código

Após ter os casos de testes executados, pode-se avaliar a cobertura para os critérios de teste desejados. O módulo *valipthread_eval* é responsável por fazer essa avaliação. Para isso ele recebe como entrada um critério de teste escolhido pelo usuário, os descritores e elementos requeridos para o critério escolhido, e as informações obtidas com a execução dos casos de teste.

Para executar, esse módulo deve-se proceder da seguinte maneira:

```
$./valipthread_eval <critério> <n> "<f>(...)" ["<f>(...)"]*
```

Como o próprio nome indica, o parâmetro <critério> define qual o critério foi escolhido pelo usuário. Os critérios de testes disponíveis são: All-nodes, All-nodes-W, All-nodes-P, All-edges, All-edges-S, All-c-uses, All-p-uses, All-comm-c-uses e All-comm-p-uses. O parâmetro *n* especifica o número de *threads* que o programa pode ter executado. Esse parâmetro, mais uma vez, tem que ser igual aquele fornecido durante o levantamento de elementos requeridos (módulo *valipthread_elem*). Por fim, é preciso especificar a lista de *threads* com seu respectivos LTIDs. Essa informação também tem que ser mantida igual àquela fornecida ao módulo *valipthread_elem*.

Como saída o módulo produzirá a lista de elementos requeridos que foram cobertos, especificando qual caso de teste foi responsável pela sua execução, ignorando a informação

de casos de testes subsequentes que também o teriam executado. Uma segunda lista relaciona os elementos requeridos não cobertos. Por fim, é mostrada a relação percentual de elementos requeridos cobertos.

Para determinar se um elemento requerido foi coberto, o módulo *valipthread_eval* utiliza duas abordagens diferentes, dependendo do critério selecionado. Porém ambas as abordagens partem da análise do *trace* (caminho) executado pelo programa.

Avaliação de cobertura por descritores

Os critérios que são baseados em fluxo de controle, fluxo de sincronização e fluxo de dados locais às *threads* (variáveis privadas) podem ser avaliados por meio de descritores. São eles: All-nodes, All-nodes-W, All-nodes-P, All-edges, All-edges-S, All-c-uses e All-p-uses. Um descritor descreve um elemento requerido. Mais detalhes relativos aos descritores foram apresentados na Seção 5.2.2 (*valipthread_elem*).

Nesse caso, um elemento requerido é dito coberto se o *trace* executado por algum dos casos de teste é reconhecido pelo autômato finito (descritor) correspondente ao elemento requerido.

Exemplo de uso

A execução do módulo *valipthread_eval* para o programa *prodcons.c* (Figura 2.6), com o critério Todas-Arestas-S e o caso de teste 1 (vide exemplo de uso do módulo *valipthread_exec*), produzirá o resultado mostrado na Figura 5.30.

```

-- ELEMENTOS REQUERIDOS COBERTOS --
1) 4-0 7-1, coberto por valipthread/test_case0001
3) 6-0 6-1, coberto por valipthread/test_case0001
4) 7-0 6-1, coberto por valipthread/test_case0001
5) 9-0 1-1, coberto por valipthread/test_case0001
6) 10-0 1-2, coberto por valipthread/test_case0001
8) 11-1 6-2, coberto por valipthread/test_case0001
9) 12-1 5-2, coberto por valipthread/test_case0001
10) 14-1 11-0, coberto por valipthread/test_case0001
14) 10-2 7-1, coberto por valipthread/test_case0001
18) 14-2 12-0, coberto por valipthread/test_case0001
-- ELEMENTOS REQUERIDOS NÃO-COBERTOS --
2) 4-0 6-2
7) 11-1 7-1
11) 14-1 12-0
12) 15-1 11-0
13) 15-1 12-0
15) 10-2 6-2
16) 11-2 6-1
17) 14-2 11-0
19) 15-2 11-0
20) 15-2 12-0

Taxa de cobertura para o critério Todas-Arestas-S: 50.00% (10 de 20)

```

Figura 5.30: Resultado da avaliação da cobertura para o critério Todas-Arestas-S, considerando o caso de teste 1 (exemplo de uso do `valipthread_exec`).

Avaliação de cobertura por *timestamps*

O módulo `valipthread_eval` faz uso de uma técnica de ordenação de eventos concorrentes para a análise dos elementos requeridos, referentes a critérios baseados em comunicação. Essa técnica funciona atribuindo-se *timestamps* a eventos concorrentes e em seguida comparando os *timestamps* para ordenar esses eventos. Um elemento requerido para esses critérios é dito coberto se o evento que faz definição da variável é o último antes do uso da variável compartilhada.

Exemplo de uso

A execução do módulo `valipthread_eval` para o programa `prodcons.c` (Figura 2.6), considerando o critério All-comm-c-uses e o caso de teste 1 (vide exemplo de uso do módulo `valipthread_exec`) produzirá o resultado mostrado na Figura 5.31.

```

-- ELEMENTOS REQUERIDOS COBERTOS --
1) <2-0, 8-1, aval>, coberto por valipthread/test_case0001 (aval)
3) <8-2, 8-1, aval>, coberto por valipthread/test_case0001 (aval)
4) <2-0, 9-1, aval>, coberto por valipthread/test_case0001 (aval)
6) <8-2, 9-1, aval>, coberto por valipthread/test_case0001 (aval)
8) <9-1, 8-2, aval>, coberto por valipthread/test_case0001 (aval)
10) <8-1, 9-2, queue>, coberto por valipthread/test_case0001 (queue)
13) <8-2, 9-2, aval>, coberto por valipthread/test_case0001 (aval)

-- ELEMENTOS REQUERIDOS NÃO-COBERTOS --
2) <9-1, 8-1, aval>
5) <9-1, 9-1, aval>
7) <2-0, 8-2, aval>
9) <8-2, 8-2, aval>
11) <2-0, 9-2, aval>
12) <9-1, 9-2, aval>
Taxa de cobertura para o critério Todos-Comm-c-Usos: 53.85% (7 de 13)

```

Figura 5.31: Resultado da avaliação da cobertura para o critério Todas-Arestas-S, considerando o caso de teste 1 (exemplo de uso do valipthread_exec).

Porém o resultado intermediário é a atribuição de *timestamps* aos eventos do *trace* executado para em seguida compará-los e chegar à cobertura para o critério selecionado. A Figura 5.32 mostra os valores de *timestamps* atribuídos aos eventos do *trace* da Figura 2.5.

# timestamp.trace.main.t0	# timestamp.trace.producer.t1	# timestamp.trace.consumer.t2
1-0 [1,0,0]	1-1 [9,1,0]	1-2 [10,0,1]
2-0 [2,0,0]	2-1 [9,2,0]	2-2 [10,0,2]
3-0 [3,0,0]	3-1 [9,3,0]	3-2 [10,0,3]
4-0-0 [4,0,0]	4-1 [9,4,0]	4-2 [10,0,4]
5-0 [5,0,0]	5-1 [9,5,0]	5-2 [10,12,5]
6-0-0 [6,0,0]	6-1 [9,6,0]	6-2 [10,12,6]
7-0-1 [7,0,0]	7-1 [9,7,0]	7-2 [10,12,7]
8-0 [8,0,0]	8-1 [9,8,0]	8-2 [10,12,8]
9-0-0 [9,0,0]	9-1 [9,9,0]	9-2 [10,12,9]
10-0-0 [10,0,0]	10-1 [9,10,0]	10-2-2 [10,12,10]
11-0 [11,25,10]	11-1-1 [9,11,0]	11-2-2 [10,12,11]
12-0 [12,25,25]	12-1-0 [9,12,0]	12-2 [10,12,12]
13-0 [13,25,25]	13-1 [9,13,0]	13-2 [10,12,13]
	4-1 [9,14,0]	4-2 [10,12,14]
	5-1 [9,15,0]	5-2 [10,22,15]
	6-1 [9,16,0]	6-2 [10,22,16]
	7-1 [10,17,10]	7-2 [10,22,17]
	8-1 [10,18,10]	8-2 [10,22,18]
	9-1 [10,19,10]	9-2 [10,22,19]
	10-1 [10,20,10]	10-2-4 [10,22,20]
	11-1-3 [10,21,10]	11-2-3 [10,22,21]
	12-1-1 [10,22,10]	12-2 [10,22,22]
	13-1 [10,23,10]	13-2 [10,22,23]
	4-1 [10,24,10]	4-2 [10,22,24]
	14-1-0 [10,25,10]	14-2-0 [10,22,25]

Figura 5.32: Timestamps atribuídos aos eventos do trace da Figura 2.5

5.3 Colocando a ValiPThread para executar

Essa seção descreve como preparar o ambiente para deixar a ValiPThread funcional. Para o correto funcionamento da ferramenta ValiPThread o seguinte ambiente deve está configurado:

- Ambiente UNIX e *shell* compatível com *bash*. As versões utilizadas no desenvolvimento são:
 - Sistema Operacional: Ubuntu 8.04 (Kernel: 2.6.24);
 - GNU bash, version 3.2.39(1)-release (i486-pc-linux-gnu).
- Arquitetura de hardware do microcomputador utilizado no desenvolvimento:
 - CPU: 4x Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz;
 - Memória RAM: 3GB;
 - Disco Rígido (HD): SATA – 320GB.
- A biblioteca PThreads instalada. A versão da biblioteca PThreads utilizada no desenvolvimento é a NPTL¹⁴ 2.7.
- IDeL, instanciada para a linguagem C. A Versão da IDeL utilizada é aquela depositada no repositório https://svn.lasdpc.icmc.usp.br/svn/ValiMPI/trunk/vali_inst/IDeL-fonte2/ sob a Revisão de número 15 .
- O pacote *libstdc++2.10-glibc2.2*, este requerido pela IDeL.
- Programa alvo dos testes em C ANSI usando a biblioteca PThreads, sintaticamente correto.
- Ferramentas auxiliares: *sed*, *indent*, *wc* e *awk*. As versões utilizadas nestas ferramentas foram: GNU sed version 4.1.5, GNU indent 2.2.9, wc (GNU coreutils) 6.10 e awk 1.3.3 Nov 1996, respectivamente.
- O interpretador de comandos Perl. A versão utilizada é v5.8.8.
- Variável de ambiente VALIPTHREAD_ROOT referenciando o diretório de

¹⁴ NPTL (*Native POSIX Thread Library*) e a nova implementação do Linux, a partir da versão 2.6 do *kernel*, para o padrão *POSIX Threads*.

instalação da ferramenta, conforme o exemplo mostrado na Figura 5.33. Nesse exemplo a ValiPthread foi instalada no sub-diretório *workspace* e a variável de ambiente `VALIPTHREAD_ROOT` deve referenciar o caminho completo: */home/fsarmanho/workspace*.

Figura 5.33: Exemplo da estrutura de diretório para a instalação da ValiPThread.

A atual versão da ferramenta está armazenada no servidor de controle de versão – Subversion – do grupo de pesquisa LaSDPC. Para obter a ferramenta basta fazer o *check out*. O seguinte comando deve ser usado para isso:

```
$ svn checkout https://svn.lasdpc.icmc.usp.br/svn/ValiPthread/trunk/ valipthread
```

Depois de obter a ferramenta é necessário compilar e instalar. Na raiz do diretório *valipthread* há um arquivo Makefile que faz a compilação de todos os módulos da ferramenta e cria no diretório `$HOME/bin` links simbólicos para cada módulo da ferramenta.

Para realizar tal *download*, faz-se necessária autorização e permissão de acesso ao servidor. Seguindo estes passos a ferramenta estará pronta para ser usada.

5.4 Considerações finais

Para conduzir as atividades de teste estrutural é necessário dispor de uma ferramenta que automatize, pelo menos, algumas das etapas dessa atividade. Na literatura não há uma grande oferta de ferramentas que tratem o teste estrutural de programas concorrentes na mesma proporção que têm para programas seqüências.

Este capítulo apresentou a ferramenta ValiPThread que implementa o modelo de teste proposto no Capítulo 4. A ferramenta trata do teste de programas *multithreads* implementados na linguagem C ANSI e usando a biblioteca PThreads. Sua principal característica é permitir a criação de seções de teste que podem ser interrompidas e retomadas a qualquer momento. Também permite salvar e executar casos de teste, além de ser capaz de avaliar a cobertura obtida para os critérios de teste baseados no grafo *PCFGsm*.

As principais contribuições do modelo de teste proposto e conseqüentemente da ValiPThread é fornecer critérios de cobertura focados especificamente para comunicação de threads, usando para isso uma técnica inovadora e complementar de ordenação de eventos concorrente baseada em *timestamps*. Também aborda a inicialização e finalização dinâmica de threads, como sincronizações. Por fim, implementa alguns critérios de cobertura baseados em fluxo de controle, sincronização e dados já conhecidos.

Os principais desafios enfrentados durante a proposta e o desenvolvimento dessa ferramenta foram: (1) lidar com as limitações da ferramenta IDeL, utilizada pelo módulo `valipthread_inst` para fazer a análise estática e instrumentar o programa concorrente; (2) investigar uma técnica para determinar possíveis comunicações em um programa *multithread*. Neste caso chegou-se à conclusão que esta tarefa extrapolava o contexto deste trabalho, decidindo-se então por considerar todas as combinações de definição e uso de variáveis compartilhadas; e (3) investigar uma técnica para determinar as comunicações exercitadas para uma execução do programa. Neste caso chegou-se à ordenação de eventos por meio de *timestamps*.

Resultados

6.1 Considerações finais deste trabalho

A programação concorrente é uma opção à demanda por maior poder e capacidade de processamento aliado ao menor tempo de respostas das aplicações. No contexto de programas *multithreads*, a popularização das arquiteturas *multicores* e o interesse em arquiteturas *petascales* tem requerido atenção nesta área.

É importante definir modelos de testes que garantam a qualidade desse tipo de programa. Muitos conhecimentos já adquiridos e consolidados para o teste de programas sequenciais podem ser utilizados para programas concorrentes. No entanto, a programação concorrente traz dificuldades adicionais que precisam ser tratadas. Esta dissertação de mestrado apresentou a proposta de um modelo de teste para o contexto de programas concorrentes *multithreads*.

A dissertação iniciou apresentando, no Capítulo 2, um estudo da área de programação concorrente contemplando aspectos de hardware e software, com uma visão *top-down*, até convergir para as arquiteturas multiprocessadas com memória compartilhada e para as bibliotecas de *threads*. O Capítulo 3 apresentou uma visão geral dos conceitos de teste de software dando maior atenção aos aspectos relativos à técnica estrutural. Em seguida

apresentou trabalhos relacionados ao teste de software aplicado a programas concorrentes, descrevendo o modelo baseado no Grafo de Fluxo de Controle Paralelo, ou PCFG em inglês, e a ferramenta ValiPar.

O modelo *PCFGsm* proposto neste trabalho está apresentado no Capítulo 4, contemplando a discussão a cerca das abordagens e decisões tomadas durante a proposta. O modelo considera a relação de causalidade (causa-e-efeito) para determinar sincronizações exercitadas, considerando a disciplina LIFO para obtenção dos pares de sincronização. Este modelo também generaliza a inicialização e a finalização de *threads* como sincronizações comuns. Critérios estruturais de cobertura são adaptados e propostos para o contexto de programas *multithreads*. É proposto também um método para revelar comunicações nesses programas, adaptando trabalhos anteriores (LAMPORT, 1978; CARVER; LEI, 1991). Nesse capítulo, a aplicação do modelo é ilustrada com um programa de exemplo. O Capítulo 5 apresentou a ferramenta ValiPThread implementada para apoiar a aplicação do modelo *PCFGsm*. Nesse Capítulo 5 são apresentadas as funcionalidades e como utilizá-la, mais uma vez ilustrando com um programa de exemplo. Os detalhes e decisões de implementação são discutidos.

Finalmente, este Capítulo 6 apresenta as considerações finais deste trabalho relacionando as principais contribuições e sugerindo trabalhos futuros.

6.2 Contribuições

O modelo de teste proposto neste trabalho é inovador por tratar as sincronizações decorrentes da inicialização e da finalização de *threads*. Elas são tratadas de forma generalizada, para isso, considerando um semáforo “virtual” para cada “tipo de *thread*”. Outra consideração que o modelo faz é que a inicialização de um semáforo é traduzida em n (valor inicial do semáforo) *post's*.

O modelo também inova por tratar a comunicação e sincronização separadamente, definindo critérios estruturais de cobertura para cada contexto. A determinação da cobertura para o critério referente à sincronização é obtido diretamente pela análise do *trace* de execução do programa. No entanto, dado que a comunicação em programas *multithreads* é implícita, para determinar a cobertura em relação aos critérios referentes à comunicação é

utilizado um método que atribui *timestamps* para eventos concorrentes e em seguida compara tais *timestamps* para determinar os pares de comunicação (definição e uso).

No momento da análise estática, quando são gerados os elementos requeridos referentes aos critérios de comunicação (Todos-comm-c-Usos e Todos-comm-p-Usos) o modelo não se preocupa com aqueles que são não-executáveis. Também não há essa preocupação para o critério referente à sincronização (Todos-sync-Usos). Para esses critérios o modelo propõe obter todas as combinações entre pontos de definição e de uso de cada variável compartilhada e todas as combinações entre *posts* e *waits* feito sobre uma mesma variável semáforo. Isso produz um elevado número de elementos requeridos e a maioria deles, geralmente, são não-executáveis. Os outros critérios de fluxo de controle e dados herdados diretamente do contexto de programas sequenciais conseguem ainda eliminar uma boa quantidade dos elementos requeridos não-executáveis por meio de análise de fluxo de dados e caminhos.

Evitar um elevado número de elementos requeridos não-executáveis é, de fato, uma necessidade do modelo que não foi tratada, dado o escopo do projeto. O refinamento da análise estática, a fim de restringir a quantidade de elementos requeridos não executáveis, é sugerido em trabalhos futuros, usando por exemplo, a execução simbólica (YOUNG; TAYLOR, 1988).

Esse mesmo problema ocorre na obtenção de elementos requeridos para os critérios referentes às sincronizações. Porém nesse sentido, trabalhos já têm sido feitos usando Redes de Petri para determinar sincronizações não-executáveis (ENDO, 2008).

A ferramenta ValiPThread implementada neste trabalho teve como principal objetivo instanciar a aplicação do modelo ***PCFGsm*** proposto. A ValiPThread foi implementada para uma versão reduzida da linguagem C ANSI, devido a algumas limitações pontuais da ferramenta IDeL, esta utilizada para a análise estática e instrumentação do código fonte. As principais funções da ValiPThread são criar, parar e retomar uma sessão de teste, gravar e executar casos de teste, e avaliar a cobertura de código de acordo com os critérios estruturais de cobertura propostos.

Resumidamente, as principais contribuições deste trabalho são:

- O modelo de teste ***PCFGsm*** proposto para o teste de programas *multithreads*.

As principais diferenças e contribuições em relação ao modelo ***PCFG*** – anteriormente

definido – são: (1) o contexto diferente – ambiente de memória compartilhada, (2) a desassociação da comunicação e da sincronização na realização da análise estática, (3) tratar as sincronizações devido à inicialização e à finalização de *threads*; (4) definir critérios de cobertura específicos para comunicação e sincronização e (5) adaptação de um método baseado em *timestamps* para avaliar a cobertura de critérios referentes à comunicação implícita que ocorre com o uso de variáveis compartilhadas.

- Os critérios estruturais de cobertura adaptados ao contexto de programas concorrentes com memória compartilhada. Os critérios que são herdados diretamente do modelo **PCFG** que por sua vez foram herdados da programação seqüencial, sofreram mínimas adaptações. São eles: All-nodes, All-nodes-P, All-nodes-W, All-edges, All-c-uses, All-p-uses, All-defs. Os critérios que tratam a comunicação e sincronização, o fazem separadamente e esses foram redefinidos, inclusive em relação ao **PCFG** que trata a sincronização e a comunicação de maneira inter-relacionada. Os critérios que tratam comunicação e sincronização são: All-edges-S, All-comm-c-uses, All-comm-p-uses e All-comm-defs.

- A adaptação de um método para ordenar parcialmente eventos concorrentes baseados nas sincronizações ocorridas, inclusive aquelas decorrentes da inicialização e da finalização de *threads*. Este método foi baseado em ordenação de eventos concorrentes gerados a partir da relação de causalidade entre os eventos considerados, sendo concebido a partir dos trabalhos de Lamport (1978) e de Lei e Carver (2006). Esse método introduz uma abordagem dinâmica à atividade de análise estática, postergando uma parte da análise para ser feita em tempo de execução. Isto viabiliza a recuperação da comunicação implícita realizada, permitindo então estabelecer a cobertura de elementos requeridos baseados nessa comunicação. Tal abordagem inédita posiciona o modelo proposto como uma versão híbrida entre estático e dinâmico.

- A implementação de uma ferramenta também inédita, a ValiPThread, que apóia a aplicação do modelo **PCFGsm** e dos critérios de cobertura; e

- A revisão bibliográfica que permitiu a atualização do conhecimento do grupo de pesquisa e colocou um novo seguimento de estudo – o de ordenação de eventos concorrentes.

Até a presente data as seguintes publicações foram geradas decorrente o desenvolvimento desse projeto de mestrado:

- SARMANHO, F. S. ; SOUZA, P. S. L. ; SOUZA, S. R. S. ; SIMAO, A. S. . Aplicação de Teste Estrutural para Programas Multithreads Baseados em Semáforos. In: **Workshop on Languages and Tools for Parallel and Distributed Programming**, 2007, Gramado, RS. Anais do Workshop on Languages and Tools for Parallel and Distributed Programming (LTPD'07, co-located with SBAC-PAD 2007), 2007.

Este artigo foi produto do estudo inicial que investigou a aplicabilidade do modelo PCFG ao contexto de memória compartilhada. Com esse estudo verificou-se que modelo PCFG poderia ser utilizado, caso mudanças fossem realizadas. A maior necessidade de alteração está relacionada à comunicação implícita da memória compartilhada, pois o PCFG trata a comunicação explícita do ambiente de passagem de mensagem.

- SARMANHO, F. S. ; SOUZA, P. S. L. ; SOUZA, S. R. S. ; SIMAO, A. S. . Structural Testing for Semaphore-Based Multithread Programs. In: **8th International Conference on Computational Science ICCS 2008, 2008**, Kraków. Lecture Notes in Computer Science. Berlin : Springer-Verlag, 2008. v. 5101. p. 337-346.

Este artigo apresenta o modelo de teste **PCFGsm** proposto por este trabalho de mestrado, destacando: o tratamento diferenciado e isolado da comunicação e da sincronização em memória compartilhada, a adaptação de um método de atribuição de *timestamps* a eventos concorrentes aplicado em um novo contexto – determinar a comunicação exercitada em programas concorrentes com memória compartilhada, e a consideração da inicialização e da finalização de *threads* como sincronizações.

6.3 Trabalhos futuros

Como continuação desse trabalho os seguintes tópicos são sugeridos para pesquisas

futuras:

- Estender o modelo para tratar outros tipos de sincronização. O tratamento das primitivas *sem_trywait*, *sem_timedwait*, seriam diretos, pois basta considerar que para essas primitivas há a possibilidade da sincronização ser ou não concretizada. Caso positivo, é como se uma sincronização comum ocorresse, caso contrário, é como se fosse um evento qualquer (sem sincronização). O tratamento de sincronizações coletivas, tais como *barreira* e *once*, podem ser feitos considerando um conjunto de *posts* e *waits*, porém análises mais detalhadas precisam ser feitas. E, finalizando, as primitivas de sincronização, tais como, semáforos leitores/escritores e variáveis de condição precisam ter um estudo específico para incorporá-las ao modelo.

- Como já foi discutido antes, o modelo gera a combinação de todas os pares de definição e uso de cada variável compartilhada para os elementos requeridos de comunicação. Porém muitos dos elementos requeridos são não-executáveis. Isso acarreta em uma explosão do número de elementos requeridos que precisam ser analisados, quanto à sua executabilidade e, se for o caso, cobertos por casos de teste. Um refinamento da análise estática fazendo, por exemplo, execução simbólica, poderia eliminar alguns dos elementos requeridos que são não-executáveis.

- O mesmo problema anterior ocorre para as sincronizações detectadas na análise estática proposta pelo modelo. Para eliminar as sincronizações não executáveis um refinamento da análise estática, como por exemplo, a modelagem por Redes de Petri, pode apresentar bons resultados.

- Durante a adaptação e desenvolvimento do método para ordenar eventos concorrentes por meio de *timestamps*, observou-se, como subproduto, que há casos de acesso concorrente às variáveis compartilhadas (ou região crítica) que podem ser detectados por ele. É sugerido que essa possibilidade seja investigada com mais propriedade a fim de verificar sua utilidade e eficiência.

- Recentemente o grupo concluiu a implementação de uma interface gráfica (SOUZA, SAWABE *et al.*, 2008) para a ferramenta ValiPVM. Dado que a ferramenta ValiPThread e ValiPVM segue a mesma arquitetura – a ValiPar – com as devidas particularidades decorrente o contexto de cada uma delas, é sugerido adaptar e estender essa interface gráfica para a ferramenta ValiPThread.

Referências Bibliográficas

- ALMASI, G. S.; GOTTLIEB A. **Highly Parallel Computing**, 2^a ed. The Benjamin Cummings Publishing Company, Inc., Redwood City, 1994.
- ANDREWS, G. R.; SCHINEIDER, F. B. Concepts and Notations for Concurrent Programming, **ACM Computing Survey**, New York, vol. 15 - 1, p. 3-43, 1983.
- AT&T Research Labs. **Graphviz - open source graph drawing software**. 2002. Disponível em: <http://www.research.att.com/sw/tools/graphviz/>. Último acesso em: 12/12/2008.
- BARNES, G. H.; BROWN, R.; KATO, M.; KUCK, D. J.; SLOTNICK, D. L.; STOKES, R. A. The Illiac IV computer, **IEEE Transactions on Computers**, IEEE Computer Society, Washington, vol. 17 - 8, p. 746-757, 1968.
- BEGUELIN, A. L. **Xab: a Tool for Monitoring PVM Programs**. Pittsburgh: Carnegie Mellon University, 1993. 6 p.
- BELL, G.; GRAY, J.; SZALAY, A. Petascale Computational Systems. **Computer**, IEEE Computer Society Press, Los Alamitos, vol. 39 – 1, p. 110-112, 2006.
- BUENO, P. M. S.; CHAIM, M. L.; JINO, M.; MALDONADO, J. C. ; VILELA, P. R. **POKE-TOOL - Manual do Usuário**. DCA-FEE-UNICAP, Campinas, 1994.
- CARVER, R. H.; TAI, K-C. Reproducible Testing of Concurrent Programs Based on Shared Variables. **Proceedings of the 6th International Conference on Distributed Computing Systems**, IEEE Computer Society Press, Cambridge, p. 428-433, 1986.
- CARVER, R. H.; TAI, K. Replay and Testing for Concurrent Programs. **IEEE Software**. IEEE Computer Society Press, Los Alamitos – CA, vol. 8 – 2, p. 66-74, 1991.
- CARVER, R. H.; TAI, K. **Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/WIN32 Programs**. John Wiley

- & Sons, Hoboken, 2006.
- CHAIM, M. L. **POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseados em Análise de Fluxo de Dados**. DCA-FEE-UNICAMP, Campinas, 1991.
- CHAPMAN, B.; JOST, G.; van der PAS, R.; KUCK, D. J. **Using OpenMP: Portable Shared Memory Parallel Programming**. The MIT Press, Cambridge, 2007.
- CHUSHO, T. Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing. **IEEE Transactions on Software Engineering**, IEEE Press Piscataway, vol. 13 – 5, p. 509-517, 1987.
- CORBETT, J. C.; DWYER, M. B.; HATCLIFF, J.; LAUBACH, S.; PĂȘĂREANU, C. S.; ROBBY; ZHENG, H. Bandera: extracting finite-state models from Java source code. **In Proceedings of the 22nd international Conference on Software Engineering (ICSE '00)**. ACM, New York, p. 439-448, 2000.
- DAMODARAN-KAMAL, S.K.; FRANCIONI, J.M., mdb: a semantic race detection tool for PVM, **Proceedings of the Scalable High-Performance Computing Conference**, p. 702-709, 1994.
- DEMILLO, R.; LIPTON, R.; SAYWARD, F. Hints on Test Data Selection: Help for the Practicing Programmer Computer. **IEEE Computer**, vol. 11 – 4, p. 34-41, 1978.
- DONGARRA, J.; FOSTER, I.; FOX, G. C.; GROPP, W.; KENNEDY, K.; TORCZON, L.; WHITE, A. **The Sourcebook of Parallel Computing**. Morgan Kaufmann Publishers Inc., San Francisco, 2002.
- DUNCAN, R. A Survey of Parallel Computer Architectures. **Computer**, IEEE Computer Society Press, Los Alamitos - CA, vol. 23 - 2, p. 5-16. 1990.
- EDELSTEIN, O.; FARCHI, E.; GOLDIN, E.; NIR, Y.; RATSABY, G.; UR, S. Framework for testing multi-threaded Java programs. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd., vol. 15 – 3-5, p. 485-499, 2003.
- ELMAS, T.; TASIRAN, S.; QADEER, S. VYRD+: verifYing concurrent programs by runtime refinement-violation detection. **In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)**. ACM, New York, p. 27-37, 2005.
- ENDO, A. **Teste de composição de web services: uma estratégia baseada em um modelo de teste de programas paralelos**. Universidade de São Paulo, Instituto de

- Ciências Matemáticas e de Computação (ICMC), São Carlos, 2008.
- FIDGE, C. Fundamentals of Distributed System Observation. **IEEE Software**. IEEE Computer Society Press, Los Alamitos, Vol. 13 – 6, p. 77-83, 1996.
- FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. Addison-Wesley Longman Publishing Co., Inc. Boston, 1995.
- FLYNN, M. J.; RUDD, K. W. Parallel architectures. **ACM Computing Surveys**, ACM Press, New York, vol. 28, p. 67-70, 1996.
- GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing**, The MIT Press, Cambridge, 1994.
- GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. **Introduction to Parallel Computing**, 2^a ed., Addison Wesley, New York, 2003.
- HANSEN, P. B. Testing a multiprogramming system. **Software – Practice and Experience**, vol. 5, p. 145-150, 1973.
- HANSEN, P. B. Reproducible testing of monitors **Software – Practice and Experience**, vol. 8, p. 721-729, 1978.
- HAUSEN, A. C. **ValiMPI: Uma Ferramenta de Teste Estrutural para Programas Paralelos no Ambiente de Passagem de Mensagem MPI**. Universidade Federal do Paraná, Paraná, 2005.
- HEATH, M. T.; ETHERIDGE, J. A. Visualizing the Performance of Parallel Programs. **IEEE Software**. IEEE Computer Society Press, Los Alamitos, vol. 8 – 5, p. 29-39, 1991.
- HWANG, K.; XU, Z. **Scalable Parallel Computing: Technology, Architecture, Programming**. WCB/McGraw-Hill, New York, 1998.
- ILMBERGER H.; THÜRMEI S.; WIEDEMANN C. P. Visit: A visualization and control environment for parallel program debugging. **Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging**, p. 199-201, 1993.
- KAHLE, J.A.; DAY, M.N.; HOFSTEE, H.P.; JOHNS, C.R.; MAEURER, T.R.; SHIPPY D. Introduction to the Cell Multiprocessor, IBM J. Res. & Dev. vol 49, nr 4/5 jul/set 2005.
- KOHL, J. A.; GEIST, G. A. **XPVM 1.0 User's Guide**. Tennessee: Oak Ridge National

- Laboratory, 1996. 45 p.
- KOPPOL, P. V.; TAI, K. An Incremental Approach to Structural Testing of Concurrent Software. **Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)**, ACM, New York, p. 14-23, 1996.
- KRAWCZYK, H.; KUZORA, P.; NEYMAN, M.; PROFICZ, J.; WISZNIEWSKI, B. **STEPS - A Tool For Testing PVM Programs**, 1998.
- LASTOVETSKY, A. **Parallel Computing on Heterogeneous Networks**. Wiley-Interscience, Hoboken, NJ, 2003.
- LEBLANC, T. J.; MELLOR-CRUMMEY, J. M. Debugging Parallel Programs with Instant Replay. **IEEE Transactions on Computers**. IEEE Computer Society Washington, vol. 36 – 4, p. 471-482, 1987.
- LEI, Y.; CARVER, R. Reachability testing of semaphore-based programs. **Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004**. IEEE Computer Society, Washington, vol. 1, p. 312-317, 2004.
- LEI, Y.; CARVER, R. Reachability testing of concurrent programs. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, vol. 32 – 6, p. 382-403, 2006.
- LEI, Y.; CARVER, R. H.; KACKER, R.; KUNG, D. A combinatorial testing strategy for concurrent programs. **Software Testing, Verification and Reliability**. John Wiley and Sons Ltd., Chichester, vol. 17 – 4, p. 207-225, 2007.
- LEWIS, B.; BERG, D. J. **Multithreaded Programming with Java Technology**. Prentice Hall PTR, Upper Saddle River – N.J., 1999.
- LEWIS, B.; BERG, D. J. **Threads Primer: A Guide to Multithreaded Programming**. Prentice Hall PTR, Mountain View, CA, 1996.
- LONG, B.; HOFFMAN, D.; STROOPER, P. Tool Support for Testing Concurrent Java Components, **IEEE Transactions on Software Engineering**, IEEE Computer Society, Los Alamitos, vol. 29 – 6, p. 555-566, 2003.
- MALDONADO, J. C. **Crterios Potenciais Usos: Uma Contribuio ao Teste Estrutural de Software Tese de Doutorado**. DCA-FEE-UNICAMP, Campinas, 1991.
- MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M. **Introduo ao Teste de Software**. Notas Didticas do

- ICMC-USP, São Paulo, 2004.
- MCDOWELL, C. E.; HELMBOLD, D. P. Debugging Concurrent Programs. **ACM Computing Survey (CSUV)**, ACM, New York, vol. 21 – 4, p. 593-622, 1989.
- MOREIRA, E. M.; SANTANA, R. H. C.; SANTANA, M. J.; e SANT'ANA, T. D. Auxiliary tool for debugging parallel programs. **In Proceedings of the 3rd International Information and Telecommunication Technologies Symposium**, São Carlos, SP, p. 1-2, 2004.
- MUSUVATHI, M.; QADEER, S.; BALL, T. **CHESS: A Systematic Testing Tool for Concurrent Software**. Redmond: Microsoft Research Technical Report, MSR-TR-2007-149, 2007. 15 p.
- MYERS, G. J. **The Art of Software Testing**. John Wiley & Sons, New York, 2004.
- NAVAUX, P. O. A. Introdução ao processamento paralelo, **RBC - Revista Brasileira de Computação**, vol. 5 nº 2, p. 31-43, 1989.
- NULL, L.; LOBUR, J. **The essentials of computer organization and architecture**. Jones and Bartlett Publishers, Massachusetts, 2003.
- OHIO Supercomputer Center. **XMPI - A Run/Debug GUI for MPI**. Ohio: Ohio Supercomputer Center, 1997. Disponível em: <http://www.lam-mpi.org/software/xmpi/>.
- The OPEN GROUP. **The Single UNIX® Specification, Version 2**. Disponível em: <http://opengroup.org/onlinepubs/007908799/>, Acessado em: 05 de janeiro de 2009.
- The OPEN GROUP. **The Open Group Base Specifications Issue 7: IEEE Std 1003.1 2008**. Disponível em: <http://www.opengroup.org/onlinepubs/9699919799/toc.htm>, Acessado em: 05 de janeiro de 2009.
- PEZZE, M.; YOUNG, M. **Software Testing and Analysis: Process, Principles and Techniques**, John Wiley & Sons, Hobokon, 2007.
- PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. McGraw-Hill, Boston, 2005.
- POSIX.1b standard. **IEEE Standard 1003.1b-1993**, 2003.
- POSIX.1c standard. **IEEE Standard 1003.1c-1995**, 2003.
- QUINN, M. **Parallel Programming in C with MPI and OpenMP**. McGraw-Hill Science/Engineering/Math, Boston, 2003.
- RAPPS, S.; WEYUKER, E. Selecting Software Test Data Using Data Flow Information.

- IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, vol. 11 – 4, p. 367-375, 1985.
- RUSSINOVICH, M.; COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. **Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation**. ACM, New York, p. 258-266, 1996.
- SIMÃO, A. da S.; VINCENZI, A. M. R.; SANTANA, A. C. L. de. A language for the description of program instrumentation and automatic generation of instrumenters. **CLEI Electronic Journal**, vol. 6 – 1, 2003.
- SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. **MPI: The Complete Reference**, The MIT Press, Disponível em: <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>, Cambridge, 1996.
- SOUZA, S.; VERGILIO, S.; SOUZA, P.; SIMÃO, A.; BLISCOSQUE, T.; Lima, A.; HAUSEN, A. ValiPar: A Testing Tool for Message-Passing Parallel Programs. **International Conference on Software Knowledge and Software Engineering (SEKE05)**, p. 386-391, 2005.
- SOUZA, S. R. S.; VERGILIO, S. R.; SIMÃO, A. S.; de SOUZA, P. S. L.; HAUSEN, A. C. Structural Testing Criteria for Message-Passing Parallel Programs. **Concurrency and Computation – Practice and Experience**, John Wiley and Sons Ltd. Chichester, vol. 20 – 16, p. 1893-1916, 2008.
- SOUZA, P. S. L. ; SAWABE, E. T. ; SIMÃO, A. S. ; SOUZA, S. R. S. ; VERGILIO, S. R. . ValiPVM - A Graphical Tool for Structural Testing of PVM Programs. **Proceedings of Euro PVM/MPI 2008**. Springer Berlin, Heidelberg, vol. 1, p. 257-264, 2008.
- STALLINGS, W. **Arquitetura e Organização de Computadores: Projeto para o desempenho**, 5ª ed., Prentice Hall Brasil, São Paulo, 2002.
- TAI, K. C. On Testing Concurrent Programs. **Proceedings of COMPSAC 85**, p. 310-317, 1985.
- TAI, K. Testing of concurrent software. **Proceedings of the 13th Annual International Computer Software and Applications Conference, COMPSAC 89**, p. 62-64, 1989.
- TAI, K. C.; CARVER, R. H.; OBAID, E. E. Debugging concurrent ada programs by deterministic execution. **IEEE Transaction on Software Engineering**. IEEE Press,

- Piscataway, vol. 17 – 1, p. 45–63, 1991.
- TANENBAUM, A. S. **Organização Estruturada de Computadores**, 5ª ed. Prentice Hall Brasil, São Paulo, 2007.
- TANENBAUM, A. S. **Sistemas Operacionais Modernos**, 2ª ed. Prentice Hall Brasil, São Paulo, 2003.
- TANENBAUM, A. S.; STEEN, M. van. **Distributed Systems: Principles and Paradigms**. Prentice Hall, Upper Saddle River - N.J., 2002.
- TAYLOR, R. N.; LEVINE, D. L.; KELLY, C. D. 1992. Structural Testing of Concurrent Programs. **IEEE Transactions on Software Engineering**. IEEE Press, Piscataway, vol. 18 – 3, p. 206-215, 1992.
- TIAN, J. **Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement**. Wiley-IEEE Computer Society Press, Hoboken, 2005.
- TOSCANI, S. S.; OLIVEIRA, R. S. de; CARISSIMI, A. da S. **Sistemas Operacionais e Programação Concorrente**, Sagra Luzzatto, Porto Alegre, 2003.
- VERGILIO, S. R.; SOUZA, S. R. S.; SOUZA, P. S. L. Coverage Testing Criteria for Message-Passing Parallel Programs. **LATW2005 - 6th IEEE Latin-American Test Workshop**, p. 161-166, 2005.
- VETTER, J. S.; SUPINSKI, B. R. Dynamic software testing of MPI applications with umpire. In **Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (Cdrom)**. Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, 2000.
- VINCENZI, A.; WONG, W.; DELAMARO, M.; MALDONADO, J. JaBUTi: a coverage analysis tool for Java programs. **XVII SBES--Brazilian Symposium on Software Engineering**, Manaus, p. 79-84, 2003.
- VINCENZI, A. M. R.; DELAMARO, M. E.; MALDONADO, J. C.; WONG, W. E. Establishing structural testing criteria for Java bytecode. **Software – Practice and Experience**, John Wiley & Sons, Inc., New York, vol. 36, p. 1513-1541, 2006.
- YANG, C. D.; POLLOCK, L. L. An Algorithm for All-du-path Testing Coverage of Shared Memory Parallel Programs. **Proceedings of the 16th Asian Test Symposium**, IEEE Computer Society, Washington, p. 263-268, 1997.
- YANG, C. D.; POLLOCK, L. L. The challenges in automated testing of multithreaded programs. In **Proceedings of the 14th International Conference on Testing**

- Computer Software**, p. 157-166, 1997.
- YANG, C. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-Path Coverage for Parallel Programs. **ISSTA 98: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis**, ACM, New York, vol. 23 – 2, p. 153-162, 1998.
- YANG, C. D.; POLLOCK, L. L. All-uses testing of shared memory parallel programs. **Software Testing, Verification and Reliability**, John Wiley & Sons, Ltd., vol. 13 – 1, p. 3-24, 2003.
- YOUNG, M.; TAYLOR, R.N. Combining static concurrency analysis with symbolic execution, **IEEE Transactions on Software Engineering**, Vol. 14 – 10, p. 1499-1511, 1988.

Arquivos de configuração usados para a IDeL

Arquivo **C.I** que define os elementos léxicos para a versão resumida da linguagem C ANSI definida pela IDeL.

```
D          [0-9]
L          [a-zA-Z_]
H          [a-zA-F0-9]
E          [Ee][+-]?{D}+
FS         (f|F|l|L)
IS         (u|U|l|L)*

%{
#include <stdio.h>
//#include "C.tab.h"

static void count();
static void comment();
}%

%%
"//".*"\n"      { }
"/"([^*]|[/])"*"/ { }

"#"(include|define|ifdef|endif).*\n" { count(); RETURN(DIRECTIVE); }
"auto"          { count(); RETURN(AUTO); }
"break"         { count(); RETURN(BREAK); }
"case"          { count(); RETURN(CASE); }
"char"          { count(); RETURN(CHAR); }
"const"         { count(); RETURN(CONST); }
"continue"      { count(); RETURN(CONTINUE); }
"default"       { count(); RETURN(DEFAULT); }
"do"            { count(); RETURN(DO); }
"double"        { count(); RETURN(DOUBLE); }
"else"          { count(); RETURN(ELSE); }
"enum"          { count(); RETURN(ENUM); }
"extern"        { count(); RETURN(EXTERN); }
```

"float"	{ count(); RETURN(FLOAT); }
"for"	{ count(); RETURN(FOR); }
"goto"	{ count(); RETURN(GOTO); }
"if"	{ count(); RETURN(IF); }
"int"	{ count(); RETURN(INT); }
"long"	{ count(); RETURN(LONG); }
"register"	{ count(); RETURN(REGISTER); }
"return"	{ count(); RETURN(RET); }
"short"	{ count(); RETURN(SHORT); }
"signed"	{ count(); RETURN(SIGNED); }
"sizeof"	{ count(); RETURN(SIZEOF); }
"static"	{ count(); RETURN(STATIC); }
"struct"	{ count(); RETURN(STRUCT); }
"switch"	{ count(); RETURN(SWITCH); }
"typedef"	{ count(); RETURN(TYPEDEF); }
"union"	{ count(); RETURN(UNION); }
"unsigned"	{ count(); RETURN(UNSIGNED); }
"void"	{ count(); RETURN(VOID); }
"volatile"	{ count(); RETURN(VOLATILE); }
"while"	{ count(); RETURN(WHILE); }
{L} ({L} {D})*	{ count(); RETURN(IDENTIFIER); }
0[xX]{H}+{IS}?	{ count(); RETURN(CONSTANT); }
0{D}+{IS}?	{ count(); RETURN(CONSTANT); }
{D}+{IS}?	{ count(); RETURN(CONSTANT); }
L?'(\\. [^\\"'])+'	{ count(); RETURN(CONSTANT); }
{D}+{E}{FS}?	{ count(); RETURN(CONSTANT); }
{D}*"."{D}+({E})?{FS}?	{ count(); RETURN(CONSTANT); }
{D}+"."{D}*({E})?{FS}?	{ count(); RETURN(CONSTANT); }
L?"(\\. [^\\""])*\"	{ count(); RETURN(STRING_LITERAL); }
"..."	{ count(); RETURN(ELLIPSIS); }
">>="	{ count(); RETURN(RIGHT_ASSIGN); }
"<<="	{ count(); RETURN(LEFT_ASSIGN); }
"+="	{ count(); RETURN(ADD_ASSIGN); }
"-="	{ count(); RETURN(SUB_ASSIGN); }
"*="	{ count(); RETURN(MUL_ASSIGN); }
"/="	{ count(); RETURN(DIV_ASSIGN); }
"%="	{ count(); RETURN(MOD_ASSIGN); }
"&="	{ count(); RETURN(AND_ASSIGN); }
"^="	{ count(); RETURN(XOR_ASSIGN); }
" ="	{ count(); RETURN(OR_ASSIGN); }
">>>"	{ count(); RETURN(RIGHT_OP); }
"<<<"	{ count(); RETURN(LEFT_OP); }
"++"	{ count(); RETURN(INC_OP); }
"--"	{ count(); RETURN(DEC_OP); }
"->"	{ count(); RETURN(PTR_OP); }
"&&"	{ count(); RETURN(AND_OP); }
" "	{ count(); RETURN(OR_OP); }
"<="	{ count(); RETURN(LE_OP); }
">="	{ count(); RETURN(GE_OP); }
"=="	{ count(); RETURN(EQ_OP); }
"!="	{ count(); RETURN(NE_OP); }
","	{ count(); RETURN(SEMICOLON); }
("{" " "<")	{ count(); RETURN(OPENCURL); }
("}" " ">")	{ count(); RETURN(CLOSECURL); }
","	{ count(); RETURN(COMMA); }
":"	{ count(); RETURN(COLON); }
"="	{ count(); RETURN(EQUAL); }
"("	{ count(); RETURN(OPENPAR); }
")"	{ count(); RETURN(CLOSEPAR); }


```

("["|"<:")          { count(); RETURN(OPENBRACKET); }
("]"|":>")          { count(); RETURN(CLOSEBRACKET); }
"."                { count(); RETURN(DOT); }
"&"               { count(); RETURN(ANPARSANT); }
"!"               { count(); RETURN(EXCLAMATION); }
"~"               { count(); RETURN(TILDE); }
"_"               { count(); RETURN(MINUS); }
"+"               { count(); RETURN(PLUS); }
"*"               { count(); RETURN(STAR); }
"/"               { count(); RETURN(SLASH); }
"%"               { count(); RETURN(PERCENT); }
"<"               { count(); RETURN(L_OP); }
">"               { count(); RETURN(G_OP); }
"^"               { count(); RETURN(CIRC); }
"|"               { count(); RETURN(BAR); }
"?"               { count(); RETURN(INTERROGATION); }

[ \t\v\n\f]        { count(); }
.                  { /* ignore bad characters */ }

%%

void comment()
{
    char c, c1;

loop:
    while ((c = getchar()) != '*' && c != 0)
        putchar(c);

    if ((c1 = getchar()) != '/' && c != 0)
    {
        unput(c1);
        goto loop;
    }

    if (c != 0)
        putchar(c1);
}

static int column = 0;

#define DEBUG
#include "Debug.h"
void count()
{
    int i;

    if(yytext != NULL)
        for (i = 0; yytext[i] != '\0'; i++)
            if (yytext[i] == '\n')
                column = 0;
            else if (yytext[i] == '\t')
                column += 8 - (column % 8);
            else
                column++;
}

```

Arquivo C.y que define a gramática livre de contexto para uma versão resumida da linguagem C ANSI.

```
%token IDENTIFIER
%token CONSTANT
%token STRING_LITERAL
%token SIZEOF
%token PTR_OP
%token INC_OP
%token DEC_OP
%token LEFT_OP
%token RIGHT_OP
%token LE_OP
%token GE_OP
%token EQ_OP
%token NE_OP
%token AND_OP
%token OR_OP
%token MUL_ASSIGN
%token DIV_ASSIGN
%token MOD_ASSIGN
%token ADD_ASSIGN
%token SUB_ASSIGN
%token LEFT_ASSIGN
%token RIGHT_ASSIGN
%token AND_ASSIGN
%token XOR_ASSIGN
%token OR_ASSIGN
%token TYPE_NAME
%token TYPEDEF
%token EXTERN
%token STATIC
%token AUTO
%token REGISTER
%token CHAR
%token SHORT
%token INT
%token LONG
%token SIGNED
%token UNSIGNED
%token FLOAT
%token DOUBLE
%token CONST
%token VOLATILE
%token VOID
%token STRUCT
%token UNION
%token ENUM
%token ELLIPSIS
%token CASE
%token DEFAULT
%token IF
%token ELSE
%token SWITCH
%token WHILE
%token DO
%token FOR
%token GOTO
%token CONTINUE
%token BREAK
%token RET
%token OPENPAR
%token CLOSEPAR
%token OPENBRACKET
%token CLOSEBRACKET
%token DOT
%token COMMA
```

```

%token ANPARSANT
%token STAR
%token PLUS
%token MINUS
%token TILDE
%token EXCLAMATION
%token SLASH
%token PERCENT
%token L_OP
%token G_OP
%token CIRC
%token BAR
%token INTERROGATION
%token COLON
%token EQUAL
%token SEMICOLON
%token OPENCURL
%token CLOSECURL
%token DIRECTIVE

%%

program
    : translation_unit
    ;

identifier
    : IDENTIFIER
    ;

constant
    : CONSTANT
    ;

primary_expression
    : identifier
    | constant
    | string
    | OPENPAR expression CLOSEPAR
    ;

string
    : STRING_LITERAL
    ;

postfix_expression
    : primary_expression
    | postfix_expression OPENBRACKET expression CLOSEBRACKET
    | postfix_expression OPENPAR CLOSEPAR
    | postfix_expression OPENPAR argument_expression_list CLOSEPAR
    | postfix_expression DOT identifier
    | postfix_expression PTR_OP identifier
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list COMMA assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression

```

```

        | SIZEOF unary_expression
        | SIZEOF OPENPAR type_name CLOSEPAR
        ;

unary_operator
: ANPARSANT
| STAR
| PLUS
| MINUS
| TILDE
| EXCLAMATION
;

cast_expression
: unary_expression
| OPENPAR type_name CLOSEPAR cast_expression
;

multiplicative_expression
: cast_expression
| multiplicative_expression STAR cast_expression
| multiplicative_expression SLASH cast_expression
| multiplicative_expression PERCENT cast_expression
;

additive_expression
: multiplicative_expression
| additive_expression PLUS multiplicative_expression
| additive_expression MINUS multiplicative_expression
;

shift_expression
: additive_expression
| shift_expression LEFT_OP additive_expression
| shift_expression RIGHT_OP additive_expression
;

relational_expression
: shift_expression
| relational_expression relational_operator shift_expression
;

relational_operator
: L_OP
| G_OP
| LE_OP
| GE_OP
;

equality_expression
: relational_expression
| equality_expression EQ_OP relational_expression
| equality_expression NE_OP relational_expression
;

and_expression
: equality_expression
| and_expression ANPARSANT equality_expression
;

exclusive_or_expression
: and_expression
| exclusive_or_expression CIRC and_expression
;

```

```

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression BAR exclusive_or_expression
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression INTERROGATION expression COLON
conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    : EQUAL
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression
    : assignment_expression
    | expression COMMA assignment_expression
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers SEMICOLON
    | declaration_specifiers init_declarator_list SEMICOLON
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator

```

```

        | init_declarator_list COMMA init_declarator
        ;

init_declarator
    : declarator
    | declarator EQUAL initializer
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

type_specifier
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | struct_or_union_specifier
    | enum_specifier
    | TYPE_NAME
    ;

struct_or_union_specifier
    : struct_or_union identifier OPENCURL struct_declaration_list
    CLOSECURL
    | struct_or_union OPENCURL struct_declaration_list CLOSECURL
    | struct_or_union identifier
    ;

struct_or_union
    : STRUCT
    | UNION
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : specifier_qualifier_list struct_declarator_list SEMICOLON
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list COMMA struct_declarator
    ;

struct_declarator

```

```

        : declarator
        | COLON constant_expression
        | declarator COLON constant_expression
        ;

enum_specifier
    : ENUM OPENCURL enumerator_list CLOSECURL
    | ENUM identifier OPENCURL enumerator_list CLOSECURL
    | ENUM identifier
    ;

enumerator_list
    : enumerator
    | enumerator_list COMMA enumerator
    ;

enumerator
    : identifier
    | identifier EQUAL constant_expression
    ;

type_qualifier
    : CONST
    | VOLATILE
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : identifier
    | OPENPAR declarator CLOSEPAR
    | direct_declarator OPENBRACKET constant_expression CLOSEBRACKET
    | direct_declarator OPENBRACKET CLOSEBRACKET
    | direct_declarator function_argument_list
    ;

function_argument_list
    : OPENPAR parameter_type_list CLOSEPAR
    | OPENPAR identifier_list CLOSEPAR
    | OPENPAR CLOSEPAR
    ;

pointer
    : STAR
    | STAR type_qualifier_list
    | STAR pointer
    | STAR type_qualifier_list pointer
    ;

type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier
    ;

parameter_type_list
    : parameter_list
    | parameter_list COMMA ELLIPSIS
    ;

parameter_list
    : parameter_declaration
    | parameter_list COMMA parameter_declaration

```

```

;

parameter_declaration
: declaration_specifiers declarator
| declaration_specifiers abstract_declarator
| declaration_specifiers
;

identifier_list
: identifier
| identifier_list COMMA identifier
;

type_name
: specifier_qualifier_list
| specifier_qualifier_list abstract_declarator
;

abstract_declarator
: pointer
| direct_abstract_declarator
| pointer direct_abstract_declarator
;

direct_abstract_declarator
: OPENPAR abstract_declarator CLOSEPAR
| OPENBRACKET CLOSEBRACKET
| OPENBRACKET constant_expression CLOSEBRACKET
| direct_abstract_declarator OPENBRACKET CLOSEBRACKET
| direct_abstract_declarator OPENBRACKET constant_expression
CLOSEBRACKET
| OPENPAR CLOSEPAR
| OPENPAR parameter_type_list CLOSEPAR
| direct_abstract_declarator OPENPAR CLOSEPAR
| direct_abstract_declarator OPENPAR parameter_type_list
CLOSEPAR
;

initializer
: assignment_expression
| OPENCURL initializer_list CLOSECURL
| OPENCURL initializer_list COMMA CLOSECURL
;

initializer_list
: initializer
| initializer_list COMMA initializer
;

statement
: labeled_statement
| compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
;

labeled_statement
: identifier COLON statement
| CASE constant_expression COLON statement
| DEFAULT COLON statement
;

compound_statement

```



```

        : OPENCURL CLOSECURL
        | OPENCURL statement_list CLOSECURL
        | OPENCURL declaration_list CLOSECURL
        | OPENCURL declaration_list statement_list CLOSECURL
        ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

statement_list
    : statement
    | statement_list statement
    ;

expression_statement
    : SEMICOLON
    | expression SEMICOLON
    ;

selection_statement
    : IF OPENPAR expression CLOSEPAR statement
    | IF OPENPAR expression CLOSEPAR statement ELSE statement
    | SWITCH OPENPAR expression CLOSEPAR statement
    ;

iteration_statement
    : WHILE OPENPAR expression CLOSEPAR statement
    | DO statement WHILE OPENPAR expression CLOSEPAR SEMICOLON
    | FOR OPENPAR expression_statement expression_statement CLOSEPAR
statement
    | FOR OPENPAR expression_statement expression_statement
expression CLOSEPAR statement
    ;

jump_statement
    : GOTO identifier SEMICOLON
    | CONTINUE SEMICOLON
    | BREAK SEMICOLON
    | RET SEMICOLON
    | RET expression SEMICOLON
    ;

translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;

external_declaration
    : directive_declaration
    | function_definition
    | declaration
    ;

directive_declaration
    : DIRECTIVE
    ;

function_definition
    : declaration_specifiers declarator declaration_list
compound_statement
    | declaration_specifiers declarator compound_statement
    | declarator declaration_list compound_statement
    | declarator compound_statement

```

```

;

%%
#include <stdio.h>

extern char yytext[];

int yyerror(const char *s)
{
    fflush(stdout);
    printf("%s", s);
    exit(0);
}

```

Arquivo de configuração contendo as regras de instrumentação para o contexto de programas C ANSI que utilizam a biblioteca PThreads: pthread.idel. Em outras palavras esse arquivo diz o que a IDeL deve fazer para instrumentar o código-fonte: o que procurar no código-fonte e por o que substituir/acrescentar.

```

#####
#   Instrumenter for C                               #
#   Adenilso Simão                                   #
#   Auri Vincenzi                                    #
#                                           Jul/2001 #
#####
#   Instrumenter for Pthread/C                       #
#   Felipe Santos Sarmanho                          #
#                                           Jul/2007 #
#####

# Este instrumentador e fortemente dependente de um pre processamento do codigo a
# ser instrumentador
# dado que algumas instrucoes do pthread (principalmente inicializacao e
# finalizacao de thread)
# precisaram de tratamento especial que não foi possivel fazer com a IDeL
# somente.

instrumenter C

## The units are the functions
unit
var
    :s      as [compound_statement]
    :name as [identifier]
    :pars as [function_argument_list]
    :type as [type_specifier]
named by
    :name
match
    [function_definition < :type :name :pars :s >]
end unit

## This step will locate the functions and create an init and an exit node for
each one
step FindFunction

pattern Function
var
    :s      as [compound_statement]
    :name as [identifier]

```

```

        :pars as [function_argument_list]
        :type as [type_specifier]
match
    [function_definition < :type :name :pars :s >]
declare node $init
declare node $exit
assignment
    assign $parameterdefinition:pars to $init
end pattern

## This step will create a begin and an end node for each statement
step MarkStatements

pattern Null
var
    :s as [statement]
match
    [statement< :s >]
declare node $begin
declare node $end
instrument
    add checkpoint $begin before self
end pattern

## This step will determine the end and begin nodes of each statement
## list from the end and begin nodes of the individual nodes
step LinkStatementList BT

pattern Statement
var
    :s as [statement]
match
    [statement_list< :s >]
assignment
    assign $begin to $begin:s
    assign $end to $end:s
end pattern

pattern List
var
    :s as [statement]
    :ss as [statement_list]
match
    [statement_list< :ss :s >]
graph
    $end:ss -> $begin:s
assignment
    assign $begin to $begin:ss
    assign $end to $end:s
end pattern

## This step will link the end node of the previous statement to the begin node
## of the next statement
step JoinStatement

pattern Join
var
    :ss as [statement_list]
match
    [statement< { :ss } >]
graph
    $begin -> $begin:ss
    $end:ss -> $end
end pattern

## This step will join the

step JoinToFunction

pattern Function1
var
    :ss as [statement_list]

```

```

        :name as [identifier]
        :pars as [function_argument_list]
        :type as [type_specifier]
match
    [function_definition < :type :name :pars { :ss } >]
graph
    $init -> $begin:ss
    $end:ss -> $exit
    mark f_valipthread_init of :name at $init
    mark definition of :name at $init
    mark susage of :name at $init

instrument
    add init $init before self
    add exit $exit after self
#    add exit $exit after :ss
end pattern

pattern Function2
var
    :ss as [statement_list]
    :name as [identifier]
    :pars as [function_argument_list]
    :decl as [declaration_list]
    :type as [type_specifier]
match
    [function_definition < :type :name :pars { :decl :ss } >]
graph
    $init -> $begin:ss
    $end:ss -> $exit
    mark f_valipthread_init of :name at $init
    mark definition of :name at $init
    mark susage of :name at $init

instrument
    add init $init before self
    add exit $exit after self
#    add exit $exit after :ss

end pattern

## This step will process the statements according to their semantics
step MakeGraph

pattern IfThen
var
    :e as [expression]
    :s as [statement]
match
    [statement<if ( :e ) :s>]
graph
    $begin -> $end
    $begin -> $begin:s
    $end:s -> $end
assignment
    assign $definition:e to $begin
    assign $pusage:e      to $begin
instrument
#    #add errorWrite $begin before self
#    add checkpoint $begin:s before :s
#    add checkpoint $end after self
    add checkpoint $end:s after :s
end pattern

pattern IfThenElse
var
    :e as [expression]
    :s1 as [statement]
    :s2 as [statement]
match
    [statement<if ( :e ) :s1 else :s2>]
graph
    $begin -> $begin:s1

```

```

        $begin -> $begin:s2
        $end:s1 -> $end
        $end:s2 -> $end
assignment
    assign $definition:e to $begin
    assign $usage:e      to $begin
instrument
#    #add errorWrite $begin before self
#    add checkpoint $begin:s1 before :s1
#    add checkpoint $begin:s2 before :s2
#    add checkpoint $end      after self
        add checkpoint $end:s1 after :s1
        add checkpoint $end:s2 after :s2
end pattern

pattern While
var
    :e as [expression]
    :s as [statement]
match
    [statement<while ( :e ) :s>]
declare node $control
graph
    $begin -> $control
    $control -> $begin:s
    $end:s -> $control
    $control -> $end
assignment
    assign $break:s      to $end
    assign $continue:s   to $control
    assign $definition:e to $control
    assign $usage:e      to $control
instrument
    add checkpoint $control before :e
#    add checkpoint $begin:s before :s
    add checkpoint $end:s after :s
#    add checkpoint $end      after self
end pattern

pattern DoWhile
var
    :e as [expression]
    :s as [statement]
match
    [statement<do :s while( :e );>]
declare node $control
graph
    $begin -> $begin:s
    $end:s -> $control
    $control -> $begin:s
    $control -> $end
assignment
    assign $break:s      to $end
    assign $continue:s   to $control
    assign $definition:e to $control
    assign $usage:e      to $control
instrument
#    add checkpoint $begin:s before :s
#    add checkpoint $end      after self
    add checkpoint $end:s after :s
    add checkpoint $control before :e
end pattern

pattern For
var
    :e1 as [expression]
    :e2 as [expression]
    :e3 as [expression]
    :s as [statement]
match
    [statement< for (:e1; :e2; :e3) :s >]
declare node $initialization
declare node $control

```

```

declare node $increment
graph
    $begin      -> $initialization
    $initialization -> $control
    $control     -> $begin:s
    $control     -> $end
    $end:s       -> $increment
    $increment   -> $control

assignment
    assign $definition:e1 to $initialization
    assign $cusage:e1     to $initialization

    assign $definition:e2 to $control
    assign $pusage:e2     to $control

    assign $definition:e3 to $increment
    assign $cusage:e3     to $increment

    assign $break:s       to $end
    assign $continue:s    to $increment
instrument
    add checkpoint $control before :e2
#    add checkpoint $begin:s before :s
    add checkpoint $end:s after :s
end pattern

pattern Switch
var
    :e as [expression]
    :s as [statement]
match
    [statement<switch ( :e ) :s>]
graph
    $begin -> $end
assignment
    assign $switch:s to $begin
    assign $break:s  to $end
    assign $pusage:e to $begin
# instrument
#    add checkpoint $end  after self
end pattern

pattern Expression
var
    :e as [expression]
match
    [statement< :e ; >]
graph
    $begin -> $end
assignment
    assign $definition:e to $begin
    assign $cusage:e     to $begin
end pattern

pattern Break
match
    [statement< break ; >]
graph
    $begin -> $break
end pattern

pattern Case
var
    :e as [constant_expression]
    :s as [statement]
match
    [statement<case :e : :s>]
graph
    $switch -> $begin
    $begin   -> $begin:s
    $end:s   -> $end
assignment

```

```

        assign $definition:e to $begin
        assign $usage:e      to $begin
instrument
    add checkpoint $begin before :s
end pattern

pattern Continue
match
    [statement< continue; >]
graph
    $begin -> $continue
end pattern

pattern Default
var
    :s as [statement]
match
    [statement< default: :s >]
graph
    $switch -> $begin
    $begin  -> $begin:s
    $end:s  -> $end
instrument
    add checkpoint $begin before :s
end pattern

# Não haverá return desse tipo, pois o pre processamento deverá torna-los todos
com o tipo: return void;
pattern Return
match
    [statement< return; >]
graph
    $begin -> $exit
# instrument
#     add exit $exit before self
end pattern

pattern ReturnVal
var
    :e as [expression]
match
    [statement< return :e; >]
graph
    $begin -> $exit
assignment
    assign $usage:e to $begin
# instrument
#     add exit $exit before self
end pattern

pattern Return_pre_valipthread
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
match
    [statement< return ( :a1 , :a2 ) ; >]
graph
    $begin -> $exit
assignment
    assign $usage:a1 to $null
# instrument
#     add exit $exit before self
end pattern

pattern Scanf
var
    :args as [argument_expression_list]
match
    [statement< scanf ( :args ) ; >]
assignment
    assign $derefddefinition:args to $begin
end pattern

```

```
#####
## 004 # patterns para detectar/instrumentar as funcoes relativas a semaforo
(post/wait/init)

pattern Mark_sem_init
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
    :a3 as [assignment_expression]

match
    [statement< sem_init( :a1 , :a2 , :a3 ) ; >]
graph
    mark f_sem_init of :a1 at $begin
    mark f_sem_init of :a2 at $begin
    mark f_sem_init of :a3 at $begin
instrument
    add add_post $begin before self
end pattern

pattern Mark_valipthread_TMARK_post_init
var
    :a1 as [assignment_expression]
match
    [statement< valipthread_TMARK_post_init( :a1 ) ; >]
graph
    mark f_sem_post of :a1 at $begin
assignment
    assign $susage:a1 to $begin
end pattern

pattern Mark_sem_post
var
    :a1 as [assignment_expression]
match
    [statement< sem_post( :a1 ) ; >]
graph
    mark f_sem_post of :a1 at $begin
assignment
    assign $susage:a1 to $begin
instrument
    add add_post $begin before self
end pattern

pattern Mark_sem_wait
var
    :a1 as [assignment_expression]
match
    [statement< sem_wait( :a1 ) ; >]
graph
    mark f_sem_wait of :a1 at $begin
assignment
    assign $susage:a1 to $begin
instrument
    add add_wait $begin before self
end pattern

pattern Mark_pthread_create
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
    :a3 as [assignment_expression]
    :a4 as [assignment_expression]
    :thdname as [string]
match
    [statement< pthread_create( :a1 , :a2 , :a3 , :a4 ) ; >]
graph
    mark f_pthread_create of :a1 at $begin
    mark f_pthread_create of :a2 at $begin
```



```

        mark f_pthread_create of :a3 at $begin
        mark f_pthread_create of :a4 at $begin
assignment
    assign $derefdefinition:a3 to $null
    assign $definition:a3 to $begin
    assign $cusage:a3 to $null
    assign $pusage:a3 to $null
    assign $susage:a3 to $begin

instrument
    add add_post $begin before self
end pattern

pattern Mark_pthread_join
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
#    :a3 as [assignment_expression]
match
    [statement< pthread_join( :a1 , :a2 ) ; >]
graph
    mark f_pthread_join of :a1 at $begin
    mark f_pthread_join of :a2 at $begin
#    mark f_pthread_join of :a3 at $begin
#    mark definition of :a1 at $begin
#    mark susage of :a1 at $begin
# assignment
#    assign $cusage:a1 to $null
#    assign $susage:a1 to $begin
instrument
    add add_wait $begin before self
end pattern

# Prepara para instrumentar returns, exit de thread (referente a sincronizacao de
# finalizacao)
pattern Mark_pthread_exit
var
    :a0 as [assignment_expression]
    :a1 as [assignment_expression]
match
    [statement< pthread_exit( :a0 , :a1 ) ; >]
graph
    mark f_pthread_exit of :a1 at $begin
    mark definition of :a0 at $begin
    mark susage of :a0 at $begin
assignment
    assign $cusage:a0 to $null
instrument
    add add_post $begin before self
end pattern

pattern Mark_exit
var
    :a1 as [assignment_expression]
match
    [statement< exit( :a1 ) ; >]
graph
    mark f_exit of :a1 at $begin
instrument
    add add_post $begin before self
end pattern

pattern Mark_return
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
match
    [statement< return( :a1, :a2 ); >]
graph
    mark return of :a2 at $begin
    mark definition of :a1 at $begin
    mark susage of :a1 at $begin
assignment

```

```

        assign $cusage:a1 to $null
instrument
    add add_post $begin before self
end pattern

## 004 # Fim
#####

## This step will mark the usages and definitions of variables
##
step Marks TB

## Ignore identifiers which is used as function name
pattern Skip_Identifier_Function
var
    :v    as [identifier]
    :args as [argument_expression_list]
match
    [postfix_expression< :v ( :args ) >]
assignment
    assign $derefdefinition:args to $begin
    assign $cusage:v to $null
    assign $pusage:v to $null
end pattern

pattern Skip_Identifier_Function_2
var
    :v    as [identifier]
match
    [postfix_expression< :v ( ) >]
assignment
    assign $cusage:v to $null
    assign $pusage:v to $null
end pattern

pattern MarkVectorDef
var
    :v1 as [identifier]
    :e2 as [assignment_expression]
    :e  as [assignment_expression]
    :op as [assignment_operator]
match
    [expression< :v1 [ :e2 ] :op :e >]
graph
    mark definition of :v1 at $begin
assignment
    assign $definition:v1 to $begin
    assign $cusage:v1 to $null
    assign $pusage:v1 to $null
    assign $cusage:e2 to $begin
    assign $pusage:e2 to $null
instrument
    add t_shared_def $begin before self
end pattern

pattern MarkVector2Def
var
    :v1 as [identifier]
    :e2 as [assignment_expression]
    :e3 as [assignment_expression]
    :e  as [assignment_expression]
    :op as [assignment_operator]
match
    [expression< :v1 [ :e2 ] [ :e3 ] :op :e >]
graph
    mark definition of :v1 at $begin
assignment
    assign $definition:v1 to $begin
    assign $cusage:v1 to $null
    assign $pusage:v1 to $null
    assign $cusage:e2 to $begin
    assign $cusage:e3 to $begin
    assign $pusage:e2 to $null

```

```

instrument
    add t_shared_def $begin before self
end pattern

pattern MarkVectorUsage
var
    :v1 as [identifier]
    :e1 as [assignment_expression]
    :e as [assignment_expression]
    :op as [assignment_operator]
match
    [postfix_expression< :v1 [ :e1 ] >]
instrument
    add t_shared_use $begin before self
end pattern

pattern MarkVector2Usage
var
    :v1 as [identifier]
    :e1 as [assignment_expression]
    :e2 as [assignment_expression]
    :e as [assignment_expression]
    :op as [assignment_operator]
match
    [postfix_expression< :v1 [ :e1 ][ :e2 ] >]
instrument
    add t_shared_use $begin before self
end pattern

pattern MarkExpression
var
    :v as [identifier]
    :e as [assignment_expression]
    :op as [assignment_operator]
match
    [expression< :v :op :e >]
graph
    mark definition of :v at $definition

assignment
    assign $cusage:v to $null
    assign $pusage:v to $null
end pattern

pattern MarkIncrement
var
    :v as [identifier]
match
    [postfix_expression< :v ++ >]
graph
    mark definition of :v at $definition
end pattern

pattern MarkDecrement
var
    :v as [identifier]
match
    [postfix_expression< :v -- >]
graph
    mark definition of :v at $definition
end pattern

pattern MarkDeref
var
    :v as [identifier]
match
    [unary_expression< & :v >]
graph
    mark definition of :v at $derefdefinition
assignment
    assign $cusage:v to $null
    assign $pusage:v to $null
end pattern

```

```

pattern MarkUsage
var
    :v as [identifier]
match
    [identifier< :v >]
graph
    mark cusage of :v at $cusage
    mark pusage of :v at $pusage
    mark susage of :v at $susage
    mark definition of :v at $parameterdefinition
end pattern

## Implementation Section
##

implementation

implement
var
    :e as [expression]
    :n as [constant]
checkpoint $node before
    [expression< :e >]
binding :n to node $node
as
    [expression< valipthread_check(valipthread_ltid, :n), (:e) >]
end implement

implement
var
    :s as [statement]
    :n as [constant]
checkpoint $node after
    [statement< :s >]
binding :n to node $node
as
    [statement< { :s valipthread_check(valipthread_ltid, :n); } >]
end implement

implement
var
    :s as [statement]
    :n as [constant]
checkpoint $node before
    [statement< :s >]
binding :n to node $node
as
    [statement< { valipthread_check(valipthread_ltid, :n); :s } >]
end implement

implement
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
    :a3 as [assignment_expression]
    :n as [constant]
    :name as [string]
    :sem as [string]
    :mut as [string]
    :ctd as [string]
add_post $node before
    [statement< sem_init ( :a1 , :a2 , :a3 ); >]
binding :n to node $node
binding :name to literal [string<"[:a1]">]
binding :sem to literal [string<"&valipthread_sem_[:a1]">]
binding :mut to literal [string<"&valipthread_mut_[:a1]">]
binding :ctd to literal [string<"valipthread_TMARK_for_[:a3]">]
as
    [statement< {
        int valipthread_index;
        sem_init ( :sem , 0 , 0 );
    } >]

```

```

        sem_init ( :mut , 0 , 1 );
        valipthread_init_repository( :sem, :mut, :name );
        sem_wait( :mut );
        sem_init ( :a1 , :a2 , :a3 );
        for (valipthread_index = 1 ; valipthread_index <= :ctd ;
valipthread_index++) {
            valipthread_check(valipthread_ltid,
valipthread_index + :n );
            valipthread_add_post( :sem, valipthread_ltid,
valipthread_index + :n );
            sem_post( :sem );
        }
        sem_post( :mut );
    } >]
end implement

implement
var
    :a1 as [assignment_expression]
    :n as [constant]
    :sem as [string]
    :mut as [string]
add_post $node before
    [statement< sem_post ( :a1 ); >]
binding :n to node $node
binding :sem to literal [string("&valipthread_sem_[:a1]")]
binding :mut to literal [string("&valipthread_mut_[:a1]")]
as
    [statement< {
        sem_wait( :mut );
        valipthread_add_post( :sem , valipthread_ltid,
:n );
        sem_post( :a1 );
        sem_post( :sem );
        sem_post( :mut );
    } >]
end implement

implement
var
    :a1 as [assignment_expression]
    :n as [constant]
    :sem as [string]
    :mut as [string]
add_wait $node before
    [statement< sem_wait ( :a1 ); >]
binding :n to node $node
binding :sem to literal [string("&valipthread_sem_[:a1]")]
binding :mut to literal [string("&valipthread_mut_[:a1]")]
as
    [statement< {
        sem_wait( :sem );
        sem_wait( :mut );
        valipthread_add_wait( :sem , valipthread_ltid,
:n );
        sem_wait( :a1 );
        sem_post( :mut );
    } >]
end implement

implement
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
    :a3 as [assignment_expression]
    :a4 as [assignment_expression]
    :n as [constant]
    :mut as [string]
    :sem as [string]
    :funcname as [string]
add_post $node before
    [statement< pthread_create( :a1 , :a2 , :a3 , :a4 ) ; >]
binding :n to node $node

```

```

binding :mut to literal [string<"&valipthread_mut_[:a3]">]
binding :sem to literal [string<"&valipthread_sem_[:a3]">]
binding :funcname to literal [string<"[:a3]">]
as
    [statement< {
        sem_wait( :mut );
        pthread_create( :a1 , :a2 , :a3 , :a4 );
        valipthread_new_thread(valipthread_ltid, :n,
:a1, :a3, :funcname );
        valipthread_add_post( :sem , valipthread_ltid, :n);
        sem_post( :sem );
        sem_post( :mut );
    } >]
end implement

implement
var
#
:a0 as [assignment_expression]
:a1 as [assignment_expression]
:a2 as [assignment_expression]
:n as [constant]
add_wait $node before
    [statement< pthread_join( :a1 , :a2 ) ; >]
binding :n to node $node
as
    [statement< {
        pthread_join( :a1 , :a2 );
        valipthread_exit_thread(valipthread_ltid, :n , :a1);
    } >]
end implement

implement
var
:ss as [statement_list]
:name as [identifier]
:pars as [function_argument_list]
:n as [constant]
:type as [type_specifier]
:reffunc as [string]
:mut as [string]
:sem as [string]
:funcname as [string]
init $init before
    [function_definition < :type :name :pars { :ss } >]
binding :n to node $init
binding :mut to literal [string<"&valipthread_mut_[:name]">]
binding :sem to literal [string<"&valipthread_sem_[:name]">]
binding :funcname to literal [string<"[:name]">]
as
    [function_definition< :type :name :pars {
        void *valipthread_pt2Function = (void *) :name;
        int valipthread_ltid;
        sem_wait( :sem );
        sem_wait( :mut );
        valipthread_ltid =
valipthread_start(valipthread_pt2Function, :funcname, :sem );
        valipthread_check(valipthread_ltid, :n );
        sem_post( :mut );
        { :ss }
    } >]
end implement

implement
var
:ss as [statement_list]
:name as [identifier]
:pars as [function_argument_list]
:decl as [declaration_list]
:n as [constant]
:type as [type_specifier]
:mut as [string]

```

```

        :sem as [string]
        :funcname as[string]
init $init before
    [function_definition < :type :name :pars { :decl :ss } >]
binding :n to node $init
binding :mut to literal [string<"&valipthread_mut_[[:name]]">]
binding :sem to literal [string<"&valipthread_sem_[[:name]]">]
binding :funcname to literal [string<"[[:name]]">]
as
    [function_definition< :type :name :pars {
        :decl
        void *valipthread_pt2Function = (void *) :name;
        int valipthread_ltid;
        sem_wait( :sem );
        sem_wait( :mut );
        valipthread_ltid =
valipthread_start(valipthread_pt2Function, :funcname, :sem );
        valipthread_check(valipthread_ltid, :n );
        sem_post( :mut );
        { :ss }
    } >]
end implement

implement
var
    :ss as [statement_list]
    :name as [identifier]
    :pars as [function_argument_list]
    :n as [constant]
    :type as [type_specifier]
exit $exit after
    [function_definition < :type :name :pars { :ss } >]
binding :n to node $init
as
    [function_definition< :type :name :pars {
        { :ss }
        valipthread_check(valipthread_ltid, :n );
        valipthread_finish(valipthread_ltid, :n,
valipthread_pt2Function);
    } >]
end implement

implement
var
    :ss as [statement_list]
    :name as [identifier]
    :pars as [function_argument_list]
    :decl as [declaration_list]
    :n as [constant]
    :type as [type_specifier]
exit $exit after
    [function_definition < :type :name :pars { :decl :ss } >]
binding :n to node $init
as
    [function_definition< :type :name :pars {
        :decl
        { :ss }
        valipthread_check(valipthread_ltid, :n );
        valipthread_finish(valipthread_ltid, :n,
valipthread_pt2Function);
    } >]
end implement

implement
var
    :a0 as [assignment_expression]
    :a1 as [assignment_expression]
    :n as [constant]
    :e as [expression]
add_post $exit before
    [statement< pthread_exit( :a0 , :a1 ) ; >]
binding :n to node $exit
as

```

```

        [statement<{
            valipthread_finish(valipthread_ltid, :n, valipthread_pt2Function);
            pthread_exit( :a1 ) ;
        }>]
end implement

implement
var
    :a1 as [assignment_expression]
    :n as [constant]
    :e as [expression]
add_post $exit before
    [statement< exit( :a1 ) ; >]
binding :n to node $exit
as
    [statement<{
        valipthread_finish(valipthread_ltid, :n, valipthread_pt2Function);
        exit( :a1 ) ;
    }>]
end implement

implement
var
    :a1 as [assignment_expression]
    :a2 as [assignment_expression]
    :n as [constant]
    :e as [expression]
add_post $exit before
    [statement< return( :a1 , :a2 ) ; >]
binding :n to node $exit
as
    [statement<{
        valipthread_finish(valipthread_ltid, :n, valipthread_pt2Function);
        return :a2 ;
    }>]
end implement

implement
var
    :n as [constant]
    :e as [expression]
exit $exit before
    [statement< return :e; >]
binding :n to node $exit
as
    [statement<{
        valipthread_finish(valipthread_ltid, :n,
valipthread_pt2Function);
        return :e;
    }>]
end implement

implement
var
    :n as [constant]
exit $exit before
    [statement< return ; >]
binding :n to node $exit
as
    [statement<{
        valipthread_finish(valipthread_ltid, :n,
valipthread_pt2Function);
        return ;
    }>]
end implement

implement
var
    :v1 as [identifier]
    :e1 as [assignment_expression]
    :e as [assignment_expression]
    :op as [assignment_operator]
    :vectname as [string]

```



```

t_shared_use $begin before
    [postfix_expression< :v1 [ :e1 ] >]
binding :vectname to literal [string<":v1">]
as
    [postfix_expression< :v1 [ valipthread_shared_use(valipthread_ltid,
: vectname , & (:v1 [ :e1 ])) , (:e1) ] >]
end implement

implement
var
    :v1 as [identifier]
    :e1 as [assignment_expression]
    :e2 as [assignment_expression]
    :e3 as [assignment_expression]
    :e as [assignment_expression]
    :op as [assignment_operator]
    :vectname as [string]
t_shared_use $begin before
    [postfix_expression< :v1 [ :e1 , (:e2) ][ :e3 ] >]
binding :vectname to literal [string<":v1">]
as
    [postfix_expression< :v1 [ :e2 ]
[ valipthread_shared_use(valipthread_ltid, :vectname , & (:v1 [ :e2 ][ :e3 ])) ,
(:e3) ] >]
end implement

implement
var
    :v1 as [identifier]
    :e1 as [assignment_expression]
    :e2 as [assignment_expression]
    :args as [argument_expression_list]
    :e as [assignment_expression]
    :op as [assignment_operator]

t_shared_def $begin before
    [expression< :v1 [ :e1 ] [ valipthread_shared_use(:args), :e2 ] :op :e
>]
as
    [expression< :v1 [ :e1 ] [ valipthread_shared_def(:args), :e2 ] :op :e
>]
end implement

implement
var
    :v1 as [identifier]
    :e1 as [assignment_expression]
    :args as [argument_expression_list]
    :e as [assignment_expression]
    :op as [assignment_operator]

t_shared_def $begin before
    [expression< :v1 [ valipthread_shared_use(:args), :e1 ] :op :e >]
as
    [expression< :v1 [ valipthread_shared_def(:args), :e1 ] :op :e >]
end implement

end instrumenter

```


Arquivos produzidos pela execução da ValiPThread para o programa prodcons.c

Apêndice

B

Programa *prodcons.c* instrumentado

Versão instrumentada do programa prodcons.c. Instrumentação feita pelo módulo valipthread_inst/IDeL.

```
#include "hvalipthread.h"
#include <valipthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX_PROD 2
#define MAX_QUEUE 2
sem_t mutex, empty, full;
int queue[2], aval;
void *producer(void);
void *consumer(void);
int main(void)
{
    pthread_t prod_h, cons_h;
    void *valipthread_pt2Function = (void *) main;
    int valipthread_ltid;
    {
        valipthread_alloc_init_vars();
        valipthread_init(VALIPTHREAD_THREADS, VALIPTHREAD_SYNCS,
valipthread_pt2Function);
        valipthread_init_repository(&valipthread_sem_consumer,
&valipthread_mut_consumer, "consumer");
        valipthread_init_repository(&valipthread_sem_producer,
&valipthread_mut_producer, "producer");
        valipthread_init_repository(&valipthread_sem_valipthread_f_thread,
&valipthread_mut_valipthread_f_thread, "valipthread_f_thread");
        valipthread_ltid = valipthread_start(valipthread_pt2Function,
"main", NULL);
        valipthread_check(valipthread_ltid, 1);
        {
            {
```

```

        valipthread_check(valipthread_ltid, 2);
        aval = 0;
    }
    {
        valipthread_check(valipthread_ltid, 3);
        {
            int valipthread_index;
            sem_init(&valipthread_sem_mutex, 0, 0);
            sem_init(&valipthread_mut_mutex, 0, 1);
            valipthread_init_repository(&valipthread_sem_mutex,
&valipthread_mut_mutex, "mutex");
            sem_wait(&valipthread_mut_mutex);
            sem_init(&mutex, 0, 1);
            for (valipthread_index = 1; valipthread_index <= 1;
valipthread_index++)
            {
                valipthread_check(valipthread_ltid, valipthread_index
+ 3);
                valipthread_add_post(&valipthread_sem_mutex,
valipthread_ltid, valipthread_index + 3);
                sem_post(&valipthread_sem_mutex);
            }
            sem_post(&valipthread_mut_mutex);
        }
    }
    {
        valipthread_check(valipthread_ltid, 5);
        {
            int valipthread_index;
            sem_init(&valipthread_sem_empty, 0, 0);
            sem_init(&valipthread_mut_empty, 0, 1);
            valipthread_init_repository(&valipthread_sem_empty,
&valipthread_mut_empty, "empty");
            sem_wait(&valipthread_mut_empty);
            sem_init(&empty, 0, 2);
            for (valipthread_index = 1; valipthread_index <= 2;
valipthread_index++)
            {
                valipthread_check(valipthread_ltid, valipthread_index
+ 5);
                valipthread_add_post(&valipthread_sem_empty,
valipthread_ltid, valipthread_index + 5);
                sem_post(&valipthread_sem_empty);
            }
            sem_post(&valipthread_mut_empty);
        }
    }
    {
        valipthread_check(valipthread_ltid, 8);
        {
            int valipthread_index;
            sem_init(&valipthread_sem_full, 0, 0);
            sem_init(&valipthread_mut_full, 0, 1);
            valipthread_init_repository(&valipthread_sem_full,
&valipthread_mut_full, "full");
            sem_wait(&valipthread_mut_full);
            sem_init(&full, 0, 0);
            for (valipthread_index = 1; valipthread_index <= 0;
valipthread_index++)
            {
                valipthread_check(valipthread_ltid, valipthread_index
+ 8);
                valipthread_add_post(&valipthread_sem_full,
valipthread_ltid, valipthread_index + 8);
                sem_post(&valipthread_sem_full);
            }
        }
    }

```

```

        }
        sem_post(&valipthread_mut_full);
    }
}
{
    valipthread_check(valipthread_ltid, 9);
    {
        sem_wait(&valipthread_mut_producer);
        pthread_create(&prod_h, 0, (void *) producer, (void *)
0);
        valipthread_new_thread(valipthread_ltid, 9, &prod_h,
(void *) producer, "producer");
        valipthread_add_post(&valipthread_sem_producer,
valipthread_ltid, 9);
        sem_post(&valipthread_sem_producer);
        sem_post(&valipthread_mut_producer);
    }
}
{
    valipthread_check(valipthread_ltid, 10);
    {
        sem_wait(&valipthread_mut_consumer);
        pthread_create(&cons_h, 0, (void *) consumer, (void *)
0);
        valipthread_new_thread(valipthread_ltid, 10, &cons_h,
(void *) consumer, "consumer");
        valipthread_add_post(&valipthread_sem_consumer,
valipthread_ltid, 10);
        sem_post(&valipthread_sem_consumer);
        sem_post(&valipthread_mut_consumer);
    }
}
{
    valipthread_check(valipthread_ltid, 11);
    {
        pthread_join(prod_h, 0);
        valipthread_exit_thread(valipthread_ltid, 11, prod_h);
    }
}
{
    valipthread_check(valipthread_ltid, 12);
    {
        pthread_join(cons_h, 0);
        valipthread_exit_thread(valipthread_ltid, 12, cons_h);
    }
}
{
    valipthread_check(valipthread_ltid, 13);
    {
        valipthread_alloc_dispose_vars();
        valipthread_finalize();
        exit(0);
    }
}
}
}
valipthread_check(valipthread_ltid, 14);
valipthread_alloc_dispose_vars();
valipthread_finalize();
}
void *producer(void)
{
    int prod, item;
    void *valipthread_pt2Function = (void *) producer;
    int valipthread_ltid;

```

```

{
    sem_wait(&valipthread_sem_producer);
    sem_wait(&valipthread_mut_producer);
    valipthread_ltid = valipthread_start(valipthread_pt2Function,
"producer", &valipthread_sem_producer);
    valipthread_check(valipthread_ltid, 1);
    sem_post(&valipthread_mut_producer);
    {
        {
            valipthread_check(valipthread_ltid, 2);
            prod = 0;
        }
        {
            valipthread_check(valipthread_ltid, 3);
            while (valipthread_check(valipthread_ltid, 4), (prod < 2))
            {
                valipthread_check(valipthread_ltid, 5);
                {
                    {
                        {
                            valipthread_check(valipthread_ltid, 5);
                            item = rand() % 1000;
                        }
                        {
                            valipthread_check(valipthread_ltid, 6);
                            {
                                sem_wait(&valipthread_sem_empty);
                                sem_wait(&valipthread_mut_empty);
                                valipthread_add_wait(&valipthread_sem_empty,
valipthread_ltid, 6);

                                sem_wait(&empty);
                                sem_post(&valipthread_mut_empty);
                            }
                        }
                        {
                            valipthread_check(valipthread_ltid, 7);
                            {
                                sem_wait(&valipthread_sem_mutex);
                                sem_wait(&valipthread_mut_mutex);
                                valipthread_add_wait(&valipthread_sem_mutex,
valipthread_ltid, 7);

                                sem_wait(&mutex);
                                sem_post(&valipthread_mut_mutex);
                            }
                        }
                        {
                            valipthread_check(valipthread_ltid, 8);
                            queue[valipthread_shared_def(valipthread_ltid,
"queue", &(queue[aval])), (aval)] = item;
                        }
                        {
                            valipthread_check(valipthread_ltid, 9);
                            aval++;
                        }
                        {
                            valipthread_check(valipthread_ltid, 10);
                            prod++;
                        }
                        {
                            valipthread_check(valipthread_ltid, 11);
                            {
                                sem_wait(&valipthread_mut_mutex);
                                valipthread_add_post(&valipthread_sem_mutex,
valipthread_ltid, 11);

                                sem_post(&mutex);
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        sem_post(&valipthread_sem_mutex);
        sem_post(&valipthread_mut_mutex);
    }
}
{
    valipthread_check(valipthread_ltid, 12);
    {
        sem_wait(&valipthread_mut_full);
        valipthread_add_post(&valipthread_sem_full,
valipthread_ltid, 12);
        sem_post(&full);
        sem_post(&valipthread_sem_full);
        sem_post(&valipthread_mut_full);
    }
}
}
    valipthread_check(valipthread_ltid, 13);
}
}
{
    valipthread_check(valipthread_ltid, 14);
    {
        valipthread_finish(valipthread_ltid, 14,
valipthread_pt2Function);
        pthread_exit(0);
    }
}
}
    valipthread_check(valipthread_ltid, 15);
    valipthread_finish(valipthread_ltid, 15, valipthread_pt2Function);
}
void *consumer(void)
{
    int cons, my_item;
    void *valipthread_pt2Function = (void *) consumer;
    int valipthread_ltid;
    {
        sem_wait(&valipthread_sem_consumer);
        sem_wait(&valipthread_mut_consumer);
        valipthread_ltid = valipthread_start(valipthread_pt2Function,
"consumer", &valipthread_sem_consumer);
        valipthread_check(valipthread_ltid, 1);
        sem_post(&valipthread_mut_consumer);
        {
            {
                valipthread_check(valipthread_ltid, 2);
                cons = 0;
            }
            {
                valipthread_check(valipthread_ltid, 3);
                while (valipthread_check(valipthread_ltid, 4), (cons < 2))
                {
                    valipthread_check(valipthread_ltid, 5);
                    {
                        {
                            {
                                valipthread_check(valipthread_ltid, 5);
                                {
                                    sem_wait(&valipthread_sem_full);
                                    sem_wait(&valipthread_mut_full);
                                    valipthread_add_wait(&valipthread_sem_full,
valipthread_ltid, 5);
                                    sem_wait(&full);

```

```

        sem_post(&valipthread_mut_full);
    }
}
{
    valipthread_check(valipthread_ltid, 6);
    {
        sem_wait(&valipthread_sem_mutex);
        sem_wait(&valipthread_mut_mutex);
        valipthread_add_wait(&valipthread_sem_mutex,
valipthread_ltid, 6);
        sem_wait(&mutex);
        sem_post(&valipthread_mut_mutex);
    }
}
{
    valipthread_check(valipthread_ltid, 7);
    cons++;
}
{
    valipthread_check(valipthread_ltid, 8);
    aval--;
}
{
    valipthread_check(valipthread_ltid, 9);
    my_item =
queue[valipthread_shared_use(valipthread_ltid, "queue", &(queue[aval])),
(aval)];
}
{
    valipthread_check(valipthread_ltid, 10);
    {
        sem_wait(&valipthread_mut_mutex);
        valipthread_add_post(&valipthread_sem_mutex,
valipthread_ltid, 10);
        sem_post(&mutex);
        sem_post(&valipthread_sem_mutex);
        sem_post(&valipthread_mut_mutex);
    }
}
{
    valipthread_check(valipthread_ltid, 11);
    {
        sem_wait(&valipthread_mut_empty);
        valipthread_add_post(&valipthread_sem_empty,
valipthread_ltid, 11);
        sem_post(&empty);
        sem_post(&valipthread_sem_empty);
        sem_post(&valipthread_mut_empty);
    }
}
{
    valipthread_check(valipthread_ltid, 12);
    printf("consumed: %d\n", my_item);
}
}
    valipthread_check(valipthread_ltid, 13);
}
}
{
    valipthread_check(valipthread_ltid, 14);
    {
        valipthread_finish(valipthread_ltid, 14,
valipthread_pt2Function);
        pthread_exit(0);
    }
}

```



```

    }
}
}
}
valipthread_check(valipthread_ltid, 15);
valipthread_finish(valipthread_ltid, 15, valipthread_pt2Function);
}

```

Arquivo `hvalipthread.h` que contém a definição das variáveis utilizadas pela instrumentação especificamente para o contexto do programa `prodcons.c`.

```

#include <semaphore.h>

#define VALIPTHREAD_SYNCS 6

sem_t valipthread_sem_mutex;
sem_t valipthread_sem_empty;
sem_t valipthread_sem_full;
sem_t valipthread_mut_mutex;
sem_t valipthread_mut_empty;
sem_t valipthread_mut_full;

// Declaracoes de semaforos/mutex referente a criacao/finalizacao de
threads
sem_t valipthread_mut_consumer;
sem_t valipthread_mut_valipthread_f_thread;
sem_t valipthread_mut_producer;
sem_t valipthread_sem_consumer;
sem_t valipthread_sem_valipthread_f_thread;
sem_t valipthread_sem_producer;

void valipthread_alloc_init_vars() {
    sem_init(&valipthread_mut_consumer, 0, 1);
    sem_init(&valipthread_mut_valipthread_f_thread, 0, 1);
    sem_init(&valipthread_mut_producer, 0, 1);
    sem_init(&valipthread_sem_consumer, 0, 0);
    sem_init(&valipthread_sem_valipthread_f_thread, 0, 0);
    sem_init(&valipthread_sem_producer, 0, 0);
}

void valipthread_alloc_dispose_vars() {
    sem_destroy(&valipthread_mut_consumer);
    sem_destroy(&valipthread_mut_valipthread_f_thread);
    sem_destroy(&valipthread_mut_producer);
    sem_destroy(&valipthread_sem_consumer);
    sem_destroy(&valipthread_sem_valipthread_f_thread);
    sem_destroy(&valipthread_sem_producer);
}

#define VALIPTHREAD_THREADS 3

```