Um estudo sobre Teste *versus* Verificação Formal de Programas Java

Leandro César Prudente

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Área de concentração: Engenharia de Software

Orientadora: Profª Drª Ana Cristina Vieira de Melo

— São Paulo, abril de 2004 —

Um estudo sobre Teste *versus* Verificação Formal de Programas Java

Este exemplar corresponde à redação final
da dissertação, devidamente corrigida e
defendida por *Leandro César Prudente*,
aprovada pela comissão julgadora.

São Paulo, 14 de abril de 2004.

Banca examinadora:

- Prof^a. Dr^a. Ana Cristina Vieira de Melo (Orientadora) – IME-USP
- Prof^a. Dr^a. Eliane Martins – IC-Unicamp
- Prof. Dr. Francisco Carlos da Rocha Reverbel – IME-USP

À minha família.

Agradecimentos

À professora Ana Cristina Vieira de Melo,
que me orientou nesse trabalho.
Aos professores Alfredo Goldman e Yoshiko Wakabayashi,
que me incentivaram e me ouviram.
Aos outros professores do Departamento de Computação,
que contribuiram para minha formação no mestrado.
Ao Robby (*Kansas State University*), que me ajudou na *Bandera*.
Aos colegas e amigos do IME, Bianka, Silvio, Lucy, M. Brito, M. Amorim, André
Gordana, Mateus, Luciano, Rosianni, Weslley, Alexandre, Marcio, Daniel,
Alessandro, Ricardo, Vera, Santos, Adrilayne, Aninha, Dani e Chico,
que compartilharam comigo momentos de estudo e descontração.
Aos amigos Douglas, Paulo, Gabriel, Fernanda, Dr. Leandro e
Dra Ana Maria que também me deram forças.
À minha irmã e minhas primas Dayane, Mariane e Roberta pelo apoio.
À minha mãe, pela força e por acreditar no meu potencial.
E por último, ao mais importante de todos: a Deus, que permitiu que
todas essas pessoas extraordinárias estivessem no meu caminho.

Resumo

Neste trabalho, estabelecemos uma correspondência entre a atividade de teste e a verificação formal de propriedades para programas concorrentes Java. Para tanto, aplicamos um dos critérios baseados em fluxo de objetos (critério de todos os pares definição-uso de objetos) e a ferramenta de verificação *Bandera*. A propridade verificada é a ausência de *deadlock*. O estudo foi feito para analisar a cobertura dos requistos identificados de cada programa, segundo o critério de teste adotado. Esta análise foi feita sobre 10 programas que apresentam o *deadlock* e logo em seguida sobre estes mesmos programas com a retirada do *deadlock*. Quando a proprieade é violada, a análise é realizada sobre os contra-exemplos gerados pela *Bandera*. Por outro lado, quando a propriedade não é violada, a análise é feita sobre os estados exercitados. Através deste estudo, percebemos que o uso de uma ferramenta de verificação formal é útil para auxiliar a cobertura do critério definido. Além disso, também mostramos que a análise de programas com e sem *deadlock* são complementares, ou seja, requisitos que não foram exercitados com a violação da propriedade foram exercitados quando esta não foi violada e vice-versa.

Abstract

In this work, we establish a relationship between a test activity and the formal verification of properties for concurrent Java programs. So, we applied one object-flow based criterion (all-definition-use-pairs-criterion) and the Bandera verification tool. The verified property is the lack of the deadlock. The goal of this study was to analyze the coverage identified requirements in each program, according to the criterion of the test adopted. This analysis was done on 10 programs which show a deadlock and then on these same programs without deadlock. When the property is violated, the analysis is done on the counter example generated by Bandera. On the other hand, when the property is not violated, the analysis is done on the exercised states. Through this study, we realized that the use of a formal verification tool is useful to help the covering defined criterion. Besides, we also shown that the analyses of the program with or without the deadlock are complementary. Requirements which had not been exercised with a violation of the property were exercised when this property was not violated and vice-versa.

Sumário

1 Introdução **1**

1.1 Motivação e Objetivos 3

1.2 Organização 4

2 Teste de Software **5**

2.1 Objetivos do Teste 5

2.2 Teste de Sistemas Orientado a Objetos 7

2.3 Algumas Adaptações para Teste OO e Novas Estratégias 7

2.4 Critérios de Teste para Sistemas OO 9

2.4.1 Critérios Baseados em Fluxo de Objetos 10

2.4.2 Critérios Baseados em Estado 16

2.5 Conclusão 16

3 Verificação Formal auxiliando o Teste **19**

3.1 Noções de Verificação Formal 19

3.2 Uso de Verificadores de Modelo no Teste 20

3.3 Teste de Programas Java usando Verificadores 22

3.4 A Ferramenta Bandera 24

3.4.1 Fatiamento de Programas 27

3.4.2 Redutor Baseado em Abstrações 28

3.4.3 Algumas Limitações 31

3.5 Conclusão 32

4 Análise de Programas com *deadlock* **33**

4.1 Teste de Programas OO 34

4.2 Análise dos Programas 37

4.2.1 Programa *START.java* 38

4.2.2 Programa *Dead.java* 40

4.2.3 Programa *MonitorDead.java* 43

4.2.4 Programa *Phill33.java* 46

4.2.5 Programa *PCSimple.java* 51

4.2.6 Programa *MainPC.java* 56

4.2.7 Programa *NestedMonitor.java* 59

4.2.8 Programa *Bounded.java* 63

4.2.9 Programa *PipeInteger.java* 66

4.2.10 Programa *StrictClockS.java* 70

4.3 Conclusão 78

5 Análise de Programas Java sem *Deadlock* **81**

5.1 Análise dos Programas 81

5.1.1 Programa *START_s.java* 83

5.1.2 Programa *Dead_s.java* 86

5.1.3 Programa *MonitorDead_s.java* 89

5.1.4 Programa *Phil33_s.java* 92

5.1.5 Programa *PCSimple_s.java* 96

5.1.6 Programa *PCMain_s.java* 99

5.1.7 Programa *NestedMonitor_s.java* 102

5.1.8 Programa *Bounded_s.java* 105

5.1.9 Programa *PipeInteger_s.java* 108

5.1.10 Programa *StrictClockS_s.java* 112

5.2 Conclusão 119

6 Exemplo: Teste *versus* Verificação **123**

6.1 Teste e Verificação com *Deadlock* 123

6.2 Teste e Verificação sem *Deadlock* 128

6.3 Análise Complementar 132

7 Conclusão **135**

7.1 Resultados Obtidos 136

7.2 Contribuições e Trabalhos Futuros 139

Referências Bibliográficas **142**

A Diagramas e Contra-exemplos **149**

A.1 Programa *Phill33.java* 149

A.2 Programa *PCSimple.java* 152

A.3 Programa *PCMain.java* 155

A.4 Programa *NestedMonitor.java* 158

A.5 Programa *Bounded.java* 160

A.6 Programa *PipeInteger.java* 163

A.7 Programa *StrictClockS.java* 170

Lista de Figuras

2.1 Um programa Java simples 12

2.2 Gráfico do Fluxo de Controle do Objeto 15

3.1 Ferramenta *Bandera* (Interface e Arquitetura Interna) 26

3.2 Exemplo de um código sem (direito) e com (esquerdo) a aplicação de fatiamento 28

3.3 Exemplo de um código concreto (esquerdo) e um código abstrato (direito) 30

4.1 Programa *START.java* 35

4.2 DFO do programa *START.java* 36

4.3 Programa *Dead.java* 41

4.4 DFO do Programa *Dead.java* 41

4.5 Programa *MonitorDead.java* 44

4.6 DFO do Programa *MonitorDead.java* 45

4.7 Programa *Phil33.java* 47

4.8 DFO do Programa *Phil33.java* 49

4.9 Programa *PCSimple.java* 52

4.10 DFO Programa *PCSimple.java* 53

4.11 Programa *MainPC.java* 57

4.12 DFO do programa *PCMain.java* 58

4.13 Programa *NestedMonitor.java* 59

4.14 DFO do programa *NestedMonitor.java* 61

4.15 Programa *Bounded.java* 63
4.16 DFO do programa *Bounded.java* 65
4.17 Programa *PipeInteger.java* 67
4.18 DFO do programa *PipeInteger.java* 68
4.19 Programa *StrictClockS.java* 71
4.20 DFO do Programa *StrictClockS.java* 76

5.1 Programa *START_s.java* 83
5.2 DFO do Programa *START_s.java* 84
5.3 Programa *Dead_s.java* 87
5.4 DFO do Programa *Dead_s.java* 87
5.5 Programa *MonitorDead_s.java* 90
5.6 DFO do Programa *MonitorDead_s.java* 91
5.7 Programa *Phil33_s.java* 93
5.8 DFO do Programa *Phil33_s.java* 94
5.9 Programa *PCSimple_s.java* 96
5.10 DFO Programa *PCSimple_s.java* 97
5.11 Programa *PCMain_s.java* 99
5.12 DFO do Programa *PCMain_s.java* 100
5.13 Programa *NestedMonitor_s.java* 102
5.14 DFO do Programa *NestedMonitor_s.java* 103
5.15 Programa *Bounded_s.java* 106
5.16 DFO do Programa *Bounded_s.java* 107
5.17 Programa *PipeInteger_s.java* 109
5.18 DFO do Programa *PipeInteger_s.java* 110
5.19 Programa *StrictClockS_s.java* 113
5.20 DFO do Programa *StrictClockS_s.java* 117

6.1 Programa *NestedMonitor.java* 124
6.2 DFO do Programa *NestedMonitor.java* 125

6.3 Programa *NestedMonitor.java* 129
6.4 DFO do Programa *NestedMonitor.java* 130

A.1 Programa *Phil33.java* 149
A.2 DFO do Programa *Phil33.java* 150
A.3 Programa *PCSimple.java* 153
A.4 DFO Programa *PCSimple.java* 153
A.5 Programa *PCMain.java* 156
A.6 DFO do Programa *PCMain.java* 156
A.7 Programa *NestedMonitor.java* 158
A.8 DFO do Programa *NestedMonitor.java* 158
A.9 Programa *Bounded.java* 160
A.10 DFO do Programa *Bounded.java* 161
A.11 Programa *PipeInteger.java* 163
A.12 DFO do Programa *PipeInteger.java* 164
A.13 Programa *StrictClockS.java* 170
A.14 DFO do Programa *StrictClockS.java* 174

Lista de Tabelas

4.1 Cobertura do programa *START.java* 40
4.2 Cobertura do programa *Dead.java* 43
4.3 Cobertura do programa *MonitorDead.java* 46
4.4 Cobertura do programa *Phil33.java* 51
4.5 Cobertura do programa *PCSimple.java* 56
4.6 Cobertura do programa *MainPC.java* 59
4.7 Cobertura do programa *NestedMonitor.java* 62
4.8 Cobertura do programa *Bounded.java* 66
4.9 Cobertura do programa *PipeInteger.java* 70
4.10 Cobertura do programa *StrictClockS.java* 77
4.11 Análise dos requisitos(Programas com *deadlock*) 79

5.1 Cobertura do Programa *START_s.java* 86
5.2 Cobertura do Programa *Dead_s.java* 89
5.3 Cobertura do Programa *MonitorDead_s.java* 92
5.4 Cobertura do Programa *Phil33_s.java* 95
5.5 Cobertura do Programa *PCSimple_s.java* 98
5.6 Cobertura do Programa *PCMain_s.java* 101
5.7 Cobertura do Programa *NestedMonitor_s.java* 105
5.8 Cobertura do Programa *Bounded_s.java* 108
5.9 Cobertura do Programa *PipeInteger_s.java* 112

5.10 Cobertura do Programa *StrictClockS_s.java* 119
5.11 Análise dos Requisitos (Programas sem *Deadlock*) 120

6.1 Cobertura do Programa *NestedMonitor.java* 132

7.1 Análise dos Requisitos (Programas com e sem *Deadlock*) 138

Capítulo 1

Introdução

O teste de software é uma atividade dispendiosa e complexa no desenvolvimento de sistemas. Com o surgimento do paradigma orientado a objetos (OO), novas características também surgiram: encapsulamento, polimorfismo, herança e hierarquia de classes. Estas características são diferentes da programação seqüencial e, com isso, a atividade de teste teve que se adaptar e até mesmo definir novas estratégias e novos critérios para revelar os erros. Os critérios de teste estabelecem requistos a serem satisfeitos, os quais orientam a seleção de teste, a avaliação da qualidade do teste e a definição de requisitos de suficiências a serem atingidas para o encerramento da atividade de teste [6, 9, 36].

Um dos novos critérios criados é o baseado no fluxo do objeto, o qual pode ser dividido em outros critérios: critério de todas as concretizações, critério de todos os pares definição-uso (du), critério de todas as definições, entre outros. No presente trabalho, o critério de todos o pares *du* [11] foi adotado para auxiliar a atividade de teste.

A maior dificuldade da atividade de teste é satisfazer todo o comportamento do sistema. Isto se deve ao fato do número de casos a serem testados ser muito grande ou infinito. Então, o teste é feito sobre a definição de requisitos que devem ser satisfeitos, testando-se o sistema apenas para algumas situações. Entretando, há grande possibilidade de que erros existentes no sistema não sejam detectados.

Na área de verificação formal, os verificadores de modelo são usados para determinar se um requisito é verdadeiro em um dado sistema. Esta ferramenta trata de sistemas

(modelos) que possuem um número finito de estados e verifica se este sistema satisfaz certos requisitos (propriedades). Estes requisitos são afirmações sobre cada estado, como "a mensagem foi enviada", "o *buffer* está cheio" ou "$X < 5$". Para o verificador de modelo, não importa o significado exato dos requisitos, apenas se eles são verdadeiros ou falsos em um certo estado [16]. Os verificadores de modelo são utilizados para avaliar diferentes propriedades de interesse do usuário e, em geral, caso a propriedade seja violada (falsa), ele fornece um contra-exemplo, ou seja, um caminho do início do modelo até onde a propriedade foi violada. A vantagem de utilizar a técnica de verificação de modelo é que esta permite considerar todos os casos possíveis, mesmo quando o número de casos é muito grande e, em alguns casos, infinito.

Com o objetivo de auxiliar a atividade de teste, os verificadores de modelo são aplicados [8, 7, 3, 38, 18] para avaliar diferentes tipos de propriedades. Caso uma dessas propriedades seja violada, o verificador fornece um contra-exemplo que auxilia a análise do problema. O objetivo do presente trabalho é utilizar um verificador de modelo para auxiliar a análise dos requisitos que são satisfeitos pelo critério baseado no fluxo do objeto em programas Java. A vantagem de se utilizar um verificador de modelo para auxiliar esta análise está na avaliação de propriedades, como por exemplo ausência de *deadlock* (impasse)[1]

Neste trabalho, a ferramenta *Bandera* [13, 40, 17, 22] é utilizada junto com o verificador de modelo Spin [24]. A *Bandera* tem como objetivo integrar as ferramentas de verificadores de modelo existentes, fornecendo uma sustentação automatizada para extração de modelos de estados finitos seguros e compactos, que são apropriados para verificação de programas Java. Na *Bandera*, caso uma propriedade seja violada, são utilizadas algumas ferramentas para auxiliar a análise destes programas. Esta análise é feita sobre os contra-exemplos gerados e sobre os caminhos do modelo percorridos pelo verificador de modelo Spin.

[1] o termo *deadlock* é usado em todo texto por ser uma expressão comum na área de computção.

1.1 Motivação e Objetivos

A principal motivação deste trabalho é usar uma ferramenta de verificação formal para auxiliar a cobertura do critério de todos os pares definição-uso e, com isso, diminuir o trabalho do *testador* [2].

A proposta deste trabalho consiste em:

- Mostrar que ferramentas de verificação formal são capazes e podem auxiliar parcialmente a atividade de teste;
- Estabelecer uma correlação entre teste e verificação formal de programas Java.

Com isso, visamos principalmente mostrar que é possível utilizar uma ferramenta de verificação formal (*Bandera*) para verificar a propriedade de *deadlock* e analisar o que esta propriedade auxilia a satisfazer do critério de todos os pares definição-uso. Além disso, também visamos integrar o teste e a verificação formal de sistema para aumentar a garantida de qualidade de programas Java.

O presente trababalho é um estudo inicial de integração da atividade de teste e a verificação formal de programas Java. O estudo é feito, inicialmente, sobre 10 programas concorrentes que violam a propriedade de *deadlock* e também sobre estes mesmos programas com a ausência de *deadlock*. Para os dois casos, são definidos os requisistos que precisam ser testados e identificados quais destes requisitos foram testados.

[2] Pessoa que realiza a atividade de teste.

1.2 Organização

O presente trabalho está organizado da seguinte forma:

Capítulo 2 Introduzimos a atividade de teste, ressaltando principalmente os critérios de teste para programas orientados a objeto: critérios baseados no fluxo do objeto e critérios baseados no estado do objeto. O objetivo deste Capítulo é auxiliar e justificar a escolha do critério adotado neste trabalho: critério de todos os pares *du*. Além disso, mostramos como este critério pode ser aplicado na prática e a vantagem de se utilizar tal critério.

Capítulo 3 Introduzimos o conceito de verificação formal e como a técnica baseada na verificação de modelo tem auxiliado a atividade de teste. Apresentamos também alguns verificadores de modelo e suas contribuições para o teste de especificação e implementação. Este Capítulo tem dois objetivos: mostrar que a verificação de modelos tem ajudado a atividade de teste e apresentar a ferramenta *Bandera*. Além disso, as principais fucionalidades desta ferramenta também são descritais e as vantagens de utilizá-la para auxiliar o teste de programas Java.

Capítulo 4 Descrevemos a análise de programas Java com *deadlock*. Neste Capítulo, é descrito como foram identificados os requisitos a serem testados, como os contra-exemplos gerados pela *Bandera* auxiliaram na cobertura de tais requisitos e os resultados desta análise.

Capítulo 5 Descrevemos a análise de programas Java sem *deadlock*. Este Capítulo tem uma similaridade com o Capítulo 4: é descrito como foram identificados os requisitos e como os estados percorridos pelo verificador de modelo auxilia a cobertura do critério. Relatamos os resultados obtidos na cobertura para o programas sem *deadlock*. Finalmente, comparamos as análises das duas situações: programas com e sem *deadlock* e os resultados obtidos desta junção.

Capítulo 6 Relatamos um exemplo, como o teste de programas Java pode ser auxiliado por uma ferramenta de verificação formal.

Capítulo 7 Relatamos a conclusão final deste trabalho, mostramos os resultados obtidos, apontamos as principais contribuições e descrevemos como este trabalho pode ser extendido de modo a tratar aspectos que não foram abordados.

Capítulo 2

Teste de Software

O desenvolvimento de sistemas de software envolve uma série de atividades de produção nas quais as oportunidades de injeção de falhas humanas[1] são enormes [36]. As atividades relacionadas à garantia de qualidade de software têm sido aplicadas ao longo do desenvolvimento do software, dentre elas a atividade de teste. A atividade de teste [4] consiste de uma análise dinâmica do produto, sendo relevante para a identificação e eliminação de erros que persistem. Segundo Pressman [36], "A atividade de teste não pode mostrar a ausência de defeitos, ela só pode mostrar se defeitos de software estão presentes". Defeito é uma deficiência no programa (um passo, um processo ou uma definição de dados incorretos) que pode provocar uma saída incorreta para algum dado de entrada [6, 4]. Este Capítulo tem como objetivo apenas elucidar a atividade de teste de sistemas orientados a objetos dentro do contexto do trabalho desenvolvido.

2.1 Objetivos do Teste

Segundo Myers [19], " Teste de software é o processo de executar um programa com o objetivo de encontrar defeitos". Um teste bem sucedido é aquele que consegue determinar *casos de teste* para os quais o programa em teste falha. Um *caso de teste* corresponde a uma situação típica sobre o qual o sistema deve ser testado, e como o sistema deve responder a tal situação. Para exercitar os casos de teste, é gerado

[1] Ação humana que produz um resultado incorreto, por exemplo, um comando incorreto em um programa

uma seqüência de entradas do sistema, segundo um critério de entrada, a qual deve ser analisada mediante a um critério de aceitação [18, 30]. Essa seqüência de entradas são formada por um conjunto de *dados de teste.* Um dado de teste é um conjunto de valores utilizado como entrada para o programa em uma determinada execução [19, 6]. Caso não seja encontrado nenhum defeito no sistema, o *testador*[2] tem sua confiança aumentada quanto à qualidade do software, desde que os testes sejam realizados de maneira rigorosa e sistemática.

O principal objetivo do teste é projetar casos de teste que descubram sistematicamente diferentes classes de erros e que tenham custos de tempo e de esforço mínimos. Para tentar reduzir o custo relacionado à atividade de teste, as *técnicas sistemáticas* têm um papel fundamental. Um dos objetivos das técnicas sistemáticas é garantir que aspectos relevantes da especificação e da implementação do sistema sejam executados (exercitados) pelo menos uma vez por algum caso de teste. Existem duas técnicas de teste bastante utilizadas:

1. *Teste funcional* (teste caixa-preta): examina o comportamento das entradas e das saídas relacionadas, desconsiderando a implementação do software [34]. Este teste tem como objetivo identificar os seguintes erros [36]: funções incorretas ou ausentes; erros de interface; erros nas estruturas de dados ou no acesso a bancos de dados externos; erros de desempenho; e erros de inicialização e término;

2. *Teste estrutural* (teste caixa-branca): é baseado na estrutura do código, utilizando o código fonte para identificar requisitos a serem cumpridos pelos casos de teste [36, 6, 37].

A impossibilidade de se testar todas as combinações de entradas possíveis (o domínio de dados de entrada é muito grande e teoricamente infinito na maioria dos casos), induz a elaboração de *critérios de teste*, cujo objetivo é ajudar o *testador* a selecionar casos de teste e a analisar o nível de cobertura[3] dos casos de teste existentes. Na Seção 2.4, os critérios de teste são discutidos detalhadamente.

[2]Pessoa que realiza a atividade de teste

[3]Porcentagem de requisitos que são satisfeitos por um critério.

2.2 Teste de Sistemas Orientado a Objetos

A programação Orientada a Objetos (OO) apresenta várias características, como encapsulamento, herança, polimorfismo, ligação dinâmica de mensagens e hierarquia de classes, as quais facilitam a reusabilidade de software e o desenvolvimento baseado em componentes. Estas características são substancialmente diferentes das programações tradicionais, as quais têm como objetivo a programação seqüencial. Uma grande vantagem do paradigma OO é que este adota formas mais próximas dos mecanismos humanos para gerenciar a complexidade inerente ao software, como abstração, classificando elementos (objetos) em grupos (classes), através de uma estrutura hierárquica [35].

Embora a abordagem OO apresente várias vantagens em relação à procedimental, a realização de teste é um dos maiores problemas encontrados no desenvolvimento de códigos OO e as falhas são difíceis de serem detectadas usando as técnicas de testes tradicionais. Por exemplo, se uma falha em um método herdado for encontrado somente no contexto da classe derivada, então esta falha não pode ser detectada se a técnica de teste usada não forçar uma invocação deste método no objeto que liga a esta classe particular [11]. Devido a essas dificuldades, adaptações de técnicas tradicionais para teste de programas OO e novas estratégias de teste de programas OO vêm sendo discutidas com o intuito de melhorar o teste.

2.3 Algumas Adaptações para Teste OO e Novas Estratégias

Como foi descrito acima, o paradigma OO é bem diferente do procedimental, e devido a isso algumas adaptações na atividade de teste OO são necessárias. Essas adaptações são claramente observadas no processo de teste, principalmente no teste de unidade e no teste de integração. No teste de unidade aplicado a programas tradicionais, cada componente é testado individualmente para que se tenha a certeza de que eles operam corretamente. Segundo Turner e Robson [35], a menor unidade de teste para sistemas orientados a objetos é a classe. Devido a isso, o conceito de unidade muda, pois o encapsulamento induz à definição de classes e objetos. Isto quer dizer que cada classe

e cada instância de uma classe (objeto) guardam os atributos (dados) e as operações (métodos) que manipulam estes dados [36]. Mas cada classe pode conter diferentes métodos e um método em particular que é implementado em outra classe diferente. Devido a isto o teste de unidade muda drasticamente, pois para testar este método em particular também é necessário testar a classe onde este foi implementado.

Já no teste de integração, a operação "um de cada vez" em uma classe é freqüentemente impossível por causa das interações diretas e indiretas dos componentes que compõem a classe [36]. Para tentar resolver esse problema, algumas estratégias estão sendo estudadas. Por exemplo, Turner e Robson [39] propõem uma estratégia onde é feita a junção de testes de unidade e integração com o teste baseado em estado. O teste baseado em estados enfatiza os valores armazenados na representação dos objetos, os quais representam o teste das interações em uma classe pela monitoração das alterações dos valores dos atributos em um objeto, e conseqüentemente alteram seu estado. Inicialmente, o teste baseado em estado é realizado em classes que não possuem dependentes (heranças ou agregação). Após realizar tal teste, um novo teste é feito sobre as classes que usam os serviços das classes que não possuem dependentes. Esta estrategia tem como objetivo adapatar os teste de unidade e integração usados em sistemas procedimental para serem aplicados em sistemas OO.

Segundo Alexander e Offutt [1] é necessário enfatizar mais o teste de integração do que o teste de unidade, pois no momento da composição de herança e agregação, especialmente quando combinado com polimorfismo, pode introduzir novos tipos de falhas na integração. Eles propõem uma estratéiga para solucionar o problema de encontrar erros no relacionamento polimorfico no momento da integração de componentes de um sistema. A estrategia é definir novos critérios de cobertura que permitam testar rotinas ao realizar a integração, visando principalmente testar os métodos polimorficos, ou seja, os médos de uma classe que implementam um interface ou uma classe abstrata.

Uma outra estratégia é a baseada no teste incremental de classes proposta por Harrold e McGregor [20] que tem como objetivo resolver o problema de herança. Essa estratégia explora a natureza hierárquica da relação de herança para testar grupos de classes relacionadas, reutilizando as informações de teste das superclasses para guiar o teste das subclasses. O principal benefício desta estratégia é que um teste completo de uma subclasse é evitado analisando o histórico de testes aplicados à classe pai, ou seja,

os testes aplicados a uma classe pai são reutilizados para projetar um acompanhamento dos testes de subclasses e, se possível, os casos de teste gerados para a classe pai são reutilizados para as subclasses.

Além disso, Pinto e Price [35] implementaram uma ferramenta chamada de `ATeste`, a qual foi implementada utiliazando a estratégia de teste, denominada teste dinâmico de caminho. Esta estratégia aplica conceitos de teste baseados em estados, aliados às vantagens da reflexão computacional[4]. A ferramenta monitora dinamicamente os métodos selecionados quando executados, ou seja, ela captura e apresenta os estados dos objetos antes e após a execução dos métodos selecionados para monitoramento. Esta ferramenta fornece ao *testador* uma visão do estado da classe sob teste, antes e após a execução dos métodos. Além disso, ela fornece uma visão textual de todo caminho executado pela aplicação, mostrando todos os métodos executados e os respectivos estados de maneira ordenada. Estas informações são utilizadas pelo *testador* para verificar os resultados obtidos e analisá-los segundo o comportamento esperado da aplicação.

2.4 Critérios de Teste para Sistemas OO

Os critérios de teste têm como objetivo auxiliar a descoberta de bons casos de teste que aumentem a probabilidade de revelação de defeitos. Essencialmente, os critérios de teste estabelecem requisitos a serem satisfeitos[5] e que podem orientar a seleção de casos de teste, a avaliação da qualidade do teste e a definição de requisitos de suficiências a serem atingidos para o encerramento da atividade de teste [6].

Os critérios estruturais, como os baseados em fluxo de dados e os baseados em fluxo de controle são relevantes para revelar falhas em programas procedimentais. Os critérios de fluxo de dados estabelecem requisitos de teste que exigem a execução de caminhos entre a *definição* e o *uso* de uma variável. Uma variável é definida quando ela for inicializada ou seu valor for alterado, enquanto que uma variável é usada quando seu valor for usado por uma computação ou um predicado. Já os critérios de fluxo de

[4]Reflexão computacional é a atividade executada por um sistema computacional quando faz computações sobre suas próprias computações.

[5]Satisfazer um critério de teste, de um modo genérico, significa satisfazer todos os requisitos de teste estabelecidos pelo critério[6].

controle estão associados aos elementos do fluxo de controle do programa, como todos os caminhos, todas as bifurcações [19, 9] e outros.

Tanto no teste procedimental quanto no orientado a objetos, os critérios de teste têm sua importância na atividade de teste. Entretanto, foram observados problemas no teste OO quando aplicados critérios de teste procedimentais: os critérios tradicionais não são suficientes para revelar as falhas porque as aplicações OO não são executadas seqüencialmente. McDermid, Clack e Kim [26] observaram isso nos critérios baseados em fluxo de controle e nos critérios baseados em fluxo de dados aplicados a programas OO. Para sistema OO, os critérios baseados no fluxo do objeto têm apresentado um bom nível de cobertura e descoberta de falhas [11, 10, 25, 21].

Chen e Kao [11] propõem critérios baseados no fluxo do objeto enfatizando somente o objeto ativo (objeto instanciado no momento da execução) para guiar a seleção de casos de teste e cuidar de problemas causados pelos atributos orientados a objetos. Dois critérios foram propostos por eles: o critério de todas as concretizações (*all-binding*) e o critério de todos os pares definição-uso (*all-du-pairs*). O critério de todas as concretizações tem como objetivo resolver casos de herança e polimorfismo e pode ajudar a identificar que métodos herdados necessitam ser reexaminados. Já o critério de todos os pares definição-uso é aplicado para monitorar o comportamento de cada um dos objetos durante seu período de vida, guiando o caminho de onde o objeto é definido e onde cada definição é referenciada.

2.4.1 Critérios Baseados em Fluxo de Objetos

Antes dos dois critérios serem descritos, é necessário algumas definições para o melhor entendimento dos mesmos, como *objeto definido*, *objeto usado* e *concretizações*.

1. **Objeto definido**: um objeto é dito definido se o seu estado é iniciado ou mudado, ou seja, se pelo menos uma das seguintes condições for válida:
   (a) o construtor do objeto é invocado;
   (b) um atributo é definido, ou seja, o valor do atributo é determinado explicitamente pelo objeto;
   (c) um método que inicializa ou modifica o(s) atributo(s) do objeto é invocado.

2. **Objeto usado**: um objeto é dito usado se pelo menos uma das seguintes condições for válida:

   (a) um de seus atributos é usado numa computação ou num predicado;

   (b) um de seus métodos que usa o atributo do objeto é invocado;

   (c) o objeto é passado como um parâmetro de um método.

3. **Concretizações**: um objeto pode ser ligado a diferentes classes em tempo de execução numa substituição polimórfica. Por exemplo, em java, uma referência a uma classe abstrata pode em tempo de execução referenciar instâncias de diferentes classes. Veja por exemplo na Figura 2.1, na linha 2 *shape_obj* é uma referência para a classe abstrata *shape* e nas linhas 13 e 15 ela pode eventualmente referenciar o objeto *Square* ou *Circle*.

Alguns critérios baseados no fluxo do objeto são:

- **Todas as Concretizações**: todas as possíveis ligações de cada objeto devem ser exercitadas pelo menos uma vez quando o objeto for *definido* ou *usado*. Caso uma instrução invoque mais de um objeto, todas as combinações de uma possível ligação também precisam ser exercitadas pelo menos uma vez [11];

- **Todos os pares definição-uso(du)**: pelo menos um caminho livre de definição de cada objeto definido para cada objeto usado desta definição deve ser exercitado sobre algum teste [11, 25];

- **Todas as definições**: pelo menos um caminho livre de definição de cada objeto definido deve ser exercitado. Este critério foi proposto para o teste de Classes [25].

Critério de todos os pares definição-uso proposto por Chen e kao

Para exemplificar este critério [11], considere o seguinte programa Java ilustrado na Figura 2.1. A classe *Shape* tem duas classes derivadas: *Square* e *Circle*. Note que alguns métodos não foram desenvolvidos, pois não são relevantes para ilustrar tais critérios (método *draw*, por exemplo). O primeiro passo é identificar os objetos e onde

```
 1: class Frame{
 2:     Shape shape_obj;
 3:     public void draw_frame()
 4:     {
 5:         shape_obj.draw();
 6:         if (shape_obj.get_param()>10)
 7:             shape_obj.double1();
 8:         shape_obj.draw();
 9:     }
10:     public Shape set_obj( int shape, int perimeter, int x, int y)
11:     {
12:         if (shape == 1){
13:             shape_obj =  new Square(perimeter,x,y);}
14:         else {
15:             shape_obj =  new Circle(perimeter,x,y);}
16:         return shape_obj;
17:     }
18: }
19: abstract class Shape{
20:     int perimeter;
21:
22:     public void set_parm( int param)
23:     { perimeter = param;};
24:     public int get_param(){ return perimeter;};
25:     public void double1() { perimeter = perimeter * 2;};
26:     public abstract int draw();
27: }
28: class Square extends Shape{
29:     protected int x,y;
30:     public Square( int x, int y, int init_perimeter){
31:         this.x=x ;
32:         this.y=y ;
33:         perimeter=init_perimeter;
34:     }
35:     public  int draw(){
36:         System.out.println( "Desenha Square" );
37:         return 0;
38:     }
39: }
40: class Circle extends Shape{
41:     int x,y;
42:     double radius;
43:     public Circle( int x, int y, int init_perimeter){
44:         this.x=x;
45:         this.y=y;
46:         perimeter=init_perimeter;
47:         this.radius = init_perimeter /(2*3.14);
48:     }
49:     public int draw(){
50:         System.out.println( "Desenha Circle" );
51:         return 0;
52:     }
53: }
54: class teste{
55:     static public void main (String[] args){
56:         Frame frame_obj =  new Frame();
57:         int shape = Integer.parseInt(args[0]);
58:         int perimeter = Integer.parseInt(args[1]);
59:         int x = Integer.parseInt(args[2]);
60:         int y =  Integer.parseInt(args[3]);
61:         frame_obj.set_obj(shape,perimeter,x,y);
62:         frame_obj.draw_frame();
63:     }
64: }
```

Figura 2.1: Um programa Java simples

eles são definidos e usados. Considere o objeto *shape_obj* da classe *Frame* que é definido nas linhas: 7 (segundo a definição 1.c), 13 e 15 (segundo a definição 1.a) e que é usado nas linhas: 5, 6, 7 e 8 (segundo a definição 2.b).

Após identificar onde o objeto *shape_obj* foi definido e onde ele foi usado, pode-se

criar os pares definição-uso(*du*) fazendo todas as possíveis combinações: (7, 5), (7, 6), (7, 8), (13, 5) (13, 6), (13, 7), (13, 8), (15, 5), (15, 6), (15, 7), (15, 8). A partir de todos os pares, devemos eliminar os que nunca serão executados, por exemplo: (7, 5) e (7, 6). Além de identificar os pares *du*, é necessário identificar onde o objeto pode ser concretizado. O objeto *shape_obj* pode ser instanciado por *Shape*, *Cicle* e *Square*, o qual pode ser identificado nas linhas 5, 6, 7, 8, 13 e 15.

O teste baseado no fluxo do objeto proposto por Chen e Kao [11] requer que todas as possíveis concretizações e que cada par *du* de cada objeto sejam identificados e exercitados. Enquanto as concretizações podem ser determinadas diretamente da hierarquia de heranças, os pares *du* precisam de um processamento especial. Isso se deve ao fato de os pares *du* serem invocados no mesmo contexto de um objeto e de um método; ou os pares *du* serem invocados no contexto de um objeto, mas chamados por diferentes métodos. Por isso, é necessário identificar dois tipos de pares *du* de objetos: intra-métodos e inter-métodos. Para identificar esses pares, é utilizado um Diagrama de Fluxo do Objeto (DFO) para o sistema em teste. Além de auxiliar na identificação dos pares *du*, o DFO também ajuda o *testador* a projetar casos de teste sobre os pares encontrados. Para definir todos os pares *du* do programa, Chen e Kao [11] propõem seis passos:

- **Passo 1**: Criar o conjunto de *definição* e o conjunto de *uso* para cada método em cada classe. O conjunto de definição de um método M contém todos os atributos definidos por M e o conjunto de uso de M contém todos os atributos cujos valores são referenciados em M;

- **Passo 2**: Consiste em construir o DFO, mas antes é preciso conhecer algumas definições. Um DFO $G = (SN, A)$ é um grafo direcionado, onde a conexão consiste de um conjunto de super nós (SN) e um conjunto de arestas A. Um *super nó* $sn = (N,AC) \in SN$ representa um método em uma classe, o qual é representado por um conjunto de nós e arestas de controle. Um nó $n \in N$ referente a uma instrução que define ou usa algum objeto é denotado pelo número da linha no programa, enquanto as arestas de controle $ac = (n_i, n_j) \in AC$ indicam que o nó n_i chega ao n_j por algum caminho de execução. Há dois tipos de arestas que descrevem o relacionamento entre os super nós: *arestas passa-mensagem* e *arestas du-método*. Uma aresta passa-mensagem $(n_i, sn_j) \in A$ significa que uma mensagem é enviada

do nó n_i para o *super nó* sn_j (o método sn_j é invocado na linha i), enquanto uma aresta du-método (sn_i, sn_j) quer dizer que há uma relação *du* entre os métodos sn_i e sn_j (sn_i define algum atributo que é usado no sn_j).

O segundo passo consiste da construção dos super nós e das arestas de controle do DFO para objetos intra-método. Em cada *super nó* criado, cria-se também os nós correspondentes a cada instrução que *define* ou *usa* um objeto. Como as arestas passa-mensagem (n_i, sn_j) pertencem às arestas de controle, nesse passo será utilizada tal definição para criar as arestas de controle. Veja na Figura 2.2, o *super nó* sn_2 (representado pelo método set_obj), o n_{13} e o n_{15} (representam as instruções da linha 13 e linha 15) são criados. Baseado na informação do fluxo dos objetos em sn_2, as arestas de controle são criadas. Por outro lado, também podem ser criadas as arestas passa-mensagem, pois o objeto *frame_obj* em n_{61} e n_{62} chama os métodos *set_obj* (sn_2) e *draw_frame* (sn_3) respectivamente;

- **Passo 3**: Neste passo, são criados os pares $du(n_i, n_j)$ para objetos intra-método. Baseado no conjunto de definição e no conjunto de uso criados no **passo 1**, considere o método *double1* da classe *Shape* (Figura 2.1) que define o atributo *perimeter*, o qual é usado no método *draw*. Com isso, tem-se o par *du* (7, 8) e similarmente o par (61, 62);

- **Passo 4**: Neste passo, são identificados os pares *du* para objetos inter-métodos. Mas, antes de identificá-los, é necessário criar todas as arestas método-du (sn_i, sn_j) e atualizar o DFO. Esta aresta é criada se existir pelo menos um par $du(n_i, n_j)$ e duas arestas passa-mensagem (n_i, sn_i) e (n_j, sn_j). Observe que na Figura 2.2, pode-se identificar duas arestas passa-mensagem (n_{61}, sn_2) e (n_{62}, sn_3) e o par *du* (61,62). Logo é criada a aresta método-du (sn_2, sn_3) e adicionado ao DFO;

- **Passo 5**: Para identificar os pares *du* para objetos inter-métodos é necessária a seguinte definição: (n_i, n_j) é um par *du* inter-métodos se um objeto for definido no nó n_i do super nó sn_k e for usado no nó n_j do super nó sn_l, onde $k \neq l$ e (sn_k, sn_l) é uma aresta método-du. Pelo **passo 4**, tem-se, por exemplo, que a aresta método-du (sn_2, sn_3) indica que existe algum par *du* entre os métodos *set_obj* e *draw_frame* da classe *Frame*. Com isso, todos os possíveis pares *du* entre *set_obj* e

draw_frame devem ser incluídos: (13, 5), (13, 6), (13, 7), (13, 8), (15, 5), (15, 6), (15, 7) e (15, 8).

- **Passo 6**: Verificar se existem novos pares *du* no **passo 5** e criar uma nova aresta método-du, voltando ao **passo 4**.

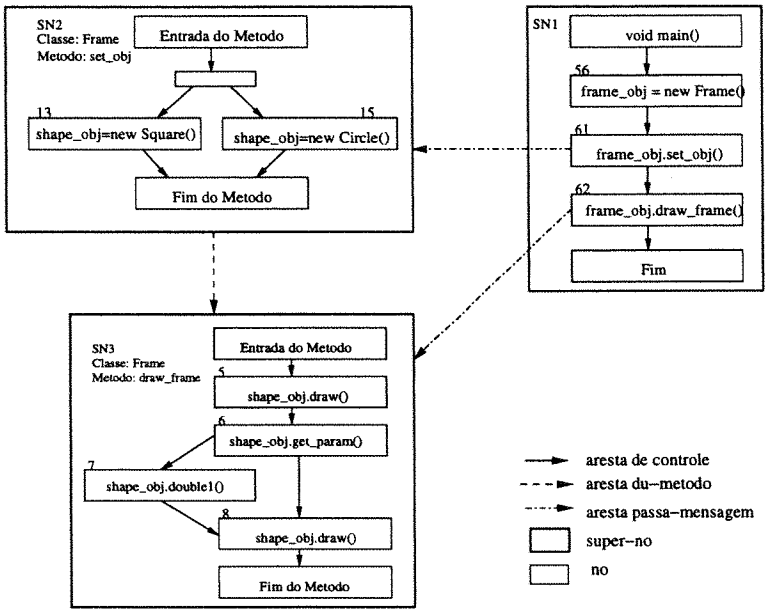


Figura 2.2: Gráfico do Fluxo de Controle do Objeto

Após identificar todos os pares *du*: (61, 62), (7, 8), (13, 5), (13, 6), (13, 7), (13, 8), (15, 5), (15, 6), (15, 7) e (15, 8), é necessário exercitá-los pelo menos uma vez e para isso são criados os seguintes casos de teste:

- **Caso 1**: 56, 61, 12, 13, 62, 5, 6, 7, 8. Dada a seguinte entrada de dados: (1, 11, 2, 2), testa os pares *du*: (61, 62), (13, 5), (13, 6), (13, 7) e (7, 8);
- **Caso 2**: 56, 61, 12, 13, 62, 5, 6, 8. Dada a seguinte entrada de dados: (1, 3, 2, 2), testa os pares *du*: (61, 62), (13, 5), (13, 6) e (13, 8);
- **Caso 3**: 56, 61, 12, 15, 62, 5, 6, 7, 8. Dada a seguinte entrada de dados: (2, 15, 2, 2), testa os pares *du*: (61, 62), (15, 5), (15, 6), (15, 7) e (7, 8);

- **Caso 4**: 56, 61, 12, 15, 62, 5, 6, 8. Dada a seguinte entrada de dados: (2, 5, 2, 2), testa os pares *du*: (61, 62), (15, 5), (15, 6) e (15, 8);

2.4.2 Critérios Baseados em Estado

Além dos critérios baseados no fluxo do objeto, há outros que também são utilizados para sistemas OO: critérios baseados no estado do objeto. O teste baseado no estado do objeto [27] enfoca o comportamento de cada objeto, incluindo a sua construção, a sua mudança de estado e a sua destruição. Este tipo de teste utiliza o Diagrama de Estado do Objeto (DEO) para representar e capturar o comportamento dependente do estado dos objetos numa programação OO. Para analisar o comportamento dos objetos, é preciso checar cada estado e cada transição para cada um dos objetos. Um estado de um objeto é definido como uma combinação dos limites de valores dos atributos definidos no objeto. Enquanto as transições dos objetos correspondem à execução de um método da classe. Dois critérios baseados no estado do objeto são propostos [28, 29]: critério de todos os estados (todos os estados devem ser exercitados pelo menos uma vez) e critérios de todas as transições (todas as transições devem ser exercitadas pelo menos uma vez).

Uma grande dificuldade desses critérios é reconhecer todos os estados do programa e sem conhecê-los é muito difícil decidir quais seqüências de estados e transições de estados exercitar, principalmente se o número de estados for muito grande [11]. Uma outra desvantagem é ao se comparar os critérios descritos anteriormente com os critérios baseados no fluxo do objeto. Os critérios baseados no fluxo do objeto exercitam a maioria dos requisitos de teste do critério descrito acima. Isto deve-se ao fato desses critérios estarem relacionados com o fluxo de controle dos objetos. Por isto, o critério baseado em fluxo do objeto será utiliazado no presente trabalho.

2.5 Conclusão

A atividade de teste é muito dispendiosa e complexa no desenvolvimento de sistemas e, principalmente, em sistema já desenvolvidos. Com o objetivo de revelar defeitos, é necessário estabelecer requisitos a serem satisfeitos, os chamados critérios de teste. Os

critérios de teste adotados devem ser definidos segundo a característica de cada sistema. Para sistemas implementados no paradigma OO, a aplicação de critérios baseados no fluxo do objeto apresenta-se de forma satisfatória na detecção de falhas dos sistemas. O critério de todos os pares definição-uso, apresentado na Seção 2.4.1, será adotado no presente trabalho para identificar os requisitos que precisam ser testados. Com isto, é possível gerar casos de teste que cubram o critério estabelecido e, conseqüentemente, também é possível gerar dados de teste que exercitem este critério.

Criar casos de teste e gerar dados de teste não é uma tarefa fácil, exige muito esforço do *testador*. Com o objetivo de minimizar tal tarefa, ferramentas de verificação formal estão sendo utilizadas para auxiliar a atividade de teste. No próximo Capítulo é retratado como estas ferramentas estão sendo usadas junto à atividade de teste e quais as vantagens e desvantagens de utilizá-las.

Capítulo 3

Verificação Formal de Modelo auxiliando o Teste de Software

Para continuar a discussão sobre a atividade de teste aplicada nesse trabalho, é necessário descrever alguns conceitos de verificação formal enfatizando a técnica de verificação formal de modelos (*Model Checking*) e como tais técnicas podem ser aplicadas junto à atividade de teste.

3.1 Noções de Verificação Formal

A verificação formal consiste em utilizar técnicas matemáticas para assegurar que uma sistema computacional apresente um certa propriedade ou satisfaça sua especificação. Estas técnicas permitem considerar todos os casos possíveis de comportamento do sistema, mesmo quando é muito grande ou infinito. A verificação de modelos é uma técnica de verificação formal que decide se uma determinada propriedade é satisfeita em um modelo. A propriedade corresponde a algum requisito de funcionamento do sistema que geralmente é representado como uma fórmula lógica, enquanto o modelo descreve o comportamento do sistema em uma linguagem formal[1] [16].

Há várias ferramentas de verificação baseadas nesta técnica, que são chamadas de verificadores de modelos (*Model Checkers*), como o SMV [12], o Spin [24] e o DSpin [15].

[1]Esta linguagem permite que o sistema seja descrito de forma abstrata, completa e precisa.

Para utilizar essas ferramentas é necessário ter a implementação do sistema escrito numa linguagem formal própria do verificador de modelo. Após conseguir essa representação, o verificador de modelo é utilizado para verificar as propriedades desejadas pelo usuário. Caso uma dessas propriedades seja violada, a ferramenta gera um contra-exemplo dessa propriedade o qual é útil para diagnosticar o problema. Quando o verificador de modelo gera um contra-exemplo, este atribui valores iniciais para as variáveis no primeiro estado e muda os valores das variáveis nos estados subseqüentes, até onde a propriedade foi violada.

Além de gerar um contra-exemplo e atribuir valores para as variáveis, alguns verificadores de modelos também geram algumas informações relevantes para a análise quando a propriedade não é violada. No verificador Spin, por exemplo, algumas informações são os estados que o Spin não percorre durante a verificação. O Spin também mostra separadamente os estados não percorridos para cada processo do sistema, a quantidade de estados percorridos e a quantidade de estados percorridos mais de uma vez. Com isto, é possível saber a quantidade total de estados percorridos. Assim, os verificadores de modelos podem fornecer informações relevantes aos testes de programas, além da verificação de propriedade que é o objetivo principal.

3.2 Uso de Verificadores de Modelo no Teste de Programas

Além de auxiliar a verificação de propriedades, os verificadores de modelo também estão auxiliando a atividade de teste. Os verificadores de modelo têm ajudado o testador na geração de dados de teste e casos de teste com objetivo de descobrir falhas na especificação, na implementação e entre a especificação e a implementação. Os verificadores de modelo também estão auxiliando na avaliação dos casos de teste de um determinado modelo, ou seja, os casos de teste são analisados com o objetivo de saber se ele é bom ou não. Toda essa análise é feita sobre o contra-exemplo, gerado pelo verificador de modelo, quando uma propriedade é violada segundo a estratégia adotada pelo testador.

Black e Ammann [2] utilizam o verificador de modelo SMV para gerar casos de teste para a implementação de sistemas através de sua especificação formal. A idéia desse trabalho é utilizar a análise de mutantes para gerar os casos de teste a serem usados

na implementação. O objetivo da análise de mutantes é desenvolver um conjunto de teste sobre algumas mudanças sintáticas do programa original [2]. Essa idéia foi adaptada para a técnica de verificação de modelo com algumas mudanças sintáticas na máquina de estado (no modelo especificado numa linguagem própria para o SMV) e na lógica temporal (na propriedade). Com estas mudanças sintáticas, o SMV gerou vários contra-exemplos que foram analisados e convertidos em casos de testes para a implementação. Além de testar a implementação, eles notaram que estes casos de teste também foram úteis para revelar inconsistência entre a especificação e a implementação. Um dos problemas encontrados nesta aplicação foi a grande quantidade de casos de teste gerados. Entretanto, a cobertura desses casos de teste foi satisfatória mediante os critérios estabelecidos.

Um outro trabalho proposto por Black e Ammann [3] utiliza as técnicas de verificação de modelo, análise de mutantes e métricas de cobertura para avaliar casos de teste. Eles desenvolveram uma métrica de cobertura que combina análise de mutantes e verificadores de modelos para avaliar a cobertura de um conjunto de casos de teste. Inicialmente, são gerados vários mutantes da especificação original e cada caso de teste é convertido para uma máquina de estado finito (MEF) que são simbolicamente executados[3] uma de cada vez. Alguns mutantes são inconsistentes, isto é, o verificador de modelo encontra uma diferença entre a execução simbólica do caso de teste e este mutante. Inicialmente é assumido que os casos de teste são consistentes com a especificação original e isso indica uma inconsistência com a especificação original. Quando um mutante revela uma inconsistência com algum dos casos de teste de um conjunto de teste, diz-se que o mutante foi "morto" por este conjunto de caso de teste. A razão dos mutantes mortos para o total de mutantes é uma métrica de cobertura que é aplicada à especificação. Quanto maior for esta razão, melhor ou mais completa é a cobertura do conjunto de teste sobre a especificação. Sobre esta razão eles analizam se os casos de teste pertencentes a este conjunto de teste são bons.

Por outro lado, o grupo do *NASA Software Reseach Laboratory* [7, 8] propõe o

[2] A análise de mutantes [3] define um conjunto de operadores de mutação, os quais são padrões para a mudança sintática. O programa mutante é produzido pela aplicação de um operador de mutação sobre o programa original.

[3] A execução simbólica determina os valores das variáveis originais dependendo de um valor de uma variável especial que controla a MEF.

uso de técnicas de verificação de modelo para monitorar o comportamento dos contra-exemplos gerados pelo verificador de modelo Spin, enfatizando principalmente os valores das variáveis com objetivo de gerar casos de teste durante o teste estrutural. Além de gerar esses casos de teste, os testadores utilizam os valores encontrados na execução do verificador de modelo e dos valores previstos para descobrir inconsistências entre a especificação e a implementação. Os casos de teste são gerados segundo uma partição de equivalência[4] sobre os valores de entrada para minimizar o número de teste. Esta partição envolve os caminhos percorridos na especificação pelo verificador de modelo segundo a combinação de requisitos, ou seja, segundo a propriedade definida.

Um dos grandes problemas desses trabalhos é que cada verificador de modelo possui uma linguagem formal própria, por exemplo, o Spin utiliza a linguagem *PROMELA*. Essas linguagens têm uma sintaxe totalmente diferente das linguagens de programação, como Java e C++, o que também pode causar interpretações errôneas dos resultados gerados pelos contra-exemplos. Com o objetivo de minimizar esses problemas, algumas ferramentas estão sendo construídas para conseguir a tradução do sistema escrito em uma linguagem comum ao usuário (C++ ou Java) para uma notação de um verificador de modelo [23, 5, 13, 14]. Além de conseguir a tradução, algumas dessas ferramentas também conseguem gerar contra-exemplos na mesma linguagem em que o sistema foi escrito, ou seja, na linguagem comum ao usuário, o que auxilia bastante a análise dos resultados gerados. Como nesse trabalho é enfocado o teste de programas Java, na próxima Seção são descritas algumas ferramentas que utilizam a tradução de programas Java para linguagens reconhecidas pelos verificadores.

3.3 Teste de Programas Java usando Verificadores de Modelos

Com o objetivo de facilitar a interface com o usuário, algumas ferramentas estão sendo criadas para auxiliar a verificação de programas Java. Sistemas escritos em Java são traduzidos para uma linguagem formal de um verificador de modelo, ou seja, para um modelo matemático de um sistema, e com isso o verificador de modelo determina se tal

[4] A partição de equivalência [36] divide o domínio de entrada em classes de dados a partir das quais os casos de teste podem ser derivados.

modelo corresponde à especificação de seus requisitos, ou seja, de suas propriedades. Através das ferramentas *JCAT* [14], *Java PathFinder 1 (JPF1)* [23] e *Bandera* [13] é possível conseguir este modelo. Com isso, pode-se utilizar verificadores de modelo (Spin, por exemplo) para gerar casos de teste [7, 8, 2] e/ou também avaliar os casos de teste [3], como nos trabalhos descritos na Seção 3.2.

Por outro lado, as ferramentas *JCAT*, *Java PathFinder 2 (JPF2)* [5] e *Bandera* podem auxiliar a análise do erro diretamente no programa Java. Isso acontece pois tais ferramentas convertem os programas Java em uma notação de um verificador de modelo, o qual é executado para verificar a propriedade desejada e se houver um erro, este é convertido novamente para o programa Java. Tais ferramentas não exigem um conhecimento profundo da linguagem formal e do verificador de modelo utilizado, pois isto está oculto para o usuário.

A ferramenta *JCAT*[5] [14] é um verificador de modelo para detectar *deadlocks* em programas Java que usam mecanismo de múltiplas linhas de controle (*multithreading*). Esta ferramenta foi feita pelo grupo de Arquitetura e de Redes de Computadores da Politécnica de Turin (*Gruppo Architetture e Reti di Calcolatori di Politecnico di Torino*). Uma abstração formal do modelo é gerada do código Java usando o tradutor Java2Spin. O modelo gerado é traduzido para a linguagem *PROMELA* e o Spin é usado para a análise deste sistema. A vantagem desta ferramenta é que ela permite fazer uma simulação do sistema e a verificação da propriedade. Tanto a simulação quanto a verificação são feitas pelo Spin, mas convertidas para o usuário em Java. Há dois problemas em relação ao uso dessa ferramenta: a checagem apenas da propriedade de *deadlock* e o tratamento da explosão dos estados não é suficiente para diminuir o tamanho do modelo gerado, pois percebe-se uma significativa parcela de tempo gasto quando se executa um simples programa Java. O tratamento é dito insuficiente, pois ele utiliza apenas da técnica de fatiamento (Seção 3.4.1) não-automática, ou seja, é necessário que o usuário identifique quais variáveis são irrelevantes aos requisitos do sistema.

Tanto a ferramenta JPF1[6] quanto a JPF2 foram criadas pelo grupo de Engenharia de Software Automatizado da NASA (*Automated Software Engineering group at NASA Ames*). O JPF1 [23] é apenas um tradutor de Java 1.0 para *PROMELA*, a linguagem do

[5] http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml
[6] http://ase.arc.nasa.gov/visser/jpf/

verificador de modelo Spin. A proposta desta ferramenta é estabelecer um *framework* para verificação e simulação de programas Java baseados na técnica de verificação de modelo. Mas esta ferramenta é apenas utilizada como um tradutor de Java para *PROMELA*. A verificação e a simulação são feitas utilizando o XSpin, que nada mais é do que um modo gráfico do Spin. A desvantagem desse sistema é que toda a análise tem que ser feita no Spin, o qual possui uma linguagem não familiar ao usuário. Outra desvantagem é que a JPF1 está limitado à versão 1.0 do Java.

Já a JPF2 [5] é um verificador de modelo para programas escritos em Java e trabalha diretamente sobre o Java *bytecode*. O verificador de modelo JPF2 também foi escrito em Java e pode checar somente *deadlocks*. O ambiente do JPF2 combina as técnicas de verificação de modelo com técnicas para divisão de estados finitos. Estas técnicas incluem: análise estática para redução parcial do conjunto de transições que será explorado pelo verificador de modelo, abstrações de predicados para abstrair os estados e análise de tempo real para apontar problemas no código Java.

Essas ferramentas são vistas como *front-end* e/ou como verificação de modelo para Java e elas também são úteis no caso de analisar o que já foi coberto por um dado critério de teste. Isso é relevante pois pode diminuir a quantidade de casos de teste a serem gerados para cobrir tal critério. Neste trabalho, será enfocado este tipo de análise com o uso da ferramenta *Bandera*.

3.4 A Ferramenta Bandera

A ferramenta *Bandera* [7] foi criada pelo grupo *SAnToS - Specification, Analysis, and Transformation of Software* (Especificação, Análise e Transformação de Software) da Universidade do Estado do Kansas (*Kansas State University*). A *Bandera* [13] é uma coleção integrada de programas de análise e de transformação de componentes que possibilita o usuário selecionar a propriedade do programa a ser verificada. Esta coleção integrada utiliza o poder dos verificadores de modelo existentes, aplicando-os à razão sobre as exigências de corretude dos programas Java. A Bandera explora os seguintes verificadores de modelo: SMV, Spin, dSpin e JPF2. Há várias vantagens em se uti-

[7] http://www.cis.ksu.edu/santos/bandera

lizar a Bandera: explora verificadores de modelo robustos, possuem uma interface amigável, aceita a verificação de todas as propriedades, auxilia a interpretação dos contra-exemplos, utiliza as técnicas de fatiamento (veja na Seção 3.4.1) e de abstração de dados para diminuir a explosão dos estados. Uma das grandes desvantagens dessa ferramenta são suas limitações. Todas essas características serão descritas no decorrer dessa Seção.

O objetivo desse trabalho é utilizar a *Bandera* para auxiliar a atividade de teste de programas Java. Ou seja, dado um critério de teste, no caso deste trabalho foram definidos os critérios baseados no fluxo do objeto, a *Bandera* será utilizada para auxiliar a análise de cobertura deste critério. Para a análise dos programas Java será utilizado o verificador de modelo Spin, pois este pode verificar a propriedade sobre sistemas OO e concorrentes. Para que a *Bandera* utilize o Spin é necessário conseguir a tradução de Java para o modelo de linguagem *PROMELA*, e este modelo tem que ser o mais compacto possível. Especificamente, a *Bandera* utiliza a técnica de fatiamento para eliminar componentes irrelevantes, interpretação das abstrações que dá suporte à abstração de dados e um gerador de modelos que permitem a flexibilidade de limites para vários componentes de sistemas. A *Bandera* também inclui uma coleção de estruturas de dados para mapear automaticamente os caminhos dos erros gerados pelo verificador de modelo de volta para o nível do código Java.

A Figura 3.1 ilustra as entradas fornecidas para a *Bandera* ao verificar um programa Java sobre algum requisito e ilustra também as estruturas internas do conjunto de ferramentas do *Bandera* [17, 13]:

1. *Java front-end*: a *Bandera* foi criada a partir do *framework Soot*[8]. A idéia do *Soot* é traduzir programas Java para uma linguagem intermediária (representação interna) chamada *Jimple*. O *Java front-end* do *Bandera* é chamado de J^3C (Java-para-Jimple-para-Compilador Java) que mantém uma correspondência entre o código do programa Java e a representação *Jimple*;

2. *Property front-end*: As propriedades a serem checadas são escritas na *Bandera Specification Language* (SBL). A especificação das propriedades dá suporte tanto à propriedade global (e.g.*deadlock*) quanto à aplicação de propriedades específicas

[8]Desenvolvido pelo grupo Sable da *McGill University*

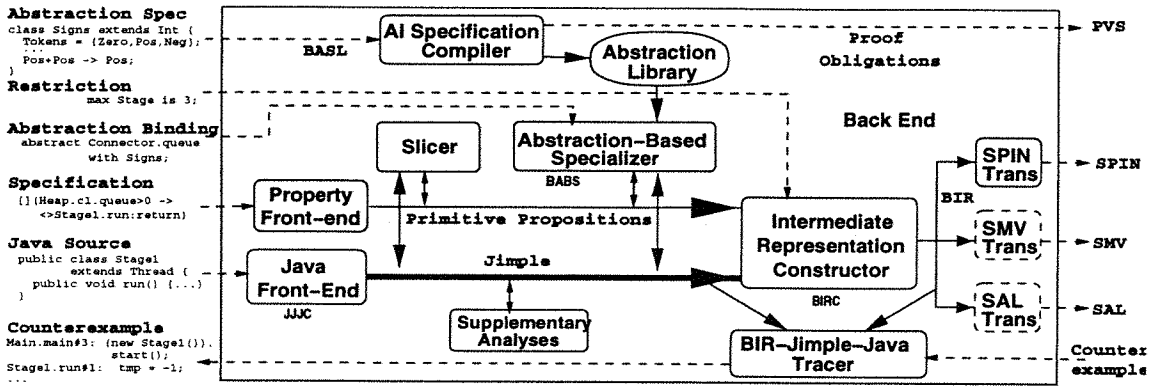


Figura 3.1: Ferramenta *Bandera* (Interface e Arquitetura Interna)

(e.g. formula de lógica temporal). O usuário define *observações* do estado de execução do programa Java como predicados. Estas observações são definidas no código Java usando a notação de comentário do Javadoc. O *property front-end* chama o *Java front-end* que extrai todas as observações declaradas no código do programa e converte essas observações num modelo de baixo nível. A *Bandera* checará a propriedade de *deadlock* (impasse)[9] caso nenhuma outra propriedade seja especificada;

3. *Fatiamento de Programa*: automatiza a eliminação de componentes do programa que são irrelevantes para a propriedade a ser checada. O critério para eliminar componentes é baseado nos atributos do programa (variáveis e pontos de controle), que são referenciados pelas observações feitas no programa. A semântica desses atributos deve ser preservada para a propriedade a ser checada, mas todos os outros componentes, que não influenciam a semântica dos atributos observados, podem ser eliminados no modelo gerado. Veja como isso é feito na Seção 3.4.1;

4. *Abstração dos dados*: A especificação da propriedade a ser verificada não depende dos valores concretos do programa, mas sim da *propriedade* desses valores. Por exemplo, um vetor que armazena um conjunto de itens, se a propriedade a ser verificada depender apenas de um item particular do vetor, pode-se abstrair o valor total de estados do vetor para um conjunto menor $\{ItemInVec-$

[9] O termo *deadlock* será utilizado neste trabalho por ser corriqueiramente utilizado no Brasil

tor,ItemNotInVector}. Com isso, garante-se a redução no tamanho do domínio dos dados sobre os tipos das variáveis do programa. Na Seção 3.4.2, são discutidas as abstrações de dados detalhadamente;

5. *Geração de Modelo Final*: é um gerador de código que transforma o programa *Jimple* em um formato de entrada do verificador de modelo selecionado, respeitando as abstrações e fatiamentos adotados. Este gerador comunica diretamente com o BIR (*Bandera Intermediate Representation*), uma linguagem intermediária, que é traduzida para uma entrada de um *model checker*, neste caso, para *PROMELA*. Quando uma propriedade for violada, o *model checker* gera um contra-exemplo que pode ser refletido de volta no código fonte do programa através do BIR.

3.4.1 Fatiamento de Programas

A *Bandera* utiliza a técnica de fatiamento [22] para gerar modelos reduzidos que são seguros e completos em relação à propriedade. Quando a *Bandera* faz a extração de um modelo de um programa P sobre uma propriedade ϕ (requisito), o programa de fatiamento coleta as instruções de interesse $C_\phi = \{s_1, ..s_k\}$ de ϕ, chamado de critérios de fatiamento para ϕ. O programa de fatiamento computará o programa P', o qual é uma versão reduzida de P pela remoção de instruções e variáveis que são irrelevantes para satisfazer ϕ, ou seja, utiliza o conjunto C_ϕ para fazer o fatiamento. Com isso, é possível afirmar que P satisfaz a propriedade especificada em ϕ se e somente se P' satisfaz a propriedade especificada em ϕ.

Observando a Figura 3.2, podemos ver um exemplos de um programa antes de aplicar a técnica de fatiamento (à direita) e o mesmo programa ao aplicar tal técnica (à esquerda). As linhas 18, 21, 29 e 32 foram eleminadas quando aplicamos a técnica de fatiamento para verificar a propriedade de *deadlock*. Estas linhas são removidas pela ferramenta porque não influênciam na verificação da propriedade em questão. Devido a isto, a ferramenta gera um modelo reduzido, pois tais linhas serão desconsideradas.

A técnica de fatiamento reduz o programa modificando o modelo dependendo da estrutura do programa. Em alguns sistemas, esta técnica remove um processo inteiro de execução e conseqüentemente diminui drasticamente a explosão dos estados. Por outro

```
1: public class Dead {
2:     static Lock lock1;
3:     static Lock lock2;
4:     static int  state;
5:
6:     public static void main(String[] args)
7:         lock1 = new Lock();
8:         lock2 = new Lock();
9:         Process1 p1 = new Process1();
10:        Process2 p2 = new Process2();
11:        p1.start();
12:        p2.start();
13:    }
14: }
15:
16: class Process1 extends Thread {
17:     public void run() {
18:         Dead.state++;
19:         synchronized (Dead.lock2) {
20:             synchronized (Dead.lock1) {
21:                 Dead.state++;
22:             }
23:         }
24:     }
25: }
26:
27: class Process2 extends Thread {
28:     public void run() {
29:         Dead.state++;
30:         synchronized (Dead.lock1) {
31:             synchronized (Dead.lock2) {
32:                 Dead.state++;
33:             }
34:         }
35:     }
36: }
37:
38: class Lock {
39: }
```

```
1: public class Dead {
2:     static Lock lock1;
3:     static Lock lock2;
4:     static int  state;
5:
6:     public static void main(String[] args)
7:         lock1 = new Lock();
8:         lock2 = new Lock();
9:         Process1 p1 = new Process1();
10:        Process2 p2 = new Process2();
11:        p1.start();
12:        p2.start();
13:    }
14: }
15:
16: class Process1 extends Thread {
17:     public void run() {
18:
19:         synchronized (Dead.lock2) {
20:             synchronized (Dead.lock1) {
21:
22:             }
23:         }
24:     }
25: }
26:
27: class Process2 extends Thread {
28:     public void run() {
29:
30:         synchronized (Dead.lock1) {
31:             synchronized (Dead.lock2) {
32:
33:             }
34:         }
35:     }
36: }
37:
38: class Lock {
39: }
```

Figura 3.2: Exemplo de um código sem (direito) e com (esquerdo) a aplicação de fatiamento

lado, quando os componentes são fortemente acoplados, ou grande parte do programa é relevante para a propriedade, a redução feita pela técnica de fatiamento é de moderada a insignificante. Entretanto, a técnica de fatiamento não é dispendiosa comparado ao custo completo do uso de verificador de modelo e ao fato dessa técnica ser totalmente automática. Para utilizar tal técnica, é preciso definir sistematicamente as instruções e as variáveis que são relevantes para a propriedade, pois uma definição errônea pode impossibilitar a verificação [22].

3.4.2 Redutor Baseado em Abstrações

O redutor baseado em abstrações, conhecido como *Bandera Abstraction-Based Specializer* (BABS), melhora o suporte automático para redução de tamanho do modelo via abstração de dados. Isto é útil quando a propriedade a ser verificada não depende dos valores concretos de alguns dados, ou seja, a propriedade depende somente da propriedade dos valores dos dados. O exemplo de um vetor que armazena um conjunto de itens, descrito anteriormente, mostra esta redução, pois apenas um valor que está

contido no vetor é relevante para a propriedade a ser verificada. Não é necessário que todos os valores concretos sejam exercitados para mostrar tal propriedade e o conjunto completo de valores concretos pode ser reduzido a valores que estão no vetor e os que não estão no vetor.

Algumas Abstrações Implementadas

A *Bandera* possui uma biblioteca de abstrações implementadas. Existem, por exemplo, abstrações para números inteiros: a abstração *sings* trata todas as operações (adição, multiplicação, subtração, entre outras) para valores inteiros. A abstração *range* trata os valores concretos entre o limite inferior e superior de dois números, ou seja, os valores são abstraídos em menor que o inferior e maior que o superior. A abstração *set* é usada quando inteiros são usados para simular tipos enumerados. A última abstração implementada é a *point*, que mapeia todos os valores concretos para um único valor. Cada tipo concreto tem um *point* associado a ele e estes são usados em situações onde o valor de uma variável particular não tem nenhum impacto significativo sobre a propriedade a ser verificada. Para os números reais, abstrações tais como: *sings* e *point* também foram implementadas.

É fundamental a atenção do usuário ao selecionar uma abstração, pois o desejo de reduzir os estados, via abstração de dados, tem que preservar a *propriedade* do dado que seja relevante para a propriedade a ser verificada. Existe ainda um universo de outras abstrações, mas neste trabalho são utilizadas apenas essas abstrações de dados, de forma que o comportamento original do programa não seja perdido.

Problema do mau uso das Abstrações

As abstrações de dados descritas anteriormente criam programas abstratos com comportamento bem próximo aos programas originais. Quando a especificação é verdadeira para o programa abstrato, esta também é verdadeira para um programa concreto. Entretanto, se a especificação é falsa para o programa abstrato, o contra-exemplo gerado pode ser um resultado de algum comportamento no programa abstrato que não é apresentado no programa original.

Um dos problemas de usar abstrações é que o resultado de algum comportamento do programa abstrato pode não aparecer no programa concreto. O resultado gerado pela *Bandera* sobre um programa abstrato é falso quando [33]:

1. o programa concreto não satisfaz a propriedade: neste caso, o contra-exemplo é praticável e indica um defeito do sistema ou;

2. a abstração não é adequada para checar a propriedade. Quando isso ocorre, o contra-exemplo abstrato não corresponde ao resultado da aproximação de um comportamento do programa concreto.

```
   class App{
    public static void main(...){
[1]  new AThread().start();   ...
[2]  int i=0;
[3]  while(i<2){...
[4]    assert(!Global.done);
[5]    i++;
    }}}
   class AThread extends Thread{
    public void run(){ ...
[6]  Global.done=true;
    }}
```

```
class App{
 public static void main(...){
  new AThread().start();   ...
  int i=Signs.ZERO;
  while(Signs.lt(i,Signs.POS))  {...
   assert(!Global.done);
   i=Signs.add(i,Signs.POS);
  }}}
class AThread extends Thread{
 public void run(){ ...
  Global.done=true;
 }}
```

Figura 3.3: Exemplo de um código concreto (esquerdo) e um código abstrato (direito)

A Figura 3.3 retrata tal problema. Observe que no programa abstrato (à direita) o não determinismo aparece no método `lt` (linha 3) que implementa a operação abstrata menor ("<"), ou seja, após passar uma vez pelo *loop* `while`, o valor abstrato de `i` passa a ser *pos* e o valor retornado por `Signs.lt(i,Signs.POS)` pode ser verdadeiro ou falso. Para exemplificar o problema, considere que inicialmente `Global.done` (linha 6) seja falso e que após o *loop* ser executado duas vezes, o valor de `Global.done` (linha 4) torna-se verdadeiro. Note com isso que a afirmação (`assert`) é violada no caminho abstrato 1-2-3-4-5-3-4-5-3-4. Observe que isso é claramente possível no programa abstrato, mas

não é possível no programa concreto. Isso ocorre porque o valor abstrato de `i` tornar-se *pos* e a condição na linha 3 é não determinística. Já no programa concreto isso não é possível, porque após executar o *loop* duas vezes, o valor de `i` é 2 e com isso a condição do *loop* seria satisfeita o que finalizaria o *loop*. Assim tem-se que o contra-exemplo gerado pelo programa abstrato é impraticável no programa concreto.

Este problema pode ser evitado modificando a abstração de dados selecionada. Cada abstração retrata a *propriedade* dos dados relevantes para a propriedade a ser verificada e quando estes dados são abstraídos, o seu comportamento também deve ser mantido. Caso isso não seja possível, não será aplicada diretamente esta técnica de redução no programa a ser testado e analisado.

3.4.3 Algumas Limitações

A *Bandera* é uma ferramenta que ainda está em fase de desenvolvimento e por isso possui algumas limitações:

1. O atual tradutor de Java para *PROMELA* admite apenas a descrição estática de *threads*, ou seja, não é permitido que dois objetos sejam instâncias de uma mesma classe que implementa uma *thread*;

2. O tradutor impõe limites no tamanho mínimo e máximo de inteiros, nos números de instâncias alocadas por um objeto e no tamanho dos vetores;

3. O tradutor de Java para *PROMELA* não aceita recursividade.

4. Os tratamentos de exceções são ignorados;

5. Os tipos abstratos de *array* não são admitidos;

6. Quanto às abstrações, é aceito somente abstrações de dados dos tipos inteiro e real;

7. As expressões booleanas complexas não podem ser decompiladas, especialmente o operador "?:" e as composições arbitrárias de operadores "&&" e "||".

3.5 Conclusão

As ferramentas de verificação formal têm sido de grande ajuda na atividade de teste. Elas têm auxiliado a descoberta de problemas na especificação e na implementação de sistemas. Estas ferramentas também auxiliam a criação de casos de teste e a identificação de dados de teste. Isto é feito através da geração de contra-exemplos quando a propriedade é violada, pois é possível identificar os caminhos percorridos, assim como identificar os valores atribuídos às variáveis durante a verificação.

Mesmo contendo tantas limitações, a *Bandera* é considerada mais eficiente do que as outras ferramentas. Isso se deve principalmente à utilização das técnicas de fatiamento e abstrações de dados para diminuir o tamanho do modelo gerado. Uma outra vantagem da *Bandera* é que ela reproduz o contra-exemplo gerado no próprio programa Java. Isto facilita a análise dos dados.

Uma das grandes dificuldades de todas essas ferramentas que trabalham diretamente com a tradução do programa Java para uma linguagem formal é analisar quais caminhos do programa foram percorridos quando a propriedade não é violada. A *Bandera* para tentar ajudar nesta análise, atribui o mesmo nome das variáveis, dos métodos e das classes para sua respectiva tradução na linguagem formal. Nos próximos Capítulos, é descrito como esta ferramenta auxilia o teste de programas concorrentes Java e quais resultados são obtidos.

Capítulo 4

Análise de Programas Java com *deadlock*

Este trabalho tem como objetivo principal estudar como o uso de verificadores de modelos para poder auxiliar ou não, a tarefa de teste de sistemas Java. Como a propriedade de *deadlock* está definida na *Bandera*, inicialmente um conjunto de programas concorrentes foi escolhido para ser analisado. Nos programas analisados, o mecanismo de monitor é utilizado para manter a exclusão mútua entre os processos concorrentes, ou seja, apenas um processo pode estar ativo dentro de um monitor. Na linguagem Java existe um mecanismo de sincronização (*synchronized*) que pode ser usado para implementar monitores diretamente e foi utilizado em alguns dos programas escolhidos.

Outro mecanismo de sincronização também utilizados em alguns dos programas é o semáforo, o qual não existe implementado diretamente na linguagem Java, mas possível de implementar usando monitores. Ao aplicar estes mecanismos a um programa concorrente, pode ocorrer um *deadlock*, ou seja, os processos nunca terminam sua execução porque estão esperando por um evento que nunca ocorrerá. Um *deadlock* ocorre se e somente se as quatro condições abaixo acontecerem simultaneamente[32]:

- Cada recurso só poderá estar alocado(bloqueado) a um único processo em um determinado instante;
- Um processo que já mantém recursos alocados, pode estar esperando por outros recursos que também já estão alocados por outro processo;

- Um recurso não pode ser liberado de um processo porque outros processos desejam o mesmo recurso, ou seja, o recurso só pode ser liberado pelo processo que o alocou;

- Um processo pode ter de esperar por um recurso alocado a outro processo e vice-versa.

Neste Capítulo analisamos um conjunto de programas que violam a propriedade de *deadlock* e como as informações fornecidas pelos contra-exemplos gerados pela *Bandera* podem auxiliar a atividade de teste dos programa segundo o critério descrito na Seção 2.4.1. A idéia geral da análise é identificar os pares *du* que foram exercitados através dos contra-exemplos gerados.

É relevante lembrar que o critério de todas as concretizações não será aplicado neste trabalho, pois na maioria dos programas escolhidos não há polimorfismo. Neste trabalho também não serão analisados os caminhos percorridos pela *Bandera* antes de gerar os contra-exemplos, pois esta ferramenta não oferece nenhum mecanismo para obtermos tais informações.

No presente trabalho foram escolhidos dez programas Java[1] para serem analisados e apenas a propriedade de *deadlock* foi verificada. Os programas utilizados e a análise dos pares *du* exercitados, quando da verificação de *deadlock*, são descritos detalhadamente nas próximas Seções. Na próxima Seção também é descritos como é feita a atividade de teste, gerando os casos de teste que precisam ser exercitados.

4.1 Teste de Programas OO

Como vimos na Seção 2.4, os critérios de teste têm como objetivo auxiliar a descoberta de casos de teste. Os critérios de teste nos auxiliam a identificar quais requisitos do sistema que precisam ser testados. Através deste requisitos, é possível identificamos situações que precisam ser testadas, ou seja, podemos criar os casos de teste necessários.

[1]Retirados de `http://bandera.projects.cis.ksu.edu/`, `http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml` e `http://ase.arc.nasa.gov/visser/jpf/`

Nesta Seção, utilizaremos o programa da Figura 4.1 para exemplificar como são identificados os requisitos e como são criados os casos de teste.

Este programa é um exemplo que tem como objetivo ilustrar a sincronização de duas tarefas. O programa contém a chamada de dois métodos sincronizados da classe *Event*. A chamada é feita pela classe *FirstTask* e *SecondTask*.

```
 1: class Event{
 2:   int count=0;
 3:   public synchronized void wait_for_event(){
 4:     try{wait();}catch(InterruptedException e){};
 5:   }
 6:   public synchronized void signal_event(){
 7:     count = count + 1;
 8:     notifyAll();
 9:   }
10: }
11:
12: class FirstTask extends java.lang.Thread{
13:   Event event1,event2;
14:   int count=0;
15:   public FirstTask(Event e1, Event e2){
16:     this.event1 = e1; this.event2 = e2;
17:   }
18:   public void run(){
19:     count = event1.count;
20:     while(true){
21:          event2.signal_event();
22:          if (count == event1.count)
23:              event1.wait_for_event();
24:          count = event1.count;
25:     }
26:   }
27: }
28:
29: class SecondTask extends java.lang.Thread{
30:   Event event1,event2;
31:   int count=0;
32:   public SecondTask(Event e1, Event e2){
33:     this.event1 = e1; this.event2 = e2;
34:   }
35:   public void run(){
36:     count = event2.count;
37:     while(true){
38:          event1.signal_event();
39:          if (count == event2.count)
40:            event2.wait_for_event();
41:          count = event2.count;
42:     }
43:   }
44: }
45:
46: class START {
47:   public static void main(String[] args){
48:     Event event1 = new Event();
49:     Event event2 = new Event();
50:     FirstTask  task1 = new FirstTask(event1,event2)
51:     SecondTask task2 = new SecondTask(event2,event1
52:     task1.start();
53:     task2.start();
54:   }
55: }
```

Figura 4.1: Programa *START.java*

O primeiro passo da atividade de teste é definir e identificar os critérios de teste. Para identificar o critério adotado, inicialmente é preciso encontrar as definições e os usos de cada objeto:

1. Objeto *event1*
   Definido : 48, 38 ;
   Usado : 50, 51, 19, 22, 24.
2. Objeto *event2*
   Definido : 49, 24;
   Usado : 50, 51, 36, 39, 41.

3. Objeto *task1*
   Definido : 50;
   Usado : 52.
4. Objeto *task2*
   Definido : 51;
   Usado : 53.

Após identificarmos as definições e os usos, nós podemos utilizar esta informação junto com o fluxo de controle dos objetos para criar o DFO (Figura 4.2). À medida em que o DFO é criado, vamos identificando os pares *du*:

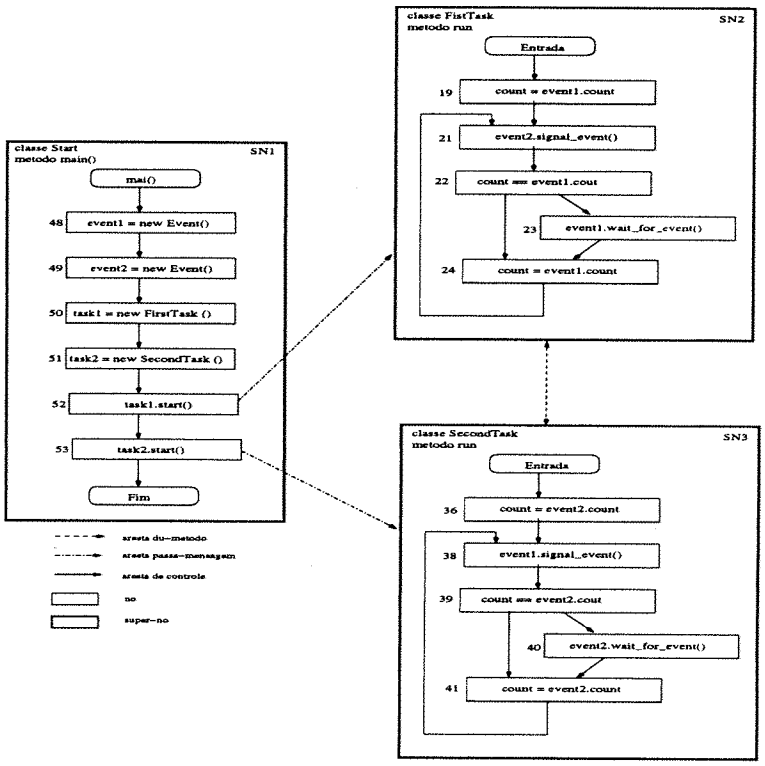


Figura 4.2: DFO do programa *START.java*

- *event1*: (48, 50), (48, 51), (48, 19), (48, 22), (48, 24), (48, 38), (38, 19), (38, 22), (38, 24);
- *event2*: (49, 50), (49, 51), (49, 36), (49, 39), (49, 41), (49, 21), (21, 36), (21, 39), (21, 41);
- *task1*: (50, 52);
- *task2*: (51, 53).

Sobre os requisitos encontrados, podemos criar os casos de teste que cubram todo o critério de teste. É relevante ressaltar que ao criar os casos de teste, nós não nos preocupamos se o programa tem ou não *deadlock*, a única preocupação foi que os casos de teste exercitassem pelo menos uma vez cada requisito identificado. Para este programa, foram identificados quatro casos de teste. Cada caso é um caminho representado por uma seqüência de números e estes números correspondem as linhas de cada requisisto que deve ser exercitado:

- **Caso 1**: 48, 49, 50, 51, 52, 19, 21, 22, 23, 24, 53, 36, 38, 39, 40, 41;
- **Caso 2**: 48, 49, 50, 51, 52, 19, 21, 22, 24, 53, 36, 38, 39, 41;
- **Caso 3**: 48, 49, 50, 51, 52, 53, 36, 38, 39, 40, 41, 19, 21, 22, 23, 24;
- **Caso 4**: 48, 49, 50, 51, 52, 53, 36, 38, 39, 41, 19, 21, 22, 24;

O próximo passo de uma atividade de teste convencional, seria gerar os dados de teste para exercitar os critérios identificados. Mas neste trabalho, nós utilizamos uma ferramenta de verificação para nos auxiliar a análise de cobertura dos requisitos identificados. A próxima Seção retrata esta análise para os programas escolhidos.

4.2 Análise dos Programas

A análise dos programas é feita da seguinte forma: primeiramente, são identificados todas as definições e todos os usos de cada objeto. Utilizando tais informações e analisando o fluxo de controle dos objetos é criado o DFO de cada programa. A medida em que o DFO é criado, são identificados os pares *du* executáveis para cada objeto. Após identificar tais pares, a ferramenta *Bandera* é utilizada para verificar a propriedade de *deadlock* no programa escolhido. Quando a propriedade é violada, a ferramenta pode gerar dois contra-exemplos: o de menor caminho e o aleatório. É relevante lembrar que estes contra-exemplos podem ser iguais. Os caminhos percorridos nestes contra-exemplos são analisados para identificarmos quais pares *du* foram exercitados. Com base nos pares *du* identificados e exercitados, é criado uma Tabela para elucidar os resultados de cada programa.

4.2.1 Programa *START.java*

O programa analisado nesta Seção é o mesmo descrito na anterior (Seção 4.1). Devido a isto, os passos iniciais da análise até a identificação dos pares *du* não serão descritos aqui. Na presente Seção, não houve a preocupação se o programa tem ou não *deadlock*. Segundo a análise feita anterioriormente, foram identificados os seguintes pares *du*:

- *event1*: (48, 50), (48, 51), (48, 19), (48, 22), (48, 24), (48, 38), (38, 19), (38, 22), (38, 24);
- *event2*: (49, 50), (49, 51), (49, 36), (49, 39), (49, 41), (49, 21), (21, 36), (21, 39), (21, 41);
- *task1*: (50, 52);
- *task2*: (51, 53).

Ao verificar a propriedade de *deadlock* para este programa, a *Bandera* gerou dois contra-exemplos: um de menor caminho e um aleatório. Ao analisar estes contra-exemplos percebemos que eles são iguais. Antes de descrevermos os contra-exemplos é necessário o entendimento do símbolo "*", o qual representa onde o programa entra em *deadlock*. É relevante lembrar, que isto foi aplicado em todo o texto. Logo a seguir, o caminho deste contra-exemplo é descrito. Este caminho foi representado usando a idéia do diagrama de sequência. Cada classe utilizada é descrita numa coluna com a sequência do número de cada linha na mesma ordem em que foi percorrida ao realizar a verificação pela *Bandera*. Os símbolos "*" significa onde ocorreu o *deadlock*:

```
START       FirstTask       SecondTask        Event
48             |                |               |
49             |                |               |
50 ----------> 16               |               |
|  <---------- |                |               |
51 ---------- | ------------> 33                |
|  <---------- | ------------- |                 |
52 ----------> 19               |               |
|              21 ------------ | ------------> 7
|              22 <----------- | ------------ 8
|              23* ----------- | ------------> 4
|  <---------- | ------------- | ------------ |
53 ---------- | ------------> 36                |
|              |                38 ----------> 7
|              |                39 <---------- 8
|              |                40* ---------> 4
```

Ao analisar o caminho percorrido no contra-exemplo, podemos dizer que dos 20 pares *du* identificados 11 foram exercitados:

- *event1*: (48, 50), (48, 51), (48, 19), (48, 22), (48, 38);
- *event2*: (49, 50), (49, 51), (21, 36), (21, 39);
- *task1*: (50, 52);
- *task2*: (51, 53).

A Tabela 4.1 ilustra um resumo da análise feita para este programa. Antes de explicar os resultados, é importante saber como Tabela foi criada. A construção da Tabela é importante para explicarmos detalhadamente como foi feita a análise de cada programa.

Para auxiliar a análise do programa, também é necessário sabermos como a Tabela foi construida:

- *Objetos* - coluna 1 e 2 - separamos os objetos em sincronizados e não sincronizados. Caso um objeto seja, em certo momento, sincronizado e em outro não, ele também será separado;
- *Métodos* - coluna 3 - foram descritos todos os possíveis métodos que estes objetos invocam;
- `NP` - coluna 4 - são identificados os números de pares *du* de cada objeto;
- `TO` - coluna 5 - é retratada a porcentagem de pares *du* do objeto em relação total de pares *du*. Esta informação é relevante para sabermos a percentagem de pares de cada objeto que precisam ser exercitados;
- `NPE` - coluna 6 - mostra o número de pares exercitados de cada objeto;
- `CO` - coluna 7 - retrata a porcentagem de pares exercitados de cada objeto em relação aos pares identificados deste objeto;
- `CTO` - coluna 8 - descreve o percentual de cobertura de pares *du* de cada objeto em relação ao total de pares;
- `CT` - coluna 9 - mostra a porcentagem de pares *du* que foram cobertos em relação ao total de pares pares *du* identificados;
- *deadlock*- coluna 10 - indica onde e como ocorreu o *deadlock*;
- *mecanismo de bloqueio* - coluna 11- indica qual mecanismo foi implementado no programa.

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
sinc	*event1*	*wait_for_event(s)*	4	20%	1	25%	5%	15%	Ocorre nas est. *while(if)*, quando os dois obj., event1 e event2 entram em estado de espera quando executam *wait_for_event*	Monitor
sinc	*event2*	*wait_for_event(s)*	4	20%	2	50%	10%			
nsinc	*event1*	-	5	25%	4	80%	20%	35%		
nsinc	*event2*	-	5	25%	2	40%	10%			
nsinc	*task1*	*run*	1	5%	1	100%	5%			
nsinc	*task2*	*run*	1	5%	1	100%	5%			
			20		11			55%		

Tabela 4.1: Cobertura do programa *START.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

É relevante lembrar que esta Tabela foi criada para cada programa analisado. A análise dos outros programas segue a mesma abordagem aplicada a este programa.

A Tabela 4.1, nos mostra que a porcentagem de pares não sincronizados (35%) exercitados é maior do que a dos pares sincronizados (15%). Se observarmos a coluna CO, podemos perceber que a porcentagem de cobertura dos pares *du* por objeto varia de 40% a 100% para os objetos não sincronizados. Enquanto para os objetos sincronizados, este intervalo está entre 25% a 50%. Com isto, podemos dizer que para este programa, os pares não sincronizados são melhor exercitados. Entretanto, também podemos concluir que a porcentagem de cobertura total dos pares *du* é de 55%. Se compararmos com uma média de cobertura de 50%, podemos dizer que a cobertura deste programa foi razoável ao utilizarmos a *Bandera*. Isto significa que o trabalho o testador foi minimizado, ou seja, o testador precisa testar apenas parte do critério que nao foi testado, pois a ferramenta testou 50% do critério para este programa.

4.2.2 Programa *Dead.java*

O programa *Dead.java* está descrito na Figura 4.3 e tem como objetivo implementar apenas o mecanismo de monitores. Neste exemplo, o *Deadlock* acontece porque os processos estão bloqueados na ordem errada: o processo "um" adquire o bloqueio do objeto *lock1*, logo em seguida do objeto *lock2*, enquanto o processo "dois" tenta adquirir o bloqueio do objeto *lock2* e seqüencialmente do objeto *lock1*. Umas das possíveis

```
 1: public class Dead {
 2:     static Lock lock1;
 3:     static Lock lock2;
 4:     static int  state;
 5:
 6:     public static void main(String[] args)
 7:         lock1 = new Lock();
 8:         lock2 = new Lock();
 9:         Process1 p1 = new Process1();
10:         Process2 p2 = new Process2();
11:         p1.start();
12:         p2.start();
13:     }
14: }
15:
16: class Process1 extends Thread {
17:     public void run() {
18:         Dead.state++;
19:         synchronized (Dead.lock2) {
20:             synchronized (Dead.lock1) {
21:                 Dead.state++;
22:             }
23:         }
24:     }
25: }
26:
27: class Process2 extends Thread {
28:     public void run() {
29:         Dead.state++;
30:         synchronized (Dead.lock1) {
31:             synchronized (Dead.lock2) {
32:                 Dead.state++;
33:             }
34:         }
35:     }
36: }
37:
38: class Lock {
39: }
```

Figura 4.3: Programa *Dead.java*

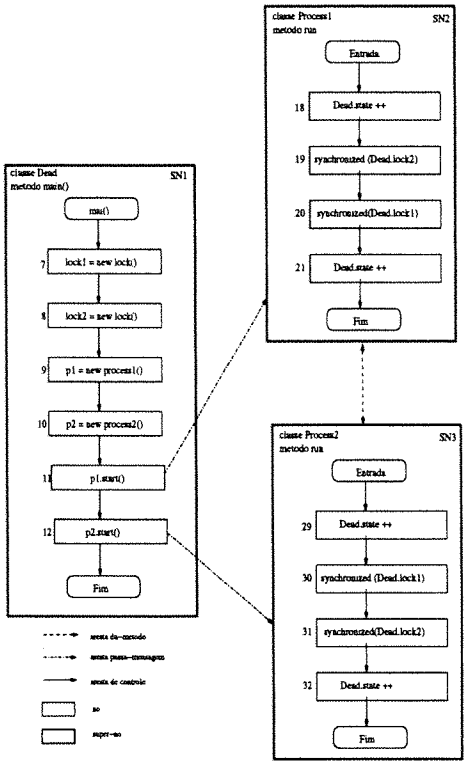


Figura 4.4: DFO do Programa *Dead.java*

situações do programa seria quando o processo "um" adquire o bloqueio do objeto *lock1* enquanto o processo "dois" adiquire o bloqueio do objeto *lock2*. Neste caso, ambos os processos ficariam à espera do desbloqueio um do outro.

Assim como foi realizado no critério baseado no fluxo do objeto, as definições e os usos de cada objeto são identificados:

1. Objeto *lock1*
   Definido : 7;
   Usado : 19, 31.
2. Objeto *lock2*
   Definido : 8;
   Usado : 20, 30.
3. Objeto *p1*
   Definido : 9;
   Usado : 11.
4. Objeto *p2*

Definido : 10;
Usado : 12.

5. Objeto *Dead*
   Definido : 18, 21, 29, 32;
   Usado : 18, 19, 20, 21, 29, 30, 31, 32.

Após estas identificações, o diagrama de fluxo de objeto (DFO), Figura 4.4, é criado para então definir todos os pares *du* passíveis de execução. É relevante lembrar que para criar o DFO também é necessário conhecer o fluxo de controle dos objetos:

- *lock1*: (7, 19), (7, 31);
- *lock2*: (8, 20), (8, 30);
- *p1*: (9, 11);
- *p2*: (10, 12);
- *Dead*: (18, 19), (18, 20), (18, 21), (18, 29), (18, 30), (18, 31), (18, 32), (21, 29), (21, 30), (21, 31), (21, 32), (29, 30), (29, 31), (29, 32), (29, 18), (29, 19), (29, 20), (29, 21), (32, 18), (32, 19), (32, 20), (32, 21).

Com base na definição dos pares, a *Bandera* é usada para verificar a propriedade de *deadlock*. Como tal propriedade é violada, dois contra-exemplos são gerados pela *Bandera*. Ao analisar os dois contra-exemplos gerados: de menor caminho e o aleatório, percebemos que eles são iguais. Os passos deste contra-exemplo são os seguintes:

```
Dead        Process1        Process2
7               |               |
8               |               |
9               |               |
10              |               |
11 ----------> 18               |
|              19*              |
12 <---------- |                |
|  ---------- | ------------> 29
|              |                30*
```

Ao analisar o contra-exemplo gerado, podemos dizer que os seguintes pares *du* foram exercitados: (7, 19), (8, 30), (9, 11), (10, 12), (18, 19), (18, 29) e (29, 30).

A Tabela 4.2 nos mostra um resumo da análise dos requisitos de teste que foram cobertos para este programa. Se observamos a coluna CT, percebemos que apenas

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
s i n c	*lock1*	-	2	7,14%	1	50%	3,57%	17,85%	Ocorre na est. Seq quando o obj. p1 está com o bloqueio do obj. *lock2* e obj. p2 está com bloqueio do obj. *lock1* então ambos ñ conseguem terminar	Monitores aninhados
	lock2	-	2	7,14%	1	50%	3,57%			
	Dead	-	22	78,57%	3	13,64%	10,71%			
ñ s i n c	*p1*	run	1	3,57%	1	100%	3,57%	7,69%		
	p2	run	1	3,57%	1	100%	3,57%			
			28		7			25%		

Tabela 4.2: Cobertura do programa *Dead.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

17,85% dos pares *du* dos objetos sincronizados foram exercitados. Isto retrata uma baixa cobertura, considerando que a quantidade de pares de objetos sincronizados é de 26. Observando a coluna NPE, os 2 pares de objetos não sincronizados foram exercitados. Devido à grande discrepância entre a quantidade de objetos sincronizados (26) e não sincronizados(2), percebemos que o nível de cobertura do critério é praticamente dependente dos pares exercitados de objetos sincronizados, que foram apenas 5. Com isto, notamos que a porcentagem de requisitos exercitados deste programa é apenas 25%.

4.2.3 Programa *MonitorDead.java*

O próximo programa (Figura 4.5) retrata a chamada aninhada de monitores. Neste exemplo, os objetos *t1* e *t2* são instâncias das classes *Thread1* e *Thread2*, respectivamente, os quais chamam um método sincronizado da classe *Wrapper* e esta faz uma chamada para outro método sincronizado da classe *Par*. Dentro do monitor interno, o *Thread1* entra em estado de espera enquanto alguma condição for verdadeira. E este continua esperando até o *Thread2* mudar sua condição. O problema deste exemplo está quando o *Thread1* libera o monitor interno mas não libera o monitor externo. Neste caso, o *Thread2* nunca poderá despertar o *Thread1* pois este não liberou o monitor externo. Antes de criar o DFO deste programa, as definições e usos de cada objeto foram identificados:

```
 1: public class NestedMonitor{
 2:     public static void main(String[] args) {
 3:         Pair p = new Pair();
 4:         Wrapper w = new Wrapper(p);
 5:         Thread1 t1 = new Thread1(w);
 6:         Thread2 t2 = new Thread2(w);
 7:
 8:         t1.start();
 9:         t2.start();
10:     }
11: }
12:
13: class Pair {
14:     private int hi = 0;
15:     private int lo = 0;
16:     public synchronized void inclo() {
17:         while (lo >= hi)
18:             try { wait(); } catch ( InterruptedException ex)
19:         lo++;
20:     }
21:     public synchronized void inchi() {
22:         hi++;
23:         notify();
24:     }
25: }
26:
27: class Wrapper {
28:     private Pair p;
29:     Wrapper(Pair x) {p = x;}
30:     public synchronized void inclo() { p.inclo(); }
31:     public synchronized void inchi() { p.inchi(); }
32: }
33:
34: class Thread1 extends Thread {
35:     private Wrapper w;
36:     Thread1(Wrapper x) {w = x;}
37:     public void run() {
38:         w.inclo();
39:     }
40: }
41:
42: class Thread2 extends Thread {
43:     private Wrapper w;
44:     Thread2(Wrapper x) {w = x;}
45:     public void run() {
46:         w.inchi();
47:     }
48: }
```

Figura 4.5: Programa *MonitorDead.java*

1. Objeto p

   Definido : 3, 30, 31 ;
   Usado : 4, 30, 31.

2. Objeto w

   Definido : 4, 38, 46;
   Usado : 5, 6, 38.

3. Objeto $t1$

   Definido : 5, 8;
   Usado : 8.

4. Objeto $t2$

   Definido : 6, 9;
   Usado : 9.

Ao analisarmos o fluxo de controle dos objetos e as identificações das definições e usos destes objetos, podemos criar o DFO (Figura 4.6) do programa descrito na Figura 4.5. Junto com a construção do DFO foram identificados todos os pares *du* que podem ser executados:

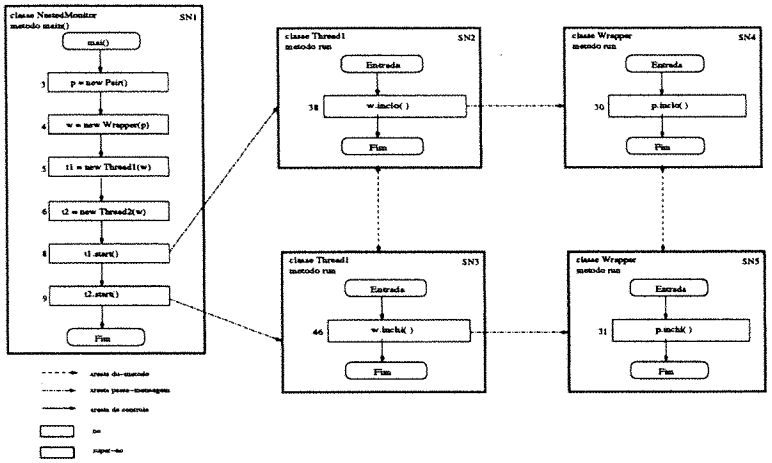


Figura 4.6: DFO do Programa *MonitorDead.java*

- *p*: (3, 4), (31, 30), (30, 31), (3, 30), (3, 31)
- *w*: (4, 5), (4, 6), (46, 38), (38, 46), (4, 38), (4, 46);
- *t1*: (5, 8);
- *t2*: (6, 9).

O próximo passo da análise, é usar a *Bandera* para verificar a ausência de *deadlock*. Ao utilizar tal ferramenta, percebemos que os contra-exemplos gerados, o de menor caminho e o aleatório, são os mesmos e os passos deste contra-exemplo são:

```
NestedMonitor       Thread1       Thread2        Wrapper          Pair
3                     |             |              |              |
4 ------------------- | ----------- | -----------> 29             |
| <------------------ | ----------- | ------------ |              |
5 ------------------> 36            |              |              |
| <------------------ |             |              |              |
6 ------------------- | ----------> 44             |              |
| <------------------ | ----------- |              |              |
8 ------------------> 38*           |              |              |
|                     | ----------- | -----------> 30             |
|                     |             |              | --------> 17
| <------------------ | ----------- | ------------ | --------- 18
9 ------------------- | ----------> 46*            |              |
```

Analisando o contra-exemplo gerado, podemos notar que dos 13 pares *du* identificados, 6 pares foram exercitados:

- *p*: (3, 4), (3, 30);
- *w*: (4, 5), (4, 6), (4, 38);
- *t1*: (5, 8);

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
sinc	*p*	*inclo*(s) *inchi*(s)	4	30,77%	1	25%	7,69%	15,38%	Ocorre na est. Seq, quando o monitor mais interno de t1 é liberado (p.inclo) e o monitor mais externo(w.inclo) não é liberado. Com isto t2 não conseguirá executar (w.inchi)	Monitores aninhados
	w	*inclo*(s) *inchi*(s)	4	30,77%	1	25%	7,69%			
nsinc	*p*	-	1	7,69%	1	100%	7,69%			
	w	-	2	15,38%	2	100%	15,38%			
	t1	*run*	1	7,69%	1	100%	7,69%	30,77%		
	t2	*run*	1	7,69%	0	0	0			
			13		6			46,15%		

Tabela 4.3: Cobertura do programa *MonitorDead.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

A Tabela 4.3, nos mostra que 46,15% dos pares *du* foram exercitados. Observando a coluna NPE desta Tabela, percebemos que a quantidade de pares sincronizados que foram exercitados (2 pares) está bem abaixo da metade, comparando com quantidade total de pares *du* sincronizados (8 pares). Por outro lado, realizando a mesma comparação para os pares não sincronizados, podemos perceber que a maioria dos pares identificados (5 pares) também foram exercitados (4 pares). Com base nestas informações, podemos dizer que a média de cobertura total dos pares *du* está abaixo dos 50%, mas se comparamos apenas os pares dos objetos não sincronizados eles apresentaram uma boa cobertura.

4.2.4 Programa *Phill33.java*

O programa *Phill33.java* retrata o problema clássico do jantar entre filósofos. Cinco filósofos estão em uma mesa circular. Cada filósofo passa sua vida alternando-se entre pensar e comer. No centro da mesa está um grande prato com macarrão tipo *spaggeti*. Um filósofo necessita de dois garfos para comer o *spaggeti*. Infelizmente, eles só possuem 5 garfos. Um garfo é colocado entre cada par de filósofos e eles concordam que cada um somente utilizará os garfos que estiverem imediatamente a sua esquerda e direita.

O problema acontece quando todos os cinco filósofos pegam, simultaneamente, o garfo que esta a sua esquerda e logo em seguida tentam pegar o garfo a sua direita. O que não é possível, pois este já está sendo utilizado. Com isto, cada filósofo fica a espera da liberação de um garfo, o que nunca ocorrerá, pois todos os filósofos encontram-se neste situação.

```
 1: class Fork{
 2:     private boolean is_free;
 3:
 4:     Fork(){
 5:         is_free = true;
 6:     }
 7:
 8:     public synchronized void take(){
 9:         while (!is_free)
10:             try {
11:                 wait();
12:             } catch (InterruptedException e) {return;}
13:
14:         is_free = false;
15:     }
16:
17:     public synchronized void leave(){
18:         is_free = true;
19:         notify();
20:     }
21: }
22:
23: class Philosopher extends Thread{
24:     private Fork left;
25:     private Fork right;
26:
27:     Philosopher(Fork f1, Fork f2){
28:         left = f1;
29:         right = f2;
30:     }
31:
32:     public void eat(){
33:         left.take();
34:         right.take();
35:     }
36:
37:     public void think(){
38:         left.leave();
39:         right.leave();
40:     }
41:
42:     public void run(){
43:         while (true)
44:             {
45:                 eat();
46:                 think();
47:             }
48:     }
49: }
50:
51: public class Phil33{
52:     public static void main(String[] args){
53:         Fork f1;
54:         Fork f2;
55:         Philosopher p1;
56:         Philosopher p2;
57:         f1 = new Fork();
58:         f2 = new Fork();
59:         p1 = new Philosopher(f1, f2);
60:         p2 = new Philosopher(f2, f1);
61:         p1.start();
62:         p2.start();
63:     }
64: }
```

Figura 4.7: Programa *Phil33.java*

Para o programa da Figura 4.7 foi considerado apenas dois filósofos e dois garfos. O *deadlock* acontece quando os dois filósofos ficam esperando a liberação de um garfo. Antes de analisar este programa é relevante o entendimento de certas terminologias.

Como foi descrito anteriormente, a ferramenta *Bandera* tem certas limitações e uma delas é que dois ou mais objetos não podem ser instâncias de uma classe que estende uma *Thread*. Para resolver tal problema é recomendado que cada objeto seja instância da mesma classe (classe original), só que com nomes diferentes. Para este trabalho será utilizando o termo classes clone quando se referir ao conjunto de classes renomeadas de uma classe *Thread*.

Veja o exemplo do programa da Figura 4.7, onde os objetos *p1* e *p2* são instâncias da classe *Philosopher*. Para que este programa seja analisado pela *Bandera*, é necessário criar uma classe clone e instanciar um destes objetos por tal classe enquanto o outro continua sendo instância da classe *Philosopher*.

Também é relevante lembrar que as classes clone serão usadas apenas para a verificação na *Bandera*, ou seja, para as análises dos programas serão considerados que todos objetos são instâncias da classe original. Para diferenciar as linhas onde o objeto é definido ou usado em cada instância da classe original, será utilizado uma letra (de "a" à "z") junto ao número da linha. Veja que no exemplo da Figura 4.7, os objetos *p1* e *p2* são instâncias da mesma classe original. Para diferenciar o fluxo dos objetos em cada instância foram consideradas as seguintes linhas:

- 33, 34, 38, 39, são os números das linhas para o objeto *p1*;
- 33a, 34a, 38a, 39a são os números das linhas para o objeto *p2*.

Para este programa foram identificadas todas as definições e os usos de cada objeto:

1. Objeto *f1*
   Definido : 57, 33, 38, 34a, 39a;
   Usado : 59, 60, 33, 34a.
2. Objeto *f2*
   Definido : 58, 33a, 38a, 34, 39;
   Usado : 59, 60, 33a, 34.
3. Objeto *p1*
   Definido : 59, 45, 46;
   Usado : 61, 45.
4. Objeto *p2*
   Definido : 60, 45a, 46a;
   Usado : 62, 45a.

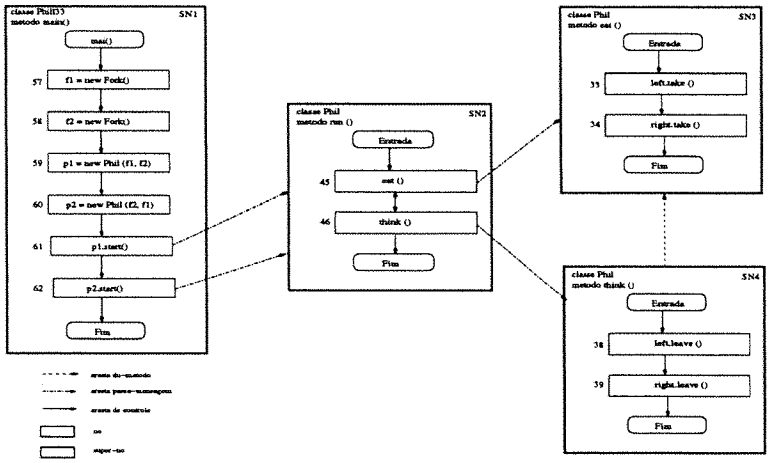


Figura 4.8: DFO do Programa *Phil33.java*

Após identificar as definições e usos de cada objeto, estes dados são utilizados junto com o fluxo de controle dos objetos para construir o DFO (Figura 4.8) deste programa. À medida em que é criado tal diagrama, também são criados os pares *du* para cada objeto:

- *f1*: (57, 59), (57, 60), (38, 33), (39a, 34a), (57,33), (57, 34a), (33, 34a), (34a, 33), (39a, 33), (38, 34a);
- *f2*: (58, 59), (58, 60), (39, 34), (38a, 33a), (58, 33a), (58, 34), (33a, 34), (34, 33a), (39, 33a), (38a, 34);
- *p1*: (59, 61), (46, 45);
- *p2*: (60, 62), (46a, 45a).

Com relação a este programa, a *Bandera* gerou dois contra-exemplos distintos: o de menor caminho e o aleatório. O contra-exemplo de menor caminho está descrito logo a seguir. Já o contra-exemplo aleatório encontra-se no Apêndice A:

```
Phil33      Philosopher        Fork
57 ---------- | ------------> 5
| <---------- | ------------- |
58 ---------- | ------------> 5
| <---------- | ------------- |
59 ---------> 28               |
|             29               |
```

```
| <---------- |              |
60 ---------> 28a            |
|             29a            |
| <---------- |              |
61 ---------> 45             |
|             33 ----------> 9
|             | <---------- 14
|             34 ----------> 9
|             | <---------- 14
|             46             |
|             38 ----------> 18
|             | <---------- 19
|             39 ----------> 18
|             | <---------- 19
|             45             |
|             33 ----------> 9
| <---------- | ------------ 14
62 ---------> 45a            |
|             33a ----------> 9
|             | <---------- 14
|             34a ----------> 9
|             | <---------- 11*
|             34 ----------> 9
|             |              11*
```

Analisando o contra-exemplo de menor caminho, percebemos que os seguintespares *du* foram exercitados:

- *f1*: (57, 59), (57, 60), (38, 33), (57, 33), (34a, 33);
- *f2*: (58, 59), (58, 60), (58, 34);
- *p1*: (59, 61), (46, 45);
- *p2*: (60, 62).

Já a análise feita sobre o contra-exemplo aleatório, que encontra no Apêndice A, temos que os seguintes pares foram exercitados:

- *f1*: (57, 59), (57, 60), (38, 33), (39a, 34a), (57, 33), (34a, 33), (39a, 33), (38, 34a);
- *f2*: (58, 59), (58, 60), (39, 34), (38a, 33a), (58, 34), (33a, 34), (34, 33a), (38a, 34);
- *p1*: (59, 61), (46, 45);
- *p2*: (60, 62), (46a, 45a).

Comparando os pares exercitados pelos dois contra-exemplos, podemos afirmar que todos os pares exercitados no contra-exemplo de menor caminho também são exercitados no contra-exemplo aleatório. Com isto, notamos que dos 24 pares *du* identificados

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
s i n c	*f1*	*take(s)* *leave(s)*	8	33,33%	6	75%	25%	45,83%	Ocorre na est. *while*, quando p1 e p2 estão com bloqueio de f1 e f2 respectivamente através da execução de take e p1 entra em estado de espera da liberação de f2 e p2 da liberação de f1	Semáforo
	f2	*take(s)* *leave(s)*	8	33,33%	5	62,5 %	20,83%			
n s i n c	*f1*		2	8,33%	2	100%	8,33%	33,33%		
	f2		2	8,33%	2	100%	8,33%			
	p1	*eat, think run*	2	8,33%	2	100%	8,33%			
	p2	*eat, think run*	2	8,33%	2	100%	8,33%			
			24		19			79,17%		

Tabela 4.4: Cobertura do programa *Phil33.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

19 foram exercitados. A Tabela 4.4 nos mostra que 79.17% dos pares *du* foram exercitados.

Ao observarmos a coluna CO da Tabela 4.4, deduzimos que a porcentagem de cobertura dos pares *du* sincronizados de cada objeto é considerada boa, pois todas encontram-se acima de 60%. Já se analisarmos os pares não sincronizados, nos podemos afirmar que a porcentagem de cobertura destes pares é ótima, pois todos foram exercitados. Se analisarmos as colunas TO e CT, temos que dos 66,66% dos pares sincronizados 45,83% foram exercitados. Através destes dados viemos confirmar que a porcentagem de cobertura dos objetos sincronizados também é considerada satisfatória.

4.2.5 Programa *PCSimple.java*

Este é um simples problema em que um objeto produtor insere um elemento no *buffer* e um objeto consumidor retira um elemento do *buffer* (Figura 4.9). As operações de inserir e de retirar são sincronizadas, ou seja, não é possível inserir um elemento enquanto a operação de retirada estiver bloqueada e vice-versa. O *Deadlock* acontece quando o produtor não desperta o consumidor para que este possa agir.

Para dar continuidade a análise deste programa, inicialmente, é preciso identificar as definições e usos de cada objeto:

```
1: public class PCSimple{
2:     public static void main(String[] args){
3:         int numIntegers = 6;
4:         Buffer b = new Buffer(2);
5:         Producer p = new Producer(numIntegers, b);
6:         Consumer c = new Consumer(numIntegers, b);
7:         p.start();
8:         c.start();
9:     }
10: }
11:
12: class Producer extends Thread
13: {
14:     private int count;
15:     private Buffer buffer;
16:
17:     public Producer(int num, Buffer b){
18:         count = num;
19:         buffer = b;
20:     }
21:
22:     public void run(){
23:         for (int i=0; i < count; i++)
24:             buffer.put(i);
25:         buffer.put(-1);
26:     }
27: }
28:
29: class Consumer extends Thread{
30:     private int count;
31:     private Buffer buffer;
32:
33:     public Consumer(int num, Buffer b){
34:         count = num+1; //for that extra -1
35:         buffer = b;
36:     }
37:
38:     public void run(){
39:      int[] received = new int[count];
40:      for(int i=0; i<count; i++){
41:            received[i] = buffer.get();
42:       if(received[i] == -1){
43:         System.out.print("Consumer.run: got last number");
44:             return;
45:           }
46:       }
47:     }
48: }
49:
```

```
50: class Buffer{
51:     protected int size;
52:     protected int[] array;
53:     protected int putPtr;
54:     protected int getPtr;
55:     protected int usedSlots;
56:
57:     public Buffer(int b){
58:       size = b;
59:       array = new int[b];
60:       putPtr = 0;
61:       getPtr = 0;
62:       usedSlots = 0;
63:     }
64:
65:     public synchronized void put(int x){
66:         while(usedSlots == size)
67:             try{
68:                 System.out.println("producer wait");
69:                 wait();
70:             }catch(InterruptedException e){};
71:
72:         array[putPtr] = x;
73:         System.out.println("Buffer.put: put "+x);
74:         putPtr = (putPtr + 1)%size;
75:
76:         usedSlots++;
77:     }
78:
79:     public synchronized int get(){
80:         while(usedSlots == 0)
81:             try{
82:                 System.out.println("consumer wait");
83:                 wait();
84:             }catch(InterruptedException e){};
85:         int x = array[getPtr];
86:         System.out.println("Buffer.get: got "+x);
87:         if(x<0)
88:             return x;
89:         getPtr = (getPtr+1)%size;
90:         if(usedSlots == size) notify();
91:         usedSlots --;
92:         return x;
93:     }
94: }
```

Figura 4.9: Programa *PCSimple.java*

1. Objeto b

   Definido : 4, 24, 25, 41;
   Usado : 5, 6, 24, 25, 41.

2. Objeto p

   Definido : 5;
   Usado : 7.

3. Objeto c

   Definido : 6;
   Usado : 8.

Para construir o DFO da Figura 4.10, foram utilizados as informações obtidas das definições e usos de cada objeto e, também, o conhecimento do fluxo de controle dos objetos. Enquanto o DFO é criado, são identificados os pares *du* passíveis de execução:

- b: (4, 5), (4, 6), (4, 24),(4, 41), (24, 41), (41, 24), (41, 25), (25, 41), (24, 25);
- p: (5, 7);

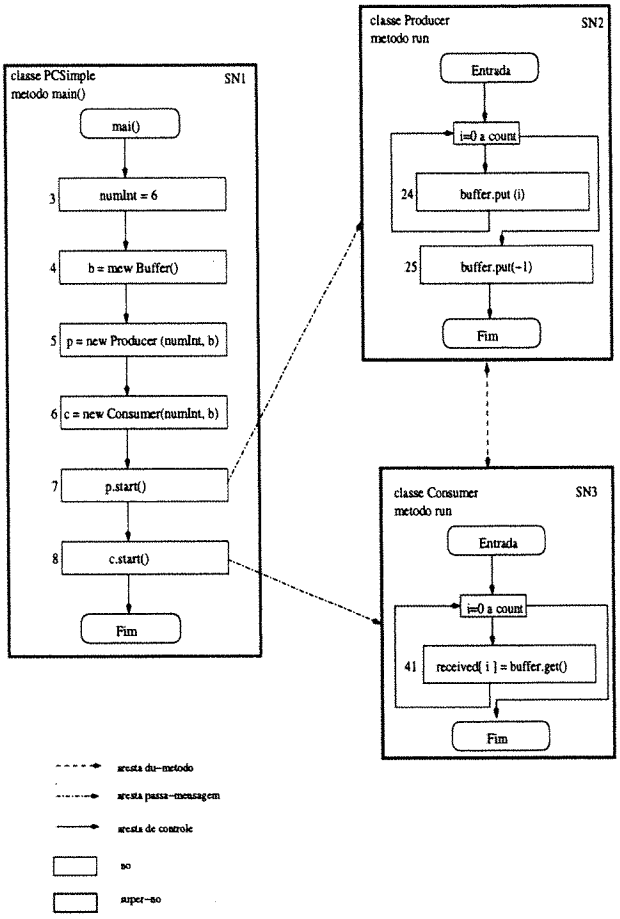
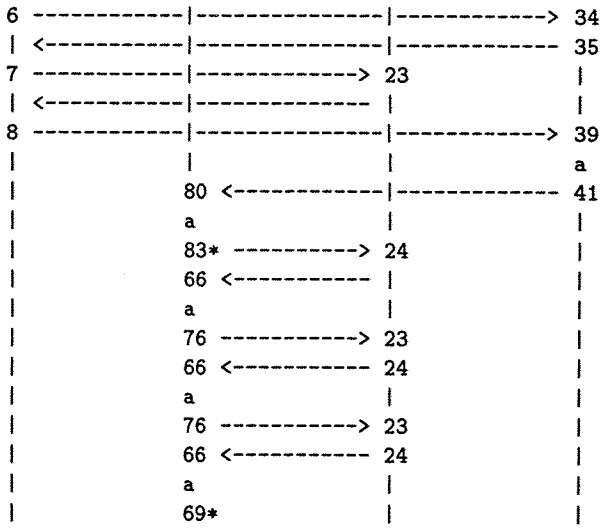


Figura 4.10: DFO Programa *PCSimple.java*

- *c*: (6, 8).

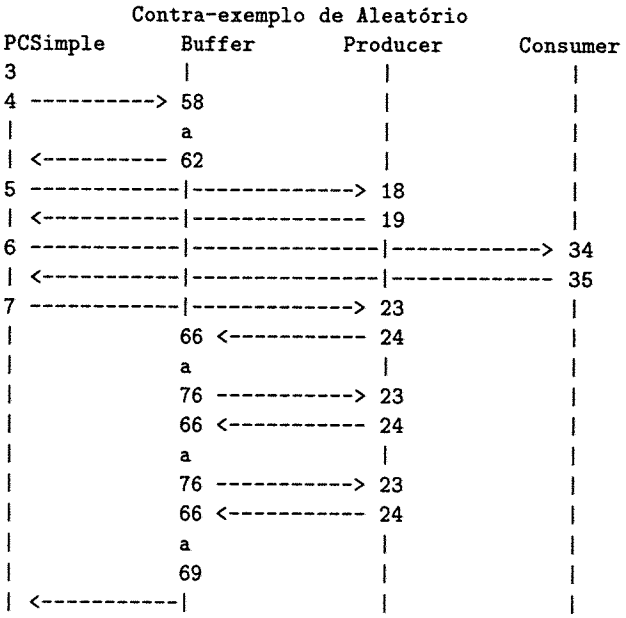
Com relação a este programa, a *Bandera* gerou dois contra-exemplos distintos: o de menor caminho e o aleatório. Estes contra-exemplos são descritos logo a seguir:

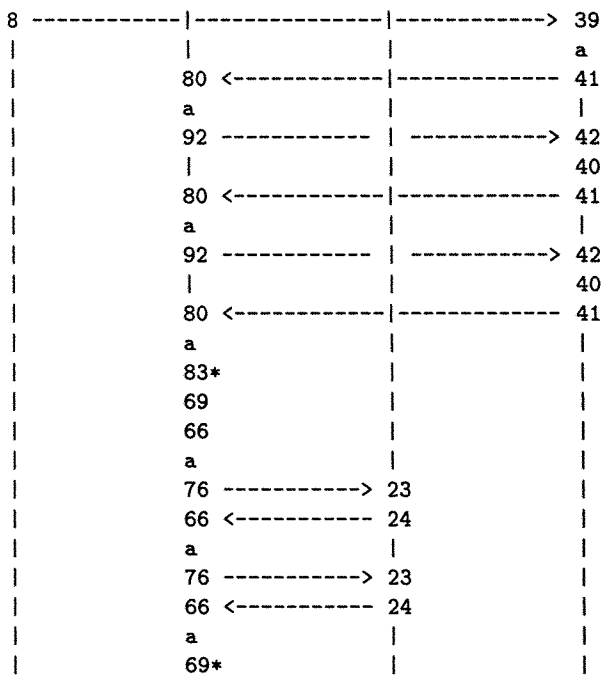
```
          Contra-exemplo de menor caminho
PCSimple       Buffer        Producer       Consumer
3              |                |              |
4 ----------> 58               |              |
|              a                |              |
| <---------- 62                |              |
5 ------------|--------------> 18              |
| <-----------|--------------- 19              |
```



Observando o contra-exemplo de menor caminho, podemos dizer que os seguintes pares *du* foram exercitados:

- *b*: (4, 5), (4, 6), (4, 41), (41, 24);
- *p*: (5, 7);
- *c*: (6, 8).





Por outro lado, ao observarmos o caminho do contra-exemplo aleatório, percebemos que todos os 6 pares exercitados pelo contra-exemplo de menor caminhotambém são exercitados por este contra-exemplo. Além disso, também é possível notar que 2 pares a mais foram exercitados por este contra-exemplo:

- b: (4, 5), (4, 6), (4, 24),(4, 41), (24, 41), (41, 24);
- p: (5, 7);
- c: (6, 8).

Com isto, temos que 8 pares *du* foram exercitados contra 11 pares identificados. Percentualmente, a Tabela 4.5 mostra que 72,73% dos pares *du* foram exercitados. Para este caso, é interessante notar que a porcentagem de pares sincronizados e não sincronizados é de 36,36%. Embora sejam a mesma, também é relevante ressaltar que todos os pares não sincronizados foram exercitados, enquanto dos 6 pares sincronizados, 4 foram exercitados. Em termos de porcentagem, isto nos mostra que dos 63,64% que deveria ser exercitados, 36,36% foram exercitados (colunas `TO` e `CT`, respectivamente). Por outro lado, mesmo que os pares *du* dos objetos sincronizados não tenham sido

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
sinc	*b*	*put(s)* *get(s)*	7	63,64%	4	57,14%	36,36%	36,36%	Ocorre na est. *for*, quando o obj. c entra em estado de espera (c.get()) e o obj. p também entra em estado de espera (p.put()) e não desperta o obj. c	Semáforo
nsinc	*b* *p* *c*	- *run* *run*	2 1 1	18,18% 9,09% 9,09%	2 1 1	100% 100% 100%	18,18% 9,09% 9,09%	36,36%		
			11		8			72,73%		

Tabela 4.5: Cobertura do programa *PCSimple.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

totalmente exercitados, a cobertura total dos requisitos foi considerada muito boa, pois encontra-se acima de 70%.

4.2.6 Programa *MainPC.java*

O programa descrito na Figura 4.11 também retrata a idéia de um produtor e um consumidor usando monitores aninhados. O problema deste programa é que ele entra em *deadlock* quando é liberado o bloqueio do monitor mais interno e não libera o bloqueio do monitor mais externo.

Segundo o critério de teste adotado, o primeiro passo é identificar as definições e usos de cada objeto:

1. Objeto *pool*

   Definido : 3, 36, 22;
   Usado : 4, 5, 22.
2. Objeto *p*

   Definido : 4;
   Usado : 6.
3. Objeto *c*

   Definido : 5;
   Usado : 7.
4. Objeto *gd*

   Definido : 75, 79, 83;
   Usado : 79.

```
 1: public class MainPC{
 2:     public static void main(String args[]){
 3:         My_Pool pool=new My_Pool();
 4:         Prod p=new Prod(pool);
 5:         Cons c=new Cons(pool,1);
 6:         p.start();
 7:         c.start();
 8:     }
 9: }
10: class Cons extends Thread{
11:     My_Pool p;
12:     int _id;
13:
14:     public Cons(My_Pool pool, int id){
15:         p=pool;
16:         _id=id;
17:     }
18:
19:     public void run(){
20:         while(true){
21:             System.out.println(_id+"Tirando algo do pool");
22:             int val=p.get();
23:             System.out.println(_id+"Tirei: "+val);
24:         }
25:     }
26: }
27: class Prod extends Thread{
28:     My_Pool p;
29:     public Prod(My_Pool pool){
30:         p=pool;
31:     }
32:
33:     public void run(){
34:         while(true){
35:            System.out.println("Estou colocando algo...");
36:            p.put(1);
37:         }
38:     }
39: }
```

```
40: class My_Guard{
41:     int slot;
42:     int status;
43:
44:     public My_Guard(){
45:         slot=0;
46:         status=0;
47:     }
48:
49:     synchronized int get(){
50:         try{
51:             while(status==0) wait();
52:             status=0;
53:         }
54:         catch(Exception e){
55:             System.out.println("Erro no get...");
56:         }
57:         return slot;
58:     }
59:
60:     synchronized void put(int i){
61:         try{
62:             slot=i;
63:             status=1;
64:             notify();
65:         }
66:         catch(Exception e){
67:             System.out.println("Erro no put");
68:         }
69:     }
70: }
71: class My_Pool{
72:     My_Guard gd;
73:
74:     public My_Pool(){
75:         gd=new My_Guard();
76:     }
77:
78:     synchronized int get(){
79:         return gd.get();
80:     }
81:
82:     synchronized void put(int i){
83:         gd.put(i);
84:     }
85: }
```

Figura 4.11: Programa *MainPC.java*

O próximo passo é criar o DFO (Figura 4.12) deste programa. Para construir o DFO, é necessário termos a informação do fluxo de controle de cada objeto, além das definições e usos dos mesmos. À medida em que este diagrama é construído, também são identificados todos os pares *du* que possam ser executados:

- *pool*: (3, 4), (3, 5), (3, 22),(36, 22);
- *p*: (4, 6);
- *c*: (5, 7);
- *gd*: (75, 79), (83, 79).

Com relação a este programa, a *Bandera* gerou dois contra-exemplos distintos: o de menor caminho e o aleatório. Logo a seguir está descrito o contra-exemplo de menor caminho. Já o contra-exemplo aleatório encontra-se elucidado no Apêndice A:

```
MainPC          Prod          Cons          My_Guard      My_Pool
```

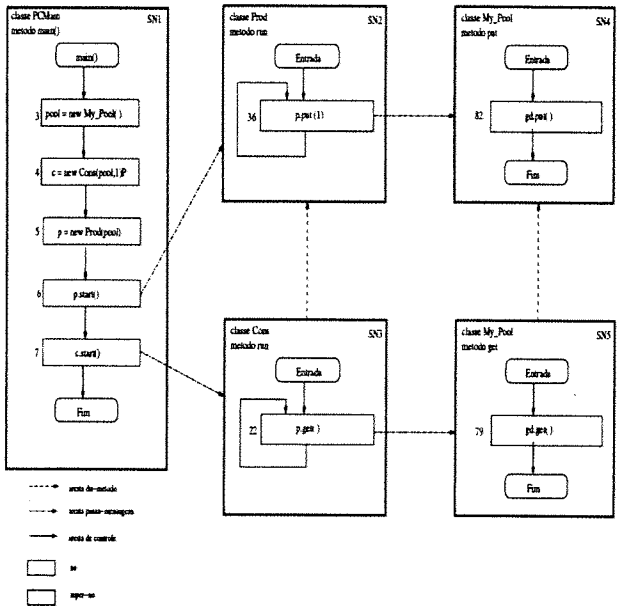


Figura 4.12: DFO do programa *PCMain.java*

```
3 ------------------ | ------------- | ------------ | --------> 75
|                    |               |              45 <-------- |
| <----------------- | ------------- | ------------ 46          |
4 ------------------> 30              |              |           |
| <----------------- |               |              |           |
5 ------------------ | ------------> 15             |           |
| <----------------- | ------------- |              |           |
6                    |               |              |           |
7 ------------------ | ------------> 22             |           |
|                    |               | ------------ | --------> 79
|                    |               |              51* <------- |
|                    36* <---------- | ------------ |           |
```

Observando os caminhos dos dois contra-exemplos gerados, podemos dizer que 6 dos 8 pares *du* identificados foram exercitados. O interessante deste análise, é perceber que 2 dos pares exercitados no contra-exemplo aleatório não é exercitado no contra-exemplo de menor caminho e vice-versa. Com isto, podemos afirmar que todos os pares *du* foram exercitados.

A Tabela 4.6 retrata bem este resultado. Também, é interessante observar que neste caso a quantidade de pares *du* de objeto sincronizados é o mesmo de objetos não sincronizados e ambos tiveram uma ótima cobertura.

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
s i n c	*gd*	*put(s)* *get(s)*	2	25%	2	100%	25%	50%	Ocorre na est. *while* quando o monitor mais interno de p é liberado (gd.put()) e o monitor mais externo (pool.put()) não é liberado. Com isso c não conseguirá executar (pool.get())	Monitores Aninhados
	pool	*put(s)* *get(s)*	2	25%	2	100%	25%			
n s i n c	*pool*	-	2	25%	2	100%	25%	50%		
	p	*run*	1	12,5%	1	100%	12,5%			
	c	*run*	1	12,5%	1	100%	12,5%			
			8		8			100%		

Tabela 4.6: Cobertura do programa *MainPC.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

4.2.7 Programa *NestedMonitor.java*

```
1:  interface Buffer {
2:      public void put(Object o)
3:          throws InterruptedException; //poe o objeto no buffer
4:      public Object get()
5:          throws InterruptedException; //retira um objeto do buffer
6:  }
7:
8:  class Semaphore {
9:      private int value;
10:     public Semaphore (int initial) {
11:         value = initial;
12:     }
13:
14:     synchronized public void up() {
15:         ++value;
16:         notifyAll();
17:     }
18:
19:     synchronized public void down() throws InterruptedException {
20:         while (value==0) {wait();}
21:         --value;
22:     }
23: }
24:
25: /********************SEMABUFFER*****************************/
26: class SemaBuffer implements Buffer {
27:     protected int size;
28:     Semaphore full; //conta o numero de itens
29:     Semaphore empty;//conta o numero de espaco
30:
31:     SemaBuffer(int size) {
32:         this.size = size;
33:         full = new Semaphore(0);
34:         empty= new Semaphore(size);
35:     }
36:     //public void put(Object o)
37:     synchronized public void put(Object o)
38:         throws InterruptedException {
39:         empty.down();
40:         full.up();
41:     }
42:     // public Object get()
43:     synchronized public Object get()
44:         throws InterruptedException{
45:         full.down();
46:         empty.up();
47:         return (null);
48:     }
49: }
50:
```

```
51: /********** Pricipal ************/
52: public class NestedMonitor  {
53:     static int SIZE = 5; /* parametro */
54:     static Buffer buf;
55:
56:     public static void main(String [] args) {
57:         buf = new SemaBuffer(SIZE);
58:         Producer p = new Producer(buf);
59:         p.start();
60:         Consumer c = new Consumer(buf);
61:         c.start();
62:     }
63: }
64:
65: /******************PRODUCER************************/
66: class Producer extends Thread {
67:     Buffer buf;
68:     Producer(Buffer b) {buf = b;}
69:
70:     public void run() {
71:       try {
72:         while(true)
73:             buf.put(null);
74:       } catch (InterruptedException e){}
75:     }
76: }
77:
78: /********************CONSUMER*****************************
79: class Consumer extends Thread {
80:     Buffer buf;
81:     Consumer(Buffer b) {buf = b;}
82:
83:     public void run() {
84:         try {
85:           while(true)
86:             buf.get();
87:       } catch(InterruptedException e ){}
88:     }
89: }
90:
```

Figura 4.13: Programa *NestedMonitor.java*

Esta é uma outra forma de implementação do programa produtor consumidor (Figura 4.13). A diferença deste exemplo, em relação ao exemplo da Seção 4.2.6, é que ele trabalha com dois *buffers* ao mesmo tempo. Um *buffer* que inicialmente é vazio e outro cheio. Nesta implementação, o *deadlock* acontece devido à presença de monitores aninhados: um objeto interno libera o bloqueio, enquanto o mais externo não. Para este programa também foram identificadas todas as definições e usos de cada objeto:

1. Objeto *buf*

   Definido :57, 73, 86;
   Usado : 58, 60, 73, 86.
2. Objeto *full*

   Definido : 33, 40, 45;
   Usado : 40, 45.
3. Objeto *empty*

   Definido : 34, 39, 46;
   Usado : 39, 46.
4. Objeto *p*

   Definido : 58;
   Usado : 59.
5. Objeto *c*

   Definido : 60;
   Usado : 61.

Com base neste dados e no fluxo de controle dos objetos, é construído o DFO (Figura 4.14) e enquanto é construído tal diagrama, nós também vamos identificando todos os possíveis pares *du*:

- *buf*: (57, 58), (57, 60), (57, 73), (57, 86), (73, 86), (86, 73);
- *full*: (33, 40), (33, 45), (40, 45), (45, 40);
- *empty*: (34, 39), (34, 46), (39, 46), (46, 39);
- *p*: (58, 59);
- *c*: (60, 61) .

Ao utilizar a *Bandera* para verificar a propriedade de *deadlock*, esta gerou um contra-exemplo de menor caminho, o qual será descrito logo a seguir, e um contra-exemplo aleatório que encontra-se no Apêndice A:

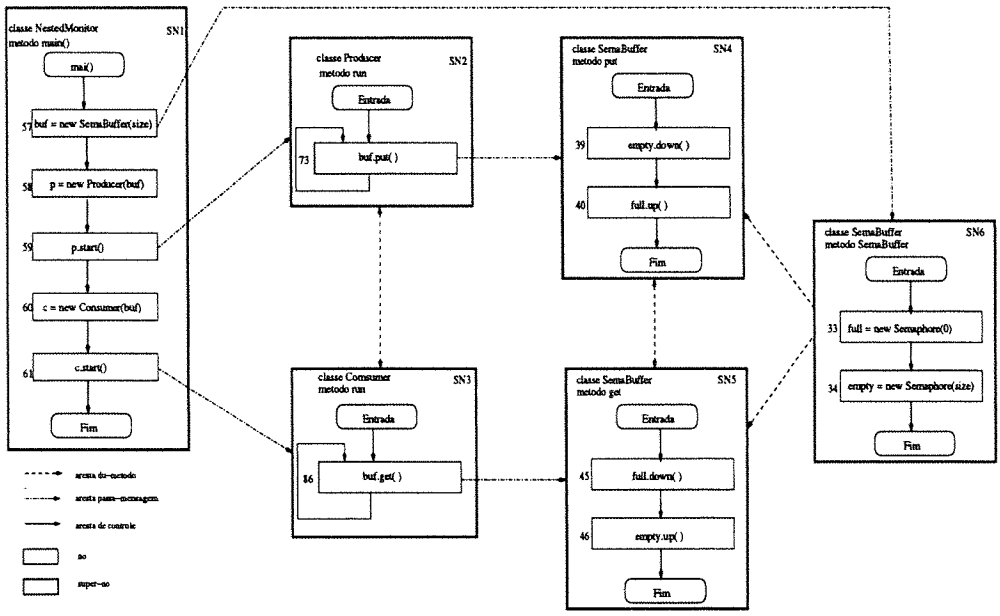


Figura 4.14: DFO do programa *NestedMonitor.java*

```
NestedMonitor      Producer         Consumer     SemaBuffer     Semaphore
57 ----------------- | ------------ | ----------> 33           |
|                    |              |             | --------> 11
|                    |              |             34 <-------- |
|                    |              |             | --------> 11
| <----------------- | ------------ | ----------- | --------- |
58 ----------------> 68             |             |           |
| <----------------- |              |             |           |
59                   |              |             |           |
60 ----------------- | ------------> 81           |           |
| <----------------- | ------------ |             |           |
61 ----------------- | ------------> 86           |           |
|                    |              | ----------> 45           |
|                    |              |             | --------> 20*
|                   73* <---------- | ----------- | --------- |
```

Ao analisar os caminhos dos contra-exemplos gerados, percebemos que alguns requisitos exercitados no contra-exemplo de menor caminho não foram exercitados no contra-exemplo aleatório e vice-versa. O contra-exemplo de menor caminho exercita 5 pares *du*, enquanto o aleatório exercita 6 pares:

- *Menor Caminho*: (57, 58), (57, 60), (57, 86), (33, 45), (60, 61);

- *Aleatório*: (57, 58), (57, 60), (57, 73), (33, 40), (34, 39), (58, 59).

Se observarmos bem os pares exercitados, podemos perceber apenas 2 pares exercitados no contra-exemplo de menor caminho também são exercitados no contra-exemplo aleatório. Com isto, afirmamos que dos 16 pares *du* identificados 9 foram exercitados:

- *buf*: (57, 58), (57, 60), (57, 73), (57, 86);
- *full*: (33, 40), (33, 45);
- *empty*: (34, 39);
- *p*: (58, 59);
- *c*: (60, 61).

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
sinc	*buf*	*put*(s) *get*(s)	4	25%	2	50%	12,5%	31,25%	Ocorre na est. *while*, quando monitor mais interno é liberado e o monitor mais externo não é liberado	Monitores Aninhados
	full	*up* (s) *down*(s)	4	25%	2	50%	12,5%			
	empty	*up* (s) *down*(s)	4	25%	1	6,25%	6,25%			
nsinc	*buf*	-	2	12,5%	2	100%	12,5%	25%		
	p	*run*	1	6,25%	1	100%	6,25%			
	c	run	1	6,25%	1	100%	6,25%			
			16		9			56,25%		

Tabela 4.7: Cobertura do programa *NestedMonitor.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

Podemos notar que em termos de porcentagem, 56,25% dos pares *du* foram exercitados. A Tabela 4.7 pode confirmar este dado. Para este caso, também é interessante ressaltar que todos os pares de objetos não sincronizados foram exercitados, o que corresponde a 25% do total de pares *du*. Observando os pares dos objetos sincronizados, podemos dizer que 31,25% do total de pares *du* foram exercitados, contra os 75% que deveriam ser exercitados. Mesmo sabendo que pouco menos da metade dos pares sincronizados foram exercitados, conseguimos atingir uma boa porcentagem de cobertura sobre o total de pares identificados.

```
 1: public class Bounded {
 2:     public static void main (String [] args) {
 3:         BoundedBuffer b1 = new BoundedBuffer(3);
 4:         BoundedBuffer b2 = new BoundedBuffer(3);
 5:         Object o1 = new Object();
 6:
 7:         b1.add(o1);
 8:         b1.add(o1);
 9:         InOut out1 = new InOut(b1,b2);
10:         InOut out2 = new InOut(b2,b1);
11:         out1.start();
12:         out2.start();
13:     }
14: }
15:
16: class BoundedBuffer {
17:     Object [] buffer;
18:     int bound, head, tail;
19:
20:     public BoundedBuffer(int b) {
21:       bound = b;
22:       buffer = new Object[bound];
23:       head = 0;
24:       tail = bound-1;
25:     }
26:
27:     public synchronized void add(Object o) {
28:         while ( tail == head )
29:             try { wait(); }
30:             catch ( InterruptedException ex) {}
31:         buffer[head] = o;
32:         head = (head+1) % bound;
33:
34:     }
35:
36:
37:      public synchronized Object take() {
38:          while ( isEmpty() )
39:              try { wait(); }
40:              catch ( InterruptedException ex) {}
41:          tail = (tail+1) % bound;
42:          notifyAll();
43:          return buffer[tail];
44:     }
45:
46:     public synchronized boolean isEmpty() {
47:         return head == ((tail+1) % bound); }
48: }
49:
50: class InOut extends Thread {
51:      BoundedBuffer in,out;
52:      public InOut(BoundedBuffer ins, BoundedBuffer outs
53:          in = ins;
54:          out = outs;
55:      }
56:      public void run() {
57:          Object tmp;
58:          while(true) {
59:              tmp = in.take();
60:              System.out.println( "Take ");
61:              out.add(tmp);
62:              System.out.println( "Put ");
63:          }
64:      }
65: }
```

Figura 4.15: Programa *Bounded.java*

4.2.8 Programa *Bounded.java*

O programa *Bounded.java* (Figura 4.15) é uma implementação de um *buffer* limitado e algumas classes que utilizam estes *buffers*. A classe *BoundBuffer* possui três métodos, os quais são usados para remover e adicionar elementos no *buffer* e verificar se o *buffer* está vazio. Para adicionar um elemento, o *buffer* não pode estar cheio e/ou possuir um bloqueio que garanta o acesso exclusivo ao *buffer*. Já para remover um elemento, o *buffer* não pode estar vazio e/ou nem possuir um acesso bloqueado. O programa cria dois *buffers* (*b1* e *b2*) e duas *threads* que trabalham com estes *buffers*. O objetivo deste programa é manter a sincronização das atividades (adicionar e remover) entre os dois *buffers*.

O *deadlock* acontece devido à seguinte situação: tanto *b1* quanto *b2* nunca desperta a operação de remover elemento do *buffer* quando este está adicionando um elemento.

O *buffer b1* passa todos os seus elementos para o *buffer b2* e entra em estado de espera e logo em seguida o *buffer b2* passa todos os seus elementos para o *buffer b1*, que também entra em estado de espera. Então, *b1* volta a ficar cheio novamente, podendo realizar a operação de remover elemento, mas em momento algum ele foi despertado quando os elementos foram adicionados.

As definições e usos de cada objeto, para que a análise deste programa seja feita, são:

1. Objeto *b1*
   Definido : 3, 7, 59, 61a;
   Usado : 7, 8, 9, 10, 59, 61a.
2. Objeto *b2*
   Definido : 4, 59a, 61;
   Usado : 9, 10, 59a, 61.
3. Objeto *o1*
   Definido : 5;
   Usado : 7, 8, 59, 61, 59a, 61a.
4. Objeto *out1*
   Definido : 9;
   Usado : 11.
5. Objeto *out2*
   Definido : 10;
   Usado : 12.

O próximo passo do critério de teste, é criar o DFO para este programa. Sobre as informações das definições e usos encontrados e o fluxo de controle de cada objeto, o DFO da Figura 4.16 foi construído e os pares *du* também foram identificados:

- *b1*: (3, 7), (7, 8), (8, 9), (8, 10), (8, 59), (8, 61a), (59, 61a), (61a, 59);
- *b2*:(4, 9), (4, 10), (4, 59a), (4, 61), (59a, 61), (61, 59a);
- *o1*: (5, 7), (5, 8), (5, 59), (5, 61), (5, 59a), (5, 61a);
- *out1*: (9, 11);
- *out2*: (10, 12).

Com relação a este programa, os contra-exemplos de menor caminho e o aleatório gerados pela *Bandera* são iguais. Logo a seguir é descrita apenas uma parte deste contra-exemplo, pois o caminho gerado é muito extenso. O contra-exemplo completo encontra-se no Apêndice A.

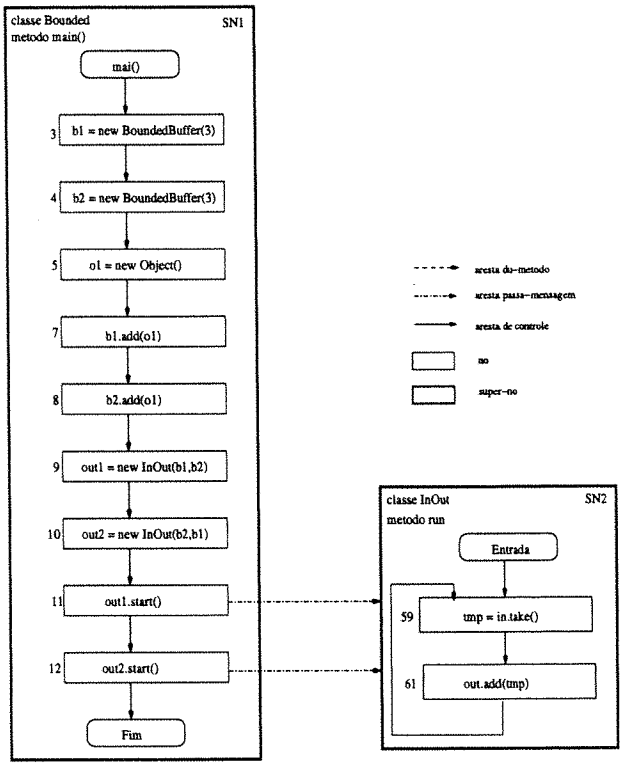


Figura 4.16: DFO do programa *Bounded.java*

```
Bounded     BoundedBuffer      InOut
3 ----------> 21                 |
|             a                  |
| <---------- 24                 |
4 ----------> 21                 |
|             a                  |
5 <---------- 24                 |
7 ----------> 28                 |
|             a                  |
| <---------- 32                 |
8 ----------> 28                 |
|             a                  |
| <---------- 32                 |
9 ----------- | ------------> 53
| <---------- | ------------- 54
10 ---------- | ------------> 53a
| <---------- | ------------- 54a
11 ---------- | ------------> 59
```

Ao analisar o contra-exemplo gerado pela *Bandera*, notamos que dos 22 pares *du* identificados 18 foram exercitados pelo menos uma vez:

- *b1*: (3, 7), (7, 8), (8, 9), (8, 10), (8, 59), (59, 61a);
- *b2*:(4, 9), (4, 10), (4, 61), (61, 59a);
- *o1*: (5, 7), (5, 8), (5, 59), (5, 61), (5, 59a), (5, 61a);
- *out1*: (9, 11);
- *out2*: (10, 12).

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
s i n c	*b1*	*take*(s) *add*(s)	6	27,27%	4	66,67%	18,18%		Ocorre na est. *while*, porque os objetos *b1* e *b2* não se despertam quando estão adicionando elementos no *buffer*	Semáforo
	b2	*take*(s) *add*(s)	4	18,18%	2	50%	9,09%			
	o1	-	4	18,18%	4	100%	18,18%	45,45%		
n s i n c	*b1*	-	2	9,09 %	2	100%	9,09 %	36,36%		
	b2	-	2	9,09%	2	100%	9,09%			
	o1	-	2	9,09%	2	100%	9,09%			
	out1	*run*	1	4,54%	1	100%	4,54%			
	out2	*run*	1	4,54%	1	100%				
			22		18			81,81%		

Tabela 4.8: Cobertura do programa *Bounded.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

Para este programa, a porcentagem de requisitos exercitados foi de 81,81%, como pode ser visto na Tabela 4.8. Com base neste dado, podemos dizer que é um bom resultado ao compararmos com a porcentagem média de 50%. Observando os pares *du* dos objetos não sincronizados, também percebemos que todos foram exercitados, coluna CT. Estes pares correspondem a 36,36% do total de pares *du* identificados. Por outro lado, analisando os pares dos objetos sincronizados, notamos que 45,45% deste pares foram exercitados. Comparando este dado com os 81,81% de cobertura total dos requisitos, podemos afirmar que os pares exercitados dos objetos sincronizados também são responsáveis por este resultado satisfatório.

4.2.9 Programa *PipeInteger.java*

O programa *PipeInteger.java* (Figura 4.17) é a implementação de um *pipeline* simples. Este programa consiste de quatro conectores (Classe *Connector*) que possuem ape-

```
 1: class Heap {
 2:     static Connector c1, c2, c3, c4;
 3: }
 4:
 5: class PipeInteger {
 6:     static public void main (String[] args) {
 7:         Heap.c1 = new Connector();
 8:         Heap.c2 = new Connector();
 9:         Stage1 s1 =  new Stage1();
10:         s1.start();
11:         Heap.c3 = new Connector();
12:         Stage2 s2 = new Stage2();
13:         s2.start();
14:         Heap.c4 = new Connector();
15:         Stage3 s3 = new Stage3();
16:         s3.start();
17:         Listener lis = new Listener();
18:         lis.start();
19:
20:         for (int i = 1; i < 5; i++)
21:             Heap.c1.add(i);
22:         Heap.c1.stop();
23:     }
24: }
25:
26: final class Connector {
27:     int queue = -1;
28:     public final synchronized int take() {
29:         int value;
30:         while (queue < 0)
31:             try {
32:                 wait();
33:             } catch (InterruptedException ex) {}
34:         value = queue;
35:         queue = -1;
36:         return value;
37:     }
38:
39:     public final synchronized void add(int o) {
40:         queue = o;
41:         notifyAll();
42:     }
43:
44:     public final synchronized void stop() {
45:         queue = 0;
46:         notifyAll();
47:     }
48: }
49:
```

```
50: final class Stage1 extends Thread {
51:     public void run() {
52:         System.out.println("Stage1 startup");
53:         int tmp = -1;
54:         while (tmp != 0) {
55:             if ((tmp = Heap.c1.take()) != 0)
56:                 Heap.c2.add(tmp + 1);
57:         }
58:         Heap.c2.stop();
59:         System.out.println("Stage1 shutdown");
60:     }
61: }
62:
63: final class Stage2 extends Thread {
64:     public void run() {
65:         System.out.println("Stage2 startup");
66:         int tmp = -1;
67:         while (tmp != 0) {
68:             if ((tmp = Heap.c2.take()) != 0)
69:                 Heap.c3.add(tmp + 1);
70:         }
71:         Heap.c3.stop();
72:         System.out.println("Stage2 shutdown");
73:     }
74: }
75:
76: final class Stage3 extends Thread {
77:     public void run() {
78:         System.out.println("Stage3 startup");
79:         int tmp = -1;
80:         while (tmp != 0) {
81:             if ((tmp = Heap.c3.take()) != 0)
82:                 Heap.c4.add(tmp + 1);
83:         }
84:         Heap.c4.stop();
85:         System.out.println("Stage3 shutdown");
86:     }
87: }
88:
89: final class Listener extends Thread {
90:     public void run() {
91:         System.out.println("Listener startup");
92:         int tmp = 0;
93:         while (tmp >= -1) {
94:             if ((tmp = Heap.c4.take()) != 0)
95:                 System.out.println("output is " + tmp);
96:         }
97:         System.out.println("Listener shutdown");
98:     }
99: }
```

Figura 4.17: Programa *PipeInteger.java*

nas um atributo (*queue*) do tipo inteiro. O programa começa com um valor positivo atribuído ao atributo do primeiro conector. Ao executar a primeira *thread* (*Stage1*), esta recebe o valor do atributo do primeiro conector e passa para o segundo conector incrementado do valor um. As outras *threads* fazem a mesma coisa com seus conectores correspondentes. A *Thread Listener* recebe o valor do atributo do último conector e o imprime como resultado.

O propósito do uso de métodos sincronizados são utilizados para que o *pipeline* tenha uma execução direta. Mas na classe *Listener*, a condição da estrutura *while* não é violada. Com isto, o último conector entra em estado de espera e numa será

despertado, pois todos os outros conectores já finalizaram sua execução. Para iniciar a análise deste programa, são identificadas as definições e usos de cada objeto:

1. Objeto *Heap.c1*

   Definido : 7, 21, 22, 55;
   Usado : 55.
2. Objeto *Heap.c2*

   Definido : 8, 56, 58, 68;
   Usado : 68.
3. Objeto *Heap.c3*

   Definido : 11, 69, 71, 81;
   Usado : 81.
4. Objeto *Heap.c4*

   Definido : 14, 82, 84, 94;
   Usado : 94.
5. Objeto *s1*

   Definido : 9;
   Usado : 10.
6. Objeto *s2*

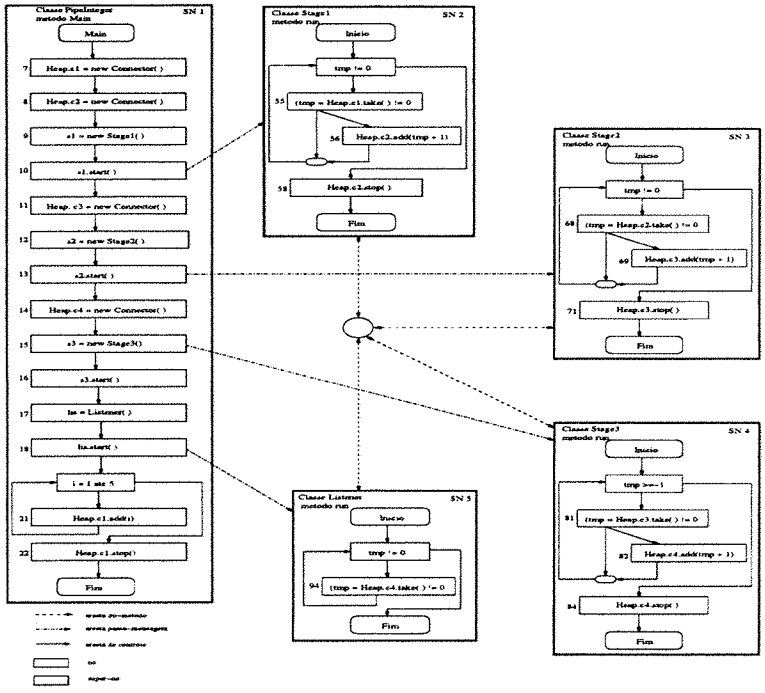


Figura 4.18: DFO do programa *PipeInteger.java*

Definido : 12;
Usado : 13.

7. Objeto *s3*

   Definido : 15 ;
   Usado : 16.

8. Objeto *Heap.c4*

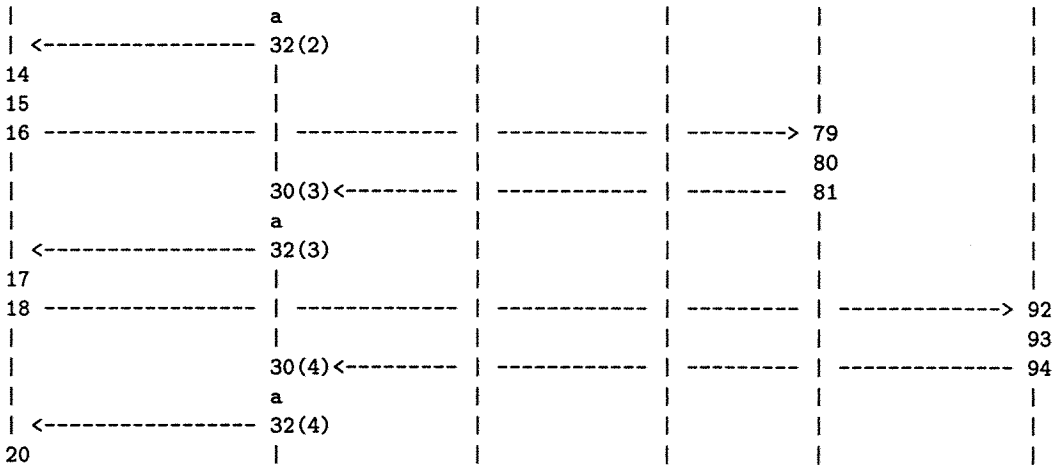
   Definido : 17;
   Usado : 18.

Analisando o fluxo de controle dos objetos e as definições e usos encontradas, podemos construir o DFO (Figura 4.18) e identificar os pares *du* passíveis de execução:

- *Heap.c1*: (7, 55), (21, 55), (22, 55);
- *Heap.c2*: (8, 68), (56, 68), (58, 68);
- *Heap.c3*: (11, 81), (69, 81), (71, 81);
- *Heap.c4*: (14, 94), (82, 94), (84, 94);
- *s1*:(9, 10);
- *s2*:(12, 13);
- *s3*:(15, 16);
- *lis*:(17, 18).

Sobre as informações dos requisitos que queremos testar, podemos usar a *Bandera* para verificar a ausência de *deadlock*. Ao fazer a verificação neste programa, a ferramenta gerou dois contra-exemplos. A descrição completa dos contra-exemplos aleatório e de menor caminho estão no Apêndice A. Logo a seguir está descrito uma parte do contra-exemplo de menor caminho:

```
                                Contra-exemplo de Menor Caminho
PipeInteger       Connector        Stage1          Stage2      Stage3          Listener
7                    |               |               |           |               |
8                    |               |               |           |               |
9                    |               |               |           |               |
10 ----------------- | ------------> 53              |           |               |
|                    |               54              |           |               |
|                    30(1)<--------- 55              |           |               |
|                    a               |               |           |               |
| <----------------- 32(1)           |               |           |               |
11                   |               |               |           |               |
12                   |               |               |           |               |
13 ----------------- | ------------- | -----------> 66           |               |
|                    |               |              67           |               |
|                    30(2)<--------- | ------------ 68           |               |
```



Ao analisar os requisitos e os contra-exemplos gerados, percebemos que todos os 16 pares *du* identificados foram exercitados. A Tabela 4.9 nos mostra claramente este resultado. É interessante observamos, que 75% dos pares *du* identificados, são de objetos sincronizados.

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
s	*Heap.c1*	*add*(s)	3	18,75%	3	100%	18,75%		Ocorre na est.	
i	*Heap.c2*	*take*(s)	3	18,75%	3	100%	18,75%	75%	*while* da classe	
n	*Heap.c3*	*stop*(s)	3	18,75%	3	100%	18,75%		*Listener*, a	
c	*Heap.c4*	-	3	18,75%	3	100%	18,75%		condição sempre é	Semáforo
n	*s1*	*run*	1	5%	1	100%	5%		satisfeita e isto	
s	*s2*	*run*	1	5%	1	100%	5%		faz com que o último	
i	*s3*	*run*	1	5%	1	100%	5%	25%	conector entre em	
n	*lis*	*run*	1	5%	1	100%	5%		estado de espera e	
c									nunca será despertado	
			16		16			100%		

Tabela 4.9: Cobertura do programa *PipeInteger.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

4.2.10 Programa *StrictClockS.java*

O programa *StricClock.java* (Figura 4.19) simula o alarme de um relógio. Neste exemplo a *thread clock* chama o método sincronizado *tick()* do monitor para avançar a hora do relógio e acordar os clientes que estão esperando. A *thread Client* chama o método

```
1  public class StrictClockS
2  {
3      public static void main(String[] args)
4      {
5          int numThreads = 2;
6          int maxTime = 5;
7          Monitor m  = new Monitor(maxTime);
8          Clock   cl = new Clock("Clock", m, maxTime);
9          cl.start();
10         Client1 c1 = new Client1("Client", m, 1);
11         Client2 c2 = new Client2("Client", m, 2);
12         c1.start();
13         c2.start();
14     }
15 }
16
17 class Clock extends Thread{
18     private String name;
19     private Monitor monitor;
20     private int max;
21     public Clock(String n, Monitor m, int maxTime){
22       name = n;
23       monitor = m;
24       max = maxTime;
25     }
26     public void run(){
27         System.out.println("I am thread " + name);
28         while(monitor.getTime() < max){
29            monitor.tick();
30            try{sleep(10);}
31            catch(InterruptedException e){System.out.println(e);}
32         }
33     }
34 }
35
36 class Client1 extends Thread{
37     private String name;
38     private Monitor monitor;
39     private int id;
40
41     public Client1(String n, Monitor m, int i) {
42         name = n;
43         monitor = m;
44         id = i;
45     }
46
47     public void run(){
48         int maxSleepTime = 5;
49         int interval;
50         System.out.println("I am thread " + name + " number " + id);
51         interval = 3;
52         monitor.wakeme(id, interval);
53         System.out.println("Client1: Awaken");
54     }
55 }
56 class Client2 extends Thread{
57     private String name;
58     private Monitor monitor;
59     private int id;
60
61     public Client2(String n, Monitor m, int i) {
62         name = n;
63         monitor = m;
64         id = i;
65     }
66
67     public void run() {
```

(a) Página 1

Figura 4.19: Programa *StrictClockS.java*

```
68          int maxSleepTime = 5;
69          int interval;
70          System.out.println("I am thread " + name + " number " + id);
71          interval = 2;
72          monitor.wakeme(id, interval);
73          System.out.println("Client2: Awaken");
74      }
75  }
76
77  class Monitor{
78      private int now;            // current time
79      private MyLinkedList waitList;
80      private int max;            // max time
81      private MyObject countMon; // for addtional synch
82      private int count;          // for additional synck
83
84      Monitor(int maxTime){
85          now = 0;
86          waitList = new MyLinkedList();
87          countMon = new MyObject(0); // object used for additional synch
88                                      // since nobody will use 0 for interval
89          count = 0;                  // var used for additional synch
90          max = maxTime;
91      }
92
93      synchronized void tick(){
94          synchronized(countMon){
95              if(count!=0){
96                  try{countMon.wait();}catch (InterruptedException e){}
97              }
98              now++;
99              if(!waitList.isEmpty()){
100                 MyObject first = waitList.firstElement();
101                 if(first.time() == now){
102                     MyObject wakeup = first;
103                     waitList.removeElementAt(0);
104                     synchronized(wakeup){
105                         wakeup.notify();
106                     }
107                 }
108             }
109         }
110     }
111
112     void wakeme(int id, int interval){
113         int waketime;
114         MyObject lock;
115
116         synchronized (this){
117             waketime = now + interval;
118             System.out.println("Monitor: Client"+id+" calling wakeme()");
119             if(waketime > max){
120                 System.out.println("Want to wake up past max");
121                 return;
122             }
123             System.out.println("Client "+id+"Want to sleep "+ interval+" want to wake up at "
    +waketime);
124             lock = waitList.getLock(waketime);
125
126             synchronized(countMon){
127                 count++;
128             }
129         }
130
131         synchronized(lock){
132             synchronized(countMon){
133                 count--;
```

(b) Página 2

Figura 4.19: Programa *StrictClockS.java* (Continuação...)

```
134                 if(count==0)countMon.notifyAll();
135             }
136             try{
137                 lock.wait();
138             }catch(InterruptedException e){}
139             System.out.println("Client awaken, time is "+ now);
140         }
141     }
142
143     int getTime(){
144         return now;
145     }
146 }
147
148 class MyObject extends Object{
149     public int time;
150     MyObject(int n){
151         time = n;
152     }
153
154     public int time(){
155         return time;
156     }
157 }
158
159 class MyLinkedList{
160     private MyObject[] list = new MyObject[2];
161     private int capacity =2;
162     private int size = 0;
163
164     public synchronized boolean isEmpty(){
165         return size==0;
166     }
167
168     public synchronized boolean isFull(){
169         return size == capacity;
170     }
171
172     public synchronized MyObject elementAt(int index){
173         return list[index];
174     }
175
176     public synchronized void insertElementAt(MyObject obj, int index){
177         if(!this.isFull()){
178             for(int i=size-1; i>=index; i--){
179                 list[i+1] = list[i];
180             }
181             list[index] = obj;
182             size++;
183         }
184     }
185
186     public synchronized void addElement(MyObject obj){
187         if(!this.isFull()){
188             list[size] = obj;
189             size++;
190         }
191     }
192
193     public synchronized MyObject firstElement(){
194         return list[0];
195     }
196
197     public synchronized void removeElementAt(int index){
198         if(!this.isEmpty()){
199             for(int i=index; i<size-1; i++){
200                 list[i] = list[i+1];
```

(c) Página 3

Figura 4.19: Programa *StrictClockS.java* (Continuação...)

```
201                 }
202                 list[size-1] = null;
203                 size--;
204             }
205         }
206
207         public synchronized MyObject getLock(int n){
208             MyObject temp;
209             int key;
210             MyObject help;
211             int i = 0;
212             for ( i=0; i<size; i++){
213                 help=list[i];
214                 key = help.time();
215                 if (n==key){
216                     temp = list[i];
217                     return temp;
218                 }
219             }
220             MyObject lock = new MyObject(n);
221             for (i=0; i<size; i++){
222                 help = list[i];
223                 key = help.time();
224                 if( n  < key ){
225                     this.insertElementAt(lock, i);
226                     return lock;
227                 }
228             }
229             this.addElement(lock); //append if greater than all others
230             return lock;
231         }
232     }
233
```

(d) Página 4

Figura 4.19: Programa *StrictClockS.java* (Fim)

wakeme() que calcula a hora de cada um acordar. Esta parte está sincronizada com o monitor, o qual garante que não haverá nenhuma interferência sobre os cálculos de outros clientes. O *deadlock* acontece quando a *thread clock* não desperta todos os clientes.

Segundo o critério de teste, é necessário que identifiquemos todas as definições e usos de cada objeto:

1. Objeto *m*

   Definido :7, 29, 52, 52a;
   Usado : 8, 10, 11, 28, 29, 52, 52a.

2. Objeto *waitlist*

   Definido : 66, 83, 104;
   Usado : 79, 80, 81, 82, 83, 104.

3. Objeto *cl*

   Definido : 8;
   Usado : 9.

4. Objeto *c1*

   Definido : 10;
   Usado : 12.

5. Objeto *c2*

   Definido : 11;
   Usado : 13.

Identificar os requisitos que precisam ser exercitados é o objetivo principal da atividade de teste. Mas antes disto, precisamos construir o DFO (Figura 4.20) e para realizar tal tarefa precisaremos das definições e usos identificados anteriormente. Logo a seguir estão os pares *du* identificados:

- *m*: (7, 8), (7, 10), (7, 11), (7, 28), (7, 29), (7, 52), (7, 52a), (29, 52), (52, 28), (52, 29), (29, 52a), (52a, 28), (52a, 29);
- *waitlist*: (66, 79), (66, 80), (66, 81), (66, 82), (66, 83), (66, 104), (83, 104), (104, 79), (104, 80), (104, 81), (104, 83), (104, 82);
- *cl*: (8, 9);
- *c1*: (10, 12);
- *c2*: (11, 13) .

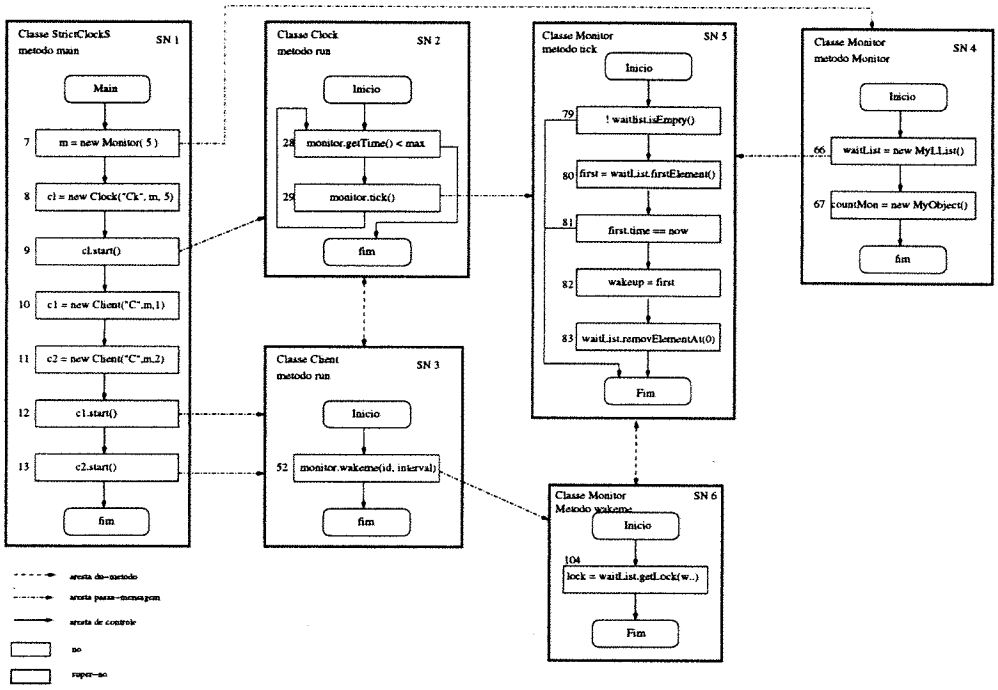
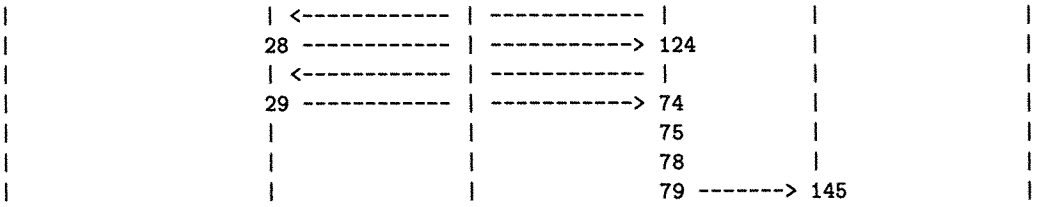


Figura 4.20: DFO do Programa *StrictClockS.java*

Ao verificar a propriedade, a *Bandera* gerou apenas um contra-exemplo, ou seja, o contra-exemplo de menor caminho é o mesmo contra-exemplo aleatório. Logo a seguir está descrita uma parte deste contra-exemplo gerado, o contra-exemplo completo encontra-se descrito no Apêndice A:

```
StrictClockS      Clock          Client          Monitor    MyLinkedList     MyObject
7 ----------------- | ------------ | ----------> 65          |                |
|                   |              |               a         |                |
| <---------------- | ------------ | ------------ 70         |                |
8 ----------------> 42             |               |         |                |
|                   43             |               |         |                |
| <---------------- 44             |               |         |                |
9 ----------------> |              |               |         |                |
|                   27             |               |         |                |
|                   28 ------------ | ----------> 124         |                |
|                   | <------------ | ------------ |          |                |
|                   29 ------------ | ----------> 74          |                |
|                   |              |               75        |                |
|                   |              |               78        |                |
|                   |              |               79 -------> 145             |
|                   |              |               | <-------- |             |
```



Analisando o caminho percorrido pelo contra-exemplo, percebemos que dos 28 pares *du* identificados, 21 foram exercitados:

- *m*: (7, 8), (7, 10), (7, 11), (7, 28), (7, 29), (29, 52), (52, 28), (29, 52a), (52a, 28),(52a, 29);
- *waitlist*: (66, 79), (66, 104), (83, 104), (104, 79) , (104, 80), (104, 81), (104, 83), (104, 82);
- *c1*: (8, 9);
- *c1*: (10, 12);
- *c2*: (11, 13) .

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
sinc	*waitlist*	*isEmpty*(s) *firstElement*(s) *removeElementAt(s)* *getlock*(s)	12	42,86%	8	66,37%	28,57%	53,57%	Ocorre na est. *while*, quando o *thread clock* não disperta todos os clientes	Monitores aninhados
	m	*getTime*(s) *tick*(s) *wakeme*	10	35,71%	7	70%	25%			
nsinc	*m*	-	3	10,71%	3	100%	10,71%	21,43%		
	c1	run	1	3,57%	1	100%	3,57%			
	c1	run	1	3,57%	1	100 %	3,57%			
	c2	run	1	3,57%	1	100%	3,57%			
			28	21				75%		

Tabela 4.10: Cobertura do programa *StrictClockS.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

Se observarmos a Tabela 4.10, percebemos que 75% dos requisitos foram identificados. Observando a coluna NP da Tabela, percebemos que 22 dos pares *du* identificados são de objetos sincronizados. Já a coluna NPE retrata que destes 22 pares, 15 foram exercitados. Em termos de porcentagem, isto corresponde a 53,57% sobre o total de

pares exercitados (coluna CT. Como o percentual de cobertura foi de 75%, podemos dizer que os pares sincronizados tem uma contribuição significativa para a cobertura dos requisitos. Por outro lado, também é relevante ressaltar que todos os pares *du* de objetos não sincronizados foram exercitados.

4.3 Conclusão

Embora os programas concorrentes Java apresentados neste trabalho sejam simples e em pequena quantidade, nós conseguimos, através deles, apresentar os principais mecanismos de sincronização e também os problemas clássicos de *deadlock* que estes mecanismos podem causar, caso não sejam utilizados adequadamente.

Ao utilizar a ferramenta *Bandera* para verificar a ausência de *deadlock*, percebemos que de modo geral, os contra-exemplos gerados são de grande ajuda na identificação dos requisitos exercitados. Por outro lado, se conseguíssemos saber quais caminhos foram percorridos antes que o *Bandera* gerasse os contra-exemplos, o auxílio a atividade de teste seria mais significativa. Pois, com isto, o número de pares *du* exercitados aumentaria.

Além disso, percebemos também que há uma discrepância na cobertura do critério de teste entre alguns programas, mas isto se deve ao fato de cada programa possuir uma característica particular.

Com relação à parte seqüencial dos programas, os pares *du* só não são exercitadas por completo porque há *threads* invocadas que não executam todos as instruções que deveriam. Isto acontece, principalmente em programas que implementam monitores aninhados.

Nos programas que implementam monitores aninhados, a quantidade de pares *du* exercitados de objetos sincronizados é menor comparado aos programas que não implementam tal estrutura. Isto acontece porque na maioria dos casos, o programa não chega a executar a segunda *thread*, pois a primeira *thread* tem o monitor mais externo bloqueado.

Por outro lado, também foi observado que nos programas que implementam a idéia

Programas	todos os pares *du*		Porcentagem de cobertura	Mecanismo de Bloqueio
	identificados	exercitados		
Dead.java	28	7	25%	Monit. Aninhados
Phil33.java	24	19	79,17%	Semáforo
START.java	20	11	55%	Monitor
Bounded.java	22	18	81,81%	Semáforo
PipeInteger.java	16	16	100%	Semáforo
NestedMonitor.java	16	9	56,25%	Monit. Aninhados
MonitorDead.java	13	6	46,15%	Monit. Aninhados
PCSimple.java	10	7	70%	Semáforo
MainPC.java	8	8	100%	Monit. Aninhados
StrictClockS.java	28	21	75%	Monit. Aninhados

Tabela 4.11: Análise dos requisitos(Programas com *deadlock*)

de semáforo, veja a Tabela 4.11, a cobertura dos pares *du* varia de 70% a 100%. Já para programas que implementam monitores aninhados, o nível de cobertura dos pares estão variando de 25% a 100%. Comparando os programas que implementam tais mecanismos, podemos observar que os percentuais dos programas que implementam Semáforos tem uma discrepância menor entre si, enquanto os que implementam monitores aninhados as porcentagens estão em um limite de variação maior.

Uma informação bastante significativa, que também pode ser vista na Tabela 4.11, é dos 10 programas analisados, 8 tiveram uma porcentagem de cobertura acima de 50%. Se pegarmos a porcentagem de cobertura de cada programa e calcularmos a média, percebemos que a média de cobertura para estes programas é de 65,59%. Isso não reflete uma medida para todos os programas, ou ainda para programas com essas caracteristicas, mas serve como um indício de que podemos diminuir significativamente os casos de teste que podem ser atendidos com o auxílio da verificação de propriedades.

Embora, até o presente momento, a análise tenha sido feita apenas sobre os programas com *deadlock*, podemos perceber que de modo geral o uso da ferramenta *Bandera* para auxiliar a atividade foi satisfatória. Podemos observar isto nos dados anteriormente discutidos, onde podemos afirmar que 80% destes programas tiveram um nível de cobertura acima de 50%. Com isto, também, podemos afirmar que a quantidade de requisitos que ainda precisam ser exercitados teve uma redução considerável.

Capítulo 5

Análise de Programas Java sem *Deadlock*

No Capítulo anterior, analisamos um conjunto de programas que violavam a propriedade de *deadlock*. Neste Capítulo, foi analisado este mesmo conjunto de programas retirando o *deadlock* de cada um. Com o intuito de facilitar o entendimento do trabalho e diferenciar dos programas analisados no Capítulo anterior, os programas sem *deadlock* foram renomeados e acrescentados "_s" ao nome original, por exemplo nome do programa *Dead.java* passa para *Dead_s.java*.

Para esta análise também será utilizada a ferramenta *Bandera* para auxiliar a atividade de teste. A diferença é que, neste caso, a ferramenta não gera um contra-exemplo e para auxiliar o teste são consideradas as informações fornecidas pelo verificador de modelo que a *Bandera* utiliza para fazer a verificação (neste caso, o verificador é o Spin). Nas próximas Seções são descritos como foi retirado o *deadlock* destes programas e como foi feita a análise destes pares.

5.1 Análise dos Programas

Com a ausência de *deadlock*, não há geração de contra-exemplos e, conseqüentemente, a análise será feita sobre os estados percorridos do código em *PROMELA* pelo verificador de modelo quando a *Bandera* o utiliza. O Spin continua sendo o verificador de modelo

que faz a verificação da propriedade do código Java representado em *PROMELA*. O código em *PROMELA* foi conseguido através da tradução da *Bandera*. Para esta tradução, foram utilizadas as técnicas de fatiamento e abstrações de dados para gerar um modelo mais compacto em *PROMELA*. Estas técnicas são utilizadas de forma que as características de cada programa sejam mantidas tanto para a verificação da propriedade quanto para a análise do critério de teste.

A análise deste programa é feita sobre os resultados gerados pelo Spin, o código em *PROMELA* traduzido e o código do programa Java. Quando uma propriedade não é violada, o verificador de modelo Spin retorna quantos estados foram encontrados e quantas vezes estes estados foram percorridos mais de uma vez.

Além disto, o Spin também retorna todos os estados que não foram percorridos do código em *PROMELA*. Esta informação será relevante para identificar quais estados realmente foram percorridos. Esta análise foi feita de forma manual, pois a ferramenta oferece pouco suporte para este tipo de análise. Para realizar a análise foram seguidos os seguintes passos:

- 1º passo: mapear quais estados representam cada instrução do código Java;
- 2º passo: identificar quais instruções do código Java foram percorridas sobre a análise dos estados percorridos em *PROMELA*;
- 3º passo: identificar se os estados não percorridos são relevantes para o código do programa Java.

A análise feita nesta Seção é semelhante à análise feita no Capítulo 4. Inicialmente é eliminado o *deadlock* do programa. Em seguida, os requisitos a serem testados são identificados segundo o critério de teste adotado. Após esta identificação, a propriedade de *deadlock* é verificada no presente programa pela ferramenta *Bandera*. Como neste caso a ferramenta não gera um contra-exemplo, a análise deve ser feita sobre os estados da linguagem *PROMELA* percorridos pelo Spin. Este verificador de modelos nos mostra apenas os estados que não foram percorridos e com isto é possível saber quais estados foram percorridos. Com base na análise dos estados percorridos, é feita uma comparação entre o código em *PROMELA* e o programa Java para identificarmos quais pares *du* foram exercitados.

5.1.1 Programa *START_s.java*

```
 1: class Event{
 2:     int count=0;
 3:     public synchronized void wait_for_event(){
 4:         try{wait();} catch(InterruptedException e){};
 5:     }
 6:     public synchronized void signal_event(){
 7:         count = count + 1;
 8:         notifyAll();
 9:     }
10: }
11:
12: class FirstTask extends java.lang.Thread{
13:   Event event1,event2;
13a:    Flag t;
14:   int count=0;
15:   public FirstTask(Event e1, Event e2, Flag t){
16:     this.event1 = e1;  this.event2 = e2;  this.t=t;
17:   }
18:   public void run(){
19:     count = event1.count;
20:     while(true){
20a:              while (!t.get()){
21:                event2.signal_event();
21a:              }
22:       if (count == event1.count)
23:         event1.wait_for_event();
24:       count = event1.count;
24a:      t.release();
25:     }
26:   }
27: }
28:
29: class SecondTask extends java.lang.Thread{
30:   Event event1,event2;
30a:  Flag t;
31:   int count=0;
32:   public SecondTask(Event e1, Event e2, Flag t){
33:       this.event1 = e1;  this.event2 = e2;  this.t=t;
34:   }
35:   public void run(){
36:     count = event2.count;
37:     while(true){
37a:         while (! t.get()){
38:            event1.signal_event();
38a:        }
39:         if (count == event2.count)
40:           event2.wait_for_event();
41:         count = event2.count;
41a:        t.release();
42:     }
43:   }
44: }
```

```
45:
46: class START_s {
47:   public static void main(String[] args){
47a:    Flag t      =   new Flag();
48:     Event event1 =  new Event();
49:     Event event2 =  new Event();
50:     FirstTask  task1 =  new FirstTask(event1,event2,t);
51:     SecondTask task2 =  new SecondTask(event2,event1,t)
52:     task1.start();
53:     task2.start();
54:   }
55: }
56:
57: class Flag{
58:     int v = 0;
59:     public synchronized boolean get(){
60:         if (v == 0){v = 1;  return true;}
61:         return false;
62:     }
63:     public synchronized void release(){
64:         v = 0;
65:     }
66: }
```

Figura 5.1: Programa *START_s.java*

O programa da Figura 5.1 contém a chamada de dois métodos sincronizados da classe *Event*. A chamada é feita pelas Classes *FirstTask* e *SecondTask*. O objetivo deste programa é sincronizar a tarefa de dois eventos. O problema do *deadlock* acontecia quando os dois eventos entravam em estado de espera e para resolver tal problema foi criada a classe *Flag* com dois métodos também sincronizados. Estes métodos têm a função de controlar a sincronização dos eventos, ou seja, não permitir que os dois eventos entrem em estado de espera simultaneamente.

Para a análise deste programa, o objeto t não foi considerado e, conseqüêntemente, os pares *du* deste objesto também não foram considerados. Além disso, os números das linhas deste programa foram mantidos os mesmos do programa com *deadlock*.

O objetivo desta manutenção é preservar as identificações das definições e usos para auxiliar a análise geral do programa:

1. Objeto *event1*
   Definido : 48, 38 ;
   Usado : 50, 51, 19, 22, 24.
2. Objeto *event2*
   Definido : 49, 24;
   Usado : 50, 51, 36, 39, 41.
3. Objeto *task1*
   Definido : 50;
   Usado : 52.
4. Objeto *task2*
   Definido : 51;
   Usado : 53.

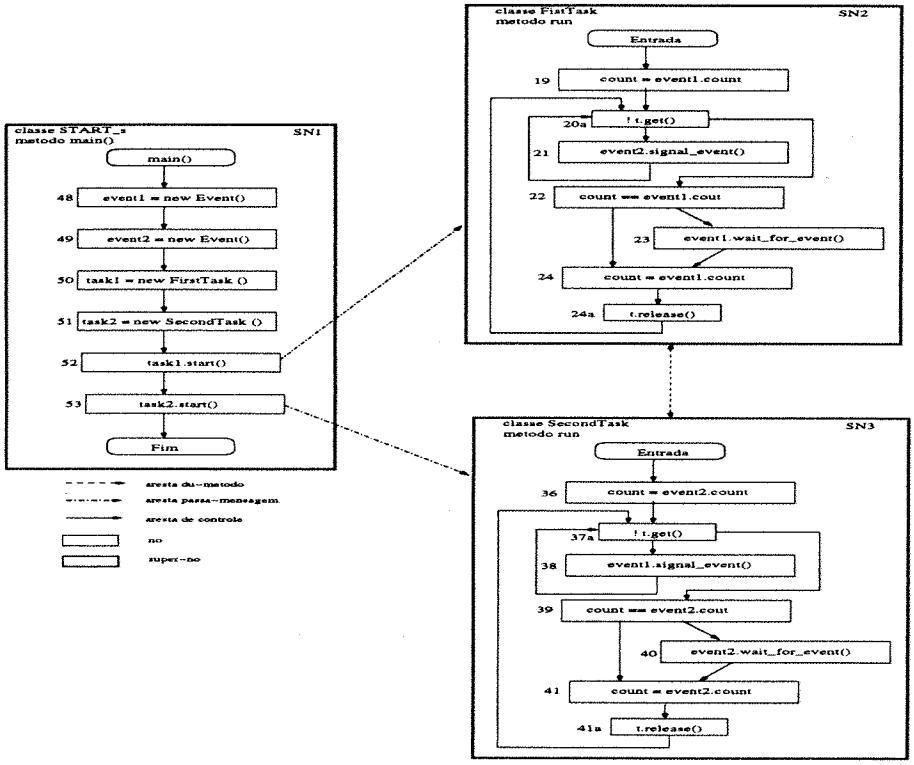


Figura 5.2: DFO do Programa *START_s.java*

Após identificar as definições e usos de cada objeto, o próximo passo do critério de teste é construir o DFO. Para construir o DFO (Figura 5.2) deste programa são considerados o fluxo de controle de cada objeto e as definições e usos encontradas. Também

é relevante lembrar que, à medida em que este diagrama é criado, são identificados os pares *du* passíveis de execução:

- *event1*: (48, 50), (48, 51), (48, 19), (48, 22), (48, 24), (48, 38), (38, 19), (38, 22), (38, 24);
- *event2*: (49, 50), (49, 51), (49, 36), (49, 39), (49, 41), (49, 21),(21, 36), (21, 39), (21, 41);
- *task1*: (50, 52);
- *task2*: (51, 53).

Ao utilizar *Bandera* para verificar a propriedade, foi aplicada a técnica de fatiamento, o que levou a uma redução significativa no tamanho do código gerado em *PROMELA*. Sem aplicarmos esta técnica foram encontrados 822 estados contra 443 estados quando a técnica foi aplicada. Já a técnica de abstração de dados não foi aplicada porque as abstrações implementadas na *Bandera* não garantem as características do programa, ou seja, o uso das abstrações geram contra-exemplos impraticáveis.

Além de diminuir o esforço da ferramenta no momento da verificação, a aplicação destas técnicas diminui o tamanho do modelo gerado em *PROMELA*, podendo minimizar o esforço da análise e do mapeamento dos estados que foram percorridos.

Como o verificador de modelo Spin mostra apenas os estados percorridos e não retrata quais seqüências de estados foram percorridos, a análise foi feita sobre uma estimativa do número mínimo de requisitos que deveriam ser exercitados para que os estados percorridos fossem exercitados pelo menos uma vez. Veja o exemplo do objeto *event1*. Nós sabemos que todas as definições e usos deste objeto foram exercitados, mas não temos certeza de qual ou quais seqüências de caminhos foram executados. Por isto, podemos descrever algumas possibilidades de caminhos que exercitem todas as instruções e, logo a seguir, os possíveis pares *du*:

- 48, 50, 51, 19, 22, 23, 24, 38 - (48, 50), (48, 51), (48, 19), (48, 22), (48, 24), (48, 38);
- 48, 50, 51, 38, 19, 22, 23, 24;- (48, 50), (48, 51), (48, 38), (38, 19), (38, 22), (38, 24);

É relevante lembrar que estes caminhos representam os possíveis caminhos que exercitam todas as definições e usos identificadas, não temos certeza se realmente são

estes os cominhos, isto representa apenas uma estimativa. Com isto podemos dizer que além do 3 pares que temos certeza que foram exercitados, no mínimo mais 3 pares deste objeto também foram exercitados. É relevante ressaltar que esta estimativa foi aplicada para todos os programas em questão. Para os outros objetos deste programa temos:

- *event1*: (48, 50), (48, 51), (48, 38) e no mínimo mais 3 pares;
- *event2*: (49, 50), (49, 51), (49, 21) e no mínimo mais 3 pares;
- *task1*: (50, 52);
- *task2*: (51, 53).

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
sinc	*event1*	*wait_for_event(s)*	4	20%	4	100%	20%	40%	Monitor
	event2	*wait_for_event(s)*	4	20%	4	100%	20%		
ñsinc	*event1*	-	5	25%	2	40%	10%	30%	
	event2	-	5	25%	2	40%	10%		
	task1	*run*	1	5%	1	100%	5%		
	task2	*run*	1	5%	1	100%	5%		
			20		14			70%	

Tabela 5.1: Cobertura do Programa *START_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

A partir desta informação, podemos afirmar que dos 20 pares *du* identificados, no mínimo 14 foram exercitados. Percentualmente isto corresponde a 70%, como mostra a Tabela 5.1. Destes 14 pares exercitados, é possível conhecermos apenas 8 destes pares *du*. Devido à incerteza de quais pares foram exercitados, é difícil sabermos quais destes pares são de objetos sincronizados e não sincronizados. Para efeito ilustrativo (Tabela 5.1), assumimos que os 3 pares (mínimo) de cada objeto, correspondem a pares de objetos sincronizados.

5.1.2 Programa *Dead_s.java*

Para este programa (Figura 5.3), o *deadlock* foi retirado invertendo a sincronização dos objetos *lock2* e *lock1* na classe *Process2*. Com isto, percebemos que os pares *du*

```
 1: public class Dead_s {
 2:     static Lock lock1;
 3:     static Lock lock2;
 4:     static int  state;
 5:
 6:     public static void main(String[] args)
 7:         lock1 = new Lock();
 8:         lock2 = new Lock();
 9:         Process1 p1 = new Process1();
10:         Process2 p2 = new Process2();
11:         p1.start();
12:         p2.start();
13:     }
14: }
15:
16: class Process1 extends Thread {
17:     public void run() {
18:         Dead.state++;
19:         synchronized (Dead.lock2) {
20:             synchronized (Dead.lock1) {
21:                 Dead.state++;
22:             }
23:         }
24:     }
25: }
26:
27: class Process2 extends Thread {
28:     public void run() {
29:         Dead.state++;
30:         synchronized (Dead.lock1) {
31:             synchronized (Dead.lock2) {
32:                 Dead.state++;
33:             }
34:         }
35:     }
36: }
37:
38: class Lock {
39: }
```

Figura 5.3: Programa *Dead_s.java*

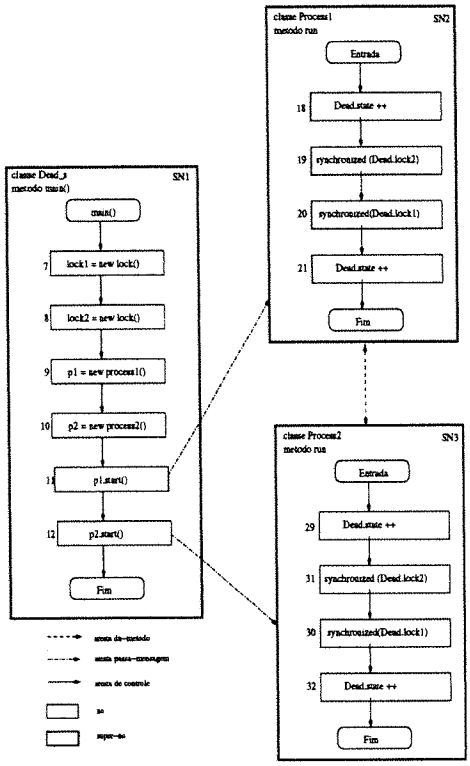


Figura 5.4: DFO do Programa *Dead_s.java*

continuam os mesmos. Os números das linhas dos objetos invertidos permaneceram os mesmos, embora a ordem tenha sido invertida. Adotamos esta estratégia para facilitar a análise do programa com e sem *deadlock*. Logo abaixo está descrito onde cada objeto foi definido e usado no programa.

1. Objeto *lock1*
   Definido : 7;
   Usado : 20, 30.
2. Objeto *lock2*
   Definido : 8;
   Usado : 19, 31.
3. Objeto *p1*
   Definido : 9;
   Usado : 11.
4. Objeto *p2*
   Definido : 10;
   Usado : 12.

5. Objeto *Dead*
   Definido : 18, 21, 29, 32;
   Usado : 18, 19, 20, 21, 29, 30, 31, 32.

O diagrama de fluxo do objeto - DFO (Figura 5.4) - foi criado baseado no fluxo de controle dos objetos e das definições e usos identificadas. No momento em que o diagrama foi construído, os pares *du* também foram identificados. Se compararmos os pares *du* com os pares deste mesmo programa com o *deadlock* podemos afirmar que são os mesmos:

- *lock1*: (7, 19), (7, 31);
- *lock2*: (8, 20), (8, 30);
- *p1*: (9, 11);
- *p2*: (10, 12);
- *Dead*: (18, 19), (18, 20), (18, 21), (18, 29), (18, 30), (18, 31), (18, 32), (21, 29), (21, 30), (21, 31), (21, 32), (29, 30), (29, 31), (29, 32), (29, 18), (29, 19), (29, 20), (29, 21), (32, 18), (32, 19), (32, 20), (32, 21).

Ao utilizar a *Bandera* para este programa, foi usada a abstração de dados *point* (Seção 3.4.2). Com a uso desta abstração houve uma redução significativa no tamanho do código em *PROMELA*, de 282 estados encontrados sem o uso de abstração para 150 estados encontrados. A técnica de fatiamento não foi empregada neste programa, pois haveria a eliminação das linhas 18, 21, 29 e 32 e isto prejudicaria a análise do critério de teste.

Após a comparação e a análise dos estados percorridos com o programa Java, podemos dizer que dos 28 pares *du* identificados no mínimo 13 foram exercitados:

- *lock1*: (7, 19), (7, 31);
- *lock2*: (8, 20), (8, 30);
- *p1*: (9, 11);
- *p2*: (10, 12);
- *Dead*: no mínimo 7 pares.

A Tabela 5.2 nos mostra que em termos de porcentagem, a cobertura foi de 46,43%. Para este programa, todos os pares de objetos não sincronizados foram exercitados. Já

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
sinc	*lock1*	-	2	7,14%	2	100%	7,14%	39,28%	Monitor aninhados
	lock2	-	2	7,14%	2	100%	7,14%		
	Dead	-	22	78,57%	7	31,82%	25%		
nsinc	*p1*	run	1	3,57%	1	100%	3,57%	7,14%	
	p2	run	1	3,57%	1	100%	3,57%		
			28		13			46,43%	

Tabela 5.2: Cobertura do Programa *Dead_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

para os objetos sincronizados, podemos dizer que no mínimo 11 pares foram exercitados e que este número pode ser maior, pois é incerto quais caminhos foram percorridos. É relevante ressaltar, que deste mínimo, 4 pares são conhecidos.

5.1.3 Programa *MonitorDead_s.java*

Para o programa *MonitorDead_s.java* descrito na (Figura 5.5), a retirada do *deadlock* foi feita na classe *Wrapper*. Para os métodos desta classe que se encontravam sincronizados (Linhas 30 e 31) foram retiradas as sincronizações. É relevante lembrar que mesmo tirando a sincronização destes métodos, o programa não perde sua característica.

Sobre este programa também é feita a análise de cobertura do critério de teste adotado e o primeiro passo para identificar os requisitos é encontrar as definições e usos de cada objeto:

1. Objeto *p*
   Definido : 3, 30, 31 ;
   Usado : 4, 30, 31.
2. Objeto *w*
   Definido : 4, 38, 46;
   Usado : 5, 6, 38.
3. Objeto *t1*
   Definido : 5, 8;
   Usado : 8.
4. Objeto *t2*

```
 1: public class NestedMonitor_s {
 2:     public static void main(String[] args) {
 3:         Pair p = new Pair();
 4:         Wrapper w = new Wrapper(p);
 5:         Thread1 t1 = new Thread1(w);
 6:         Thread2 t2 = new Thread2(w);
 7:
 8:         t1.start();
 9:         t2.start();
10:     }
11: }
12:
13: class Pair {
14:     private int hi = 0;
15:     private int lo = 0;
16:     public synchronized void inclo() {
17:         while (lo >= hi)
18:             try { wait(); } catch ( InterruptedException ex)
19:         lo++;
20:     }
21:     public synchronized void inchi() {
22:         hi++;
23:         notify();
24:     }
25: }
26:
27: class Wrapper {
28:     private Pair p;
29:     Wrapper(Pair x) {p = x;}
30:     public  void inclo(){ p.inclo();}
31:     public  void inchi(){ p.inchi();}
32: }
33:
34: class Thread1 extends Thread {
35:     private Wrapper w;
36:     Thread1(Wrapper x) {w = x;}
37:     public void run() {
38:         w.inclo();
39:     }
40: }
41:
42: class Thread2 extends Thread {
43:     private Wrapper w;
44:     Thread2(Wrapper x) {w = x;}
45:     public void run() {
46:         w.inchi();
47:     }
48: }
```

Figura 5.5: Programa *MonitorDead_s.java*

Definido : 6, 9;
Usado : 9.

Estas definições e usos encontrados junto com o fluxo de controle dos objetos são relevantes para construir o DFO (Figura 5.6) deste programa. Enquanto este diagrama é construído, também são identificados todos os pares *du* de cada objetos:

- *p*: (3, 4), (31, 30), (30, 31), (3, 30), (3, 31)
- *w*: (4, 5), (4, 6), (46, 38), (38, 46), (4, 38), (4, 46);
- *t1*: (5, 8);
- *t2*: (6, 9).

Ao executar a *Bandera* sobre este programa, tentamos aplicar a abstração de dados *sings*[1] (Seção 3.4.2) para os atributos da classe *Par*, mas o uso desta abstração gerou

[1]Abstração dos números inteiros

um contra-exemplo impraticável. Entretanto, a técnica de fatiamento foi aplicada com sucesso, embora a redução no tamanho do código em *PROMELA* tenha sido pouco significativa: de 127 estados para 113.

Com base neste código e nos estados percorridos pelo verificador de modelo Spin, percebemos que no mínimo 10 dos 13 pares *du* foram exercitados. Logo a seguir são retratados os pares que sabemos que foram exercitados e a quantidade mínima de pares de cada objeto que deveria ser exercitada:

- *p*: (3, 4), (31, 30), (30, 31) e no mínimo 1 par
- *w*: (4, 5), (4, 6) e no mínimo 2 pares;
- *t1*: (5, 8);
- *t2*: (6, 9).

Se analisarmos a Tabela 5.3, percebemos que tanto os objetos sincronizados quanto

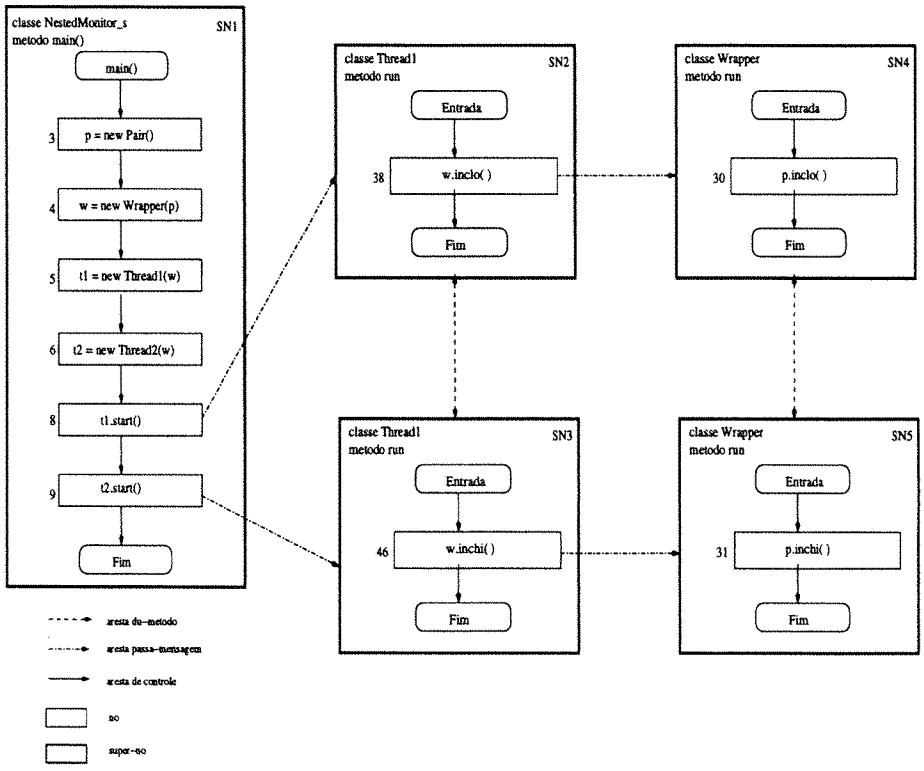


Figura 5.6: DFO do Programa *MonitorDead_s.java*

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
sinc	p	*inclo*(s) *inchi*(s)	4	30,77%	3	75%	23,08%	23,08%	Monitor
nsinc	w	*inclo* *inchi*	6	46,15%	4	66,67%	30,77%	53,85%	
	p	-	1	7,69%	1	100%	7,69%		
	t1	*run*	1	7,69%	1	100%	7,69%		
	t2	*run*	1	7,69%	1	100%	7,69%		
			13		10			76,92%	

Tabela 5.3: Cobertura do Programa *MonitorDead_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

os não sincronizados tiveram uma boa cobertura, pois todos os objetos tiveram um percentual acima de 65%, como pode ser visto na coluna CO da Tabela em questão. Devido à incerteza da quantidade de pares *du* exercitados, sabemos que no mínimo 76,92% dos pares identificados foram exercitados.

5.1.4 Programa *Phil33_s.java*

Para eliminar o *deadlock* do programa jantar entre filósofos, foi invertida a ordem dos garfos que o segundo filósofo deve pegar e soltar. Se observamos as linhas 59 e 60 da Figura 5.7, os dois filósofos pegam os garfos sempre na mesma ordem.

Antes de analisarmos os pares *du*, é necessário identificar todas as definições e usos para cada objeto para auxiliar a identificação destes pares:

1. Objeto *f1*
   Definido : 57, 45, 46, 33, 38, 33a, 38a;
   Usado : 59, 60, 45, 33, 33a.
2. Objeto *f2*
   Definido : 58, 45a, 46a, 34a, 39a, 34, 39;
   Usado : 59, 60, 45a, 34a, 39a.
3. Objeto *p1*
   Definido : 59;
   Usado : 61.
4. Objeto *p2*
   Definido : 60;

```
 1: class Fork{
 2:     private boolean is_free;
 3:
 4:     Fork(){
 5:         is_free = true;
 6:     }
 7:
 8:     public synchronized void take(){
 9:         while (!is_free)
10:             try {
11:                 wait();
12:             } catch (InterruptedException e) {return;}
13:
14:         is_free = false;
15:     }
16:
17:     public synchronized void leave(){
18:         is_free = true;
19:         notify();
20:     }
21: }
22:
23: class Philosopher extends Thread{
24:     private Fork left;
25:     private Fork right;
26:
27:     Philosopher(Fork f1, Fork f2){
28:         left = f1;
29:         right = f2;
30:     }
31:
32:     public void eat(){
33:         left.take();
34:         right.take();
35:     }
36:
37:     public void think(){
38:         left.leave();
39:         right.leave();
40:     }
41:
42:     public void run(){
43:         while (true)
44:             {
45:                 eat();
46:                 think();
47:             }
48:     }
49: }
50:
51: public class Phil33_s{
52:     public static void main(String[] args){
53:         Fork f1;
54:         Fork f2;
55:         Philosopher p1;
56:         Philosopher p2;
57:         f1 = new Fork();
58:         f2 = new Fork();
59:         p1 = new Philosopher(f1, f2);
60:         p2 = new Philosopher(f1, f2);
61:         p1.start();
62:         p2.start();
63:     }
64: }
```

Figura 5.7: Programa *Phil33_s.java*

Usado : 62.

Com base nos dados adquiridos anteriormente, podemos utilizar a informação do fluxo de controle dos objetos para construir o diagrama deste programa (Figura 5.8) e identificar os pares *du* que podem ser executados:

- *f1*: (57, 59), (57, 60), (57, 33), (57, 33a),(38, 33), (38a, 33a), (33, 33a), (33a, 33), (38a, 33), (38, 33a);
- *f2*: (58, 59), (58, 60), (58, 34), (58, 34a), (39, 34), (39a, 34a), (39, 34a), (39a, 34), (34, 34a), (34a, 34) ;
- *p1*: (59, 61), (46, 45);
- *p2*: (60, 62), (46a, 45a).

O próximo passo da análise é verificar a propriedade usando a ferramenta *Bandera*. A técnica de fatiamento foi aplicada e o modelo gerado em *PROMELA* manteve as

mesmas características e comportamento do programa Java. Além disso, o tamanho do modelo diminuiu de 1218 estados para 1018 estados. Por outro lado, a técnica de abstração de dados não foi aplicada pelo fato de não existir abstrações implementadas na ferramenta que possam ser aplicadas às variáveis e/ou atributos deste programa. Após analisarmos os estados percorridos, percebemos que os seguintes pares *du* foram exercitados:

- *f1*: (57, 59), (57, 60) e no mínimo mais 2 pares;
- *f2*: (58, 59), (58, 60) e no mínimo mais 2 pares;
- *p1*: (59, 61), (46, 45);
- *p2*: (60, 62), (46a, 45a).

Dos 24 pares *du* identificados podemos dizer que, no mínimo, metade deles foram exercitados. Desta metade conhecemos apenas 6 pares, como foi descrito anteriormente. Em termos de porcentagem, a Tabela 5.4 mostra que 50% dos pares foram exercitados,

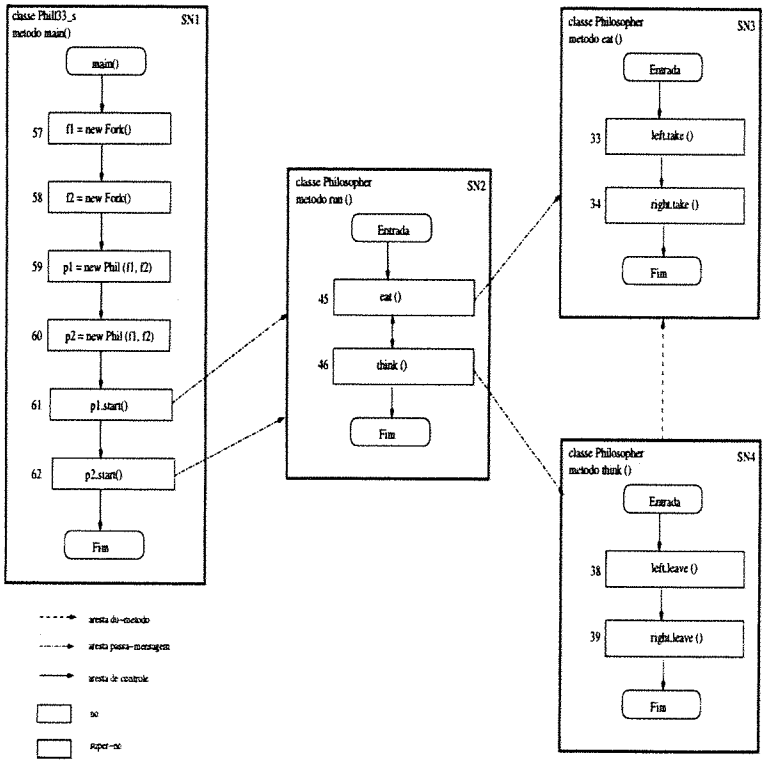


Figura 5.8: DFO do Programa *Phil33_s.java*

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
s i n c	*f1*	*take(s)* *leave(s)*	8	33,33%	2	25%	8,33%	16,67%	Semáforo
	f2	*take(s)* *leave(s)*	8	33,33%	2	25%	8,33 %		
n s i n c	*f1*	-	2	8,33%	2	100%	8,33%	33,33%	
	f2	-	2	8,33%	2	100%	8,33%		
	p1	*eat, think run*	2	8,33%	2	100%	8,33%		
	p2	*eat, think run*	2	8,33%	2	100%	8,33%		
			24		12			50%	

Tabela 5.4: Cobertura do Programa *Phil33_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

embora este número possa ser maior devido a estimativa de cobertura feita sobre os requisitos. Para este problema, podemos notar que todos os pares de objetos não sincronizados foram exercitados (coluna `CO`).

5.1.5 Programa *PCSimple_s.java*

```
1: public class PCSimple_s{
2:     public static void main(String[] args){
3:         int numIntegers = 6;
4:         Buffer b = new Buffer(2);
5:         Producer p = new Producer(numIntegers, b);
6:         Consumer c = new Consumer(numIntegers, b);
7:         p.start();
8:         c.start();
9:     }
10: }
11:
12: class Producer extends Thread
13: {
14:     private int count;
15:     private Buffer buffer;
16:
17:     public Producer(int num, Buffer b){
18:         count = num;
19:         buffer = b;
20:     }
21:
22:     public void run(){
23:         for (int i=0; i < count; i++)
24:             buffer.put(i);
25:         buffer.put(-1);
26:     }
27: }
28:
29: class Consumer extends Thread{
30:     private int count;
31:     private Buffer buffer;
32:
33:     public Consumer(int num, Buffer b){
34:         count = num+1; //for that extra -1
35:         buffer = b;
36:     }
37:
38:     public void run(){
39:      int[] received = new int[count];
40:      for(int i=0; i<count; i++){
41:             received[i] = buffer.get();
42:       if(received[i] == -1){
43:         System.out.println("Consumer.run: got last number received[i]");
44:             return;
45:           }
46:       }
47:     }
48: }
49:
```

```
50: class Buffer{
51:     protected int size;
52:     protected int[] array;
53:     protected int putPtr;
54:     protected int getPtr;
55:     protected int usedSlots;
56:
57:     public Buffer(int b){
58:       size = b;
59:       array = new int[b];
60:       putPtr = 0;
61:       getPtr = 0;
62:       usedSlots = 0;
63:     }
64:
65:     public synchronized void put(int x){
66:         while(usedSlots == size)
67:             try{
68:                 System.out.println("producer wait");
69:                 wait();
70:             }catch(InterruptedException e){};
71:
72:         array[putPtr] = x;
73:         System.out.println("Buffer.put: put "+x);
74:         putPtr = (putPtr + 1)%size;
75:         if(usedSlots == 0) notify();
76:         usedSlots++;
77:     }
78:
79:     public synchronized int get(){
80:         while(usedSlots == 0)
81:             try{
82:                 System.out.println("consumer wait");
83:                 wait();
84:             }catch(InterruptedException e){};
85:         int x = array[getPtr];
86:         System.out.println("Buffer.get: got "+x);
87:         if(x<0)
88:             return x;
89:         getPtr = (getPtr+1)%size;
90:         if(usedSlots == size) notify();
91:         usedSlots --;
92:         return x;
93:     }
94: }
```

Figura 5.9: Programa *PCSimple_s.java*

Este é um simples programa que implementa a idéia do produtor-consumidor usando um *buffer* (Figura 5.9). O problema do *deadlock* acontece porque o produtor não desperta o consumidor. Para eliminar este problema do programa, na linha 75 o produtor desperta o consumidor quando não tem nenhum elemento para produzir.

Para este programa, foram identificadas as definições e usos de cada objeto com o intuito de auxiliar a identificação dos requisitos:

1. Objeto *b*

   Definido : 4, 24, 25, 41;
   Usado : 5, 6, 24, 25, 41.
2. Objeto *p*

Definido : 5;
Usado : 7.

3. Objeto *c*

Definido : 6;
Usado : 8.

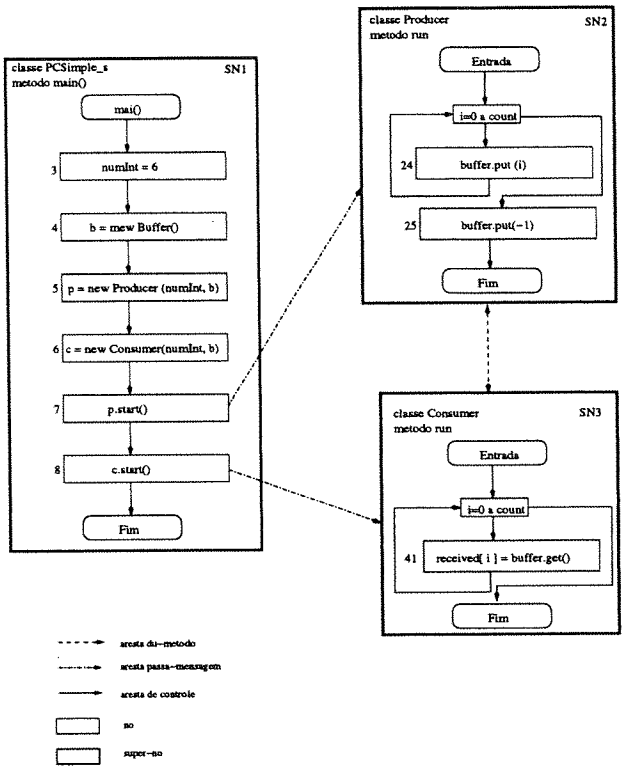


Figura 5.10: DFO Programa *PCSimple_s.java*

Através da construção do diagrama da Figura 5.10, também foram identificados os possíveis pares *du* que precisam ser exercitados. Para construir este diagrama, foram considerados o fluxo de controle do objeto e as definições e usos encontradas anteriormente. Logo a seguir são descritos os requisitos identificados:

- *b*: (4, 5), (4, 6), (4, 24),(4, 41), (24, 41), (41, 24), (41, 25), (25, 41), (24, 25);
- *p*: (5, 7);
- *c*: (6, 8).

Ao verificar a ausência de *deadlock* pela *Bandera*, não foi aplicada a técnica de fatiamento porque esta técnica remove parte do código que é relevante para o critério de

teste, embora não seja relevante para a propriedade verificada. A técnica de abstração de dados também não foi aplicada a este programa, pois não há nenhuma abstração implementada na *Bandera* que possa ser aplicada. Com isto, para a verificação e análise deste programa em específico, não podemos contar com o auxílio das técnicas de redução de modelo.

Para o código em *PROMELA* gerado foram percorridos 2221 estados. Ao compararmos estes estados com o código Java, percebemos que os seguintes pares *du* foram exercitados:

- *b*: (4, 5), (4, 6), (24, 41), (41, 24) e no mínimo mais 2;
- *p*: (5, 7);
- *c*: (6, 8).

É relevante ressaltar, que dos 8 pares *du* exercitados, nós sabemos exatamente quais são 6 destes pares. Os outros 2 dois não temos certeza de quais ou quantos são, mas sabemos a qual objetos eles estão vinculados. A Tabela 5.5 nos mostra que no mínimo 72,73% dos pares foram exercitados. Para este caso, todos os pares de objetos não sincronizados foram exercidos, o que corresponde a 36,36%. Já dos 7 pares sincronizados identificados, sabemos que no mínimo 4 foram exercitados, isto também corresponde a 36,36%.

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
sinc	*b*	*put(s)* *get(s)*	7	63,64%	4	57,14%	36,36%	36,36%	Semáforo
nsinc	*b*	-	2	18,18%	2	100%	18,18%	36,36%	
	p	*run*	1	9,09%	1	100%	9,09%		
	c	*run*	1	9,09%	1	100%	9,09%		
			11		8			72,73%	

Tabela 5.5: Cobertura do Programa *PCSimple_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

```
1: public class MainPC_s{
2:     public static void main(String args[]){
3:         My_Pool pool=new My_Pool();
4:         Prod p=new Prod(pool);
5:         Cons c=new Cons(pool,1);
6:         p.start();
7:         c.start();
8:     }
9: }
10: class Cons extends Thread{
11:     My_Pool p;
12:     int _id;
13:
14:     public Cons(My_Pool pool, int id){
15:         p=pool;
16:         _id=id;
17:     }
18:
19:     public void run(){
20:         while(true){
21:             System.out.println(_id+"Tirando algo do pool");
22:             int val=p.get();
23:             System.out.println(_id+"Tirei: "+val);
24:         }
25:     }
26: }
27: class Prod extends Thread{
28:     My_Pool p;
29:     public Prod(My_Pool pool){
30:         p=pool;
31:     }
32:
33:     public void run(){
34:         while(true){
35:            System.out.println("Estou colocando algo...");
36:            p.put(1);
37:         }
38:     }
39: }
```

```
40: class My_Guard{
41:     int slot;
42:     int status;
43:
44:     public My_Guard(){
45:         slot=0;
46:         status=0;
47:     }
48:
49:     synchronized int get(){
50:         try{
51:             while(status==0) wait();
52:             status=0;
53:         }
54:         catch(Exception e){
55:             System.out.println("Erro no get...");
56:         }
57:         return slot;
58:     }
59:
60:     synchronized void put(int i){
61:         try{
62:             slot=i;
63:             status=1;
64:             notify();
65:         }
66:         catch(Exception e){
67:             System.out.println("Erro no put");
68:         }
69:     }
70: }
71: class My_Pool{
72:     My_Guard gd;
73:
74:     public My_Pool(){
75:         gd=new My_Guard();
76:     }
77:
78:    int get(){
79:         returngd.get();
80:     }
81:
82:     void put(int i){
83:         gd.put(i);
84:     }
85: }
```

Figura 5.11: Programa *PCMain_s.java*

5.1.6 Programa *PCMain_s.java*

Para o programa da Figura 5.11, foram retiradas as sincronizações dos métodos da classe *My_Pool* (Linhas 78 e 82) para que o programa não entre em *deadlock*. Após retirar o *deadlock* podemos usar a ferramenta *Bandera* para verificar a propriedade e analisarmos o que foi coberto do critério de teste. Primeiro, é necessário identificar os requisitos que precisam ser testados e o primeiro passo é encontrar as definições e usos de cada objeto:

1. Objeto *pool*
   Definido : 3, 36, 22;
   Usado : 4, 5, 22.
2. Objeto *p*
   Definido : 4;
   Usado : 6.
3. Objeto *c*

Definido : 5;
Usado : 7.

4. Objeto *gd*

   Definido : 75, 79, 83;
   Usado : 79.

O segundo passo do critério é identificar os pares *du*, mas para isto é necessário construir o DFO (Figura 5.12) para auxiliar a identificação. Como foi feita nas Seções anteriores, para criar o diagrama é necessário ter o conhecimento do fluxo de controle dos objetos e das definições e usos destes. Para este programa temos os seguintes pares *du* passíveis de execução:

- *pool*: (3, 4), (3, 5), (3, 22),(36, 22);
- *p*: (4, 6);
- *c*: (5, 7);
- *gd*: (75, 79), (82, 79).

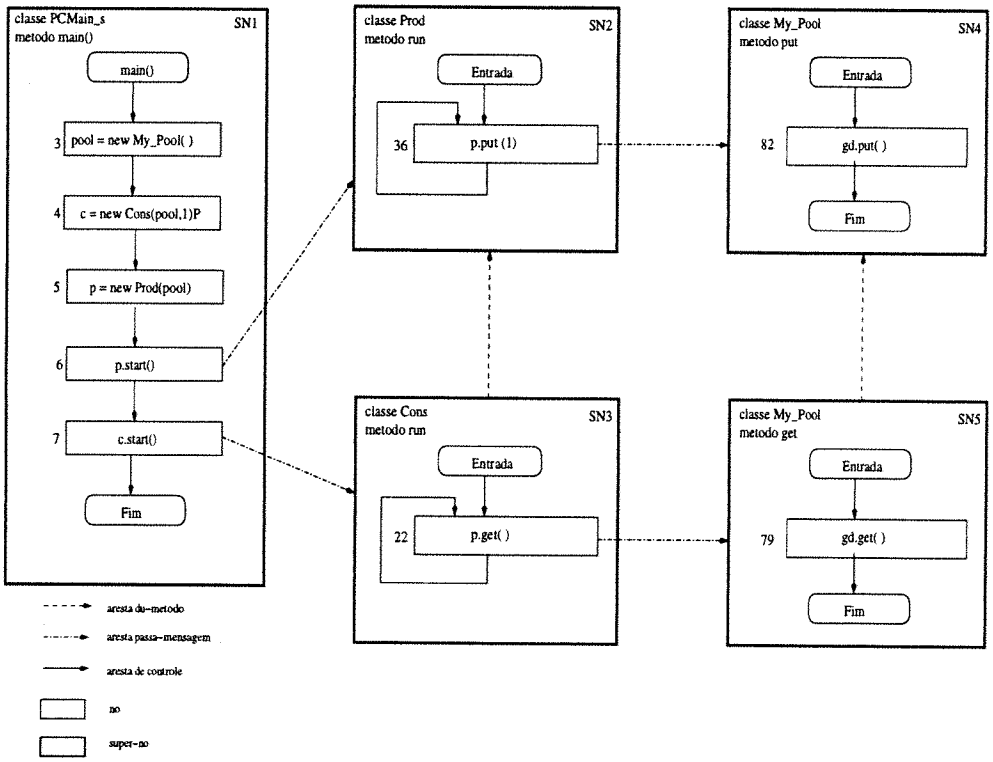


Figura 5.12: DFO do Programa *PCMain_s.java*

Ao utilizar a *Bandera* para verificar a ausência de *deadlock*, foi aplicada a técnica de abstração de dados chamada *point* (Seção 3.4.2). Antes de aplicar esta técnica, o número de estados encontrados era de 757 e após aplicar a técnica o número de estados encontrados passou a ser de 580. A técnica de fatiamento não foi aplicada porque as instruções removidas eram relevantes para a análise do critério de teste.

Comparando os estados percorridos em *PROMELA* com as instruções de cada linha do programa Java, uma estimativa é feita sobre alguns caminhos que exercitam as definições e usos de cada objeto. Através das informações obtidas nessa comparação, podemos dizer que os seguintes pares foram exercitados:

- *pool*: (3, 4), (3, 5) e no mínimo mais 1 par;
- *p*: (4, 6);
- *c*: (5, 7);
- *gd*: (75, 79), (82, 79).

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
sinc	gd	*put*(s) *get*(s)	2	25%	2	100%	25%	25%	Semáforo
nsinc	pool	*put* *get*	4	50%	3	75%	37,5%	62,5%	
	p *c*	*run* *run*	1 1	12,5% 12,5%	1 1	100% 100%	12,5% 12,5%		
			8		7			87,5%	

Tabela 5.6: Cobertura do Programa *PCMain_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

Para este programa foram identificados 8 pares e no mínimo 7 pares *du* foram exercitados, como pode ser visto na Tabela 5.6. Destes 7 pares, é possível sabermos quem são 6 pares. Através da Tabela, também podemos perceber que no caso deste programa, todos os pares sincronizados foram exercitados, isto corresponde a 25% do total.

Além disso, para os pares não sincronizados é possível fazermos apenas uma estimativa sobre o mínimo exercitado, o que pode vir a ser maior. Mesmo com a análise

feita sobre uma estimativa mínima de pares exercitados, é possível afirmarmos que o nível de cobertura deste programa é de 87,5%, o que é consideravelmente bom.

5.1.7 Programa *NestedMonitor_s.java*

```
 1: interface Buffer {
 2:     public void put(Object o)
 3:         throws InterruptedException; //poe o objeto no buffer
 4:     public Object get()
 5:         throws InterruptedException; //retira um objeto do buffer
 6: }
 7:
 8: class Semaphore {
 9:     private int value;
10:     public Semaphore (int initial) {
11:         value = initial;
12:     }
13:
14:     synchronized public void up() {
15:         ++value;
16:         notifyAll();
17:     }
18:
19:     synchronized public void down() throws InterruptedException {
20:         while (value==0) {wait();}
21:         --value;
22:     }
23: }
24:
25: /*********************SEMABUFFER*****************************/
26: class SemaBuffer implements Buffer {
27:     protected int size;
28:     Semaphore full; //conta o numero de itens
29:     Semaphore empty;//conta o numero de espaco
30:
31:     SemaBuffer(int size) {
32:         this.size = size;
33:         full = new Semaphore(0);
34:         empty= new Semaphore(size);
35:     }
36:     //public void put(Object o)
37:     public void put(Object o)
38:         throws InterruptedException {
39:          empty.down();
40:          full.up();
41:     }
42:     // public Object get()
43:     public Object get()
44:         throws InterruptedException{
45:         full.down();
46:         empty.up();
47:         return (null);
48:     }
49: }
50:
51: /************ Pricipal *************/
52: public class NestedMonitor_s  {
53:     static int SIZE = 5; /* parametro */
54:     static Buffer buf;
55:
56:     public static void main(String [] args) {
57:         buf = new SemaBuffer(SIZE);
58:         Producer p = new Producer(buf);
59:         p.start();
60:         Consumer c = new Consumer(buf);
61:         c.start();
62:     }
63: }
64:
65: /*******************PRODUCER*********************/
66: class Producer extends Thread {
67:     Buffer buf;
68:     Producer(Buffer b) {buf = b;}
69:
70:     public void run() {
71:       try {
72:         while(true)
73:             buf.put(null);
74:       } catch (InterruptedException e){}
75:     }
76: }
77:
78: /*******************CONSUMER*********************/
79: class Consumer extends Thread {
80:     Buffer buf;
81:     Consumer(Buffer b) {buf = b;}
82:
83:     public void run() {
84:         try {
85:           while(true)
86:             buf.get();
87:       } catch(InterruptedException e ){}
88:     }
89: }
90:
```

Figura 5.13: Programa *NestedMonitor_s.java*

Para o programa *NestedMonitor_s.java* descrito na Figura 5.13, o *deadlock* foi eliminado removendo a sincronização dos métodos *put* e *get* da classe *SemaBuffer* (linhas 37 e 43, respectivamente). Com isto elimina-se a presença de monitores aninhados, mas as características do programa são mantidas.

Como o código Java foi modificado quando se retirou o *deadlock*, os passos parar identificar os pares *du* são feitos novamente. Inicialmente, é necessário encontrar todas as definições e usos de cada objeto:

1. Objeto *buf*
   Definido :57, 73, 86;
   Usado : 58, 60, 73, 86.
2. Objeto *full*
   Definido : 33, 40, 45;
   Usado : 40, 45.
3. Objeto *empty*
   Definido : 34, 39, 46;
   Usado : 39, 46.
4. Objeto *p*
   Definido : 58;
   Usado : 59.
5. Objeto *c*
   Definido : 60;
   Usado : 61.

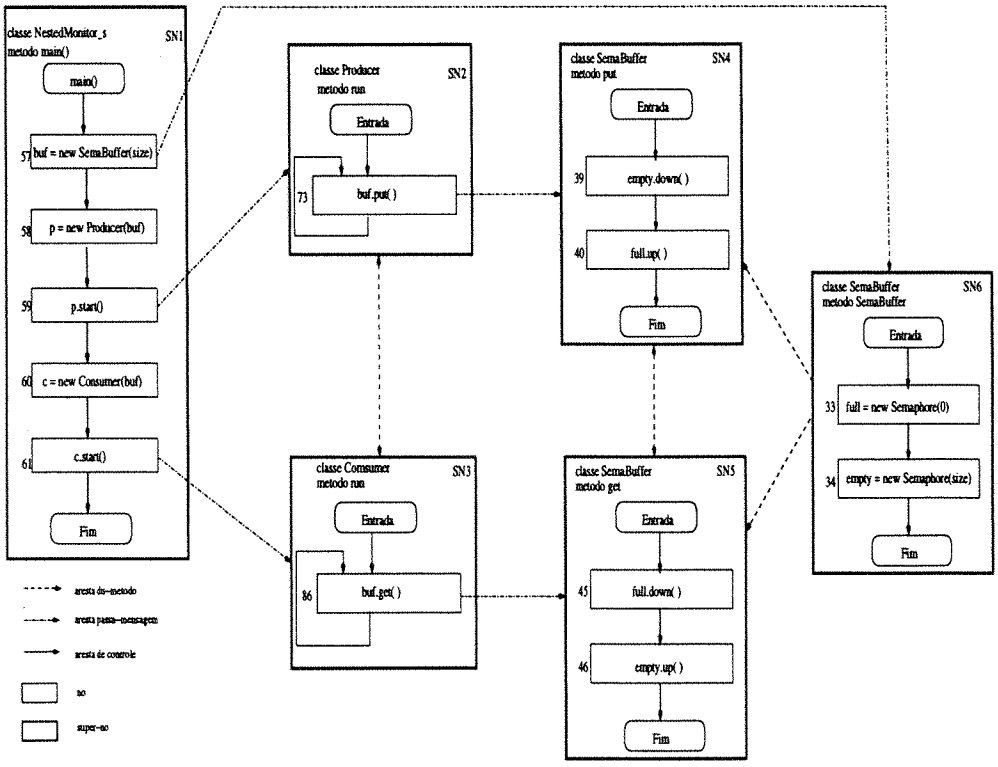


Figura 5.14: DFO do Programa *NestedMonitor_s.java*

Após identificar tais dados, estas informações e o fluxo de controle dos objetos são usados para criar o diagrama da Figura 5.14. Com isto, também são identificados os pares *du*, como mostrado a seguir:

- *buf*: (57, 58), (57, 60), (57, 73), (57, 86), (73, 86), (86, 73);
- *full*: (33, 40), (33, 45), (40, 45), (45, 40);
- *empty*: (34, 39), (34, 46), (39, 46), (46, 39);
- *p*: (58, 59);
- *c*: (60, 61).

Com base nos pares identificados, podemos utilizar a ferramenta para verificar a ausência de *deadlock*. Mas antes, também podemos aplicar as técnicas de redução de modelos para este programa. A técnica de redução baseada em abstração de dados foi utilizada com sucesso, a melhor abstração aplicada para este caso foi a *ranger04* (Veja na Seção 3.4.2). Por outro lado, a técnica de fatiamento não foi utilizada pelo fato de remover instruções do programa que são relevantes para análise dos requisitos. Para o caso específico deste programa, as técnica de redução utilizada não trouxe vantagens, pois o número de estados encontrados manteve-se constante (147 estados).

Ao fazer a verificação, podemos confirmar que o programa realmente não entra em *deadlock*. Mas infelizmente não podemos confirmar quais caminhos foram percorridos, apenas quais estados foram executados. Por causa dessa incerteza, é feita uma estimativa sobre quais e quantos pares *du* deveriam ser exercitados para que todos os 147 estados fossem exercitados. Com isto, é possível ter a seguinte informação sobre quais pares foram exercitados:

- *buf*: (57, 58), (57, 60) e no mínimo mais 2 pares;
- *full*: no mínimo 2 pares;
- *empty*: no mínimo 2 pares;
- *p*: (58, 59);
- *c*: (60, 61).

A Tabela 5.7 retrata que pelo menos 62,5% dos pares identificados foram exercitados. Este número pode ser maior, pois como dissemos anteriomente, foi feita uma estimativa sobre os possíveis pares exercitados. Observando a coluna CT da Tabela, podemos dizer que no mínimo 25% dos pares sincronizados foram exercitados. A desvantagem da análise sobre os pares sincronizados é não termos certeza de quais pares foram exercitados.

Por outro lado, sabemos que no mínimo 37,5% dos pares não sincronizados foram exercitados. Sobre estes pares, temos o conhecimento de 4 pares que foram exercitados.

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
s i n c	*full*	*up* (s) *down*(s)	4	25%	2	50%	12,5%	25%	
	empty	*up* (s) *down*(s)	4	25%	2	50%	12,5%		
n s i n c	*buf*	*put* *put*	6	37,5%	4	66,67%	25%	37,5%	
	p	*run*	1	6,25%	1	100%	6,25%		
	c	*run*	1	6,25%	1	100%	6,25%		
			16		10			62,5%	

Tabela 5.7: Cobertura do Programa *NestedMonitor_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

5.1.8 Programa *Bounded_s.java*

O programa *Bounded_s.java* (Figura 5.15) refere-se à implementação de um simples *buffer* limitado e algumas classes que realizam operações sobre este *buffer*. Para remover o *deadlock*, o produtor adiciona os elementos e libera a vez para o consumidor, veja na linha 33.

Para realizar a análise deste programa é preciso identificar os pares *du* a serem testados. Inicialmente, é necessário encontrar todas as definições e usos de cada objeto:

1. Objeto *b1*

   Definido : 3, 7, 59, 61a;
   Usado : 7, 8, 9, 10, 59, 61a.

2. Objeto *b2*

   Definido : 4, 59a, 61;
   Usado : 9, 10, 59a, 61.

3. Objeto *o1*

   Definido : 5;
   Usado : 7, 8, 59, 61, 59a, 61a.

4. Objeto *out1*

   Definido : 9;
   Usado : 11.

5. Objeto *out2*

   Definido : 10;
   Usado : 12.

```
 1: public class Bounded_s {
 2:     public static void main (String [] args) {
 3:         BoundedBuffer b1 = new BoundedBuffer(3);
 4:         BoundedBuffer b2 = new BoundedBuffer(3);
 5:         Object o1 = new Object();
 6:
 7:         b1.add(o1);
 8:         b1.add(o1);
 9:         InOut out1 = new InOut(b1,b2);
10:         InOut out2 = new InOut(b2,b1);
11:         out1.start();
12:         out2.start();
13:     }
14: }
15:
16: class BoundedBuffer {
17:     Object [] buffer;
18:     int bound, head, tail;
19:
20:     public BoundedBuffer(int b) {
21:       bound = b;
22:       buffer = new Object[bound];
23:       head = 0;
24:       tail = bound-1;
25:     }
26:
27:     public synchronized void add(Object o) {
28:         while ( tail == head )
29:             try { wait(); }
30:             catch ( InterruptedException ex) {}
31:         buffer[head] = o;
32:         head = (head+1) % bound;
33:         notify();
34:     }
35:
36:
37:      public synchronized Object take() {
38:         while ( isEmpty() )
39:             try { wait(); }
40:             catch ( InterruptedException ex) {}
41:         tail = (tail+1) % bound;
42:         notifyAll();
43:         return buffer[tail];
44:     }
45:
46:     public synchronized boolean isEmpty() {
47:         return head == ((tail+1) % bound); }
48: }
49:
50: class InOut extends Thread {
51:     BoundedBuffer in,out;
52:     public InOut(BoundedBuffer ins, BoundedBuffer outs
53:         in = ins;
54:         out = outs;
55:     }
56:     public void run() {
57:         Object tmp;
58:         while(true) {
59:             tmp = in.take();
60:             System.out.println("Take ");
61:             out.add(tmp);
62:             System.out.println("Put ");
63:         }
64:     }
65: }
```

Figura 5.15: Programa *Bounded_s.java*

Se analisarmos as informações encontradas anteriormente e o fluxo de controle dos objetos, é possível construirmos o DFO (Figura 5.16) deste programa. Enquanto o diagrama é criado, também são identificados os pares *du* passíveis de execução:

- *b1*: (3, 7), (7, 8), (8, 9), (8, 10), (8, 59), (8, 61a), (59, 61a), (61a, 59);
- *b2*:(4, 9), (4, 10), (4, 59a), (4, 61), (59a, 61), (61, 59a);
- *o1*: (5, 7), (5, 8), (5, 59), (5, 61), (5, 59a), (5, 61a);
- *out1*: (9, 11);
- *out2*: (10, 12).

O próximo passo é fazer a verificação da propriedade de *deadlock*. Antes de realizar tal tarefa, é possível aplicarmos a técnica de fatiamento de programa, sem que seja removido nenhuma instrução relevante para o critério a ser analisado. Ao aplicar

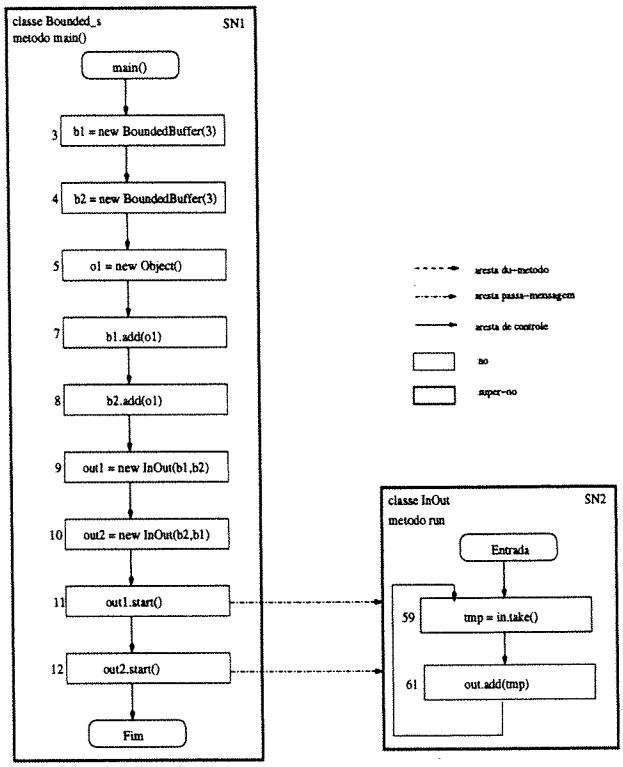


Figura 5.16: DFO do Programa *Bounded_s.java*

esta técnica houve uma redução significativa nos estados gerados, de 5027 estados encontrados em *PROMELA* reduziu-se para 3325 estados. Por outro lado, a técnica de abstração de dados não foi utilizada, pois não há uma abstração adequada implementada na *Bandera* que possa ser aplicada a este programa.

Após aplicarmos as técnicas de redução de modelo, podemos comparar os 3325 estados percorridos com cada linha de instrução do código Java para mapearmos o que foi percorrido em Java. Como foi dito nas Seções anteriores, quando a propriedade não é violada é possível saber apenas quais estados foram percorridos e não os caminhos. Através da geração de possíveis caminhos que exercitam estes estados, podemos identificar alguns pares que foram percorridos e também estimar o número mínimo de pares *du* que foram exercitados, como pode ser visto a seguir:

- *b1*: (3, 7), (7, 8), (8, 9), (8, 10), (59, 61a), (61a, 59) e no mínimo mais 1 par;

- *b2*:(4, 9), (4, 10), (59a, 61), (61, 59a) e no mínimo mais 1 par;
- *o1*: (5, 7), (5, 8), (5, 59), (5, 61), (5, 59a), (5, 61a);
- *out1*: (9, 11);
- *out2*: (10, 12).

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
s	*b1*	*take*(s) *add*(s)	6	27,27%	5	83,33%	22,73%		
i n c	*b2*	*take*(s) *add*(s)	4	18,18%	3	75%	13,64%	54,54%	
	o1	-	4	18,18%	4	100%	18,18%		
n	*b1*	-	2	9,09 %	2	100%	9,09 %		Semáforo
s	*b2*	-	2	9,09%	2	100%	9,09%		
i	*o1*	-	2	9,09%	2	100%	9,09%		
n	*out1*	*run*	1	4,54%	1	100%	4,54%	36,36%	
c	*out2*	*run*	1	4,54%	1	100%			
			22		20			90,90%	

Tabela 5.8: Cobertura do Programa *Bounded_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

Ao analisarmos este programa, podemos dizer que 36,36% do total de pares são de objetos não sincronizados e que todos foram exercitados. É possível concluir tal informação se verificarmos a coluna CT da Tabela 5.8. Além disto, sabemos que, no mínimo, 54,54% dos pares de objetos sincronizados também foram exercitados. Se observarmos de um modo geral, podemos afirma que 90,90% dos pares *du* encontrados foram exercitados. Como foi feita uma estimativa mínima, é possível que esta porcentagem seja maior.

5.1.9 Programa *PipeInteger_s.java*

O programa *PipeInteger_s.java* descrito na Figura 5.17 retrata a idéia de um simples *pipeline*. Este programa entrava em *deadlock* porque a condição do *loop* da classe *Listener* (linha 93) não era violada no momento em que deveria e, conseqüentemente, o objeto *Heap.c4* entrava em estado de espera e nunca era despertado. Para remover este problema, a condição da linha 93 foi alterada.

Para realizar a análise deste programa é necessário identificar os requisitos que precisam ser exercitados. O primeiro passo para que esta análise seja realizada é identificar

```
 1: class Heap {
 2:     static Connector c1, c2, c3, c4;
 3: }
 4:
 5: class PipeInteger_s {
 6:     static public void main (String[] args) {
 7:         Heap.c1 = new Connector();
 8:         Heap.c2 = new Connector();
 9:         Stage1 s1 =  new Stage1();
10:         s1.start();
11:         Heap.c3 = new Connector();
12:         Stage2 s2 = new Stage2();
13:         s2.start();
14:         Heap.c4 = new Connector();
15:         Stage3 s3 = new Stage3();
16:         s3.start();
17:         Listener lis = new Listener();
18:         lis.start();
19:
20:         for (int i = 1; i < 5; i++)
21:             Heap.c1.add(i);
22:         Heap.c1.stop();
23:     }
24: }
25:
26: final class Connector {
27:     int queue = -1;
28:     public final synchronized int take() {
29:         int value;
30:         while (queue < 0)
31:             try {
32:                 wait();
33:             } catch (InterruptedException ex) {}
34:         value = queue;
35:         queue = -1;
36:         return value;
37:     }
38:
39:     public final synchronized void add(int o) {
40:         queue = o;
41:         notifyAll();
42:     }
43:
44:     public final synchronized void stop() {
45:         queue = 0;
46:         notifyAll();
47:     }
48: }
49:
```

```
50: final class Stage1 extends Thread {
51:     public void run() {
52:         System.out.println("Stage1 startup");
53:         int tmp = -1;
54:         while (tmp != 0) {
55:             if ((tmp = Heap.c1.take()) != 0)
56:                 Heap.c2.add(tmp + 1);
57:         }
58:         Heap.c2.stop();
59:         System.out.println("Stage1 shutdown");
60:     }
61: }
62:
63: final class Stage2 extends Thread {
64:     public void run() {
65:         System.out.println("Stage2 startup");
66:         int tmp = -1;
67:         while (tmp != 0) {
68:             if ((tmp = Heap.c2.take()) != 0)
69:                 Heap.c3.add(tmp + 1);
70:         }
71:         Heap.c3.stop();
72:         System.out.println("Stage2 shutdown");
73:     }
74: }
75:
76: final class Stage3 extends Thread {
77:     public void run() {
78:         System.out.println("Stage3 startup");
79:         int tmp = -1;
80:         while (tmp != 0) {
81:             if ((tmp = Heap.c3.take()) != 0)
82:                 Heap.c4.add(tmp + 1);
83:         }
84:         Heap.c4.stop();
85:         System.out.println("Stage3 shutdown");
86:     }
87: }
88:
89: final class Listener extends Thread {
90:     public void run() {
91:         System.out.println("Listener startup");
92:         int tmp = -1;
93:         while (tmp != 0) {
94:             if ((tmp = Heap.c4.take()) != 0)
95:                 System.out.println("output is " + tmp);
96:         }
97:         System.out.println("Listener shutdown");
98:     }
99: }
```

Figura 5.17: Programa *PipeInteger_s.java*

as definições e usos de cada objeto:

1. Objeto *Heap.c1*

   Definido : 7, 21, 22, 55;
   Usado : 55.

2. Objeto *Heap.c2*

   Definido : 8, 56, 68, 58;
   Usado : 68.

3. Objeto *Heap.c3*

   Definido : 11, 69, 71, 81;
   Usado : 81.

4. Objeto *Heap.c4*

   Definido : 14, 82, 84, 94;

Usado : 94.

5. Objeto *s1*

   Definido : 9;
   Usado : 10.

6. Objeto *s2*

   Definido : 12;
   Usado : 13.

7. Objeto *s3*

   Definido : 15 ;
   Usado : 16.

8. Objeto *Heap.c4*

   Definido : 17;
   Usado : 18.

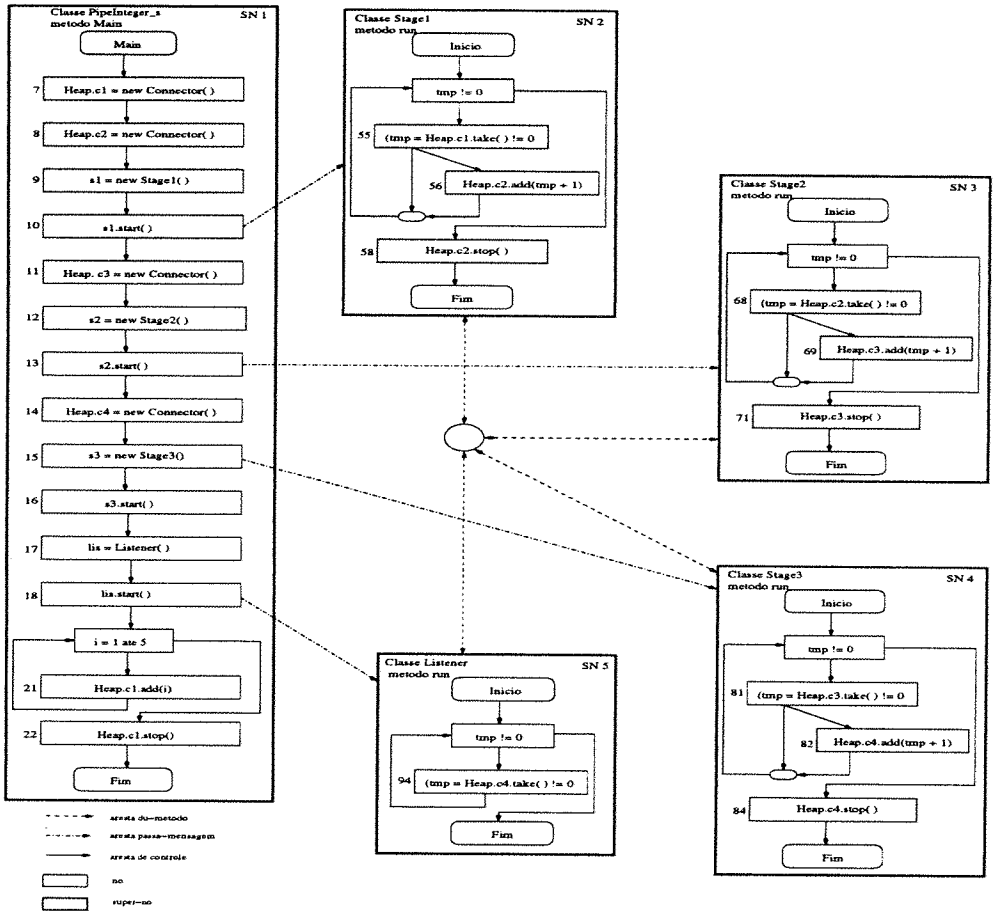


Figura 5.18: DFO do Programa *PipeInteger_s.java*

O segundo passo é construir os DFO (Figura 5.18) e identificar os pares *du* que podem ser executados. Para que este segundo passo seja realizado, é preciso conhecermos o fluxo de controle dos objetos e as definições e usos de cada um. Ao realizar esta atividade, encontramos os seguintes pares:

- *Heap.c1*: (7, 55), (21, 55), (22, 55);
- *Heap.c2*: (8, 68), (56, 68), (58, 68);
- *Heap.c3*: (11, 81), (69, 81), (71, 81);
- *Heap.c4*: (14, 94), (82, 94), (84, 94);
- *s1*:(9, 10);
- *s2*:(12, 13);
- *s3*:(15, 16);
- *lis*:(17, 18).

Ao utilizar o *Bandera* para realizar a verificação, a técnica de fatiamento foi utilizada sem remover nenhuma instrução que seja relevante para o critério a ser analisado. Entretanto, a técnica de abstração de dados não foi utilizada, pois não há uma abstração adequada implementada na *Bandera* que possa ser aplicada a este programa e as que existem, caso fossem aplicadas, geraram contra-exemplos impraticáveis.

Com a aplicação da técnica de redução descrita anteriormente, houve uma redução significativa nos estados gerados em *PROMELA*: de 20128 reduziu para 14620. Através do conhecimento dos estado percorridos é possível mapearmos quais instruções no código Java estes estados representam. Conseqüentemente, também é possível conhecer quais instruções foram exercitadas. Mas não é possível saber os caminhos percorridos quando os estados foram percorridos. Devido a esta incerteza, foi feita uma estimativa da quantidade mínima de pares *du* que deveriam ser exercitados para que os estados fossem percorridos. Além desta estimativa, os pares que temos certeza que foram exercitados serão descritos separadamente:

- *Heap.c1*: no mínimo 1 par;
- *Heap.c2*: no mínimo 1 par;
- *Heap.c3*: no mínimo 1 par;
- *Heap.c4*: no mínimo 1 par;
- *s1*:(9, 10);
- *s2*:(12, 13);
- *s3*:(15, 16);

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
s i n c	*Heap.c1*	*add*(s)	4	18,75%	1	25%	5%	20%	Semáforo
	Heap.c2	*take*(s)	4	18,75%	1	25%	5%		
	Heap.c3	*stop*(s)	4	18,75%	1	25%	5%		
	Heap.c4		4	18,75%	1	25%	5%		
n s i n c	*s1*	*run*	1	5%	1	100%	5%	20%	
	s2	*run*	1	5%	1	100%	5%		
	s3	*run*	1	5%	1	100%	5%		
	lis	*run*	1	5%	1	100%	5%		
			20		8			40%	

Tabela 5.9: Cobertura do Programa *PipeInteger_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

- *lis*:(17, 18).

Para este programa, a Tabela 5.9 ilustra uma análise detalhada. Ao observarmos a coluna `CO`, concluímos que todos os pares de objetos não sincronizados foram exercitados. Já para os objetos sincronizados, sabemos que pelo menos um par de cada um destes objetos foi exercitado. Com isto, podemos dizer que no mínimo 20% deste pares foram exercitados, mas é possível que esta porcentagem seja maior. Devido a esta incerteza, notamos que a porcentagem de cobertura mínima para este programa é de apenas 40%.

5.1.10 Programa *StrictClockS_s.java*

O programa da Figura 5.19 simula o alarme de um relógio que dispara para acordar 2 clientes. Para eliminar o *deadlock* deste programa, no método *tick*, ao invés de acordar apenas um cliente, o comando *notifyall* é usado para despertar todos os clientes (linha 85), neste caso apenas 2.

Inicialmente, é preciso identificar todas as definições e usos de cada objeto para que os requisitos do critério de teste também sejam identificados:

1. Objeto *m*
   Definido :7, 29, 52, 52a;
   Usado : 8, 10, 11, 28, 29, 52, 52a.
2. Objeto *waitlist*

```
1  public class StrictClockS
2  {
3      public static void main(String[] args)
4      {
5          int numThreads = 2;
6          int maxTime = 5;
7          Monitor m  = new Monitor(maxTime);
8          Clock   c1 = new Clock("Clock", m, maxTime);
9          c1.start();
10         Client1 c1 = new Client1("Client", m, 1);
11         Client2 c2 = new Client2("Client", m, 2);
12         c1.start();
13         c2.start();
14     }
15 }
16
17 class Clock extends Thread{
18     private String name;
19     private Monitor monitor;
20     private int max;
21     public Clock(String n, Monitor m, int maxTime){
22       name = n;
23       monitor = m;
24       max = maxTime;
25     }
26     public void run(){
27         System.out.println("I am thread " + name);
28         while(monitor.getTime() < max){
29            monitor.tick();
30            try{sleep(10);}
31            catch(InterruptedException e){System.out.println(e);}
32         }
33     }
34 }
35
36 class Client1 extends Thread{
37     private String name;
38     private Monitor monitor;
39     private int id;
40
41     public Client1(String n, Monitor m, int i) {
42         name = n;
43         monitor = m;
44         id = i;
45     }
46
47     public void run(){
48         int maxSleepTime = 5;
49         int interval;
50         System.out.println("I am thread " + name + " number " + id);
51         interval = 3;
52         monitor.wakeme(id, interval);
53         System.out.println("Client1: Awaken");
54     }
55 }
56 class Client2 extends Thread{
57     private String name;
58     private Monitor monitor;
59     private int id;
60
61     public Client2(String n, Monitor m, int i) {
62         name = n;
63         monitor = m;
64         id = i;
65     }
66
67     public void run() {
```

(a) Página 1

Figura 5.19: Programa *StrictClockS_s.java*

```
68          int maxSleepTime = 5;
69          int interval;
70          System.out.println("I am thread " + name + " number " + id);
71          interval = 2;
72          monitor.wakeme(id, interval);
73          System.out.println("Client2: Awaken");
74      }
75  }
76
77  class Monitor{
78      private int now;            // current time
79      private MyLinkedList waitList;
80      private int max;            // max time
81      private MyObject countMon; // for addtional synch
82      private int count;          // for additional synck
83
84      Monitor(int maxTime){
85          now = 0;
86          waitList = new MyLinkedList();
87          countMon = new MyObject(0); // object used for additional synch
88                                      // since nobody will use 0 for interval
89          count = 0;                  // var used for additional synch
90          max = maxTime;
91      }
92
93      synchronized void tick(){
94          synchronized(countMon){
95              if(count!=0){
96                  try{countMon.wait();}catch (InterruptedException e){}
97              }
98              now++;
99              if(!waitList.isEmpty()){
100                 MyObject first = waitList.firstElement();
101                 if(first.time() == now){
102                     MyObject wakeup = first;
103                     waitList.removeElementAt(0);
104                     synchronized(wakeup){
105                         wakeup.notifyall();
106                     }
107                 }
108             }
109         }
110     }
111
112     void wakeme(int id, int interval){
113         int waketime;
114         MyObject lock;
115
116         synchronized (this){
117             waketime = now + interval;
118             System.out.println("Monitor: Client"+id+" calling wakeme()");
119             if(waketime > max){
120                 System.out.println("Want to wake up past max");
121                 return;
122             }
123             System.out.println("Client "+id+"Want to sleep "+ interval+" want to wake up at "
+waketime);
124             lock = waitList.getLock(waketime);
125
126             synchronized(countMon){
127                 count++;
128             }
129         }
130
131         synchronized(lock){
132             synchronized(countMon){
133                 count--;
```

(b) Página 2

Figura 5.19: Programa *StrictClockS_s.java* (Continuação...)

```
134                 if(count==0)countMon.notifyAll();
135             }
136             try{
137                 lock.wait();
138             }catch(InterruptedException e){}
139             System.out.println("Client awaken, time is "+ now);
140         }
141     }
142
143     int getTime(){
144         return now;
145     }
146 }
147
148 class MyObject extends Object{
149     public int time;
150     MyObject(int n){
151         time = n;
152     }
153
154     public int time(){
155         return time;
156     }
157 }
158
159 class MyLinkedList{
160     private MyObject[] list = new MyObject[2];
161     private int capacity =2;
162     private int size = 0;
163
164     public synchronized boolean isEmpty(){
165         return size==0;
166     }
167
168     public synchronized boolean isFull(){
169         return size == capacity;
170     }
171
172     public synchronized MyObject elementAt(int index){
173         return list[index];
174     }
175
176     public synchronized void insertElementAt(MyObject obj, int index){
177         if(!this.isFull()){
178             for(int i=size-1; i>=index; i--){
179                 list[i+1] = list[i];
180             }
181             list[index] = obj;
182             size++;
183         }
184     }
185
186     public synchronized void addElement(MyObject obj){
187         if(!this.isFull()){
188             list[size] = obj;
189             size++;
190         }
191     }
192
193     public synchronized MyObject firstElement(){
194         return list[0];
195     }
196
197     public synchronized void removeElementAt(int index){
198         if(!this.isEmpty()){
199             for(int i=index; i<size-1; i++){
200                 list[i] = list[i+1];
```

(c) Página 3

Figura 5.19: Programa *StrictClockS_s.java* (Continuação...)

```
201                 }
202                 list[size-1] = null;
203                 size--;
204             }
205         }
206
207         public synchronized MyObject getLock(int n){
208             MyObject temp;
209             int key;
210             MyObject help;
211             int i = 0;
212             for ( i=0; i<size; i++){
213                 help=list[i];
214                 key = help.time();
215                 if (n==key){
216                     temp = list[i];
217                     return temp;
218                 }
219             }
220             MyObject lock = new MyObject(n);
221             for (i=0; i<size; i++){
222                 help = list[i];
223                 key = help.time();
224                 if( n  < key ){
225                     this.insertElementAt(lock, i);
226                     return lock;
227                 }
228             }
229             this.addElement(lock); //append if greater than all others
230             return lock;
231         }
232     }
233
```

(d) Página 4

Figura 5.19: Programa *StrictClockS_s.java* (Fim)

Definido : 66, 83, 104;
Usado : 79, 80, 81, 82, 83, 104.

3. Objeto *cl*

   Definido : 8;
   Usado : 9.

4. Objeto *c1*

   Definido : 10;
   Usado : 12.

5. Objeto *c2*

   Definido : 11;
   Usado : 13.

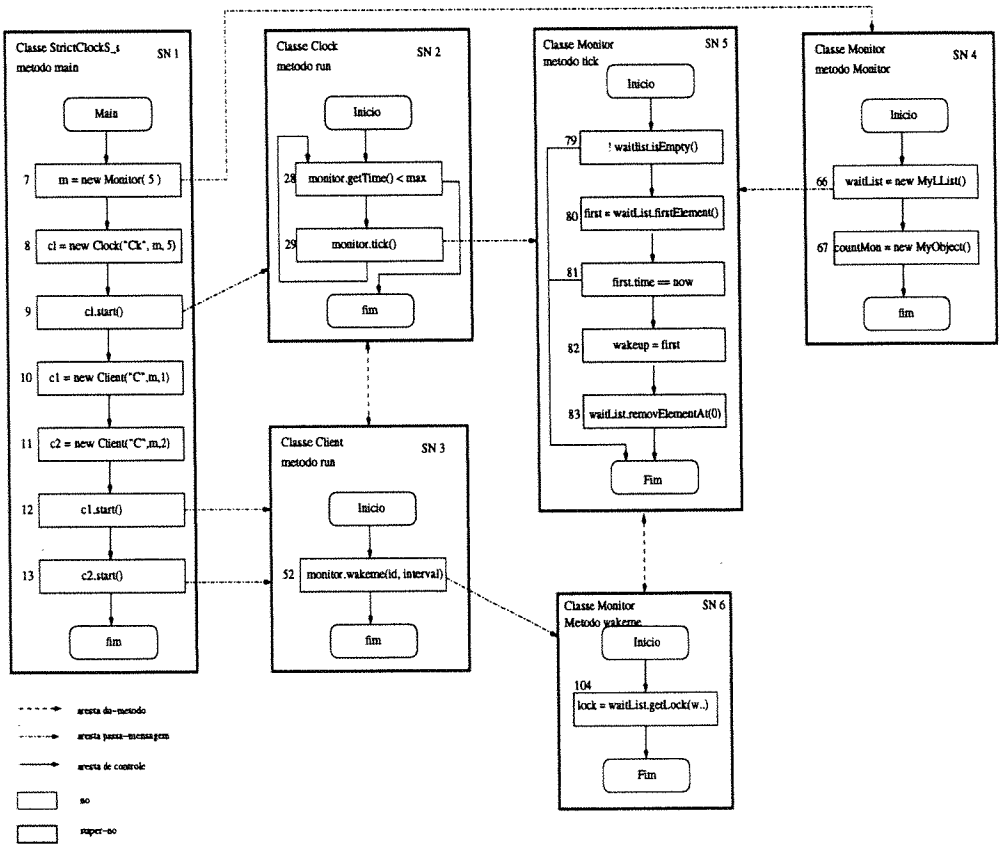


Figura 5.20: DFO do Programa *StrictClockS_s.java*

Além destas definições e usos, é preciso também conhecer o fluxo de controle dos objetos. Com base nestas duas informações podemos criar o DFO (Figura 5.20) para este programa. À medida que o diagrama é criado, também, é possível identificarmos os pares *du* que precisam ser exercitados:

- *m*: (7, 8), (7, 10), (7, 11), (7, 28), (7, 29),(7, 52), (7, 52a), (29, 52), (52, 28), (52, 29), (29, 52a), (52a, 28), (52a, 89);
- *waitlist*: (66, 79), (66, 80), (66, 81), (66, 82), (66, 83), (66, 104), (83, 104), (104, 79) , (104, 80), (104, 81), (104, 83), (104, 82);
- *cl*: (8, 9);
- *c1*: (10, 12);
- *c2*: (11, 13).

Tanto a técnica de fatiamento quanto a de abstrações de dados não foram utilizadas pela *Bandera* ao realizar a verificação da propriedade. Ao aplicar o fatiamento de programas, instruções relevantes para o critério foram removidos. Além disso, a técnica de abstração de dados também não foi utilizada, pois não há uma abstração adequada implementada na ferramenta que possa ser aplicada a este programa. O fato de nenhuma destas técnicas serem aplicadas a este programa, tornou a análise muito trabalhosa, pois 80875 estados foram encontrado, ou seja, isto significa que este programa traduzido para *PROMELA* ficou muito extenso.

Mesmo com tal dificuldade, o mapeamento dos estados em *PROMELA* para as instruções do código Java foi realizada. Ao comparar os estados percorridos e as instruções, podemos concluir que todas as definições e usos foram exercitadas. Mas devido ao desconhecimento de quais caminhos que foram percorridos para que estes dados fossem exercitados, fizemos uma estimativa do número mínimo de pares *du* que devem ser exercitados:

- *m*: (7, 8), (7, 10), (7, 11) e no mínimo mais 3 pares;
- *waitlist*: no mínimo 6 pares;
- *cl*: (8, 9);
- *c1*: (10, 12);
- *c2*: (11, 13).

Comparando os pares de objetos sincronizados e não sincronizados, podemos perceber que todos os não sincronizados foram exercitados, o que corresponde a 21,43% como pode ser visto na Tabela 5.10. Se observarmos a coluna CT desta Tabela, também podemos perceber que 32,24% dos pares sincronizados foram exercitados. Mas é relevante lembrar que este dado é uma estimativa mínima e, no entanto, esta porcentagem pode

	Objetos	Métodos	Cobertura dos pares *du*						Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]	
s i n c	*waitlist*	*isEmpty(s)* *firstElement(s)* *removeElementAt(s)* *getlock(s)*	12	42,86%	6	50%	21,43%	32,14%	Monitores Aninhados
	m	*getTime(s)* *tick(s)* *wakeme*	10	35,71%	3	30%	10,71%		
n s i n c	*m*	-	3	10,71%	3	100%	10,71%	21,43%	
	c1	run	1	3,57%	1	100%	3,57%		
	c1	run	1	3,57%	1	100%	3,57%		
	c2	run	1	3,57%	1	100%	3,57%		
			28		15			53,57%	

Tabela 5.10: Cobertura do Programa *StrictClockS_s.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

ser maior. Outro fator que pode ser analisado neste programa, é a não possíbilidade de conhecemento de nenhum par dos objetos sincronizados.

Para a análise geral deste programa, notamos que dos 28 pares *du* identificados no mínimo 15 foram exercitados. Percentualmente, isto corresponde a uma cobertura de 53,57%. Se compararmos com a média de 50%, podemos dizer que a cobertura é razoável.

5.2 Conclusão

No Capítulo anterior, concluímos que a verificação formal feita sobre programas que violam a propriedade de *deadlock* é de grande ajuda para o auxílio da atividade de teste. Segundo as análises feitas neste Capítulo, também podemos concluir que a *Bandera* auxilia o teste de programa Java quando a propriedade de *deadlock* não é violada, embora os resultados obtidos sejam valores estimados.

O problema destes resultados serem uma estimativa mínima está vinculado ao trabalho manual no mapeamento entre os estados em *PROMELA* e código Java. Como foi dito nas Seções anteriores, quando a *Bandera* verifica uma propriedade sobre um programa e esta verificação resulta em sucesso, o verificador de modelo Spin fornece apenas

os estados que não foram percorridos. Através da análise dos estados não percorridos, é possível saber quais estados foram percorridos, mas não é possível identificar quais caminhos foram percorridos. Através do mapeamento dos estados em *PROMELA* para Java, também podemos saber quais instruções em Java foram executas.

Com o objetivo de auxiliar a análise dos programas, foram identificados algumas possíveis seqüencias de caminhos que executam todas as instruções em Java (veja na Seção anterior). Através da análise destes possíveis seqüencias e do mapeamento para Java, é possível descobrir algumas partes dos caminhos que serão executadas, por exemplo quando um objeto entra em estado de espera, ele só será despertado quando executar uma instrução *notify* ou *notifyall*. Entretanto, para os outros caminhos, é difícil identificarmos quem são, pois todos os programas são concorrentes. Devido a esta incerteza de quais seqüências foram percorridas, a análise do critério de teste foi feita com uma estimativa mínima de cobertura.

Um outro fato que é relevante ressaltar está relacionado com as técnicas de redução de código. Se observarmos, para nenhum programa conseguimos aplicar as duas técnicas simultaneamente. Isto está relacionado com o fator de preservação das características originais de cada programa e principalmente em manter as instruções que são relevantes para o critério de teste analisado. Isto não significa que as técnicas de redução não possam ser aplicadas simultaneamente, estamos apenas ressaltando que para os programas escolhidos isto não ocorreu.

Programas	todos os pares *du*		Porcentagem de cobertura
	identificados	exercitados	
Dead_s.java	28	13	46,43%
Phil33_s.java	24	12	58,33%
START_s.java	20	14	70%
Bounded_s.java	22	20	90,90%
PipeInteger_s.java	16	8	50%
NestedMonitor_s.java	16	10	62,5%
MonitorDead_s.java	13	10	76,92%
PCSimple_s.java	11	8	70%
MainPC_s.java	8	7	87,5%
StrictClockS_s.java	28	15	53,57%

Tabela 5.11: Análise dos Requisitos (Programas sem *Deadlock*)

A Tabela 5.11 mostra que 80% dos programas analisados, tiveram uma porcentagem mínima de cobertura acima de 50%. Enquanto, apenas 2 programas tiveram a

porcentagem igual ou abaixo dos 50%. Mesmo fazendo uma estimativa mínima, podemos concluir que a ferramenta *Bandera* também tem uma importância significativa na análise de programas sem *deadlock*. Caso fosse possível descobrirmos os caminhos percorridos durante esta verificação, os resultados seriam mais precisos e concisos.

No caso deste Capítulo, a análise foi feita sobre os mesmos programas sem o *deadlock*. Mesmo alterando os programas para retirar o *deadlock*, eles continuam sendo simples e em pequena quantidade, mas são significativos para mostrar que o uso da *Bandera* ajuda a atividade de teste. Um dos problemas desta análise está nos casos de teste que podem ser atendidos, devido à incerteza de quais caminhos foram percorridos durante a verificação é difícil concluirmos com precisão quais as partes ou quais destes casos de teste não precisam ser exercitados.

Além deste problema, também há os possíveis erros que podem acontecer no momento do mapeamento dos estados de *PROMELA* para Java e, conseqüentemente, isto interfere na precisão da análise dos programas. Isto acontece devido ao processo manual e à dificuldade de realizar tal tarefa. É considerado difícil porque um simples programa com poucas linhas de código em Java pode tornar-se 100 vezes maior quando traduzido para *PROMELA*, devido à explosão de estados.

Observando os programas analisados, percebemos claramentente esta dificuldade. Por exemplo, os programas *PipeInteger_s.java* e *StrictClockS_s.java* são maiores e considerados mais complexos comparados aos outros programas escolhidos e, conseqüentemente, o código gerado em *PROMELA* para estes programas são extensos. O programa *StrictClockS_s.java* com 233 linhas de códigos passou a ter aproximadamente 80875 estados quando convertidos para *PROMELA*. Com isso, podemos concluir que dependendo do tamanho do programa Java, a análise do código traduzido para *PROMELA* pode ser inviável mesmo quando aplicadas as técnicas de redução de modelo.

Capítulo 6

Exemplo: a Atividade de Teste *versus* Verificação Formal

Nos Capítulos 4 e 5, foi mostrado, separadamente, como a ferramenta *Bandera* pode auxiliar a atividade de teste para cada programa. É relevante lembrar que, na realidade, as análises de cada programa devem ser feitas juntas e de forma complementar uma da outra. A idéia deste Capítulo é mostrar como deve ser o teste de um programa com o auxílio de uma ferramenta de verificação formal. É relevante lembrar que neste caso os nomes dos programas com e sem *deadlock* serão mantidos os originais (mesmo adotados no Capítulo 4). Isto foi adotado para facilitar o entendimento deste exemplo.

6.1 Teste e Verificação com *Deadlock*

Para exemplificar tal estudo, foi escolhido o programa *NestedMonitor.java* (Figura 6.1). É relevante ressaltar que a maioria das informações descritas aqui já estão nos Capítulos 4 e 5. Este programa implementa dois *buffers* ao mesmo tempo. A diferença entre os dois é que um é inicializado como vazio e outro como cheio.

Inicialmente, vamos identificar as definições e os usos de cada objeto:

1. Objeto *buf*
   Definido :57, 73, 86;
   Usado : 58, 60, 73, 86.

```
 1: interface Buffer {
 2:     public void put(Object o)
 3:         throws InterruptedException; //poe o objeto no buffer
 4:     public Object get()
 5:         throws InterruptedException; //retira um objeto do buffer
 6: }
 7:
 8: class Semaphore {
 9:     private int value;
10:     public Semaphore (int initial) {
11:         value = initial;
12:     }
13:
14:     synchronized public void up() {
15:         ++value;
16:         notifyAll();
17:     }
18:
19:     synchronized public void down() throws InterruptedException {
20:         while (value==0) {wait();}
21:         --value;
22:     }
23: }
24:
25: /*********************SEMABUFFER*****************************/
26: class SemaBuffer implements Buffer {
27:     protected int size;
28:     Semaphore full; //conta o numero de itens
29:     Semaphore empty;//conta o numero de espaco
30:
31:     SemaBuffer(int size) {
32:         this.size = size;
33:         full = new Semaphore(0);
34:         empty= new Semaphore(size);
35:     }
36:     //public void put(Object o)
37:     synchronized public void put(Object o)
38:         throws InterruptedException {
39:         empty.down();
40:         full.up();
41:     }
42:     // public Object get()
43:     synchronized public Object get()
44:         throws InterruptedException{
45:         full.down();
46:         empty.up();
47:         return (null);
48:     }
49: }
50:
```

```
51: /********** Pricipal ************/
52: public class NestedMonitor  {
53:     static int SIZE = 5; /* parametro */
54:     static Buffer buf;
55:
56:     public static void main(String [] args) {
57:         buf = new SemaBuffer(SIZE);
58:         Producer p = new Producer(buf);
59:         p.start();
60:         Consumer c = new Consumer(buf);
61:         c.start();
62:     }
63: }
64:
65: /*******************PRODUCER************************/
66: class Producer extends Thread {
67:     Buffer buf;
68:     Producer(Buffer b) {buf = b;}
69:
70:     public void run() {
71:       try {
72:         while(true)
73:             buf.put(null);
74:       } catch (InterruptedException e){}
75:     }
76: }
77:
78: /*******************CONSUMER*************************
79: class Consumer extends Thread {
80:     Buffer buf;
81:     Consumer(Buffer b) {buf = b;}
82:
83:     public void run() {
84:         try {
85:           while(true)
86:             buf.get();
87:       } catch(InterruptedException e ){}
88:     }
89: }
90:
```

Figura 6.1: Programa *NestedMonitor.java*

2. Objeto *full*
   Definido : 33, 40, 45;
   Usado : 40, 45.
3. Objeto *empty*
   Definido : 34, 39, 46;
   Usado : 39, 46.
4. Objeto *p*
   Definido : 58;
   Usado : 59.
5. Objeto *c*
   Definido : 60;
   Usado : 61.

Com base nestes dados e no fluxo de controle dos objetos, é criado o DFO (Figura 6.2) e com isto também são identificados os pares *du*:

- *buf*: (57, 58), (57, 60), (57, 73), (57, 86), (73, 86), (86, 73);

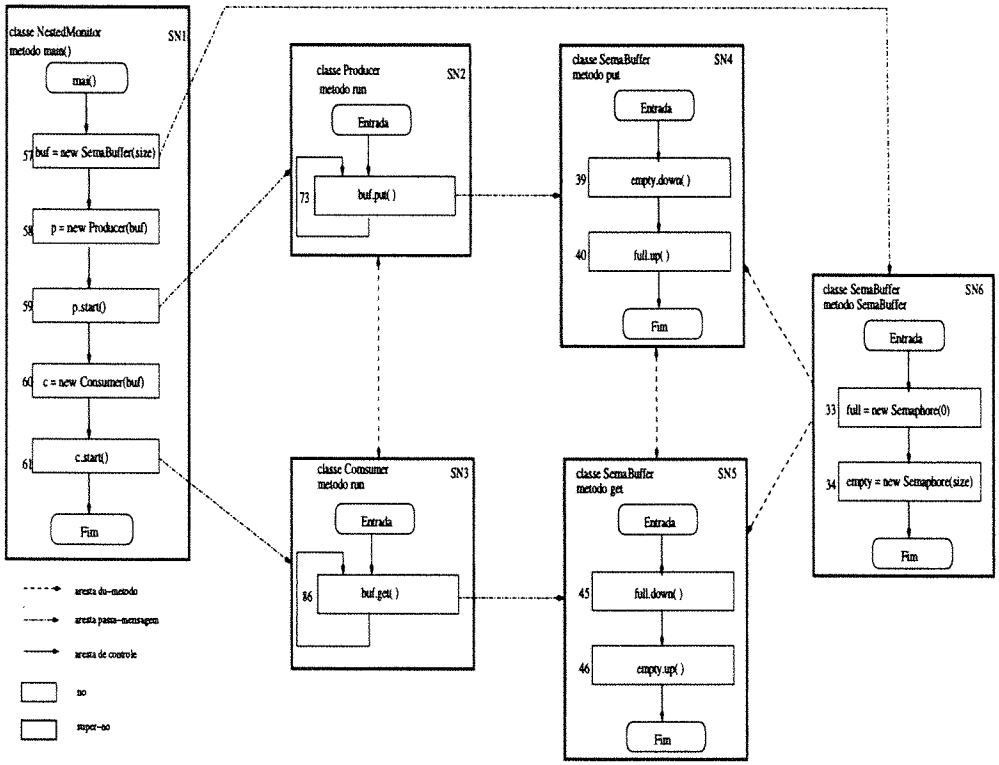


Figura 6.2: DFO do Programa *NestedMonitor.java*

- *full*: (33, 40), (33, 45), (40, 45), (45, 40);
- *empty*: (34, 39), (34, 46), (39, 46), (46, 39);
- *p*: (58, 59);
- *c*: (60, 61).

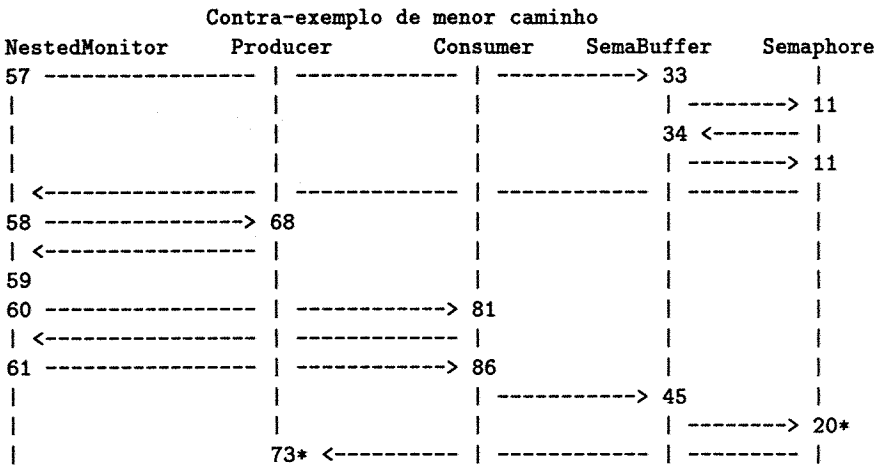
Após identificar os pares *du*, podemos criar os casos de teste que exercitam pelo menos uma vez todos os requisitos identificados:

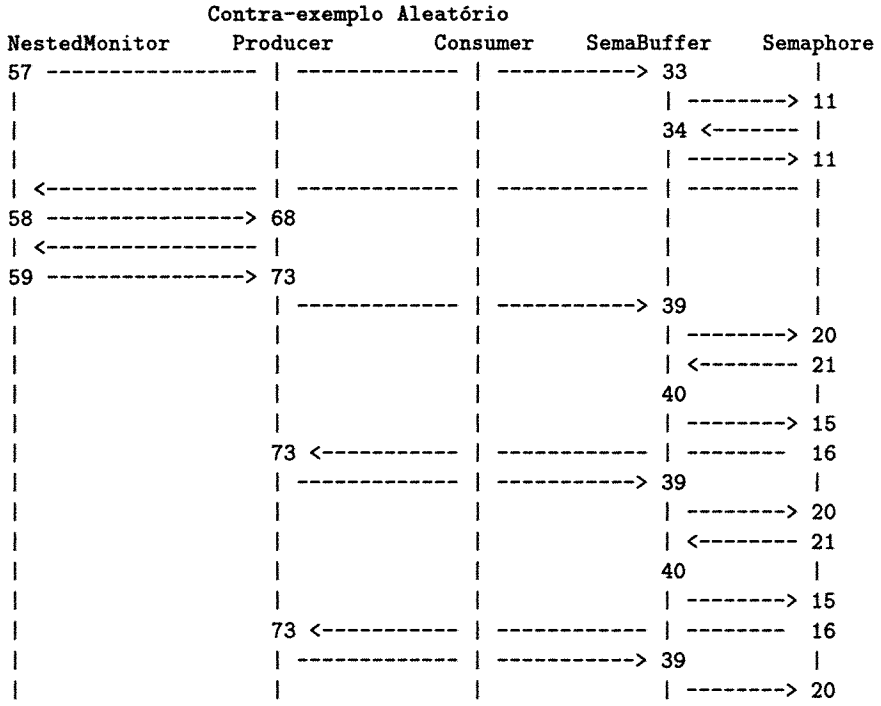
- **Caso 1**: 57, 33, 34, 58, 59, 60, 60, 73, 39, 40, 86, 45, 46;
- **Caso 2**: 57, 33, 34, 58, 59, 60, 61, 86, 45, 46, 73, 39, 40.

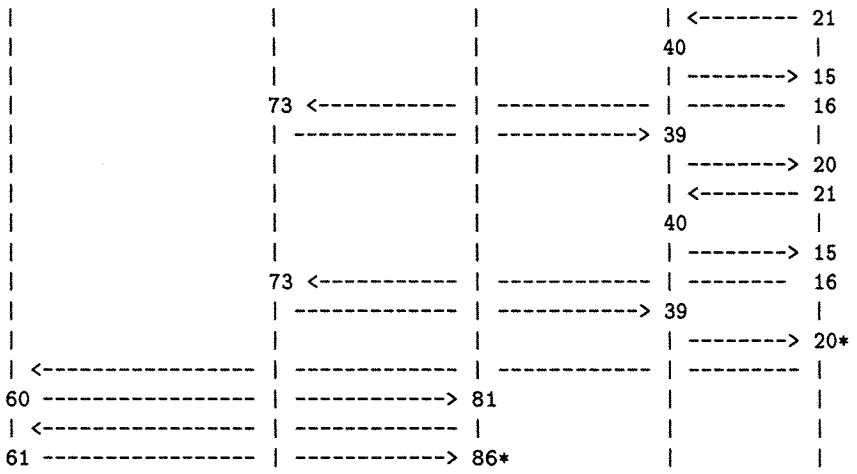
Numa atividade de teste convencional, após identificar estes casos de teste, seria necessário a geração de dados de teste para exercitar estes casos. Mas antes de gerar

estes dados, vamos fazer a verificação de ausência de *deadlock* utilizando a ferramenta *Bandera*. Com isto, analisaremos o que já foi testado e o que ainda precisa ser testado.

Ao verificar a propriedade, a *Bandera* revelou que este programa em algum momento entra em *deadlock*. Além disso, a *Bandera* gerou dois contra-exemplos distintos: o de menor caminho e o aleatório, os quais são descritos a seguir:







Com base na análise de cada contra-exemplo, percebemos que os seguintes pares *du* foram exercitados:

- *Menor Caminho*: (57, 58), (57, 60), (57, 86), (33, 45), (60, 61);
- *Aleatório*: (57, 58), (57, 60), (57, 73), (33, 40), (34, 39), (58, 59).

Ao analisar os caminhos dos contra-exemplos gerados, percebemos que alguns requisitos exercitados no contra-exemplo de menor caminho não foram exercitados no contra-exemplo aleatório e vice-versa. Com isto, podemos afirmar que estes contra-exemplos são complementares para sabermos quais e quantos pares *du* foram exercitados do critério de teste:

- *buf*: (57, 58), (57, 60), (57, 73), (57, 86);
- *full*: (33, 40), (33, 45);
- *empty*: (34, 39);
- *p*: (58, 59);
- *c*: (60, 61).

Com relação à informação anterior, podemos dizer que nenhum dos dois casos de teste gerados foi totalmente exercitado, mas podemos afirmar que uma boa parte de cada caso foi exercitada. Logo abaixo é descrito o que foi exercitado dos casos de teste:

- **Caso 1**: 57, 33, 34, 58, 59, 60, 61, 73, 39, 40;
- **Caso 2**: 57, 33, 34, 58, 59, 60, 61, 86, 45.

É relevante lembrar que a análise feita até o momento foi sobre o programa quando este entra em *deadlock*. Ao descobrir que o programa entra em *deadlock*, o próximo passo é eliminar o *deadlock* deste programa e repetir a atividade de teste com o auxílio da *Bandera* para o programa modificado.

6.2 Teste e Verificação sem *Deadlock*

Para retirar o *deadlock* deste programa, foram eliminados monitores aninhados. Veja nas linhas 37 e 43 da Classe *SemaBuffer* da Figura 6.1, os métodos *put* e *get* deixariam de ser sincronizados e com isto elimina-se os monitores aninhados. Isto acontece porque estes métodos invocam objetos que também chamam médodos sincronizados. A Figura 6.3 nos mostra como fica o programa ao eliminar a sincronização destes métodos.

Com a retirada do *deadlock*, repete-se a atividade de teste. Tal tarefa faz se necessária porque, ao retirar o *deadlock*, podem aparecer pares *du* que não são exercitados quando o programa tinha a propriedade violada, pois o código foi modificado. Ao retirar o *deadlock*, devemos ter o cuidado de manter as características do programa mantendo os mesmos pares *du* tanto quanto possível. O próximo passo é identificar novamente as definições e os usos de cada objeto:

1. Objeto *buf*
   Definido :57, 73, 86;
   Usado : 58, 60, 73, 86.
2. Objeto *full*
   Definido : 33, 40, 45;
   Usado : 40, 45.
3. Objeto *empty*
   Definido : 34, 39, 46;
   Usado : 39, 46.
4. Objeto *p*
   Definido : 58;
   Usado : 59.
5. Objeto *c*

```
 1: interface Buffer {
 2:     public void put(Object o)
 3:         throws InterruptedException; //poe o objeto no buffer
 4:     public Object get()
 5:         throws InterruptedException; //retira um objeto do buffer
 6: }
 7:
 8: class Semaphore {
 9:     private int value;
10:     public Semaphore (int initial) {
11:         value = initial;
12:     }
13:
14:     synchronized public void up() {
15:         ++value;
16:         notifyAll();
17:     }
18:
19:     synchronized public void down() throws InterruptedException {
20:         while (value==0) {wait();}
21:         --value;
22:     }
23: }
24:
25: /*********************SEMABUFFER*****************************/
26: class SemaBuffer implements Buffer {
27:     protected int size;
28:     Semaphore full; //conta o numero de itens
29:     Semaphore empty; //conta o numero de espaco
30:
31:     SemaBuffer(int size) {
32:         this.size = size;
33:         full = new Semaphore(0);
34:         empty= new Semaphore(size);
35:     }
36:     //public void put(Object o)
37:     public void put(Object o)
38:         throws InterruptedException {
39:          empty.down();
40:          full.up();
41:     }
42:     // public Object get()
43:     public Object get()
44:         throws InterruptedException{
45:         full.down();
46:         empty.up();
47:         return (null);
48:     }
49: }
50:
```

```
51: /********** Pricipal ************/
52: public class NestedMonitor_s  {
53:     static int SIZE = 5; /* parametro */
54:     static Buffer buf;
55:
56:     public static void main(String [] args) {
57:         buf = new SemaBuffer(SIZE);
58:         Producer p = new Producer(buf);
59:         p.start();
60:         Consumer c = new Consumer(buf);
61:         c.start();
62:     }
63: }
64:
65: /********************PRODUCER*********************
66: class Producer extends Thread {
67:     Buffer buf;
68:     Producer(Buffer b) {buf = b;}
69:
70:     public void run() {
71:       try {
72:         while(true)
73:             buf.put(null);
74:       } catch (InterruptedException e){}
75:     }
76: }
77:
78: /********************CONSUMER*********************
79: class Consumer extends Thread {
80:     Buffer buf;
81:     Consumer(Buffer b) {buf = b;}
82:
83:     public void run() {
84:         try {
85:           while(true)
86:             buf.get();
87:       } catch(InterruptedException e ){}
88:     }
89: }
90:
```

Figura 6.3: Programa *NestedMonitor.java*

Definido : 60;
Usado : 61.

Ao analisar o fluxo de controle do programa da Figura 6.3 e as definições e usos de cada objeto, é criado o DFO (Figura 6.4). Comparando o DFO do programa com *deadlock* (Figura 6.2) com o DFO quando não há o *deadlock*, percebemos que eles são iguais. Conseqüentemente, os pares *du* identificados também são os mesmos:

- *buf*: (57, 58), (57, 60), (57, 73), (57, 86), (73, 86), (86, 73);
- *full*: (33, 40), (33, 45), (40, 45), (45, 40);
- *empty*: (34, 39), (34, 46), (39, 46), (46, 39);
- *p*: (58, 59);
- *c*: (60, 61).

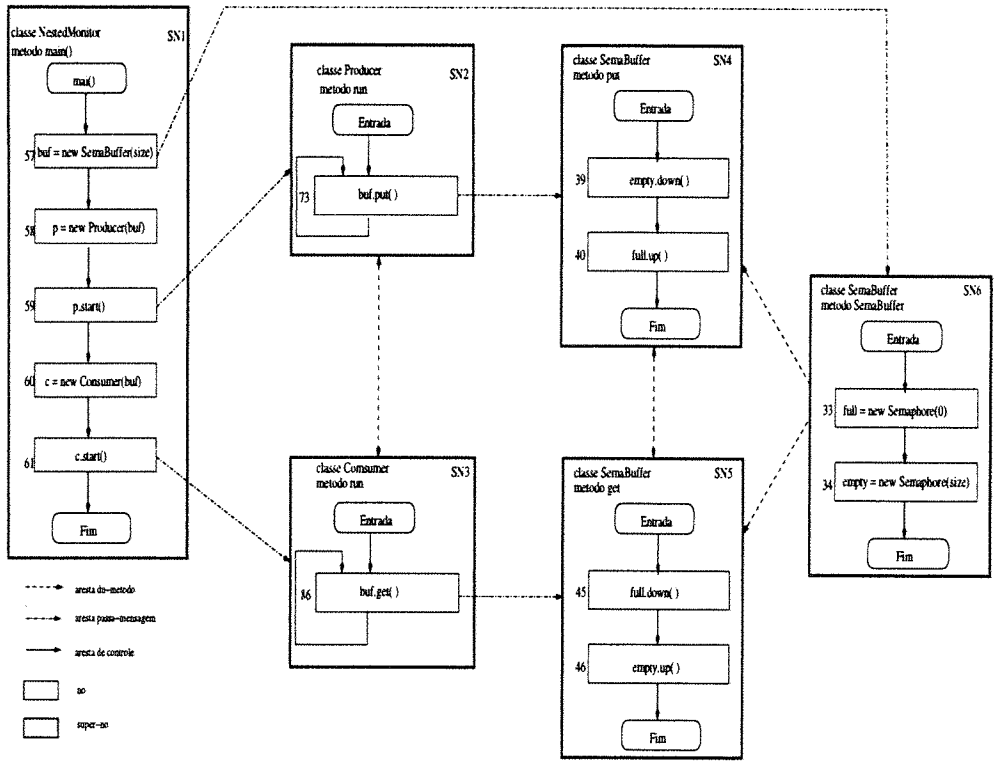


Figura 6.4: DFO do Programa *NestedMonitor.java*

Como os pares são os mesmos, os casos de teste que precisam ser exercitados para este programa também são os mesmos:

- **Caso 1**: 57, 33, 34, 58, 59, 60, 60, 73, 39, 40, 86, 45, 46;
- **Caso 2**: 57, 33, 34, 58, 59, 60, 61, 86, 45, 46, 73, 39, 40.

Após estas identificações, usamos a *Bandera* para verificar a propriedade de *deadlock*. Ao verificar a propriedade, a ferramenta não nos mostra um contra-exemplo e sim que a propriedade foi verificada com sucesso para o programa em questão. Com base nesta resposta, podemos dizer que este programa nunca entra em *deadlock*.

No caso da não violação da propriedade, a análise deve ser feita sobre os estados da linguagem *PROMELA* percorridos pelo Spin. Este verificador de modelos nos mostra os estados que não foram percorridos e com isto é possível saber quais estados foram

percorridos. Contudo, não é possível saber em qual seqüência os estados foram percorridos. É possível saber também quantos estados foram percorridos uma única vez e quantos foram percorridos mais de uma vez.

Como foi dito no Capítulo 5, a análise para este caso é manual e trabalhosa, pois é feita sobre a nossa comparação entre o código em *PROMELA* e o programa Java. Com o intuito de diminuir a explosão de estados e minimizar o trabalho manual, a técnica de abstração de dados foi utilizada com sucesso, a melhor abstração aplicada para este caso foi a *ranger04* (Veja na Seção 3.4.2). Por outro lado, a técnica de fatiamento não foi utilizada pelo fato de remover instruções do programa que são relevantes para análise do critério de teste. Mesmo utilizando a abstração de dados, para este caso, o número de estados percorridos manteve-se constante de 147.

Devido à falta de informação sobre quais seqüências de estados foram percorridos, a análise foi feita sobre uma estimativa do número mínimo de requisitos que deveriam ser exercitados para que os estados percorridos fossem exercitados pelo menos uma vez. Através do exemplo do objeto *full*, é possível identificarmos todas as definições e usos exercitados deste objeto, mas não conhecemos qual ou quais seqüências de caminhos foram percorridas. Com isso, podem existir as seguintes possibilidades de caminhos e, conseqüentemente, os possível pares *du*:

- 33, 40, 45 - (33, 40), (40, 45).
- 33, 45, 40 - (33, 45), (45, 40).

Com isto podemos afirmar que no mínimo 2 pares deste objeto foram exercitados pelo menos uma vez. Para os outros objetos temos:

- *buf*: (57, 58), (57, 60) e no mínimo mais 2 pares;
- *full*: no mínimo 2 pares;
- *empty*: no mínimo 2 pares;
- *p*: (58, 59);
- *c*: (60, 61).

Podemos dizer que nenhum dos dois casos de teste gerados foi totalmente exercitado, mas podemos perceber que parte dos caminhos em comum entre os dois casos foram exercitados. Veja a seguir:

- **Caso 1**: 57, 33, 34, 58, 59, 60, 61.
- **Caso 2**: 57, 33, 34, 58, 59, 60, 61.

6.3 Análise Complementar

	Objetos	Métodos	Cobertura dos pares *du*						*deadlock*	Mecanismo de bloqueio
			NP[a]	% TO[b]	NPE[c]	% CO[d]	% CTO [e]	%CT[f]		
Com *deadlock*										
sinc	*buf*	*put*(s) *get*(s)	4	25%	2	50%	12,5%	31,25%	Ocorre na est. *while*, quando monitor mais interno é liberado e o monitor mais externo não é liberado	Monitores Aninhados
	full	*up* (s) *down*(s)	4	25%	2	50%	12,5%			
	empty	*up* (s) *down*(s)	4	25%	1	6,25%	6,25%			
ñsinc	*buf*	-	2	12,5%	2	100%	12,5%	25%		
	p	*run*	1	6,25%	1	100%	6,25%			
	c	run	1	6,25%	1	100%	6,25%			
			16		9			56,25%		
Sem *deadlock*										
sinc	*full*	*up* (s) *down*(s)	4	25%	2	50%	12,5%	25%		
	empty	*up* (s) *down*(s)	4	25%	2	50%	12,5%			
ñsinc	*buf*	*put* *put*	6	37,5%	4	66,67%	25%	37,5%		
	p	*run*	1	6,25%	1	100%	6,25%			
	c	*run*	1	6,25%	1	100%	6,25%			
			16		10			62,5%		

Tabela 6.1: Cobertura do Programa *NestedMonitor.java*

[a]Números de pares *du* de cada objeto
[b]Percentagem de pares *du* do objeto em relação ao total de pares *du*
[c]Número de pares exercitados de cada objeto
[d]Percentual de cobertura de pares *du* do objeto em relação aos pares deste objeto
[e]Percentual de cobertura de pares *du* do objeto em relação ao total de pares *du*
[f]Percentual de cobertura dos pares *du* em relação ao total de pares

A Tabela 6.1 nos mostra os resultados obtidos através das análises do programa com e sem *deadlock*. Primeiramente comentamos a parte da tabela em que o programa entra em *deadlock*. Para este caso, percebemos que dos 16 pares *du* identificados 9 foram exercitados. Isto significa que em termos percentuais 56,25% destes pares foram exercitados.

Também podemos notar que a maioria dos pares *du* são sincronizados. Devido a presença de *Thread* e principalmente de monitores aninhados, a proporção de pares sincronizados exercitados é menor do que os pares não sincronizados. Observando a Tabela 6.1, temos que 31,25% dos pares *du* sincronizados foram exercitados, o que está

um pouco abaixo da metade, pois a porcentagem de pares sincronizados identificados é de 75%. Entretando, 25% dos pares não sincronizados foram exercitados, o que corresponde à porcentagem de todos os pares não sincronizados.

Por outro lado, percebemos que no mínimo 62,5% dos pares *du* para o programa sem *deadlock* foram exercitados. Uma das dificuldades desta análise é a precisão, pois como foi discutido anteriormente, trabalhamos com uma estimativa mínima. Devido à eliminação de monitores aninhados, podemos perceber que a quantidade de pares sincronizados diminuiu e, conseqüentemente, a proporção de pares sincronizados e não sincronizados que deveriam ser exercitados ficou na faixa de 50% para cada um. Comparando os pares sincronizados e não sincronizados que foram exercitados, percebemos que a quantidade de pares exercitados estão próximos.

Ao analisar este programa de uma forma geral, percebemos que a análise do programa com *deadlock* é complementar à análise do programa sem *deadlock*. Para a análise do programa com *deadlock* temos certeza do que realmente foi exercitado e isto é relevante para a conclusão da atividade de teste. Devido à imprecisão da informação na análise de programas sem *deadlock*, podemos dizer que para este programa o nível de cobertura é no mínimo de 62,5%, mas tal cobertura pode ser ainda maior.

Portanto, através deste exemplo, concluimos que a verificação formal de propriedades auxilia a atividade de teste. É possível percebermos isto através da quantidade de requisitos que foram exercitados e da diminuição significativa dos casos de teste que ainda precisam ser exercitados.

Capítulo 7

Conclusão

Neste trabalho apresentamos a idéia de como uma ferramenta de verificação formal pode auxiliar a atividade de teste de programas concorrentes Java. Com base nos critérios de teste baseados no fluxo de objetos para programas orientados a objetos, o critério de todos os pares definição-uso de Chen e Kao foi adotado para definir e selecionar os requisitos de suficiência a serem atingidos para a realização e finalização da atividade de teste. Além disso, a ferramenta de verificação formal, *Bandera*, foi escolhida para auxiliar quais requisitos do critério são exercitados quando a propriedade ausência de *deadlock* é verificada.

O critério de todos os pares *du* foi escolhido entre os principais critérios para programas orientados a objetos: critérios baseados no fluxo de objeto e critérios baseados no estado de objeto . Esse critério tem apresentado bons resultados na revelação de falhas nos sistemas OO.

A fim de analisar o nível de cobertura do critério adotado, estudamos algumas ferramentas de verificação formal de modelos para programas Java (*JCAT*, *Java PathFinder 1 (JPF1)*, *Java PathFinder 2 (JPF2)* e *Bandera*). O estudo foi realizado com o intuito de encontrar uma ferramenta que oferecesse os melhores recursos para auxiliar a atividade de teste, tais como técnicas de redução do modelo e geração de contra-exemplos refletidos no código Java. A ferramenta escolhida foi a *Bandera*.

7.1 Resultados Obtidos

Inicialmente analisamos 10 programas com *deadlock*. Com o intuito de melhorar o entendimento do presente trabalho, os 10 programas analisados foram renomeados ao eliminarmos o problema de *deadlock*. Como nesta Seção estão apresentados os resultados gerais desses programas (com e sem *deadlock*), os nomes originais dos programas serão mantidos.

Como vimos nos Capítulos anteriores, para fazer a análise é necessário identificar os requisitos a serem testados. Para realizar tal tarefa, é preciso conhecer todas as definições e usos de cada objeto e construir o diagrama de fluxo do objeto de cada programa. Toda esta tarefa é feita manualmente e, à medida que estes requisitos são identificados, alguns erros e/ou inconsistências podem aparecer, como por exemplo esquecer de retratar um par *du* ou retratar um par que não existe e com o intuito de resolver este tipo de problema as identificações eram refeitas.

Inicialmente, a análise foi feita sobre um conjunto de programas concorrentes Java que violam a propriedade de *deadlock*. Para estes programas foram identificados os requisitos as serem testados e, conseqüentemente, através dos contra-exemplos gerados quando a propriedade é violada, foi analisado o nível de cobertura dos requisitos. Para este conjunto de programas, o nível de cobertura foi satisfatório; a maioria dos programas teve um valor acima de 50% (Tabela 7.1).

Geralmente, para os programas que implementam monitores aninhados, não tiveram um nível de cobertura tão bom quanto os programas que implementam apenas semáforos. No caso deste trabalho, não foi realizado um estudo criterioso, com provas matemáticas, da relação entre estes mecanismos de bloqueios e o critério de teste adotado. Esta conclusão foi feita sobre os resultados obtidos da análise destes programas.

Além disso, a análise foi feita sobre o mesmo conjunto de programas concorrentes eliminando-se o *deadlock* de cada um. Para estes programas, também foram identificados os requisitos que precisam ser testados para o critério adotado e analisado o nível de cobertura destes requisitos. Como no caso quando a propriedade é satisfeita a *Bandera* não gera um contra-exemplo, a cobertura do critério foi feita sobre os estados percorridos em *PROMELA* quando a verificação é realizada. A análise é feita sobre o

mapeamento destes estados para cada instrução do programa Java.

Como não é possível saber quais caminhos foram percorridos e sim apenas quais instruções foram exercitadas, é feita uma estimativa mínima de cobertura do critério. Para este conjunto de programas sem *deadlock*, a porcentagem de cobertura mínima também demonstrou ser satisfatória, a maioria superior a 50% (Tabela 7.1).

Além da dificuldade em sabermos quais seqüências de caminhos foram percorridos, há também a questão do extenso código gerado em *PROMELA*. Mesmo aplicando as técnicas de redução de modelo, ao compararmos estes simples programas, o trabalho manual de mapear os estados de *PROMELA* para Java torna-se dispendioso à medida que os programas Java aumentam de tamanho e de complexidade.

Outro fato que também deve ser ressaltado é a probabilidade de erros ao mapear e analisar estados. Como é um trabalho manual, percebemos algumas inconsistências após o mapeamento e a forma de resolver isto era refazer o mapeamento. Além de ser um trabalho dispendioso, estas dificuldades nos levaram a concluir que dependendo da extensão do programa Java, a análise deste programa quando a propriedade não é violada pode ser inviável.

Mesmo com a presença de tais dificuldades, a análise geral foi realizada sobre este conjunto de programas comparando os dados obtidos nas duas análises em separado. Ao realizar esta junção, podemos perceber que o nível de cobertura da maioria dos programas teve um acréscimo significativo, com isto podemos dizer que uma análise é complementar à outra. É relevante lembrar que este acréscimo pode vir a ser maior, pois a análise feita nos programas sem *deadlock* é uma estimativa mínima.

Inicialmente, a análise geral foi feita sobre os programas com *deadlock* e logo em seguida os mesmos programas sem o *deadlock*. Percebemos (Tabela 7.1) que a média de cobertura dos pares *du* foi de 14,2, com um desvio padrão de 4,66. Através destas informações, podemos dizer que para qualquer conjunto de dados, aproximadamente 68% deles estão no intervalo de [9,54 a 18,86]. Este intervalo significa a distância de 2 desvio padrão da média, i.e. $\overline{x} - 2s$ e $\overline{x} + 2s$, onde $\overline{x}$ e $\mathbf{s}$ é a média e o desvio padrão, repectvamente [31].Com isto, podemos concluir que a maioria dos pares exercitados encontram se dentro deste intevalo que corresponde a aproximadamente 68%.

Programas	Quantidade de pares *du*					Porcentagem de cobertura
	identificados	ECD[a]	ESD[b]	EC[c]	ET[d]	
Dead.java	28	7	13	7	13	46,43%
Phil33.java	24	19	12	12	19	79,17%%
START.java	20	11	14	11	14	70%
Bounded.java	22	18	20	18	20	90,90%
PipeInteger.java	16	16	8	8	16	100%
NestedMonitor.java	16	9	10	7	12	75%
MonitorDead.java	13	6	10	6	10	76,92%
PCSimple.java	11	8	8	7	9	81,81%
MainPC.java	8	8	7	7	8	100%
StrictClockS.java	28	21	15	15	21	75%
Média	-	12,2	11,7	-	14,2	-
Desvio Padrão	-	5,71	3,97	-	4,66	-
Média Percentual	-	65,59%	62,9%	-	76,34%	76,34%

Tabela 7.1: Análise dos Requisitos (Programas com e sem *Deadlock*)

[a]Exercitados com *deadlock*
[b]Exercitados sem *deadlock*
[c]Exercitados em comum
[d]Exercitados no total

Também podemos concluir que uma análise é complementar à outra. A Tabela 7.1 pode nos confirmar tal afirmação. A porcentagem de cobertura para os programas com *deadlock* foi de 65,59% enquanto para os programas sem *deadlock* foi de 62,9%. Juntando os pares analisados para os casos, percebemos que o percentual subiu para 76,34%. Isto é possível porque pares que não foram exercitados quando o programa tinha *deadlock* foram exercitados quando a propriedade não foi violada e vice-versa.

A veracidade da informação anterior pode ser confirmada se observarmos a análise feita nos programas *NestedMonitor.java* e *PCSimple.java*. Para estes dois programas, a quantidade de pares *du* exercitados no total é maior que a quantidade para cada caso específico. Por isso, é possível afirmar que alguns pares que não foram exercitados quando a propriedade foi violada, foram exercitados quando esta mesma propriedade foi verificada e vice-versa.

Também podemos notar, através da Tabela 7.1, que os programas *MainPC.java* e *PipeInteger.java* (linhas 8 e 5, respectivamente) tiveram todos os pares *du* exercitados quando o programa tinha *deadlock*.

Além disso, também notamos que apenas um programa teve a porcentagem de

cobertura inferior a 50%, enquanto os outros 9 tiveram a porcentagem igual ou acima de 70%. Com isto podemos concluir que em média 76,34% dos pares foram exercitados, ressaltando que estas porcentagens podem ser maiores, pois a análise geral sobre os programas também está vinculada à estimativa feita sobre os programas sem *deadlock*.

Com isto, podemos afirmar que o uso da ferramenta *Bandera* é de grande ajuda na análise de cobertura do critério de teste adotado. Além disso, também é possível sabermos o que já foi testado, quando a *Bandera* verificou a propriedade de *deadlock*, e conseqüentemente, diminui a trabalho do *testador* que precisa se preocupar em testar apenas os requisitos que não foram exercitados.

7.2 Contribuições e Trabalhos Futuros

A principal contribuição deste trabalho está na utilização de uma ferramenta de verifação formal (*Bandera*) aplicada a uma linguagem comum ao usuário (Java, neste caso), para auxiliar o teste de programas desta linguagem. A maioria dos trabalhos estudados envolviam vericadores de modelos que auxiliam o teste de especificação ou de requisitos. Neste caso, é necessário que o sistema esteja especificado em uma linguagem formal (linguagem não familiar ao usuário) específica de um verificador. Outros trabalhos, utilizam a verificação formal de programas implementados em C++ ou Java para verificar propriedades e estudos sobre alternativas para facilitar e melhorar a verificação de programas escritos em tais linguagens.

Imaginamos que verificação formal de sistemas como auxílio a atividade de teste é uma área para muitas pesquisas, e que a *Bandera* pode ser uma ferramenta valiosa neste auxílio. Como trabalho futuro, está sendo desenvolvida uma ferramenta para a geração automática dos diagramas de fluxo de controle e de todos os pares *du* passíveis de execução. Esta ferramenta está sendo desenvolvida para auxiliar a identificação dos requisitos dos critérios baseados em fluxo de objetos propostos por Chen e Kao. Com o auxílio desta ferramenta, imaginamos que as inconsistências e os erros gerados devido ao trabalho manual diminua significativamente.

Pretendemos também estudar como identificar as seqüências de caminhos que foram percorridas antes que a ferramenta gere um contra-exemplo. Através deste estudo,

podemos desenvolver uma ferramenta que trabalhe junto com a *Bandera* para identificar automaticamente estes caminhos. Dessa forma, ao utilizarmos estas ferramentas para auxiliar a cobertura de critérios de teste, deduzimos que a quantidade de requisitos testados aumente e o trabalho do *testador* diminua.

Além disso, um estudo para identificar quais caminhos percorridos quando a propriedade é satisfeita, também pode aumentar o número de requisitos testado e a precisão de cobertura dos mesmos. Sobre este estudo poderemos desenvolver uma ferramenta que automatize o mapeamento dos estados em *PROMELA* para o código Java.

Um outro trabalho que podemos propor é realização de um estudo criterioso da relação de cobertura dos pares *du* e a verificação da propriedade de *deadlock* para programas com características específicas, como programas que implementam os mecanismos de bloqueio (semáforo e monitores). Se conseguirmos provar que existe uma relação do critério e da propriedade analisado com algum mecanismo de bloqueio, isto nos auxiliaria significativamente na análise dos programas com tais características, pois nestes casos os resultados seriam precisos e concisos.

Também podemos propor outras extensões:

- realizar um estudo comparativo entre os critérios baseados em fluxo de objetos e os critérios baseados em estado de objetos, ao utilizar a verificação de propriedades;
- utilizar a verificação de programas Java para auxiliar a geração de casos e/ou dados de teste. Neste estudo, é interessente ressaltar as vantagens e desvantagens e a viabilidade de tal auxílio;
- estudar a verificação de outras propriedades e analisar uma relação entre elas e a cobertura do critério de todos os pares *du*;
- estudar a possibilidade de empregarmos tal estudo para um novo conjunto de programas Java com características não concorrentes;
- realizar um estudo sobre o uso da ferramenta *Bandera* para analisarmos outros critérios de teste e realizarmos uma comparação entre a cobertura de cada um;
- desenvolver uma ferramenta que integre com a *Bandera* e que tenha como objetivo informar quais requisitos do critério de todos os pares *du* foram exercitados

quando a verificação de uma propriedade for realizada.

Referências Bibliográficas

[1] R. Alexander and J.Offutt. Criteria for testing polymorphic relationships, 2000.

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specications. In *Proc. 2nd IEEE Intern. Conf. on Formal Engineering Methods (ICFEM'98), Brisbane, Australia.*, pages 46–54. IEEE Computer Society, December 1998.

[3] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceeding of Fourth IEEE Ineternational High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.

[4] Ellen Francine Barbosa, Auri Marcelo Rizzo Vincenzi, and José Carlos Maldonado. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C. In *XII Simpósio Brasileiro de Engenharia de Software*, pages 103–120, Outubro 1998.

[5] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - a second generation of a Java model checker. Workshop on Advances in Verification, July 2000.

[6] Paulo Marcos Siqueira Bueno. Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software. Master's thesis, Faculdade de Engenharia Elétrica e de Computação da Universidade de Campinas, 1999.

[7] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[8] John R. Callahan, Steve M. Easterbrook, and Todd L. Montgomery. Generating test oracles via model checking, 1998. Unpublished, working paper. Available at urlhttp://research.ivv.nasa.gov/projects/blurbs/FormalTesting.html.

[9] Marcos Lordello Chaim. *Depuração de Programas Baseada em Informação de Teste Estrutural.* PhD thesis, Faculdade de Engenharia Elétrica e de Computação da Universidade de Campinas, Novembro 2001.

[10] R. Chatterjee and B. Ryder. Data-flow-based testing of object-oriented libraries, 1999.

[11] Mei-Hwa Chen and Howard M. Kao. Testing object-oriented programs - an integrated approach. In *Proceedings of the Tenth International Symposium on Software Reliability Engineering.* IEEE Computer Society Press, 1999.

[12] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[13] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[14] C. Demartini, R. Iosif, and R. Sisto. A Deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, July 1999.

[15] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.

[16] David Déharbe, Anamaria M. Moreira, Leila Ribeiro, and Vanderlei M. Rodrigues. Introdução a Métodos Formais: Especificação, Semântica e Verificação de sistemas concorrentes. *RITA - Edição especial: Métodos Formais*, 7(1):7–48, Setembro 2000.

[17] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[18] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Joint 7th European Software Eng. Conference and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Eng.*, pages 146–162, September 1999. Toulouse, France.

[19] G.J.Myers. *The Art of Software Testing.* J. Wiley, 5^a edition, 1979.

[20] Mary Jean Harrold, John D. McGregor, and Kevin J. Fitzpatrick. Incremental testing of object-oriented class structures. In *International Conference on Software Engineering*, pages 68–80. IEEE Computer Society Press, 1992.

[21] Mary Jean Harrold and Gregg Rothermel. Performing dataflow testing on classes. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 154–163, 1994.

[22] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent Java software. *Lecture Notes in Computer Science*, 2154:39–??, June 2001.

[23] K. Havelund and T. Pressburger. Model checking java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(3), April 2000.

[24] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(3):279–294, May 1997.

[25] Pei Hsia, Xiaolin Li, and David C. Kung. Augmenting data flow criteria for class testing. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 9. IBM Press, 1997.

[26] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the applicability of traditional test adequacy criteria for object oriented programs. In *Proceedings of the ObjectDays 2000*, October 2000. http://www-users.cs.york.ac.uk/ jac/.

[27] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. In *In the Proceedings of the Mutation 2000*, October 2000. http://www-users.cs.york.ac.uk/ jac/.

[28] D. Kung, N. Suchak, P. Hsia, Y. Toyoshima, and C. Chen. On object state testing. In *Proc. of COMPSAC'94*. IEEE Computer Society Press, 1994.

[29] D. Kung, N. Suchak, P. Hsia, Y. Toyoshima, and C. Chen. Object state testing for object-oriented programs. In *Proc. of COMPSAC'95*. IEEE Computer Society Press, 1995.

[30] Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst.*, 13(4):365–398, 1995.

[31] Pedro A. Morettin and Wilton de O. Bussab. *Estatistica Básica.* Saraiva, 5ª edition, 2002.

[32] Romulo Oliveira, Avelino Zorzo, and Antônio M. P. Barcellos. Sistemas operacionais - notas de aula. http://www.ulbra.tche.br/ elgios/fundamentos/.

[33] C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible counter-examples when model checking Java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems,*, April 2001.

[34] Shari Lawrence Pfleeger. *Software Engineering: the production of quality software.* Prentice-Hall, 1991.

[35] Ivete Martins Pinto and Ana Maria de Alencar Price. Um sistema de apoio ao teste de programas orientados a objetos com uma abordagem reflexiva. In *XII Simpósio Brasileiro de Engenharia de Software*, pages 87–102, Outubro 1998. Maringa-PR.

[36] R. S. Pressman. *Software Engineering.* McGraw-Hill, 1997.

[37] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1996.

[38] Nigel Tracey, John Clark, John McDermid, and Keith Mander. Integrating safety analysis with automatic test-data generation for software safety verification. In *In the proceedings of 17th Internation System Safety Conference*, pages 128–137. System Safety Society, August 1999.

[39] C. D. Turner and D. J. Robson. The Testing of Object-Oriented Programs. Technical Report TR-13/92, Durham, GB, 1993.

[40] W. Visser, S. Park, and J. Penix. Using predicate abstraction to model check object-oriented programs. In *3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000.

Apêndice A

Programas com *Deadlock*: Diagramas e Contra-exemplos

A.1 Programa *Phill33.java*

```
 1: class Fork{
 2:     private boolean is_free;
 3:
 4:     Fork(){
 5:         is_free = true;
 6:     }
 7:
 8:     public synchronized void take(){
 9:         while (!is_free)
10:             try {
11:                 wait();
12:             } catch (InterruptedException e) {return;}
13:
14:         is_free = false;
15:     }
16:
17:     public synchronized void leave(){
18:         is_free = true;
19:         notify();
20:     }
21: }
22:
23: class Philosopher extends Thread{
24:     private Fork left;
25:     private Fork right;
26:
27:     Philosopher(Fork f1, Fork f2){
28:         left = f1;
29:         right = f2;
30:     }
31:
32:     public void eat(){
33:         left.take();
34:         right.take();
35:     }
36:
37:     public void think(){
38:         left.leave();
39:         right.leave();
40:     }
41:
42:     public void run(){
43:         while (true)
44:             {
45:                 eat();
46:                 think();
47:             }
48:     }
49: }
50:
51: public class Phil33{
52:     public static void main(String[] args){
53:         Fork f1;
54:         Fork f2;
55:         Philosopher p1;
56:         Philosopher p2;
57:         f1 = new Fork();
58:         f2 = new Fork();
59:         p1 = new Philosopher(f1, f2);
60:         p2 = new Philosopher(f2, f1);
61:         p1.start();
62:         p2.start();
63:     }
64: }
```

Figura A.1: Programa *Phil33.java*

Diagrama do fluxo do objeto construído:

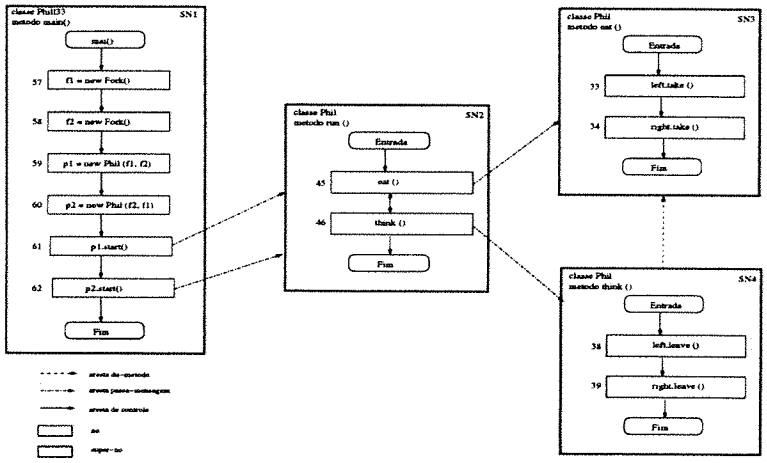


Figura A.2: DFO do Programa *Phil33.java*

Contra-exemplo aleatório gerado pela *Bandera*:

```
      Contra-exemplo Aleatório
Phil33      Philosopher        Fork
57 ---------- | ------------> 5
| <---------- | ------------- |
58 ---------- | ------------> 5
| <---------- | ------------- |
59 ---------> 28                |
|             29                |
| <---------- |                 |
60 ---------> 28a               |
|             29a               |
| <---------- |                 |
61 ---------> 45                |
|             33 ------------> 9
|             | <------------ 14
|             34 ------------> 9
|             | <------------ 14
|             46                |
|             38 ------------> 18
|             | <------------ 19
|             39 ------------> 18
|             | <------------ 19
|             45                |
|             33 ------------> 9
|             | <------------ 14
|             34 ------------> 9
```

```
|              | <------------ 14
| <---------- 46               |
62 ---------> 45a              |
|             33a ----------> 9
|             | <------------ 11
|             38 ----------> 18
|             | <------------ 19
|             39 ----------> 18
|             | <------------ 19
|             | ------------> 9
|             | <------------ 14
|             34a ----------> 9
|             | <------------ 14
|             46a              |
|             38a ----------> 18
|             | <------------ 19
|             39a ----------> 18
|             | <------------ 19
|             45a              |
|             33a ----------> 9
|             | <------------ 14
|             34a ----------> 9
|             | <------------ 14
|             46a              |
|             45               |
|             38a ----------> 18
|             | <------------ 19
|             39a ----------> 18
|             | <------------ 19
|             45a              |
|             33a ----------> 9
|             | <------------ 14
|             34a ----------> 9
|             | <------------ |
|             33 ----------> |
|             | <------------ 14
|             46a              |
|             38a ----------> 18
|             | <------------ 19
|             | ------------> 9
|             | <------------ 11
|             39a ----------> 18
|             | <------------ 19
|             45a              |
|             33a ----------> 9
|             | <------------ 14
|             | ------------> 9
|             | <------------ 14
|             34a ----------> 9
|             | <------------ 11*
|             34 ----------> 9
```

```
|              | <------------ 11*
```

Além disso, também temos o contra-exemplo de menor caminho

```
     Contra-exemplo de menor caminho
Phil33     Philosopher        Fork
57 ---------- | ------------> 5
| <---------- | ------------- |
58 ---------- | ------------> 5
| <---------- | ------------- |
59 ---------> 28               |
|             29               |
| <---------- |                |
60 ---------> 28a              |
|             29a              |
| <---------- |                |
61 ---------> 45               |
|             33 -----------> 9
|             | <------------ 14
|             34 -----------> 9
|             | <------------ 14
|             46               |
|             38 -----------> 18
|             | <------------ 19
|             39 -----------> 18
|             | <------------ 19
|             45               |
|             33 -----------> 9
| <---------- | ------------- 14
62 ---------> 45a              |
|             33a ----------> 9
|             | <------------ 14
|             34a ----------> 9
|             | <------------ 11*
|             34 -----------> 9
|             |                11*
```

A.2 Programa *PCSimple.java*

```
1: public class PCSimple{
2:     public static void main(String[] args){
3:         int numIntegers = 6;
4:         Buffer b = new Buffer(2);
5:         Producer p = new Producer(numIntegers, b);
6:         Consumer c = new Consumer(numIntegers, b);
7:         p.start();
8:         c.start();
9:     }
10: }
11:
12: class Producer extends Thread
13: {
14:     private int count;
15:     private Buffer buffer;
16:
17:     public Producer(int num, Buffer b){
18:         count = num;
19:         buffer = b;
20:     }
21:
22:     public void run(){
23:         for (int i=0; i < count; i++)
24:             buffer.put(i);
25:         buffer.put(-1);
26:     }
27: }
28:
29: class Consumer extends Thread{
30:     private int count;
31:     private Buffer buffer;
32:
33:     public Consumer(int num, Buffer b){
34:         count = num+1; //for that extra -1
35:         buffer = b;
36:     }
37:
38:     public void run(){
39:      int[] received = new int[count];
40:      for(int i=0; i<count; i++){
41:             received[i] = buffer.get();
42:       if(received[i] == -1){
43:         System.out.println("Consumer.run: got last number"+received[i]);
44:             return;
45:           }
46:       }
47:     }
48: }
49:
```

```
50: class Buffer{
51:     protected int size;
52:     protected int[] array;
53:     protected int putPtr;
54:     protected int getPtr;
55:     protected int usedSlots;
56:
57:     public Buffer(int b){
58:       size = b;
59:       array = new int[b];
60:       putPtr = 0;
61:       getPtr = 0;
62:       usedSlots = 0;
63:     }
64:
65:     public synchronized void put(int x){
66:         while(usedSlots == size)
67:             try{
68:                 System.out.println("producer wait");
69:                 wait();
70:             }catch(InterruptedException e){};
71:
72:         array[putPtr] = x;
73:         System.out.println("Buffer.put: put "+x);
74:         putPtr = (putPtr + 1)%size;
75:
76:         usedSlots++;
77:     }
78:
79:     public synchronized int get(){
80:         while(usedSlots == 0)
81:             try{
82:                 System.out.println("consumer wait");
83:                 wait();
84:             }catch(InterruptedException e){};
85:         int x = array[getPtr];
86:         System.out.println("Buffer.get: got "+x);
87:         if(x<0)
88:             return x;
89:         getPtr = (getPtr+1)%size;
90:         if(usedSlots == size) notify();
91:         usedSlots --;
92:         return x;
93:     }
94: }
```

Figura A.3: Programa *PCSimple.java*

Representação do DFO criado:

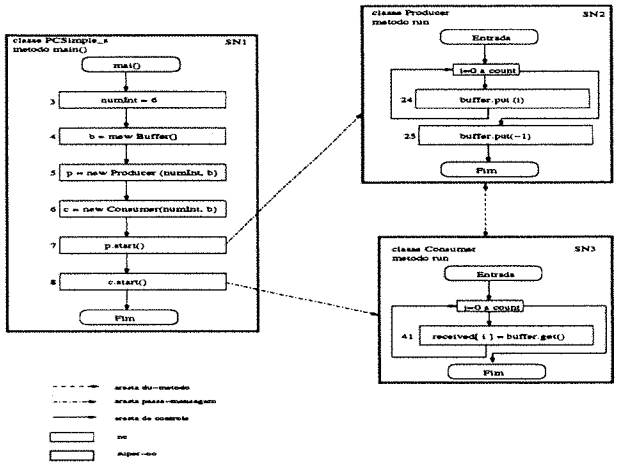
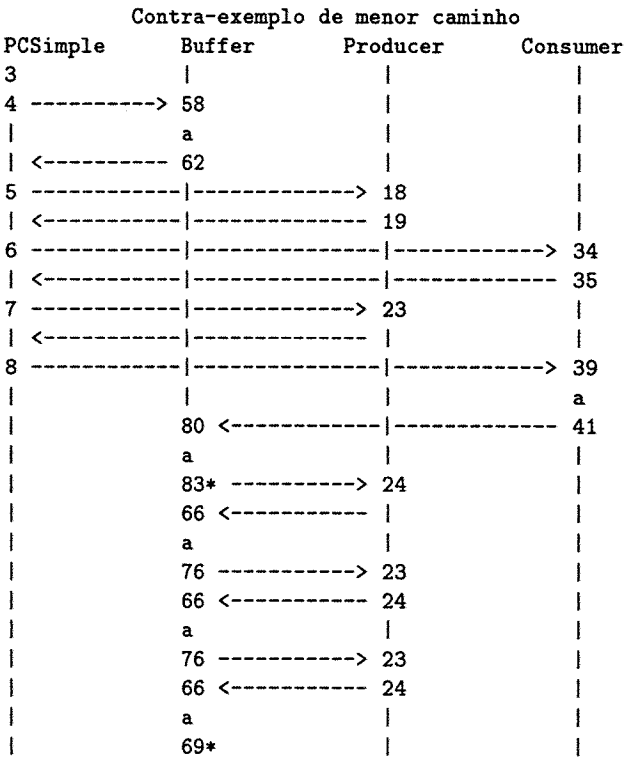
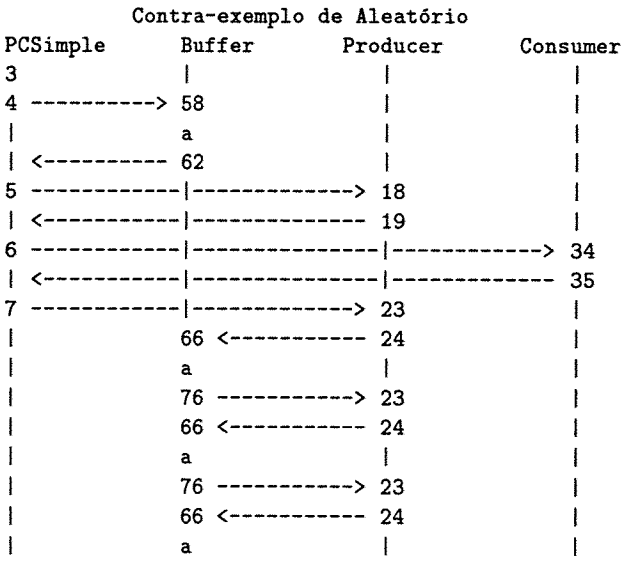


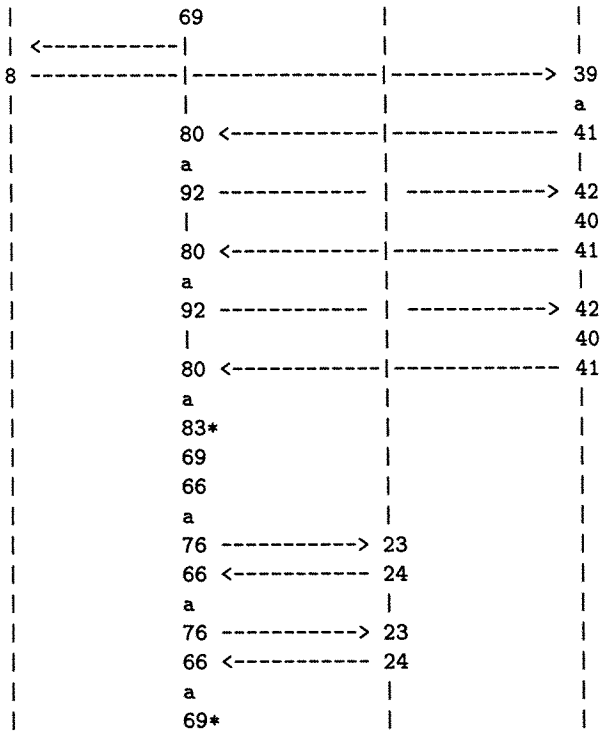
Figura A.4: DFO Programa *PCSimple.java*

Descrição do contra-exemplo de menor caminho gerado pela *Bandera*:



Também temos a representação do contra-exemplo aleatório:





A.3 Programa *PCMain.java*

```
1:  public class MainPC{
2:      public static void main(String args[]){
3:          My_Pool pool=new My_Pool();
4:          Prod p=new Prod(pool);
5:          Cons c=new Cons(pool,1);
6:          p.start();
7:          c.start();
8:      }
9:  }
10: class Cons extends Thread{
11:     My_Pool p;
12:     int _id;
13:
14:     public Cons(My_Pool pool, int id){
15:         p=pool;
16:         _id=id;
17:     }
18:
19:     public void run(){
20:         while(true){
21:             System.out.println(_id+"Tirando algo do pool");
22:             int val=p.get();
23:             System.out.println(_id+"Tirei: "+val);
24:         }
25:     }
26: }
27: class Prod extends Thread{
28:     My_Pool p;
29:     public Prod(My_Pool pool){
30:         p=pool;
31:     }
32:
33:     public void run(){
34:         while(true){
35:            System.out.println("Estou colocando algo...");
36:            p.put(1);
37:         }
38:     }
39: }
```

```
40: class My_Guard{
41:     int slot;
42:     int status;
43:
44:     public My_Guard(){
45:         slot=0;
46:         status=0;
47:     }
48:
49:     synchronized int get(){
50:         try{
51:             while(status==0) wait();
52:             status=0;
53:         }
54:         catch(Exception e){
55:             System.out.println("Erro no get...");
56:         }
57:         return slot;
58:     }
59:
60:     synchronized void put(int i){
61:         try{
62:             slot=i;
63:             status=1;
64:             notify();
65:         }
66:         catch(Exception e){
67:             System.out.println("Erro no put");
68:         }
69:     }
70: }
71: class My_Pool{
72:     My_Guard gd;
73:
74:     public My_Pool(){
75:         gd=new My_Guard();
76:     }
77:
78:     synchronized int get(){
79:         return gd.get();
80:     }
81:
82:     synchronized void put(int i){
83:         gd.put(i);
84:     }
85: }
```

Figura A.5: Programa *PCMain.java*

Diagrama do fluxo do objeto criado:

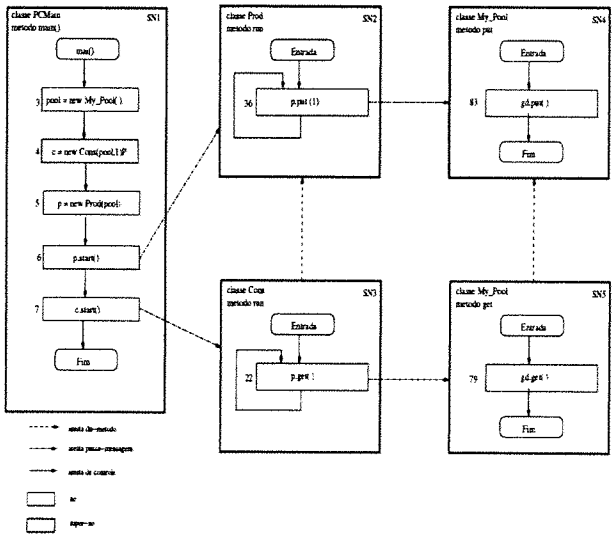
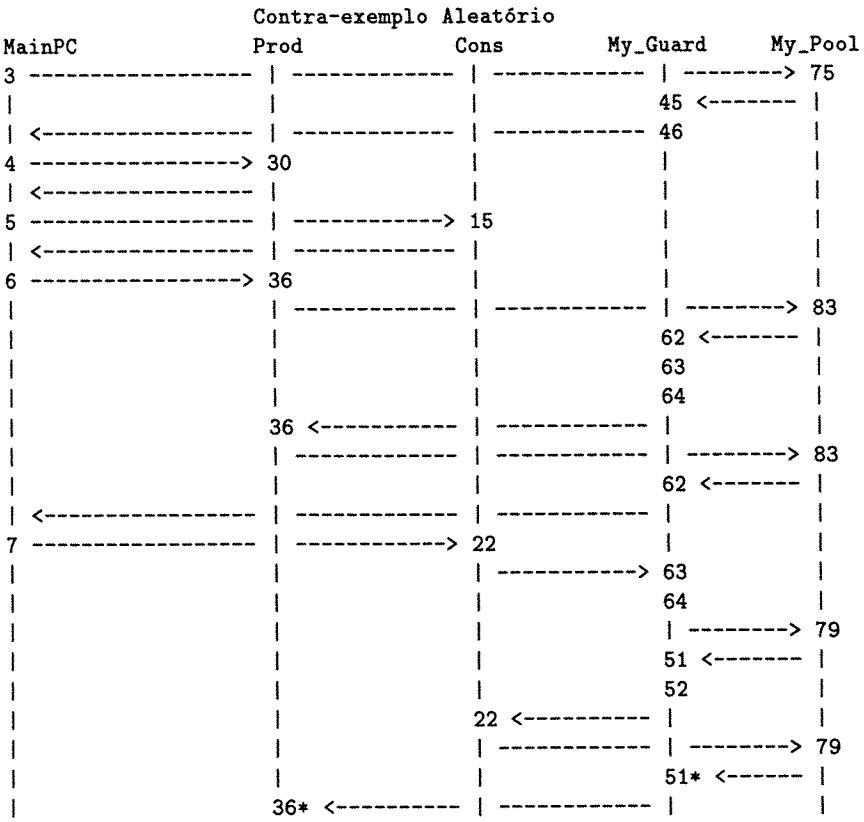
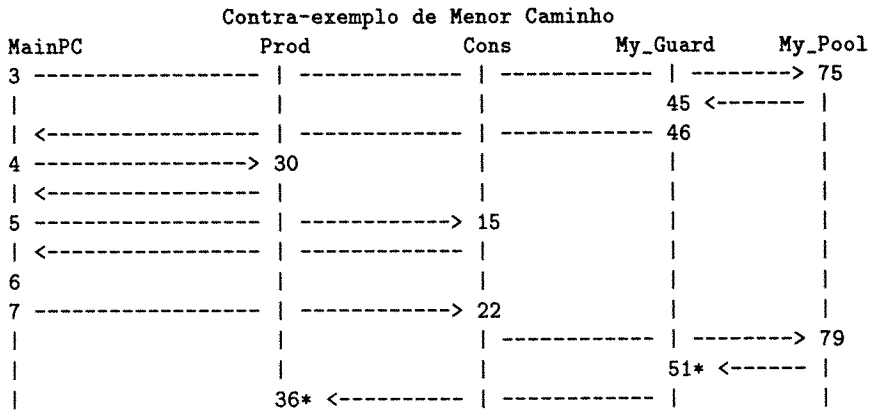


Figura A.6: DFO do Programa *PCMain.java*

Descrição do contra-exemplo aleatório gerado pela *Bandera*:



Além disso, retratamos o contra-exemplo de menor caminho:



A.4 Programa *NestedMonitor.java*

```
 1: public class NestedMonitor{
 2:     public static void main(String[] args) {
 3:         Pair p = new Pair();
 4:         Wrapper w = new Wrapper(p);
 5:         Thread1 t1 = new Thread1(w);
 6:         Thread2 t2 = new Thread2(w);
 7:
 8:         t1.start();
 9:         t2.start();
10:     }
11: }
12:
13: class Pair {
14:     private int hi = 0;
15:     private int lo = 0;
16:     public synchronized void inclo() {
17:         while (lo >= hi)
18:             try { wait(); } catch ( InterruptedException ex)
19:         lo++;
20:     }
21:     public synchronized void inchi() {
22:         hi++;
23:         notify();
24:     }
25: }
26:
27: class Wrapper {
28:     private Pair p;
29:     Wrapper(Pair x) {p = x;}
30:     public synchronized void inclo() { p.inclo(); }
31:     public synchronized void inchi() { p.inchi(); }
32: }
33:
34: class Thread1 extends Thread {
35:     private Wrapper w;
36:     Thread1(Wrapper x) {w = x;}
37:     public void run() {
38:         w.inclo();
39:     }
40: }
41:
42: class Thread2 extends Thread {
43:     private Wrapper w;
44:     Thread2(Wrapper x) {w = x;}
45:     public void run() {
46:         w.inchi();
47:     }
48: }
```

Figura A.7: Programa *NestedMonitor.java*

Representação do diagrama de fluxo de objetos:

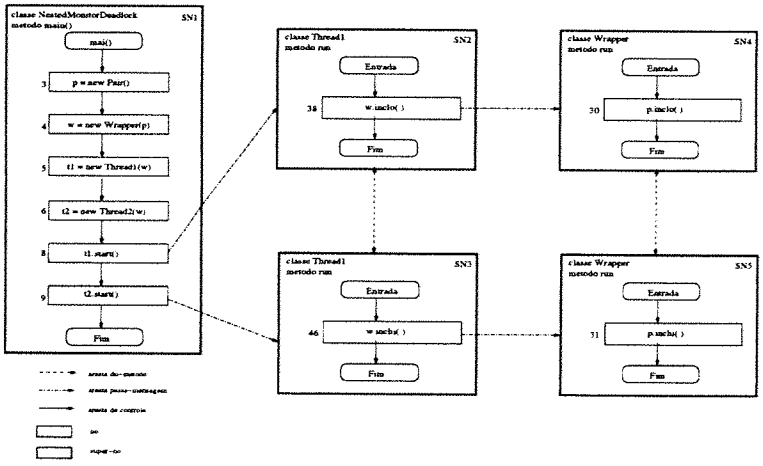
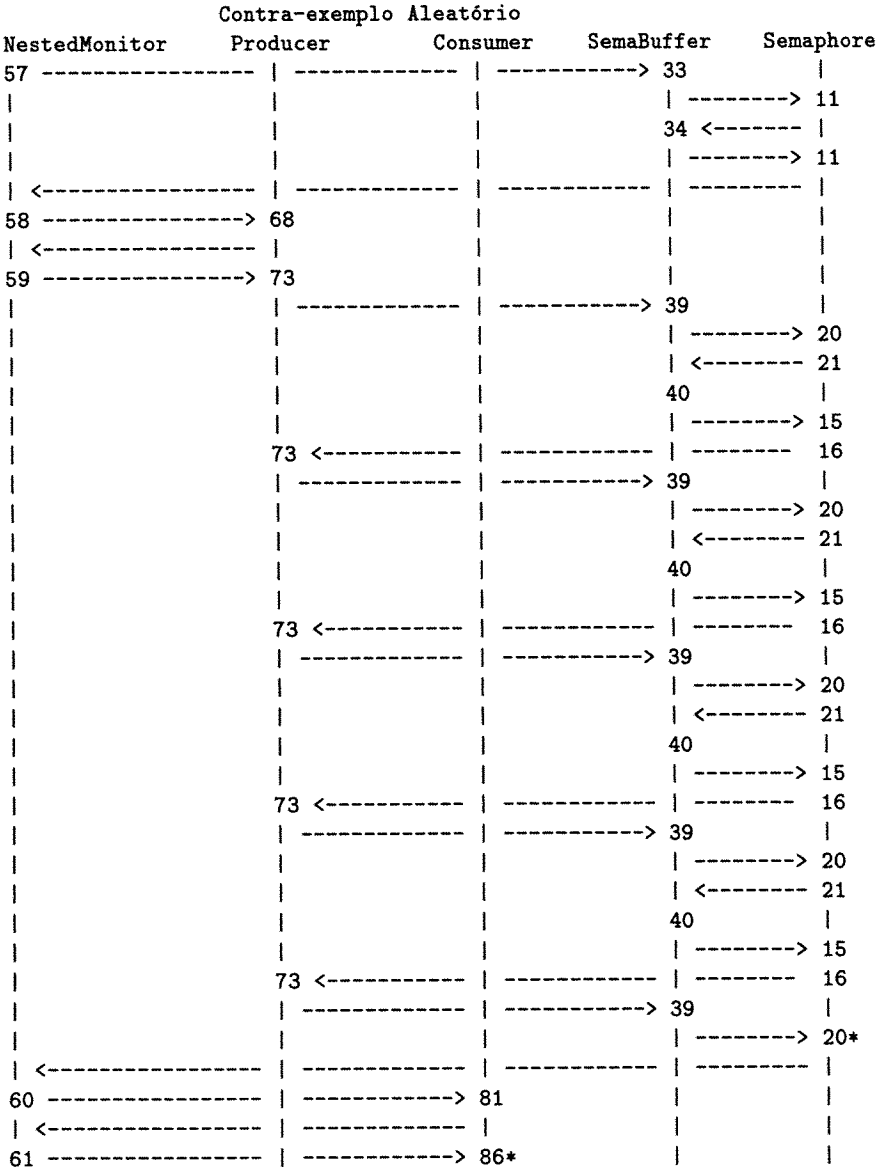
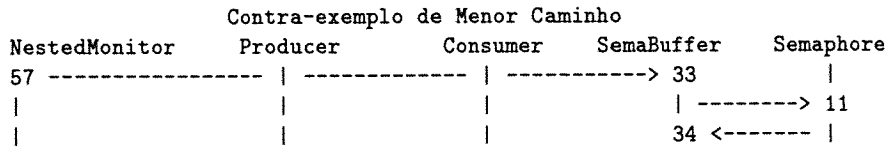


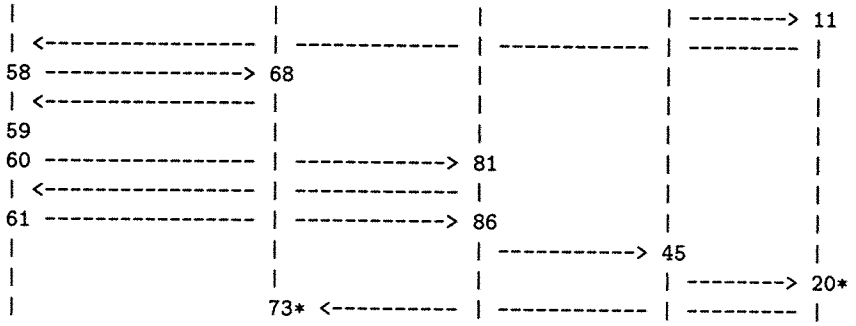
Figura A.8: DFO do Programa *NestedMonitor.java*

Contra-exemplo aleatório gerado pela ferramenta:



Também temos a representação do contra-exemplo de menor caminho:





A.5 Programa *Bounded.java*

```
 1: public class Bounded {
 2:     public static void main (String [] args) {
 3:         BoundedBuffer b1 = new BoundedBuffer(3);
 4:         BoundedBuffer b2 = new BoundedBuffer(3);
 5:         Object o1 = new Object();
 6:
 7:         b1.add(o1);
 8:         b1.add(o1);
 9:         InOut out1 = new InOut(b1,b2);
10:         InOut out2 = new InOut(b2,b1);
11:         out1.start();
12:         out2.start();
13:     }
14: }
15:
16: class BoundedBuffer {
17:     Object [] buffer;
18:     int bound, head, tail;
19:
20:     public BoundedBuffer(int b) {
21:       bound = b;
22:       buffer = new Object[bound];
23:       head = 0;
24:       tail = bound-1;
25:     }
26:
27:     public synchronized void add(Object o) {
28:         while ( tail == head )
29:             try { wait(); }
30:             catch ( InterruptedException ex) {}
31:         buffer[head] = o;
32:         head = (head+1) % bound;
33:
34:     }
35:
36:
37:      public synchronized Object take() {
38:         while ( isEmpty() )
39:             try { wait(); }
40:             catch ( InterruptedException ex) {}
41:         tail = (tail+1) % bound;
42:         notifyAll();
43:         return buffer[tail];
44:     }
45:
46:     public synchronized boolean isEmpty() {
47:         return head == ((tail+1) % bound); }
48: }
49:
50: class InOut extends Thread {
51:     BoundedBuffer in,out;
52:     public InOut(BoundedBuffer ins, BoundedBuffer outs
53:         in = ins;
54:         out = outs;
55:     }
56:     public void run() {
57:         Object tmp;
58:         while(true) {
59:             tmp = in.take();
60:             System.out.println("Take ");
61:             out.add(tmp);
62:             System.out.println("Put ");
63:         }
64:     }
65: }
```

Figura A.9: Programa *Bounded.java*

Ilustração do DFO:

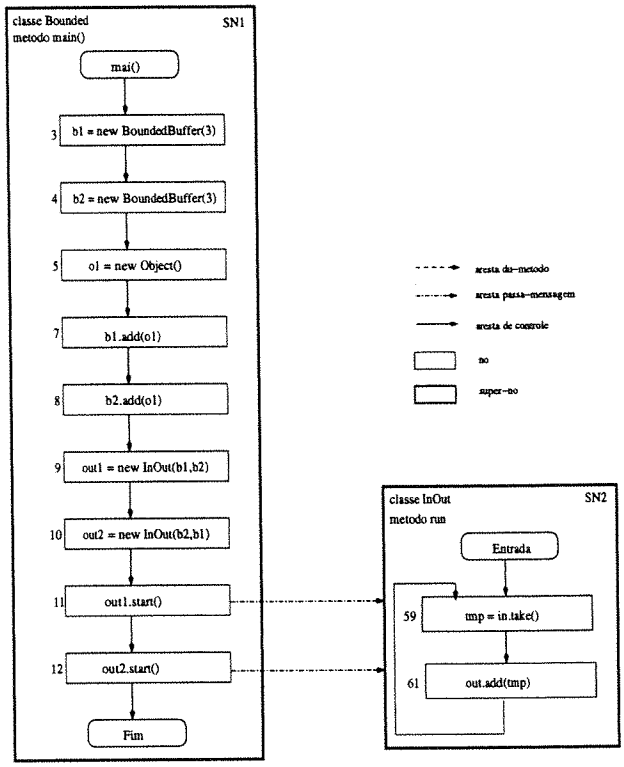
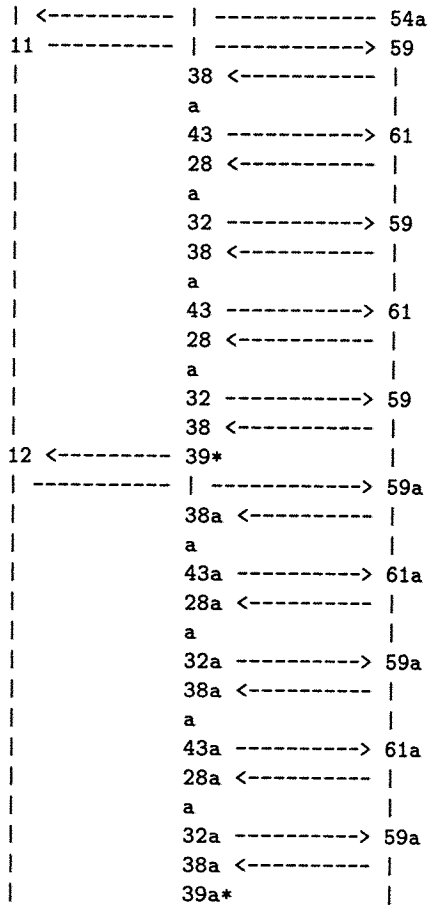


Figura A.10: DFO do Programa *Bounded.java*

Neste caso, o contra-exemplo de menor caminho e o aletório são identicos:

```
Bounded     BoundedBuffer      InOut
3 ----------> 21                 |
|             a                  |
| <---------- 24                 |
4 ----------> 21                 |
|             a                  |
5 <---------- 24                 |
7 ----------> 28                 |
|             a                  |
| <---------- 32                 |
8 ----------> 28                 |
|             a                  |
| <---------- 32                 |
9 ----------- | ------------> 53
| <---------- | ------------- 54
10 ---------- | ------------> 53a
```


54a
59
38
a
43
61
28
a
32
59
38
a
43
61
28
a
32
59
38
39*
59a
38a
a
43a
61a
28a
a
32a
59a
38a
a
43a
61a
28a
a
32a
59a
38a
39a*

A.6 Programa *PipeInteger.java*

```
 1: class Heap {
 2:     static Connector c1, c2, c3, c4;
 3: }
 4:
 5: class PipeInteger {
 6:     static public void main (String[] args) {
 7:         Heap.c1 = new Connector();
 8:         Heap.c2 = new Connector();
 9:         Stage1 s1 =  new Stage1();
10:         s1.start();
11:         Heap.c3 = new Connector();
12:         Stage2 s2 = new Stage2();
13:         s2.start();
14:         Heap.c4 = new Connector();
15:         Stage3 s3 = new Stage3();
16:         s3.start();
17:         Listener lis = new Listener();
18:         lis.start();
19:
20:         for (int i = 1; i < 5; i++)
21:             Heap.c1.add(i);
22:         Heap.c1.stop();
23:     }
24: }
25:
26: final class Connector {
27:     int queue = -1;
28:     public final synchronized int take() {
29:         int value;
30:         while (queue < 0)
31:             try {
32:                 wait();
33:             } catch (InterruptedException ex) {}
34:         value = queue;
35:         queue = -1;
36:         return value;
37:     }
38:
39:     public final synchronized void add(int o) {
40:         queue = o;
41:         notifyAll();
42:     }
43:
44:     public final synchronized void stop() {
45:         queue = 0;
46:         notifyAll();
47:     }
48: }
49:
```

```
50: final class Stage1 extends Thread {
51:     public void run() {
52:         System.out.println("Stage1 startup");
53:         int tmp = -1;
54:         while (tmp != 0) {
55:             if ((tmp = Heap.c1.take()) != 0)
56:                 Heap.c2.add(tmp + 1);
57:         }
58:         Heap.c2.stop();
59:         System.out.println("Stage1 shutdown");
60:     }
61: }
62:
63: final class Stage2 extends Thread {
64:     public void run() {
65:         System.out.println("Stage2 startup");
66:         int tmp = -1;
67:         while (tmp != 0) {
68:             if ((tmp = Heap.c2.take()) != 0)
69:                 Heap.c3.add(tmp + 1);
70:         }
71:         Heap.c3.stop();
72:         System.out.println("Stage2 shutdown");
73:     }
74: }
75:
76: final class Stage3 extends Thread {
77:     public void run() {
78:         System.out.println("Stage3 startup");
79:         int tmp = -1;
80:         while (tmp != 0) {
81:             if ((tmp = Heap.c3.take()) != 0)
82:                 Heap.c4.add(tmp + 1);
83:         }
84:         Heap.c4.stop();
85:         System.out.println("Stage3 shutdown");
86:     }
87: }
88:
89: final class Listener extends Thread {
90:     public void run() {
91:         System.out.println("Listener startup");
92:         int tmp = 0;
93:         while (tmp >= -1) {
94:             if ((tmp = Heap.c4.take()) != 0)
95:                 System.out.println("output is " + tmp);
96:         }
97:         System.out.println("Listener shutdown");
98:     }
99: }
```

Figura A.11: Programa *PipeInteger.java*

Descrição do DFO criado:

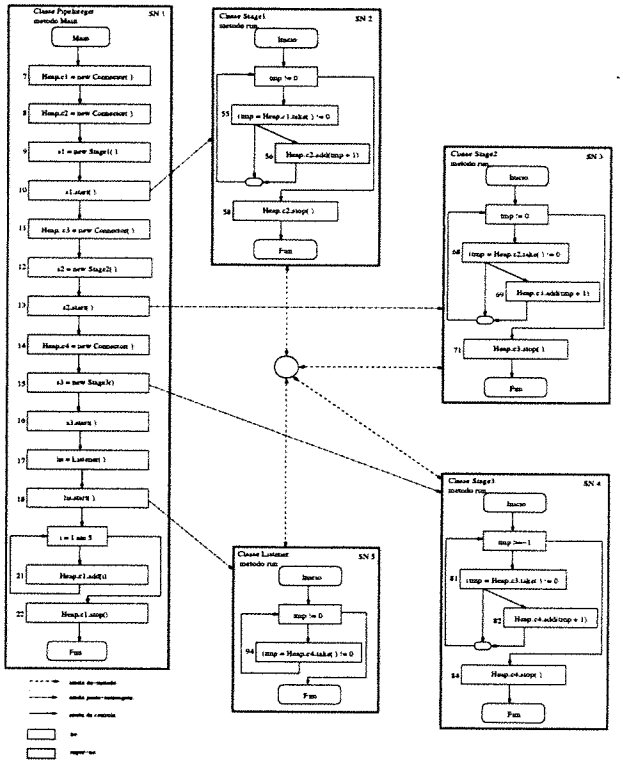
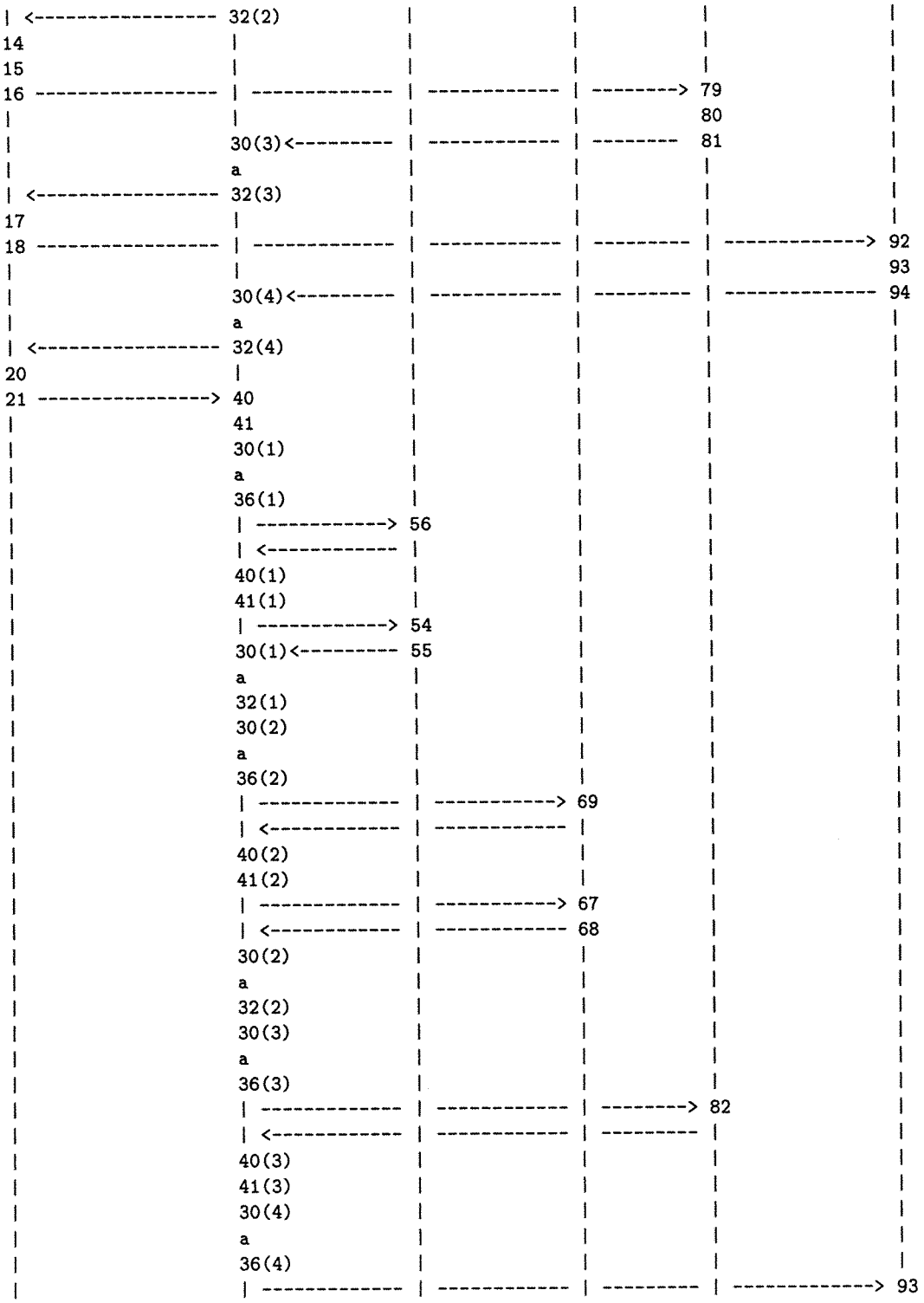
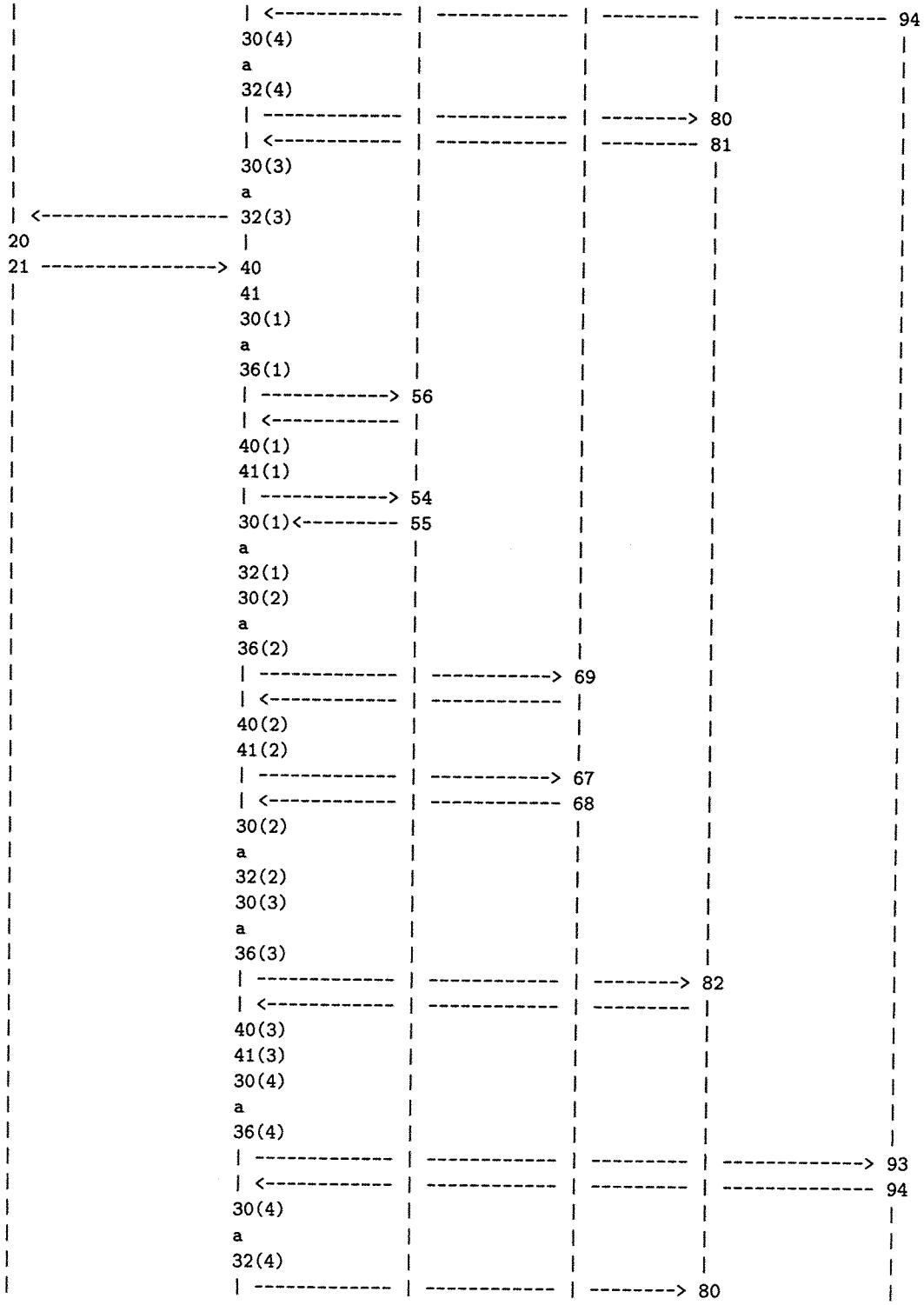


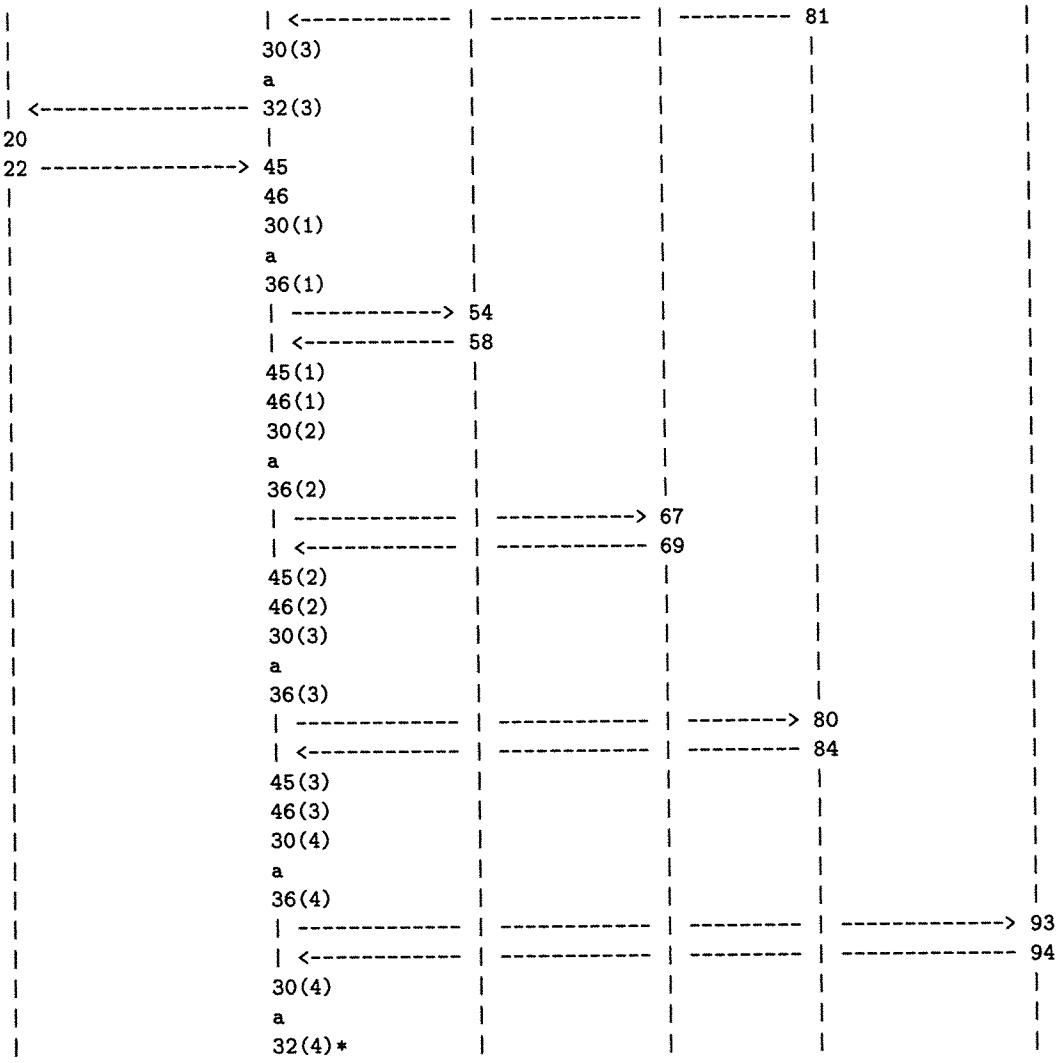
Figura A.12: DFO do Programa *PipeInteger.java*

Representação do contra-exemplo aleatório gerado pela *Bandera*:

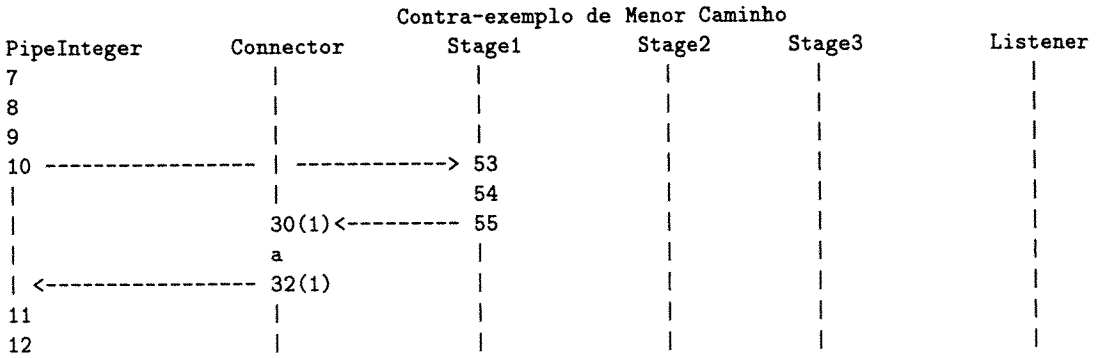
```
                              Contra-exemplo Aleatório
PipeInteger        Connector          Stage1          Stage2          Stage3          Listener
7                      |                |               |               |               |
8                      |                |               |               |               |
9                      |                |               |               |               |
10 ------------------  | ------------> 53               |               |               |
|                      |               54               |               |               |
|                   30(1)<---------- 55                 |               |               |
|                   a                |                |               |               |
| <------------------ 32(1)           |                |               |               |
11                     |                |               |               |               |
12                     |                |               |               |               |
13 ------------------  | -------------- | ------------> 66               |               |
|                      |                |               67               |               |
|                   30(2)<---------- | ------------- 68               |               |
|                   a                |                |               |               |
```

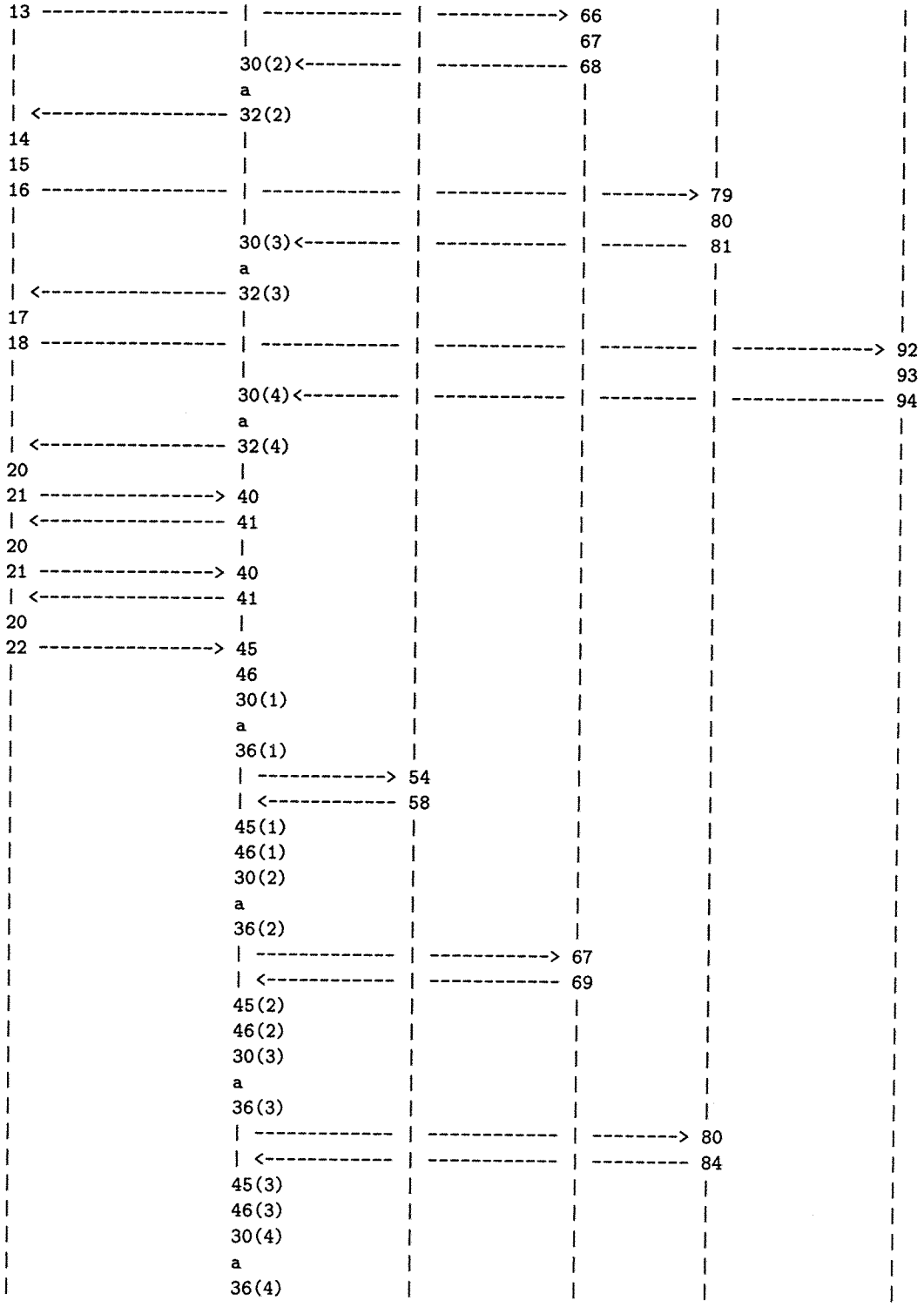

32(2)
14
15
16
79
80
30(3)
81
a
32(3)
17
18
92
93
30(4)
94
a
32(4)
20
21
40
41
30(1)
a
36(1)
56
40(1)
41(1)
54
30(1)
55
a
32(1)
30(2)
a
36(2)
69
40(2)
41(2)
67
30(2)
68
a
32(2)
30(3)
a
36(3)
82
40(3)
41(3)
30(4)
a
36(4)
93

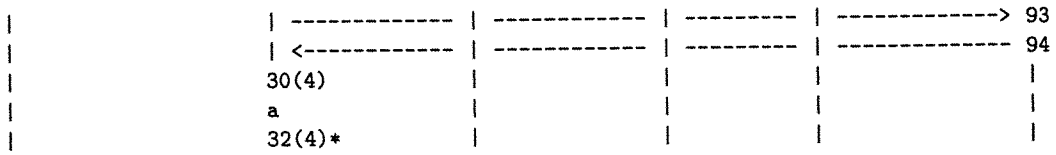

30(4)
a
32(4)
30(3)
a
32(3)
20
21
40
41
30(1)
a
36(1)
56
40(1)
41(1)
54
30(1)
55
a
32(1)
30(2)
a
36(2)
69
40(2)
41(2)
67
68
30(2)
a
32(2)
30(3)
a
36(3)
82
40(3)
41(3)
30(4)
a
36(4)
93
94
30(4)
a
32(4)
80
81
80
94



Além disso, temos o contra-exemplo de menor caminho:




13
30(2)
a
32(2)
14
15
16
30(3)
a
32(3)
17
18
30(4)
a
32(4)
20
21
40
41
20
21
40
41
20
22
45
46
30(1)
a
36(1)
54
58
45(1)
46(1)
30(2)
a
36(2)
67
69
45(2)
46(2)
30(3)
a
36(3)
80
84
45(3)
46(3)
30(4)
a
36(4)
66
67
68
79
80
81
92
93
94


30(4)
a
32(4)*
93
94

A.7 Programa *StrictClockS.java*

```
1  public class StrictClockS
2  {
3      public static void main(String[] args)
4      {
5          int numThreads = 2;
6          int maxTime = 5;
7          Monitor m  = new Monitor(maxTime);
8          Clock   c1 = new Clock("Clock", m, maxTime);
9          c1.start();
10         Client1 c1 = new Client1("Client", m, 1);
11         Client2 c2 = new Client2("Client", m, 2);
12         c1.start();
13         c2.start();
14     }
15 }
16
17 class Clock extends Thread{
18     private String name;
19     private Monitor monitor;
20     private int max;
21     public Clock(String n, Monitor m, int maxTime){
22       name = n;
23       monitor = m;
24       max = maxTime;
25     }
26     public void run(){
27         System.out.println("I am thread " + name);
28         while(monitor.getTime() < max){
29            monitor.tick();
30            try{sleep(10);}
31            catch(InterruptedException e){System.out.println(e);}
32         }
33     }
34 }
35
36 class Client1 extends Thread{
37     private String name;
38     private Monitor monitor;
39     private int id;
40
41     public Client1(String n, Monitor m, int i) {
42         name = n;
43         monitor = m;
44         id = i;
45     }
46
47     public void run(){
48         int maxSleepTime = 5;
49         int interval;
50         System.out.println("I am thread " + name + " number " + id);
51         interval = 3;
52         monitor.wakeme(id, interval);
53         System.out.println("Client1: Awaken");
54     }
55 }
56 class Client2 extends Thread{
57     private String name;
58     private Monitor monitor;
59     private int id;
60
61     public Client2(String n, Monitor m, int i) {
62         name = n;
63         monitor = m;
64         id = i;
65     }
66
67     public void run() {
```

(a) Página 1

Figura A.13: Programa *StrictClockS.java*

```
68          int maxSleepTime = 5;
69          int interval;
70          System.out.println("I am thread " + name + " number " + id);
71          interval = 2;
72          monitor.wakeme(id, interval);
73          System.out.println("Client2: Awaken");
74      }
75  }
76
77  class Monitor{
78      private int now;            // current time
79      private MyLinkedList waitList;
80      private int max;            // max time
81      private MyObject countMon; // for addtional synch
82      private int count;          // for additional synck
83
84      Monitor(int maxTime){
85          now = 0;
86          waitList = new MyLinkedList();
87          countMon = new MyObject(0); // object used for additional synch
88                                      // since nobody will use 0 for interval
89          count = 0;                  // var used for additional synch
90          max = maxTime;
91      }
92
93      synchronized void tick(){
94          synchronized(countMon){
95              if(count!=0){
96                  try{countMon.wait();}catch (InterruptedException e){}
97              }
98              now++;
99              if(!waitList.isEmpty()){
100                 MyObject first = waitList.firstElement();
101                 if(first.time() == now){
102                     MyObject wakeup = first;
103                     waitList.removeElementAt(0);
104                     synchronized(wakeup){
105                         wakeup.notify();
106                     }
107                 }
108             }
109         }
110     }
111
112     void wakeme(int id, int interval){
113         int waketime;
114         MyObject lock;
115
116         synchronized (this){
117             waketime = now + interval;
118             System.out.println("Monitor: Client"+id+" calling wakeme()");
119             if(waketime > max){
120                 System.out.println("Want to wake up past max");
121                 return;
122             }
123             System.out.println("Client "+id+"Want to sleep "+ interval+" want to wake up at "
    +waketime);
124             lock = waitList.getLock(waketime);
125
126             synchronized(countMon){
127                 count++;
128             }
129         }
130
131         synchronized(lock){
132             synchronized(countMon){
133                 count--;
```

(b) Página 2

Figura A.13: Programa *StrictClockS.java* (Continuação...)

```
134                 if(count==0)countMon.notifyAll();
135             }
136             try{
137                 lock.wait();
138             }catch(InterruptedException e){}
139             System.out.println("Client awaken, time is "+ now);
140         }
141     }
142
143     int getTime(){
144         return now;
145     }
146 }
147
148 class MyObject extends Object{
149     public int time;
150     MyObject(int n){
151         time = n;
152     }
153
154     public int time(){
155         return time;
156     }
157 }
158
159 class MyLinkedList{
160     private MyObject[] list = new MyObject[2];
161     private int capacity =2;
162     private int size = 0;
163
164     public synchronized boolean isEmpty(){
165         return size==0;
166     }
167
168     public synchronized boolean isFull(){
169         return size == capacity;
170     }
171
172     public synchronized MyObject elementAt(int index){
173         return list[index];
174     }
175
176     public synchronized void insertElementAt(MyObject obj, int index){
177         if(!this.isFull()){
178             for(int i=size-1; i>=index; i--){
179                 list[i+1] = list[i];
180             }
181             list[index] = obj;
182             size++;
183         }
184     }
185
186     public synchronized void addElement(MyObject obj){
187         if(!this.isFull()){
188             list[size] = obj;
189             size++;
190         }
191     }
192
193     public synchronized MyObject firstElement(){
194         return list[0];
195     }
196
197     public synchronized void removeElementAt(int index){
198         if(!this.isEmpty()){
199             for(int i=index; i<size-1; i++){
200                 list[i] = list[i+1];
```

(c) Página 3

Figura A.13: Programa *StrictClockS.java* (Continuação...)

```
201                 }
202                 list[size-1] = null;
203                 size--;
204             }
205         }
206
207         public synchronized MyObject getLock(int n){
208             MyObject temp;
209             int key;
210             MyObject help;
211             int i = 0;
212             for ( i=0; i<size; i++){
213                 help=list[i];
214                 key = help.time();
215                 if (n==key){
216                     temp = list[i];
217                     return temp;
218                 }
219             }
220             MyObject lock = new MyObject(n);
221             for (i=0; i<size; i++){
222                 help = list[i];
223                 key = help.time();
224                 if( n  < key ){
225                     this.insertElementAt(lock, i);
226                     return lock;
227                 }
228             }
229             this.addElement(lock); //append if greater than all others
230             return lock;
231         }
232 }
233
```

(d) Página 4

Figura A.13: Programa *StrictClockS.java* (Fim)

O diagrama de fluxo do objeto criado é:

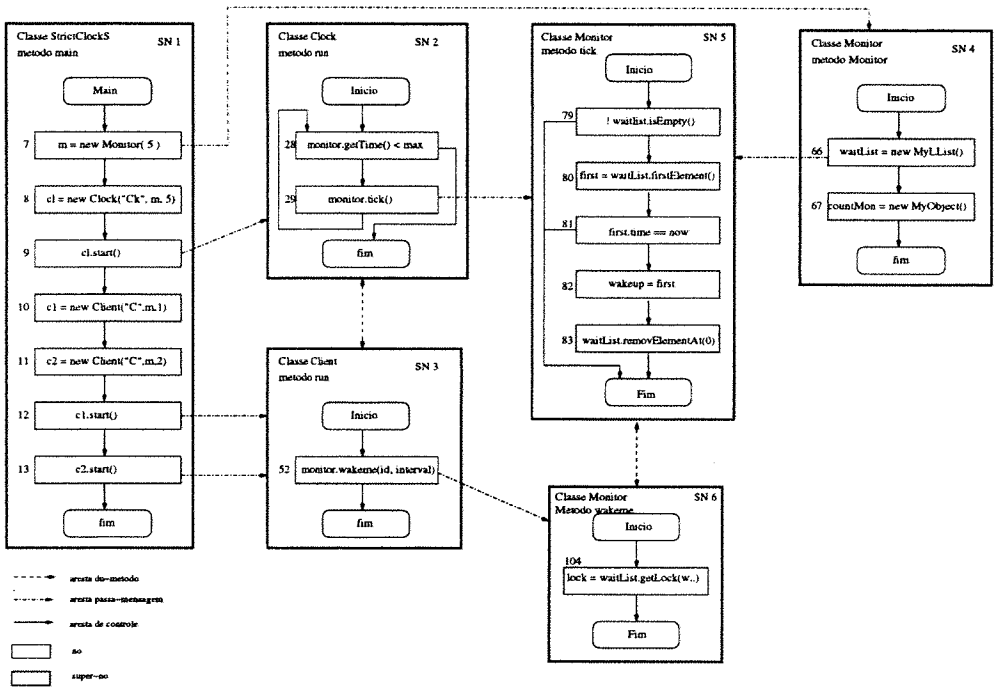
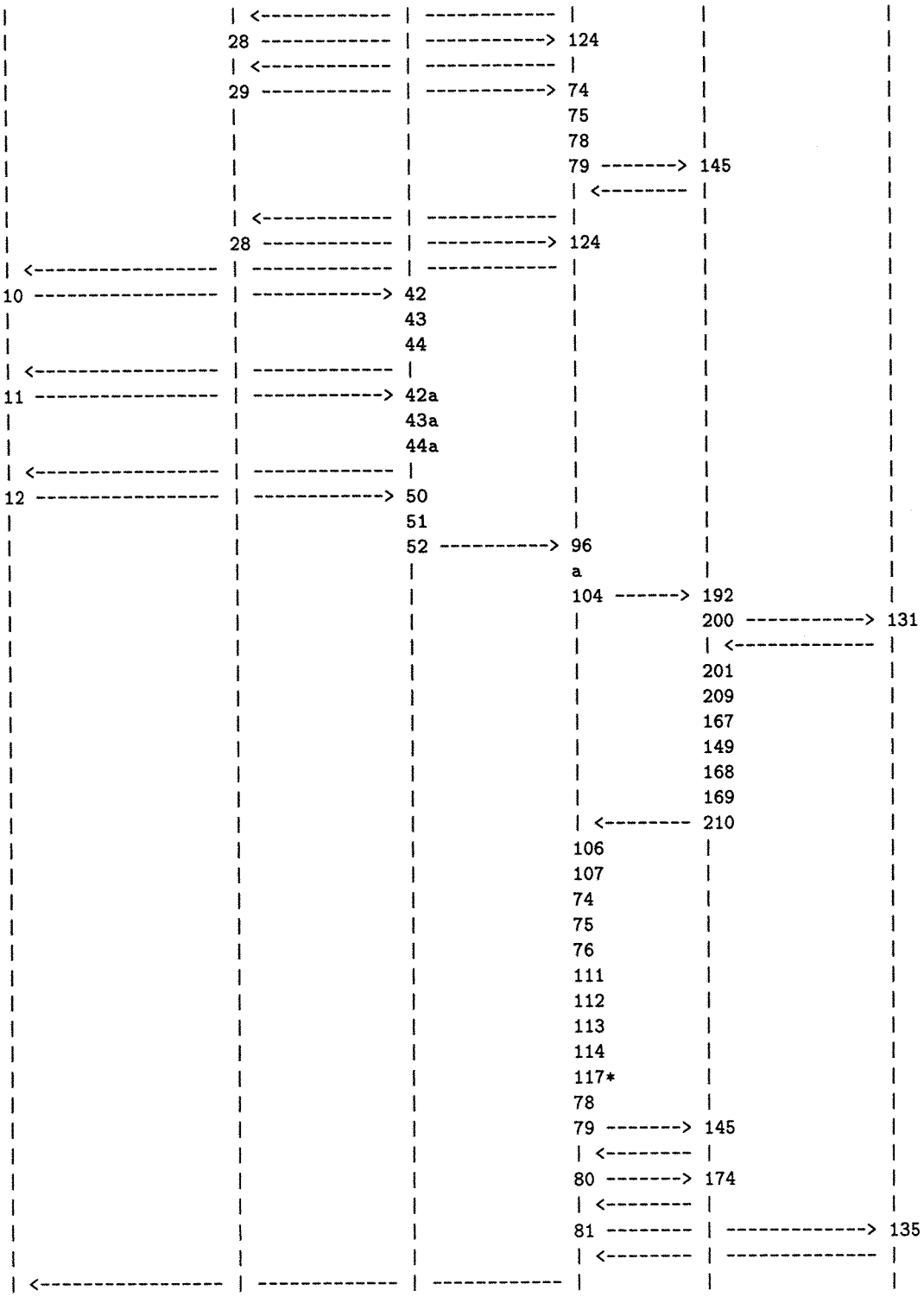
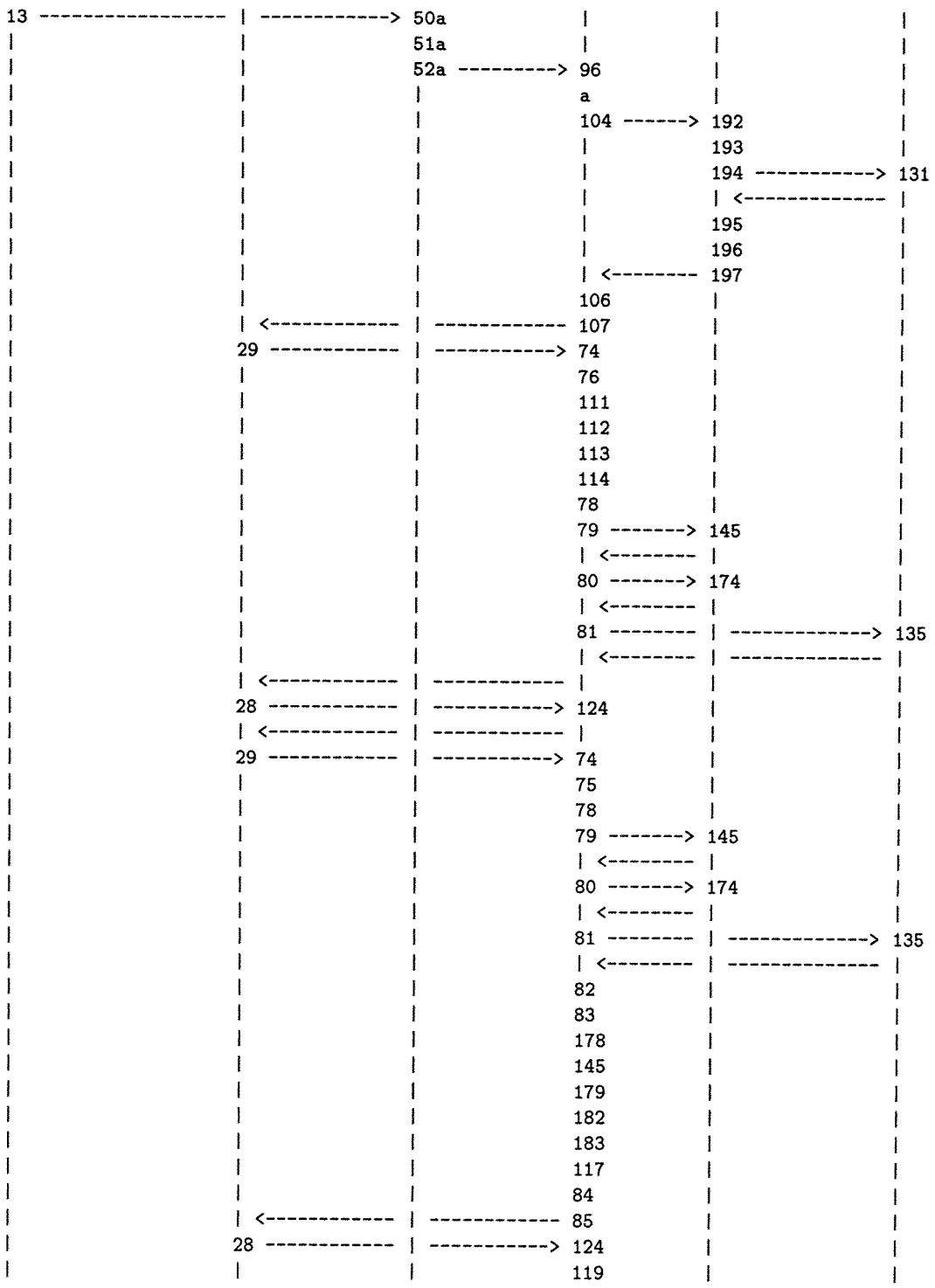


Figura A.14: DFO do Programa *StrictClockS.java*

O contra-exemplos aleatório e de menor caminho são iguais e logo seguir este encontra-se descrito:

```
StrictClockS       Clock           Client          Monitor     MyLinkedList      MyObject
7 ----------------- | ------------ | ----------> 65              |                 |
|                   |              |               a             |                 |
| <---------------- | ------------ | ------------ 70             |                 |
8 ----------------> 42             |                |            |                 |
|                   43             |                |            |                 |
| <---------------- 44             |                |            |                 |
9 ----------------> |              |                |            |                 |
|                   27             |                |            |                 |
|                   28 ------------ | ----------> 124             |                 |
|                   | <----------- | ------------ |              |                 |
|                   29 ------------ | ----------> 74              |                 |
|                   |              |               75            |                 |
|                   |              |               78            |                 |
|                   |              |               79 --------> 145               |
|                   |              |               | <-------- |                 |
```


28
29
124
74
75
78
79
145
28
124
10
42
43
44
11
42a
43a
44a
12
50
51
52
96
a
104
192
200
131
201
209
167
149
168
169
210
106
107
74
75
76
111
112
113
114
117*
78
79
145
80
174
81
135


13
50a
51a
52a
96
a
104
192
193
194
131
195
196
197
106
107
29
74
76
111
112
113
114
78
79
145
80
174
81
135
28
124
29
74
75
78
79
145
80
174
81
135
82
83
178
145
179
182
183
117
84
85
28
124
119