Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Aumentando a Confiança nos Resultados de Testes de Sistemas Multi-threaded:
Evitando Asserções Antecipadas e Tardias

Ayla Débora Dantas de Souza Rebouças

Trabalho de tese submetido à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computacão.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Francisco Brasileiro e Walfredo Cirne
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

R292a Rebouças, Ayla Débora Dantas de Souza.
Aumento a confiança nos resultados de testes de sistemas multi-threaded: evitando asserções antecipadas e tardias/Ayla Débora Dantas de Souza Rebouças. – Campina Grande, 2010.
287 f.: il.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientadores: Profº. Francisco Brasileiro, Profº. Walfredo Cirne.
Referências.

1. Software - Confiabilidade. 2. Testes de Software. 3. Sistemas Multi-threaded. I. Título.

CDU 004.052(043)

Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA - COPIN
Av. Aprigio Veloso, 882 - 58.109-970 Campina Grande, PB
Fone: (++55) (+83) 3310-1124 - e-mail: copin@dsc.ufcg.edu.br

PARECER FINAL DO JULGAMENTO DA TESE DA DOUTORANDA

AYLA DÉBORA DANTAS DE SOUZA REBOUÇAS

TÍTULO: "AUMENTANDO A CONFIANÇA NOS RESULTADOS DE TESTES DE SISTEMAS MULTI-THREADED: EVITANDO ASSERÇÕES ANTECIPADAS E TARDIAS"

COMISSÃO EXAMINADORA	CONCEITO
Francisco Vilar Brasileiro **PROF. FRANCISCO VILAR BRASILEIRO, Ph.D** **Orientador**	APROVADO
Patrícia Duarte de Lima Machado **PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D** **Examinadora**	APROVADO
Dalton Serey **PROF. DALTON DARIO SEREY GUERRERO, D.Sc** **Examinador**	APROVADO
Marcelo B. d'Amorim **PROF. MARCELO BEZERRA D'AMORIM, Ph.D** **Examinador**	APROVADO
Roberta Coelho **PROFª ROBERTA DE SOUZA COELHO , Drª** **Examinadora**	Aprovada

Campina Grande, 30 de agosto de 2010

Resumo

Testar sistemas com múltiplas threads é uma atividade que envolve vários desafios. O fato de serem inerentemente não determinísticos torna tanto a implementação desses sistemas quanto a implementação de seus testes suscetível a erros. É comum existirem testes nestes sistemas que falham em apenas algumas execuções, sem que as causas dessas falhas sejam faltas na aplicação (também conhecidas como defeitos ou *bugs*), mas sim devido a problemas nos testes. Por exemplo, isso pode acontecer se a fase dos testes em que são feitas as verificações (asserções) for executada em momentos inadequados. Isso é freqüente quando os testes envolvem operações assíncronas. Deve-se evitar que nestes testes as asserções sejam feitas antes que essas operações tenham sido concluídas ou também que sejam feitas em um momento tardio, quando o sistema mudou de estado e as condições verificadas não são mais satisfeitas, gerando assim falsos positivos. Testes que não são confiáveis, como os que falham devido a tais problemas, levam os desenvolvedores a desperdiçar muito tempo procurando faltas de software que não existem. Além disso, os desenvolvedores podem perder a confiança nos testes parando de acreditar que falhas em certos testes são devidas a faltas, mesmo quando este é o caso. A existência de tais cenários foi o que motivou este trabalho, que tem como objetivo evitar que asserções em testes sejam feitas cedo ou tarde demais. Para atingir esse objetivo, apresenta-se uma abordagem baseada na monitoração e controle das threads da aplicação e que dê suporte ao desenvolvimento de testes corretos de sistemas multi-threaded. A abordagem visa facilitar a implementação de testes envolvendo operações assíncronas e aumentar a confiança dos desenvolvedores nos seus resultados. Esta abordagem foi avaliada através de estudos de caso utilizando uma ferramenta de suporte ao teste de sistemas multi-threaded, desenvolvida para este trabalho, e também através de sua modelagem formal utilizando a linguagem TLA+, com o objetivo de demonstrar que asserções antecipadas e tardias não ocorrem quando ela é utilizada.

Abstract

Testing multi-threaded systems is quite a challenge. The inherent non-determinism of these systems makes their implementation and the implementation of their tests far more susceptible to error. It is common to have tests of these systems that may not sometimes pass and whose failures are not caused by application faults (bugs), but by problems with the tests. For instance, this can happen when there are asynchronous operations whose corresponding test verifications (assertions) are performed at inappropriate times. Unreliable tests make developers waste their time trying to find non-existing bugs, or else make them search for bugs in the wrong place. Another problem is that developers may cease to believe that certain test failures are caused by software bugs even when this is the case. Scenarios like these have motivated this work. Our main objective is to avoid test failures that are caused, not by application defects, but by test assertions performed either too early or too late. In order to achieve this goal, we present an approach whose basic idea is to use thread monitoring and control in order to support the development of multi-threaded systems tests involving asynchronous operations. This approach is intended to make it easier the development of correct tests for these systems and also to improve developers' confidence on the results of their tests. The proposed approach has been evaluated through case studies using a tool to support the development of multi-threaded systems tests (developed for this work) and also by formally modeling the approach using the TLA+ language in order to prove that early and late assertions do not occur when this approach is used.

Agradecimentos

Gostaria de agradecer a Deus, que me deu a chance de chegar até aqui e de poder aprender tantas coisas, contando sempre com pessoas especiais ao meu lado em todos os momentos de minha vida.

Obrigada ao meu esposo que tanto amo, Rodrigo, por ser meu companheiro, colega e melhor amigo, e por estar sempre perto me incentivando a continuar e a acreditar em mim.

Obrigada a minha filhinha Ana Luísa, por aguentar ainda em meu ventre a tensão que acompanha o final de um trabalho de doutorado.

Aos meus pais e irmãs, por seu amor sem medida e pela confiança que em mim sempre depositaram. Sou o que sou graças a essas pessoas tão especiais.

Aos pais de Rodrigo e aos meus cunhados Raquel e Daniel, por sua paciência e apoio principalmente no final desta caminhada de doutorado.

Aos meus orientadores, Fubica e Walfredo, por sua paciência e atenção e por tudo que me ensinaram ao longo de meu processo de formação.

Aos amigos do LSD, inclusive os que já saíram, por contribuírem para que este laboratório seja um ambiente fantástico de trabalho. Dentre esses amigos, alguns foram fundamentais para este trabalho. Matheus Gaudêncio foi um destes, em especial por topar o desafio de trabalhar de perto comigo e prover um suporte sem medida que ia desde a execução dos estudos de caso e discussão de artigos até discussões sobre filosofia e psicologia. Ajudaram também demais no trabalho os meninos do projeto OurBackup, em especial Marcelo Iury, Eduardo Colaço, Flávio Vinicius, Alexandro Soares e Paolo Victor. Sem eles os estudos de caso envolvendo esta ferramenta não teriam sido possíveis. Várias outras pessoas do LSD ajudaram das mais diversas formas, seja com um sorriso ou com uma idéia trocada nos corredores ou nas salas desse laboratório do qual nunca irei me esquecer.

Aos demais colegas do mestrado e doutorado, por todos os momentos de alegria e agonia compartilhados e que ficarão sempre guardados em meu coração.

Aos colegas da UFPB por sua compreensão em meus momentos de ausência devidos à dedicação a este trabalho de tese.

Aos membros da banca de proposta de qualificação e de tese, por todos as contribuições dadas no intuito de melhorar a qualidade desse trabalho. Contribuíram nesse processo os professores Dalton Serey, Marcelo D'Amorim, Patrícia Machado, Thaís Vasconcelos e Roberta Coelho.

À COPIN e ao Centro de Engenharia Elétrica e Informática como um todo, pelo suporte técnico e acadêmico durante esses anos e pela enorme atenção dispensada por todos.

Ao CNPq, pelo auxílio financeiro através de bolsa de estudos e taxa de bancada e que possibilitou o desenvolvimento deste trabalho.

Obrigada também a todos os amigos e familiares que estando perto ou à distância apostaram e torceram por mim. Temo ser injusta ao citar nomes (pois sei que acabarei esquecendo o nome de alguém), mas gostaria aqui de registrar algumas das pessoas acima não citadas nominalmente e que fizeram a diferença nessa fase de minha vida por tantos momentos compartilhados: Lívia, Raquel, Eliane, Degas, Andréa, Aninha, Vera, João, Paulo, Lauro, Mila, Guga, Zane, Ricardo, Marcus, Moisés, Bárbara, William, Keka, Cleide, Ana, Tomás, Alan, Giovani, Guiga, Flávio Barata, Peruca, Manel, Nazareno, Lile, Pryscilla, Leo, Camilo, Dalton, Eloi, Celso, Alexandre, Gustavo, Carla, Pablo, Vinícius, Dona Inês, e muitos outros. Gente, obrigada por tudo! Que Deus sempre os abençoe!

Conteúdo

1 Introdução **1**

1.1 Motivação e Relevância 1

1.2 Objetivos 5

1.3 Metodologia 9

1.4 Organização do Documento 11

2 Fundamentação Teórica **12**

2.1 Contextualização 12

2.1.1 Conceitos Básicos da Área de Testes 12

2.1.2 Este Trabalho no Contexto da Área de Testes 14

2.2 Esperando antes de Asserções 15

2.3 O Problema Tratado 19

2.4 Programação Orientada a Aspectos 20

2.5 Aplicações Multi-threaded em Java 22

2.6 Efeito da Monitoração 25

2.7 Conclusões Parciais 25

3 Trabalhos Relacionados **27**

3.1 Uso do Padrão *Humble Object* para Evitar Testes Assíncronos 28

3.2 Mecanismos de Sincronização em Testes 30

3.3 Controle de Testes 32

3.4 Monitoração de Threads 35

3.5 Problemas com Asserções na Área de Projeto por Contrato 36

3.6 Identificação de Defeitos de Concorrência 38

3.6.1 Detecção de Condições de Corrida 39
3.6.2 Análise Estática e Verificação de Modelos em Programas Concorrentes 40
3.6.3 Técnicas de Re-execução 41
3.6.4 Técnicas para forçar determinados escalonamentos 42
3.6.5 Geradores de Ruído 43
3.7 Conclusões parciais 45

4 Modelagem do Problema e da Solução Proposta 46
4.1 Modelagem do Problema 47
4.2 Requisitos para Resolver o Problema 53
4.3 Uma Solução Baseada em Monitores 57
4.3.1 Descrição da Solução 58
4.3.2 Prova 59
4.4 Conclusões Parciais 60

5 A Abordagem *Thread Control for Tests* 61
5.1 O Arcabouço ThreadServices 61
5.2 A Abordagem *Thread Control for Tests* 63
5.3 O Arcabouço de Testes *ThreadControl* 67
5.3.1 Arquitetura 68
5.3.2 Usando o *ThreadControl* 70
5.3.3 Especificando Estados de Espera 72
5.3.4 Monitorando e Controlando as Threads 75
5.4 Conclusões Parciais 79

6 Avaliação Inicial da Abordagem *Thread Control for Tests* 82
6.1 Estudo 1: Estudo de Caso sobre o Uso da Abordagem *Thread Control for Tests* em Casos de Teste Sintéticos, mas Reproduzindo Casos de Teste Reais 83
6.1.1 Planejamento do Estudo de Caso 83
6.1.2 Execução e Resultados do Primeiro Estudo de Caso 86
6.2 Estudo 2: Casos de Teste com Testes Reais do OurBackup 91
6.2.1 Planejamento do Segundo Estudo de Caso 93

6.2.2 Execução e Resultados do Segundo Estudo de Caso 97
6.3 Conclusões Parciais 103

7 Avaliação da Abordagem *Thread Control for Tests* usando TLA+ 104
7.1 Motivação para usar TLA+ 105
7.2 A Especificação *Test Execution* usando TLA+ 107
7.3 A Especificação *Monitored Test Execution* utilizando TLA+ 112
7.4 A Especificação em TLA+ da Execução de Teste Checando Invariante de Asserções 116
7.5 Análise de Estados Alcançáveis 116
7.6 Análise de Comportamentos 120
7.7 Conclusões Parciais 121

8 Combinando a Abordagem *Thread Control for Tests* com *Geradores de Ruído* 122
8.1 A Ferramenta ConTest 123
8.2 Aplicação Exemplo Utilizada: *Meu Gerenciador de Backup* 124
8.3 Testando o Gerenciador de Backup 125
8.4 Estudo de Caso para Avaliar o Uso da abordagem *Thread Control for Tests* utilizando Geradores de Ruído 130
8.4.1 Objetivos 130
8.4.2 Hipóteses 131
8.4.3 Variáveis 131
8.4.4 Sujeitos e Objetos 131
8.4.5 Instrumentação 132
8.4.6 Procedimento de coleta dos dados 132
8.4.7 Procedimento de análise 133
8.4.8 Execução 133
8.4.9 Análise 134
8.5 Conclusões Parciais 138

9 Considerações Finais 142
9.1 Conclusões 142

9.2 Trabalhos Futuros 145

Referências Bibliográficas 158

A Especificação em TLA+ de Execução de Testes sem Monitoração 159

B Especificação em TLA+ de Execução de Teste Monitorando Threads segundo Abordagem *Thread Control for Tests* 163

C Especificação em TLA+ de Execução de Testes sem Monitoração mas com Checagem de Asserções Antecipadas e Tardias 166

D Saída do Programa StatesAnalyzer 167

E Exemplos de Asserções Antecipadas Identificadas 179

F Exemplos de Asserções Tardias Identificadas 183

G Código do Arcabouço ThreadControl 189

Lista de Símbolos e Terminologia

Adendos - *Advice, com relação à linguagem AspectJ*

Análise em tempo de execução - *Runtime analysis*

AOP - *Aspect-Oriented Programming, ou Programação Orientada a Aspectos*

Arcabouço - *Framework*

Atrasos Explícitos - *Explicit Delays*

Condições de corrida - *Race conditions*

Controlador - *Driver*

DBC - *Design by Contract, ou seja, Projeto por Contrato*

Efeito da monitoração - *Probe effect, ou seja, a diferença em comportamento entre um objeto monitorado e sua versão não instrumentada*

Espera Ocupada - *Busy wait*

Fluxo de Execução - *Thread*

Geradores de ruído - *Noise makers*

Impasse - *Deadlock*

Habilidade de Controle de Testes - *Test Controllability*

Habilidade de Observação de Testes - *Test Observability*

Intercalação - *Interleaving*

IRM - *Inlined Reference Monitor ou monitor de referências incorporado, uma técnica através da qual o código para monitorar referências de uma aplicação é inserido no código da aplicação de maneira segura*

JML - *Java Modeling Language, uma linguagem que permite a especificação de comportamentos estáticos e comportamentais de interfaces em um código Java.*

Mecanismo de gravar e mandar executar - *Record-playback*

Re-execução - *Replay*

SUT - *System Under Test, ou seja, o sistema sendo testado*

Tranca - *Lock*

TLA - *Temporal Logic of Actions, ou Lógica Temporal de Ações, uma lógica para lidar com sistemas concorrentes*

TLA+ - *Linguagem completa de especificação e que usa como base a Lógica Temporal de Ações (TLA)*

Verificação de Modelos - *Model Checking, uma família de técnicas baseadas na exploração exaustiva do espaço de estados para verificar propriedades em sistemas concorrentes.*

Lista de Figuras

2.1 Exemplo: Threads criadas por uma invocação feita por um teste 16
2.2 Processo de combinação do código da aplicação com o código definido nos aspectos 22

5.1 Visão Geral da Abordagem 65
5.2 Algumas operações monitoradas e estados relacionados a estas operações. . 67
5.3 Arquitetura geral do arcabouço *ThreadControl* 69
5.4 Arquitetura do arcabouço *ThreadControl* visualizando suas classes principais 70
5.5 Diagrama de classes com operações da interface *SystemConfiguration* e a classe *ListOfThreadConfigurations*, que a implementa 72
5.6 Diagrama de classes representando a classe *ThreadConfiguration* e a classe *ThreadState* 73
5.7 Ação de adendos do ThreadControlAspect 76

8.1 Classes do Serviço Gerenciador de Backup Pessoal 125
8.2 Análise do comportamento das falhas no Ambiente 1 139
8.3 Análise do comportamento das falhas no Ambiente 2 140
8.4 Análise do comportamento das falhas no Ambiente 3 141

9.1 Arquitetura inicial proposta para o arcabouço *Distributed ThreadControl* . . 148

Lista de Tabelas

3.1 Visão Geral dos Trabalhos Relacionados 29

4.1 Exemplo de matriz de escalonamento M que mostra a distribuição dos eventos no tempo 49

5.1 Mapeamento entre a descrição da abordagem *Thread Control for Tests* e o modelo de solução baseada em monitores proposto na Seção 4.3 81

6.1 Frequência de falhas para o cenário 1 89
6.2 Probabilidade de falhas e tempo médio de espera para cada versão ao executar o cenário 2 90
6.3 Configuração do Ambiente para execução do estudo de caso 97
6.4 Número de falhas em testes para cada tratamento considerando 10.000 execuções de cada 100
6.5 Número de falhas classificadas como falhas causadas por asserções antecipadas e tardias para cada tratamento considerando 10.000 execuções de cada 101

8.1 Configuração do Ambiente 1 para execução do estudo de caso: Macbook . . 134
8.2 Configuração do Ambiente 2 para execução do estudo de caso: Máquina virtualizada 135
8.3 Configuração do Ambiente 3 para execução do estudo de caso: Desktop LSD 135
8.4 Resultados de Execuções de Teste no Ambiente 1 (MacBook) 136
8.5 Resultados de Execuções de Teste no Ambiente 2 (Máquina Virtualizada) . 136
8.6 Resultados de Execuções de Teste no Ambiente 3 (Desktop) 136

Capítulo 1

Introdução

Neste capítulo é apresentada a motivação para este trabalho de doutorado, são enumerados os seus objetivos e é descrita a forma como está organizado este documento.

1.1 Motivação e Relevância

Testar é uma atividade fundamental na Engenharia de Software [11] e ajuda a dar mais indícios de que a qualidade do produto final de software é a que se espera. Testes fazem parte das técnicas de verificação e validação (V&V), as quais buscam verificar se o produto de software produzido atende a sua especificação e contempla a funcionalidade esperada por seus clientes [88].

Dentre as abordagens para verificação e análise de sistemas de V&V, testes caracterizam-se como uma técnica em que o software é exercitado em busca de possíveis defeitos, visando aumentar a confiança de que ele irá se comportar corretamente quando em produção. É importante destacar que eles não servem para mostrar a ausência de defeitos, mas apenas sua presença [27]. A idéia por trás de testes é que os defeitos devem ser encontrados o mais cedo possível e idealmente antes que o software seja utilizado por seus usuários reais.

Ao se testar um software, vários cenários de uso do sistema devem ser pensados e especificados através de casos de teste. Os casos de teste definem ações e verificações que devem ser executadas no sistema para exercitá-lo e verificar como ele se comporta.

Considerando isso, pode-se resumir as atividades de teste em três etapas: i) projetar casos de teste; ii) executar o software testado com esses casos de teste; e iii) examinar os resultados

produzidos por estas execuções [46].

Sistemas com múltiplos fluxos de execução (*multi-threaded*) são normalmente difíceis de serem testados [91]. Isso ocorre devido à assincronia de algumas operações exercitadas nos testes. Em geral, não é trivial determinar nestes sistemas em que momento os resultados produzidos por sua execução devem ser examinados através de verificações, também chamadas de asserções (etapa "iii" dos testes). Nem sempre é fácil saber o quanto esperar antes de executar as asserções. Dependendo da abordagem usada para fazer a thread do teste esperar enquanto o sistema está sendo exercitado, falsos positivos podem acontecer.

Um exemplo simples de teste com assincronia é um teste que exercita a inicialização de um sistema composto por vários serviços assíncronos, representados por diferentes threads. Ao testar a inicialização de tal sistema, normalmente uma operação do tipo `start` é invocada e, depois de um tempo, deve-se verificar através de uma asserção se os serviços internos foram inicializados corretamente e estão prontos para receber requisições (ex: uma chamada no estilo `assertTrue(wasMySystemCorrectlyInitialized())`). Porém, para que tal asserção seja verdadeira e permita que o teste passe, é preciso que as threads envolvidas nos serviços que compõem o sistema tenham sido iniciadas e comecem a rodar alterando o estado do sistema que será verificado no teste.

Em alguns casos, devido ao fato de se testar operações assíncronas, testes podem falhar em algumas de suas execuções porque a verificação dos resultados produzidos por estas operações assíncronas não é feita em um momento apropriado. Por exemplo, as asserções (verificações) podem ser feitas muito cedo (quando a operação ainda está em andamento) ou muito tarde (quando a condição sendo verificada não é mais satisfeita, porque o estado do sistema foi alterado por alguma thread).

Considerando o exemplo de inicialização do sistema, contando com as operações `mySystem.start` e `assertTrue(wasMySystemCorrectlyInitialized())`, é possível que a última asserção não seja verdadeira em algumas execuções do teste. A falha nestes casos pode ocorrer pois no momento em que se verificou na asserção se o serviço estava ativo e pronto para receber requisições, a inicialização ainda não tinha sido concluída.

Por causa do problema das asserções antecipadas e tardias, verificações de estado do sistema ou dos resultados que produzem acabam não sendo verdadeiras no momento em que são executadas, gerando falsos positivos nos testes. Isso é sinalizado através de uma falha

obtida como resultado da execução do teste, que tem como finalidade indicar a existência de um defeito no software sendo testado.

Alguns exemplos de abordagens que podem levar a falsos positivos são atrasos explícitos (e.g. chamadas a operações como `Thread.sleep` para fazer a thread do teste esperar por um tempo determinado) e espera ocupada (e.g. atrasos explícitos dentro de laços em que uma condição geral está sendo verificada).

Embora ao observar essas técnicas já possa ser claro que estas possam levar a falsos positivos, é muito comum encontrar o seu uso na prática, em especial pela simplicidade de implementá-las. Em um trabalho de estágio realizado por um aluno da UFCG [38] em paralelo a este trabalho de doutorado, percebeu-se o uso de atrasos explicícitos e esperas ocupadas em testes de alguns sistemas de código aberto populares, como o Apache Hadoop [36], o Tomcat [35], o Ant [34] e o JBoss [49]. Além disso, o uso de tais técnicas foi também observado em testes automáticos desenvolvidos em diferentes sistemas do Laboratório de Sistemas Distribuídos (LSD) [63] e projetados por diferentes desenvolvedores.

Uma alternativa à técnica de atrasos explícitos e espera ocupada também encontrada na prática é o uso de sincronização explícita entre o código de teste e o do sistema, uma solução em geral intrusiva (por requerer alterações de código no sistema) e que alguns testadores tentam evitar. Outra solução proposta ainda é que se evitem testes assíncronos, como proposto por Meszaros [66] através do padrão *Humble Object*, que demanda que o sistema sob teste seja refatorado de forma que o teste só interaja com o sistema de maneira síncrona. O intuito dessa solução é tanto evitar falsos positivos quanto evitar testes que demorem muito a executar devido ao uso de outras abordagens de espera como espera ocupada ou atrasos explícitos, as quais são comumente usadas.

Nesta tese, houve a motivação para também atacar o problema dos falsos positivos em testes de aplicações *multi-threaded* pelo fato de tais aplicações estarem se tornando cada vez mais comuns, em especial com a popularização dos processadores com múltiplos núcleos (*multicore*). Tal fato tem levado a um aumento da complexidade do desenvolvimento de software, aumentando a demanda por melhores ferramentas para sistematicamente encontrar defeitos, auxiliar na depuração de programas, encontrar gargalos de desempenho e auxiliar em testes [91][33].

A demanda por ferramentas para testes nesse contexto existe devido aos desafios em tes-

tar aplicações *multi-threaded*, em especial por seu não determinismo [62], o que pode levar um mesmo caso de teste a execuções distintas do software dependendo do escalonamento das *threads* do sistema [64]. Porém, o sistema deve executar corretamente em qualquer um dos escalonamentos possíveis. Falhas que ocorrem esporadicamente nos testes podem ser consequências de defeitos da aplicação que são de difícil reprodução e correção. Tal dificuldade de correção existe porque, em geral, falhas assim demandam em sua depuração uma investigação dos vários caminhos de execução que a aplicação sendo testada pode ter seguido.

Quando o problema que leva o teste a falhar é com o próprio teste, pode-se ter desperdício de tempo por parte dos desenvolvedores tentando encontrar defeitos que não existem ou procurando-os nos locais errados. Um outro efeito negativo é que os desenvolvedores podem não mais confiar nos resultados de teste. Por exemplo, quando uma falha de teste acontece, eles podem não acreditar que ela se deve a um defeito no software, mesmo quando este é o caso [89].

O interesse em realizar esta pesquisa surgiu da observação do problema das falhas em testes por asserções feitas em momentos inadequados e das consequências deste problema. Viu-se que as técnicas comumente utilizadas na prática para fazer os testes esperarem acabam acarretando danos ao desenvolvimento e considerou-se importante investigar este tema criteriosamente.

Considerando o problema apresentado e sua relevância, este trabalho de doutorado demonstra que não se podem evitar asserções antecipadas e tardias sem monitoração e um nível de controle mínimo das threads do teste e da aplicação sob teste evitando que asserções ocorram em momentos inadequados. Ele apresenta também uma abordagem que dá suporte aos testadores no desenvolvimento de testes automáticos sem a necessidade de alterações no código do sistema sob teste e que funciona como um mecanismo transparente de sincronização entre o teste e a aplicação sendo testada.

O intuito de demonstrar formalmente a existência do problema dos falsos positivos por asserções feitas em momentos inadequados, quando não há monitoração de threads, é desestimular o uso de abordagens comuns na prática como atrasos explícitos e espera ocupada.

A abordagem que é apresentada neste trabalho de doutorado visa evitar a implementação de testes que apresentem falsos positivos por asserções realizadas em momentos inadequa-

dos. Ela foca no uso de monitoração e controle das threads da aplicação, mas com o cuidado de tentar deixar os testes simples (no sentido de não precisarem implementar barreiras de sincronização de maneira explícita) e fazer com que na sua implementação não haja a necessidade de mudanças de código no sistema sendo testado (como as que o padrão *Humble Object* exige, por exemplo). A idéia geral é fazer os testadores pensarem em fases na execução do sistema sob teste onde asserções devem ser feitas e que são mapeadas em estados de suas threads. Por exemplo, a fase em que se deve verificar se a inicialização de um certo sistema foi feita corretamente e que ele está pronto para receber requisições é o momento em que suas threads representando serviços que recebem requisições foram iniciadas e estão em um estado de espera. Portanto, uma asserção a respeito da inicialização do sistema só deve ser feita quando esse estado das threads tiver sido alcançado.

A abordagem apresentada pode ser vista como um mecanismo geral e transparente de sincronização entre o teste e o sistema testado e no qual condições de corrida nos momentos de asserção são evitadas. Sua idéia básica é que sejam oferecidas aos testadores algumas primitivas para fazer seus testes esperarem (ex: `waitUntilStateIsReached`) antes que sejam executadas as asserções. Tais primitivas devem ser oferecidas por ferramentas de teste que monitorem as threads do sistema de maneira não invasiva ao desenvolvimento (como com o uso de técnicas de instrumentação como Programação Orientada a Aspectos [54]).

Além de trazer como contribuição a abordagem em si, neste trabalho é apresentada também uma ferramenta de testes de suporte à abordagem e que pode ser utilizada em diferentes sistemas, e que foi inspirada em uma solução menos geral, denominada *ThreadServices* [25], para o problema dos falsos positivos em testes no sistema OurGrid. A abordagem apresentada foi avaliada tanto através de estudos de caso práticos utilizando essa ferramenta, quanto através de sua modelagem formal utilizando a linguagem TLA+ [58], modelagem esta que teve o propósito de demonstrar que a abordagem consegue evitar o problema das asserções antecipadas e tardias.

1.2 Objetivos

De maneira geral, este trabalho apresenta uma abordagem para evitar o problema dos falsos positivos por asserções antecipadas e tardias em testes, aumentando a confiança em seus

resultados. Pretende-se ajudar a garantir que quando um teste reporte uma falha, esta falha tenha sido causada por um defeito no software e não por uma asserção feita em um momento inapropriado. Embora o problema dos falsos negativos seja também relevante, este não é o foco deste trabalho.

Buscando aumentar a confiança dos desenvolvedores nos resultados dos testes produzidos, este trabalho visa responder a seguinte **questão de pesquisa**:

- ***Como evitar falhas em testes de sistemas multi-threaded causadas por asserções realizadas em momentos inapropriados?***

Observando esta questão de pesquisa, este trabalho investiga a seguinte **hipótese**:

- ***Não é possível evitar falhas em testes com assincronia em sistemas multi-threaded causadas por asserções feitas em momentos não apropriados sem monitorar e controlar as threads do sistema sob teste.***

Essa hipótese foi avaliada através da modelagem formal do problema das asserções antecipadas e tardias e dos requisitos de uma solução para resolvê-lo. Além disso, foi apresentada também uma abordagem para o teste de sistemas com operações assíncronas e se demonstrou formalmente que testes que usem tal abordagem não apresentam esse problema. Através de estudos de caso, conseguiu-se também mostrar que a abordagem, que recebeu o nome de *Thread Control for Tests*, pode ser utilizada na prática. Viu-se também que ela conseguiu, nos estudos feitos, evitar o problema e sem demandar alterações no código dos sistemas sob teste.

De maneira geral, pode-se descrever o **objetivo geral** desta tese da seguinte forma:

Prover mecanismos aos testadores para evitar que sejam implementados testes que falhem por asserções feitas em momentos inapropriados, evitando-se assim testes não confiáveis (que levem a falsas suspeitas sobre o sistema sendo testado).

Para atingir esse objetivo geral, algumas etapas intermediárias foram necessárias, as quais estão descritas nos **objetivos específicos** a seguir:

- *Demonstrar que sem monitorar e controlar as threads da aplicação sob teste (criando barreiras de sincronização entre o teste e a aplicação) não se pode evitar o problema*

das asserções antecipadas e tardias. Para essa demonstração, modelou-se formalmente o problema das falhas em testes devidas a asserções feitas em momentos inadequados e os requisitos para uma solução que resolva este problema. As demonstrações feitas buscaram desestimular o uso de estratégias de teste inapropriadas e também melhorar a compreensão acerca do problema das asserções antecipadas e tardias. Esta modelagem formal está descrita no Capítulo 4;

- *Apresentar uma abordagem para testes de sistemas multi-threaded que exercitem operações assíncronas e que se baseia no provimento de primitivas de teste que permitam a espera por parte da thread do teste até que determinados estados das threads do sistema sejam alcançados (ex: todas paradas, todas finalizadas, uma thread específica parada, etc)*. Para atingir este objetivo foi apresentada a abordagem *Thread Control for Tests*, que está descrita na Seção 5.2;

- *Avaliar a abordagem Thread Control for Tests e a viabilidade de seu uso na prática para observar se com ela é possível evitar falhas em testes por asserções feitas cedo ou tarde demais*. A avaliação da abordagem foi feita de diferentes maneiras:

  – Através da modelagem formal da abordagem para testes utilizando para isso a linguagem TLA+ (Temporal Logic of Actions) [58], que permite que verificações (como a ausência de asserções antecipadas ou tardias) possam ser feitas no modelo. Esta modelagem que demonstra que a abordagem *Thread Control for Tests* evita falsos positivos por asserções feitas em momentos inapropriados está apresentada no Capítulo 7;

  – Através do projeto e implementação de uma ferramenta que pode ser usada na prática em diferentes projetos e que não exige alteração no código da aplicação sendo testada. Além da ferramenta em si, que se encontra disponibilizada como uma ferramenta de código aberto no endereço `http://code.google.com/p/threadcontrol`, esse trabalho traz também como contribuição a descrição de sua arquitetura e de alguns detalhes de implementação com o intuito de auxiliar no desenvolvimento de outras ferramentas que dêem suporte ao desenvolvimento de testes envolvendo assincronia de acordo com a abordagem. A descrição

da ferramenta, denominada *ThreadControl* e de sua arquitetura são mostradas na Seção 5.3;

– Com estudos de caso utilizando testes reais onde os problemas que são alvo desta pesquisa são observados, visando verificar se o problema consegue ser resolvido nos casos de uso da abordagem através da ferramenta desenvolvida. Para isso, nesses estudos observou-se a ocorrência de falhas devidas a asserções antecipadas e tardias. O Capítulo 6 trata da avaliação através de estudos de caso da abordagem *Thread Control for Tests* através do uso da ferramenta *ThreadControl* e demonstra que esta pode ser utilizada em casos reais e que conseguiu evitar as falhas causadas por asserções antecipadas e tardias identificadas nesses testes;

– Com um estudo de caso demonstrando o uso da abordagem na prática combinado com uma outra técnica utilizada nos testes de sistemas multi-threaded, os geradores de ruído, e que têm um propósito diferente de nossa abordagem, que é o de fazer com que defeitos de concorrência se manifestem com mais frequência durante a execução de testes. O estudo de caso que foi realizado utilizou uma ferramenta para incentivar diferentes escalonamentos das threads durante re-execuções dos testes (um gerador de ruído - *noise maker*), o ConTest [2], visando investigar formas de minimizar o possível efeito (dependente do ambiente de execução) que a monitoração de threads pode trazer de esconder possíveis defeitos (já que ela influi no escalonamento das threads). Esse estudo ajudou a demonstrar que é possível combinar a abordagem *Thread Control for Tests* com ferramentas como o ConTest, que se utilizam de testes existentes para ajudar na detecção de defeitos de concorrência, como condições de corrida e impasses. Essa é uma contribuição importante, visto que com a abordagem aqui apresentada, os testes utilizados por tais ferramentas serão mais confiáveis (sem falsos positivos por asserções feitas em momentos não apropriados). Além disso, o estudo feito também demonstrou que é importante combinar a abordagem *Thread Control for Tests* com ferramentas como o ConTest visando aumentar a frequência com que determinadas falhas se manifestam em testes, e minimizando assim o efeito de monitoração (diminuição de ocorrência de falhas em testes devido a alteração no escalonamento das threads causado pelo uso de monitores). Este

estudo está detalhado no Capítulo 8.

1.3 Metodologia

De maneira geral, a metodologia utilizada para o desenvolvimento deste trabalho foi a seguinte:

- Revisão constante da literatura focando, em especial, no teste de sistemas *multi-threaded* e no problema específico em que se foca este trabalho, identificando os trabalhos relacionados;
- Identificação de cenários de teste reais onde ocorre o problema de falhas em testes de sistemas *multi-threaded* por asserções feitas em momentos inadequados (asserções antecipadas e tardias);
- Definição e contínuo refinamento de uma abordagem geral para tratar o problema baseada no provimento de primitivas de teste aos desenvolvedores de testes *multi-threaded* que façam com que estes esperem adequadamente antes de realizar asserções;
- Modelagem formal do problema tratado neste trabalho e da abordagem de testes com o intuito de verificar se ela consegue resolvê-lo;
- Avaliação da abordagem através do projeto e implementação de uma ferramenta de suporte ao desenvolvimento de testes que sigam a abordagem;
- Realização de estudos de caso utilizando a ferramenta produzida com o objetivo principal de identificar a diminuição no percentual de ocorrência de falhas por falsos positivos em re-execuções de testes que apresentavam o problema das asserções antecipadas e tardias. A idéia básica foi utilizar testes em que se verificam os problemas de que trata este trabalho. Foram utilizados nos estudos de caso testes de basicamente duas categorias: i) testes pré-existentes, de sistemas reais; ii) testes sintéticos (preparados apenas para o estudo de caso), baseados em testes de sistemas reais e onde se têm maior controle sobre a causa da falha esporádica do teste (falta na aplicação ou problema no teste). Maiores detalhes sobre a metodologia utilizada nos estudos de caso estão descritos no Capítulo 6.

- Realização de estudo de caso para avaliar o uso da abordagem para testes aqui apresentada combinado com técnicas que incentivem diferentes escalonamentos das threads durante re-execuções dos testes. A intenção desse estudo foi investigar a diminuição do possível efeito que a monitoração de threads pode trazer de mascarar ou mesmo de esconder possíveis defeitos (já que ela influi no escalonamento das threads).

- Publicação e apresentação dos resultados parciais da pesquisa à medida que eram obtidos com o intuito de obter contribuições da comunidade e poder assim melhorar o trabalho. Neste sentido, foram preparados os seguintes trabalhos:

  – Um artigo apresentando os benefícios de se usar Programação Orientada a Aspectos no suporte à monitoração de threads durante os testes e que apresenta uma solução utilizada para problemas no teste do sistema OurGrid e que inspirou a abordagem apresentada neste trabalho e o seu arcabouço de suporte [25]. Este trabalho foi publicado no Jornal da Sociedade Brasileira de Computação.

  – Um artigo descrevendo a versão inicial da abordagem *Thread Control for Tests* e o primeiro estudo de caso realizado para avaliá-la, e que foi apresentado no ICST 2008 [24].

  – Um artigo mais detalhado, apresentado no SBES 2008 [26], descrevendo uma versão evoluída da abordagem e do arcabouço que lhe dá suporte e também resultados de um outro estudo de caso utilizando testes de um sistema real.

  – Dois artigos resumidos apresentados em fóruns de estudantes com o objetivo de divulgar o trabalho e coletar feedback de duas comunidades distintas, a comunidade de Programação Orientada a Objetos e Aplicações [22] e a comunidade de Tolerância a Falhas e Confiabilidade de Sistemas [23].

  – Um artigo que engloba a formalização do problema das asserções antecipadas e tardias, a sua proposta de solução com a abordagem *Thread Control for Tests* e a validação dessa abordagem utilizando TLA+ e que está para ser submetido para um jornal.

Maiores detalhes sobre a metodologia utilizada para o desenvolvimento de partes específicas deste trabalho estão descritos ao longo desta tese.

1.4 Organização do Documento

Este documento está organizado em nove capítulos. O Capítulo 2 apresenta a fundamentação teórica, fornecendo uma visão geral da área de testes e um detalhamento do problema de que trata este trabalho; o Capítulo 3 discute os trabalhos relacionados; o Capítulo 4 apresenta a formalização do problema e dos requisitos para uma solução que o resolva; o Capítulo 5 apresenta a abordagem *Thread Control for Tests* e detalhes sobre a versão atual da ferramenta de apoio ao desenvolvimento de testes e que foi desenvolvida para suportar essa abordagem; o Capítulo 6 apresenta as avaliações através de estudos de caso inicialmente desenvolvidas utilizando essa ferramenta; o Capítulo 7 apresenta uma avaliação formal da abordagem utilizando a linguagem TLA+; o Capítulo 8 apresenta um estudo de caso em que se avaliou o uso combinado da abordagem com geradores de ruído; e por fim, o Capítulo 9 apresenta as considerações finais deste trabalho e algumas sugestões de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo será dada uma introdução a alguns conceitos que serão necessários para a melhor compreensão deste trabalho. Em especial, serão apresentados alguns conceitos gerais da área de testes e a sua relação com o problema de que trata esta pesquisa de doutorado. Além disso, o problema da espera em testes que envolvem assincronia será melhor detalhado, juntamente com algumas técnicas comumente usadas em testes que exercitam sistemas com múltiplas *threads*.

2.1 Contextualização

Considerando que existem diferentes semânticas para alguns termos da área de testes, o início deste capítulo se dedica a apresentar as definições que serão usadas neste texto para alguns destes termos, acompanhadas também de uma breve descrição de seu uso neste trabalho.

2.1.1 Conceitos Básicos da Área de Testes

Testes são definidos como a atividade através da qual um sistema ou componente é executado sob determinadas condições e onde os resultados são observados ou gravados e se faz uma avaliação, considerando algum aspecto do sistema ou componente [8]. **Artefatos de teste** são os resultados do teste de software, como casos de teste (conjuntos de entradas de teste, condições de execução e resultados esperados), planos de teste, dados e ferramentas de teste,

dentre outros.

Testes existem porque humanos cometem **erros** no desempenho de atividades relacionadas ao desenvolvimento de software. Uma **falta** é definida como o resultado de um erro cometido por alguém, ou mais precisamente, falta é a representação do erro, ou seja, sua forma de expressão, seja em um texto narrativo, em diagramas de fluxo de dados, diagramas de hierarquia, código fonte (por exemplo, o uso de > ao invés de >=), etc. Neste contexto, **defeito** e "***bug***" são alguns sinônimos para falta.

Para tentar encontrar faltas com o apoio de testes, os testadores exercitam o sistema e verificam se ele se comportou conforme o esperado. Este processo pode ser manual ou automático.

A **falha** é o que ocorre quando a falta causada por um erro é executada [51]. Ela é a observação de que o sistema fugiu de seu comportamento esperado [79]. Por exemplo, quando um trecho de código implementado de forma incorreta é executado fazendo o sistema produzir um resultado não esperado, diz-se que houve uma falha.

Depuração (*Debugging*), por sua vez, corresponde a "Detectar, localizar e corrigir faltas em um programa computacional" [8].

Verificação é um termo também usado neste texto e é importante lembrar que ele pode ser usado em duas concepções: como "(1) o processo de avaliar um sistema ou componente para determinar se os produtos de uma determinada fase de desenvolvimento satisfazem as condições impostas no início daquela fase." e também como "(2) prova formal da corretude do programa" [8].

Os **testadores** são definidos como sendo as pessoas que têm como objetivo principal encontrar faltas no software e garantir que as faltas encontradas foram corrigidas de alguma forma [77].

Testes automáticos correspondem à gerência e cumprimento de atividades de teste que incluem o desenvolvimento e execução de scripts de teste com o propósito de verificar requisitos de teste utilizando uma ferramenta automática [28] (e.g. JUnit [65]). Tais atividades se mostram de grande valor quando o desenvolvimento é marcado por mudanças contínuas, atuando, aí, como um mecanismo de controle importante para garantir a estabilidade do software a cada nova reconstrução (*build*).

Quando se desenvolve testes automáticos de software, é comum a utilização da expressão

asserção, definida como: "Uma expressão lógica que especifica um estado que deve existir no programa ou um conjunto de condições que as variáveis do programa devem satisfazer em um ponto particular durante a sua execução" [8].

O termo "asserção", para a área de testes, se popularizou com o uso de ferramentas como o arcabouço JUnit [65] ou outras ferramentas da família XUnit que oferecem operações como `assertTrue(expressaoBooleana)` ou `assertEquals(resultadoEsperado,sistema.getResultadoProduzido())`.

A idéia básica de operações de asserção em testes automáticos é verificar como o sistema se comportou depois de exercitado, observando se os resultados produzidos equivalem ao que se esperava de uma execução correta. Caso haja divergência entre o que se esperava e o que foi obtido, ou seja, se a asserção falhou, diz-se que o **teste falhou** [66].

2.1.2 Este Trabalho no Contexto da Área de Testes

Considerando os conceitos que foram apresentados e o capítulo de introdução desta tese, é importante destacar o seguinte:

*O trabalho descrito nesta tese se insere na área de **testes**. De maneira geral, o trabalho visa dar suporte aos **testadores** no desenvolvimento de **testes automáticos** que têm como objetivo exercitar o sistema a ser testado e **verificar** se ele se comporta como previsto (definição 1 de verificação). Através da melhoria da qualidade dos testes, deixando-os mais confiáveis e evitando que falhem por **asserções** feitas em momentos inadequados, este trabalho busca evitar falsos positivos em testes. Poder-se-á assim evitar desperdício de tempo de alguma **depuração** motivada por testes que falham devido a problemas com o próprio teste e não com a aplicação em si.*

Este trabalho foca na **verificação automática** do comportamento do software, por ser considerada uma atividade que traz grandes benefícios para o aumento da produtividade do desenvolvimento, melhoria da qualidade do software e por evitar que o software se torne quebradiço (*brittle*) [66]. Além disso, é uma atividade que apresenta vários desafios.

Um dos desafios da verificação automática, considerando em especial o contexto dos testes de sistemas *multi-threaded*, é que em diferentes execuções dos testes exercitando uma determinada operação do sistema, é possível encontrar diferentes maneiras em que as threads do sistema foram escalonadas para executar tal operação. Porém, independente do escalona-

mento, há determinadas propriedades que devem ser verdadeiras ao fim da operação e que podem ser verificadas automaticamente via asserções. Quando elas não são verdadeiras em algumas das execuções, têm-se as falhas esporádicas em testes, cuja causa é normalmente difícil de ser detectada.

No entanto, algumas falhas esporádicas não são devidas a defeitos no software, mas sim a problemas com os testes. Para melhor compreender tais problemas, a seguir serão introduzidos o processo de testes de operações assíncronas e as técnicas comumente utilizadas para se determinar o momento de se fazer asserções em tais testes.

2.2 Esperando antes de Asserções

Em geral, quando há operações assíncronas a se testar, as seguintes fases devem estar presentes nos testes:

1. *Exercitar o sistema*;
2. *Esperar*; e
3. *Fazer asserções*.

Para ilustrar estas fases, pode-se considerar mais uma vez o exemplo do teste da inicialização de um sistema composto por vários serviços para saber se tais serviços foram iniciados e ficaram prontos para receber requisições.

Na **fase 1** de tal teste teria-se a seguinte operação: `mySystem.initialize()`. Essa operação inicia o sistema, e para isso, cria outras *threads*, que podem, por sua vez, criar outras, representando os diferentes serviços. Todas essas *threads* irão executar algumas operações e então esperar por outros estímulos. Este exemplo está ilustrado pela Figura 2.1. Nesta figura mostra-se um caso em que a operação `mySystem.initialize()` levou à iniciação de uma thread B, que por sua vez iniciou duas threads do tipo A. Uma dessas threads A criou duas threads do tipo C.

Considerando esse mesmo exemplo, a fase de asserções (**fase 3**) deste teste pode apresentar o seguinte código: `assertTrue(isCorrectlyInitialized(mySystem))`. Esse código verifica se o sistema foi corretamente iniciado, podendo verificar, por exemplo,

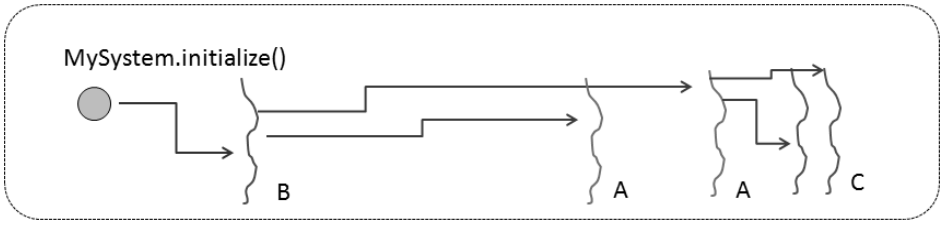


Figura 2.1: Exemplo: Threads criadas por uma invocação feita por um teste

se o estado de seus serviços, representado por variáveis do sistema, está "ATIVO"e se suas filas de requisições estão vazias. Caso o sistema não tenha sido corretamente iniciado (por exemplo, estando em um estado diferente ou apresentando requisições em sua fila), o teste irá falhar.

Entretanto, como a inicialização é assíncrona, antes de verificar se o sistema foi corretamente iniciado, deve-se esperar até que o processo de iniciação tenha sido concluído, uma fase da execução do sistema que pode ser mapeada em um certo estado para as suas threads (e.g. threads do tipo C finalizadas e threads dos tipos A e B em espera). Identificar quanto tempo esperar antes das asserções, especialmente quando múltiplas *threads* são criadas, pode ser desafiador e requerer muitas mudanças do código. Além disso, dependendo da abordagem utilizada para fazer o teste esperar, algumas falhas de testes podem acontecer, como será detalhado adiante.

De maneira geral, quando há operações assíncronas em um teste, algumas das formas comuns de implementar a fase de espera (**fase 2**) antes das asserções são:

1. *Atrasos explícitos.* Invocações a operações que fazem a thread do teste esperar por um tempo fixo, passado como parâmetro. Ex:

```
Thread.sleep(t);
```

2. *Espera ocupada.* Laços que apresentam atrasos explícitos em seu interior e que verificam de tempos em tempos uma determinada propriedade a respeito do sistema até que a condição verificada no laço seja verdadeira. Ex:

```
while (!stateReached())
  Thread.sleep(t);
```

3. *Controle das threads.* Barreiras de sincronização ou operações que fazem a thread do teste esperar enquanto outras threads atingem um estado determinado, como por exemplo o estado "terminadas" (*finished*). Ex:

```
myThread.join();
```

Atrasos explícitos e espera ocupada

Atrasos explícitos (*explicit delays*) podem ser facilmente encontrados em testes, especialmente por serem a abordagem de mais fácil implementação. No entanto, seu uso pode resultar nos seguintes problemas: testes que não passam devido ao uso de intervalos insuficientes de espera; ou testes que podem levar muito tempo para executar devido ao uso de intervalos de espera muito longos [66] (p. 255). Essa técnica assume que após um certo intervalo de tempo t o sistema estará em um estado específico, o que nem sempre é verdade dependendo do ambiente de execução dos testes. Um outro problema que o uso desta técnica pode gerar são asserções tardias, caso a espera seja grande demais e o sistema não esteja mais no estado esperado. O problema principal desta técnica é que o tempo adequado para esperar depende da máquina sendo utilizada e da carga a que está sendo submetida, o que faz com que dependendo do intervalo de tempo a esperar adotado, um teste falhe em uma máquina e passe em outra.

O uso da técnica de espera ocupada (*busy wait*) também é comum, especialmente quando a condição sendo verificada a cada iteração de seu laço é facilmente obtida do sistema. É fácil encontrá-la em códigos Open Source e discussões na Internet [76], por exemplo. Porém, o uso de espera ocupada apresenta algumas desvantagens [60] (p. 196-197): i) tal técnica pode levar a um desperdício de tempo de CPU devido a várias execuções do laço sem necessidade (o que nem sempre é crítico, dependendo do tipo de teste); ii) pode-se passar do ponto certo em que a condição verificada é verdadeira por alguns instantes (algo que pode acontecer também com atrasos explícitos); iii) podem-se usar determinadas construções de linguagens de programação que podem não ser efetivas no sentido de deixar outras threads executarem (como o `Thread.yield()` de Java, que depende das políticas da JVM sendo usada).

É importante destacar também que quando a condição de espera utilizada é a mesma que se pretende verificar nas asserções, é indicado que se use uma versão mais elaborada de

espera ocupada baseada em estouro de tempo (*timeouts*). Isso deve ser feito para evitar que a execução do teste entre em um laço infinito caso a aplicação apresente um defeito que a impeça de atingir aquele estado. Essa situação também pode ser desejável mesmo quando a condição de espera não é a mesma da asserção. Um exemplo de código com espera ocupada baseada em estouro de tempo é mostrado a seguir:

```
begin = Now();
while (! condicao() && (Now() - begin) < TEST_TIMEOUT) {
    Thread.sleep(t);
}
```

Controle das *Threads*

Uma maneira mais segura de saber quando uma asserção pode ser executada é através do controle das *threads* criadas pelo teste. Deve-se garantir que elas terminaram ou que estão em um estado estável (ex: todas já iniciaram e alcançaram um estado de espera enquanto requisições não chegam) antes que as asserções aconteçam. Quando se tem no teste as instâncias dos objetos que representam as *threads*, isso é mais simples, e operações como `join()` da classe `Thread` de Java podem ser invocadas. Tal operação faz com que o teste só prossiga quando a *thread* tiver concluído o seu trabalho.

No entanto, nem sempre é possível ter acesso a todas as instâncias de todas as *threads* criadas por um teste. Algumas vezes, uma monitoração adicional das *threads* da aplicação é necessária para indicar quando fazer as asserções. Além disso, precisa-se, por vezes, de algum mecanismo de controle, como barreiras de sincronização, para que se garanta que a asserção seja feita de forma segura, sem que se passe do ponto, evitando assim que algumas *threads* acordem no momento em que asserções estão sendo executadas e possam interferir nos resultados de teste, fazendo com que embora o estado esperado tenha sido alcançado, o sistema já tenha saído desse estado no momento em que a asserção é executada. Por exemplo, no teste da inicialização do sistema em que se verifica se seus serviços estão ativos e se as filas de requisição estão vazias, tais filas podem não mais estar nesse estado pelo fato de alguma thread ter acordado e gerado novas requisições.

2.3 O Problema Tratado

Durante as execuções de testes automáticos de alguns sistemas sendo desenvolvidos no Laboratório de Sistemas Distribuídos (LSD) da UFCG, percebeu-se que era comum encontrar testes que apresentavam falhas intermitentes devido ao seguinte problema nos testes: *os resultados produzidos para as operações exercitadas nestes testes eram verificados (através de asserções) em momentos inadequados, o que tornava tais testes não confiáveis*.

Em conversas informais com profissionais da indústria, percebeu-se que esse problema também se manifestava nas execuções de baterias de teste de empresas. Além disso, a observação de testes automáticos de projetos de código aberto que fez parte de um trabalho de estágio supervisionado pela autora desta tese também indicava que o mesmo problema poderia acontecer [38].

Observou-se que isso acontecia, em vários dos casos, pelo fato das técnicas de atrasos explícitos e espera ocupada serem bastante comuns na prática. Porém, observando-se tais técnicas, viu-se que elas podem levar aos seguintes problemas: i) testes que levam muito tempo para executar, devido à escolha de valores altos para tempo de espera [66]; ii) ou testes que falham devido a intervalos de espera insuficientes ou por terem passado do ponto ideal para verificação (o efeito *miss of opportunity to fire*) [60].

Quando o segundo caso ocorre, o que corresponde aos falsos positivos em testes, os desenvolvedores da aplicação que dependem de testes para identificar se mudanças afetaram negativamente outras partes do código já existentes passam a perder a confiança nos resultados de testes, ou então desperdiçam muito tempo depurando a aplicação por atribuírem a falha do teste a uma falta na aplicação.

Quando não há mecanismos que monitorem e ofereçam um controle mínimo da execução do teste para identificar quando asserções podem ser feitas acaba-se tendo uma condição de corrida (*race condition*) durante o teste, se considerarmos a definição de MacDonald et al. [64]. Tal definição afirma que existe uma condição de corrida se há duas operações que devem executar numa ordem específica, mas não há sincronização suficiente para garantir essa ordem. Trazendo para a execução do teste, pode-se dizer que o problema das asserções feitas em momentos inadequados pode ser visto como a falta de sincronização entre a operação de asserção e as operações do sistema que garantirão que o que será checado na asserção

tenha sido produzido.

Porém, nem sempre é desejável ou mesmo possível introduzir mecanismos de sincronização explícitos entre os testes e a aplicação, sendo interessante buscar meios menos invasivos para isso, como é o caso da Programação Orientada a Aspectos, um paradigma que será explicado adiante.

2.4 Programação Orientada a Aspectos

A separação de interesses (*separation of concerns*) é um dos princípios da Engenharia de Software que se aplicam ao longo do processo de desenvolvimento de software [40] [92]. Ele permite que se lide com diferentes aspectos de um problema, concentrando-se em cada um separadamente. A Programação Orientada a Aspectos (*Aspect-Oriented Programming - AOP*) [54] surgiu como uma nova forma de prover separação de interesses [31] e tem como objetivo principal tornar o projeto do software e seu código modularizado em situações onde técnicas puras de Programação Orientada a Objetos levariam a problemas como espalhamento e entrelaçamento de código. Tais situações são comuns especialmente quando os interesses a serem implementados são transversais (*crosscutting*) e sua implementação envolve diferentes objetos ou operações espalhados ao longo da aplicação.

Um exemplo de linguagem de programação orientada a aspectos é AspectJ, que é uma extensão da linguagem Java [53]. AspectJ dá suporte ao conceito de pontos de junção (*join points*), que são pontos bem definidos no fluxo de execução de um programa [93]. Essa linguagem também apresenta uma forma de identificar pontos de junção particulares (denominados *pointcuts* ou pontos de corte) e mudar o comportamento da aplicação nesses pontos, o que é especificado através de declarações denominadas adendos (*advice*).

Um exemplo de ponto de corte é o que coleciona os pontos ao longo da execução de uma aplicação em que o método `sleep` da classe `Thread` é chamado. Esse ponto de corte pode ser definido da seguinte forma:

```
pointcut chamadasSleep():
    call (public static void Thread.sleep(long));
```

Declarações de adendos são usadas para definir trechos de código que devem ser executados quando os pontos identificados por um ponto de corte são alcançados. Por exemplo,

pode-se definir que antes que alguma thread da aplicação comece a dormir, seja impressa uma mensagem na tela. Isso pode ser feito através da seguinte definição de adendo:

```
before(): chamadasSleep() {
    System.out.println("Uma thread irá dormir");
}
```

Além do `before`, AspectJ também provê adendos dos tipos `after` e `around`. O primeiro executa depois que os pontos de junção identificados pelo ponto de corte concluem sua computação. O segundo executa assim que o ponto de junção é atingido, e tem controle sobre se a execução da computação envolvida naquele ponto deve proceder ou não (podendo ser inclusive substituída pelo código definido no adendo do tipo `around`) [93].

AspectJ também pode afetar estaticamente um programa. Isso se dá através de declarações inter-tipo (*Inter-type declarations*). Por exemplo, pode-se mudar ou acrescentar membros, como métodos e variáveis, a uma classe ou alterar o relacionamento entre classes.

Em AspectJ há também o conceito de um aspecto, que é uma unidade modular de implementação transversal (*crosscutting*). Um aspecto é definido de forma semelhante à de uma classe em Java, e pode ter métodos, construtores, blocos de inicialização, pontos de corte, adendos e declarações inter-tipo. Uma definição de aspecto pode ser feita da seguinte forma:

```
public aspect MonitoracaoDeThreads {
...
}
```

O código definido pelo aspecto é combinado com o código da aplicação através de um *weaver* [100] ou combinador. No caso de AspectJ, esse processo é feito por um combinador chamado **ajc**, uma ferramenta que faz a combinação dos códigos e a compilação, gerando um bytecode que uma vez executado incorpora o código adicionado pelos adendos. O processo de instrumentar o código de uma aplicação e de seus testes com os adendos especificados através de aspectos está ilustrado na Figura 2.2.

Alguns trabalhos têm explorado a Programação Orientada a Aspectos no contexto de testes. Exemplos desses trabalhos são os de Copty e Ur [20], Rocha et al. [81] e MacDonald [64], os quais investigam o quão apropriado é o uso de AOP para a implementação

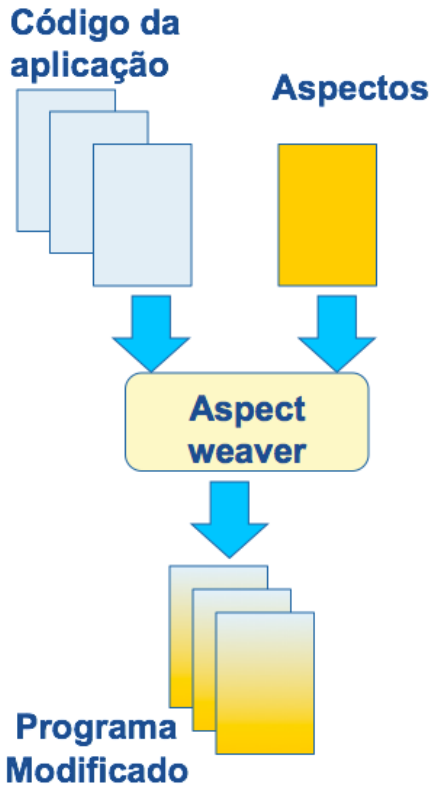


Figura 2.2: Processo de combinação do código da aplicação com o código definido nos aspectos

de ferramentas de teste. No primeiro trabalho, AspectJ foi usada para implementar a ferramenta ConTest. Essa ferramenta é utilizada para forçar diferentes escalonamentos de *threads* durante várias re-execuções de casos de teste, pretendendo, assim, revelar problemas de concorrência. O segundo trabalho [81] apresenta a J-Fut, uma ferramenta para o teste funcional de programas Java e que usa AOP para instrumentar e analisar o programa sendo testado. No último trabalho [64], AOP é combinada com outra tecnologia denominada CSSAME para executar casos de teste de forma determinística, exercitando diferentes caminhos numa condição de corrida.

2.5 Aplicações Multi-threaded em Java

A plataforma Java foi projetada para suportar programação concorrente através da linguagem de programação em si e também através de bibliotecas da linguagem. A partir de sua versão

5.0, a plataforma Java também incluiu bibliotecas de concorrência em alto nível [101].

Na programação concorrente, há basicamente duas unidades de execução: processos e threads. Em Java, a programação concorrente foca-se principalmente em threads. Processos são normalmente vistos como sinônimos para programas e aplicações. Threads são algumas vezes chamadas de *processos light*, uma vez que sua criação requer menos recursos que a criação de um processo. Na verdade threads existem dentro de um processo e cada processo tem ao menos uma thread (do ponto de vista do usuário), a thread principal.

Cada thread está associada com uma instância da classe `Thread` de Java. Para criar uma instância dessa classe, uma aplicação deve prover o código que irá executar naquela thread. Há duas formas de fazer isso:

- **Criando um objeto *Runnable*.** Essa interface define um único método, o `run`, com o código para ser executado na thread. O objeto `Runnable` que é criado é passado para o construtor da Thread. Ex: `(new Thread(new MeuRunnable())).start()`

- **Criando uma subclasse da classe `Thread`.** A classe `Thread` por si só já implementa a interface `Runnable`, mas a intenção de criar uma classe que a estenda é sobrescrever seu método `run`, definindo as ações da thread uma vez que passa a executar. Ex: `(new MinhaThread()).start()`

Pelos exemplos, pode-se verificar que para iniciar uma thread chama-se o método `start` da classe Thread e uma vez que esse método é chamado o escalonador do Sistema Operacional pode colocar a thread para rodar (executando as instruções definidas no `run`).

Para pausar a execução da thread corrente por um período de tempo especificado, utiliza-se o método estático `Thread.sleep`.

A classe `Thread` apresenta ainda um método para fazer uma thread esperar pela conclusão da execução de uma outra. Sendo `t` uma `Thread`, a invocação do método `t.join()` faz com que a thread corrente pause sua execução até que a thread `t` termine.

Para sincronizar atividades entre threads e evitar problemas como erros de consistência de memória, pode-se utilizar o recurso de métodos ou blocos sincronizados da linguagem Java. Para tornar um método sincronizado, basta adicionar a palavra-chave `synchronized` à sua declaração. Isso faz com que duas invocações a métodos sincronizados no mesmo objeto não

possam ocorrer ao mesmo tempo. A sincronização nesses casos é baseada em uma tranca (*lock*) intrínseca ou monitor, pois cada objeto já apresenta uma tranca intrínseca associada a ele. Quando uma thread invoca um método sincronizado, ela adquire a tranca para o método daquele objeto e libera essa tranca quando o método retorna.

Para criar blocos sincronizados, deve-se especificar o objeto que provê o lock intrínseco. Exemplo:

```
public void meuMetodo(){
  synchronized(objeto){
     cont++;
  }
}
```

Um outro aspecto relacionado à forma de coordenar ações entre threads são bloqueios com guarda (*guarded blocks*). Em Java, uma forma de prover guarda é através do método `wait` da classe `Object` para suspender a thread corrente. A invocação de `wait` não retorna enquanto outra thread não tiver gerado uma notificação de que algum evento especial ocorreu (não obrigatoriamente o evento que se está esperando) e para se invocar o método `wait` sobre um objeto é preciso que se tenha a tranca intrínseca sobre esse objeto.

A notificação por parte de outras threads é feita através da operação `notifyAll` ou `notify` da classe Object. No caso da primeira, todas as threads esperando naquela tranca serão informadas de que algo aconteceu. No caso da `notify` apenas uma das threads esperando por uma tranca será despertada.

Um outro ponto importante relacionado ao desenvolvimento de aplicações multi-threaded em Java é que a linguagem oferece também em sua biblioteca objetos de alto nível para prover concorrência, que foram introduzidos na versão 5.0 de Java e estão implementados nos pacotes `java.util.concurrent` e em estruturas de dados concorrentes do *Java Collections Framework*. Exemplos de novos elementos introduzidos à linguagem foram a interface `Blocking Queue`, com operações bloqueantes como `put` e `take` e a classe `Semaphore`, que representa um semáforo, com operações como `acquire` e `release`.

2.6 Efeito da Monitoração

Como a proposta de solução apresentada neste trabalho baseia-se em monitoração de threads durante os testes, é importante também explicar um efeito conhecido durante a monitoração.

Quando se tem um teste que revela uma falta, nem sempre é fácil reproduzir as condições que levaram esta a ocorrer na hora de depurar o problema detectado. Embora se consiga reproduzir, a própria depuração pode mascarar o defeito, o que se chama de efeito do observador (*observer effect*) [10]. Por exemplo, adicionar a impressão de alguma mensagem de depuração pode alterar as condições de tempo para executar alguma tarefa e assim mascarar defeitos relacionados a concorrência.

O mesmo ocorre com monitoração, que é a técnica utilizada neste trabalho de tese. Tanto esta técnica quanto a depuração podem introduzir atrasos ou mecanismos de sincronização que podem evitar que o defeito se revele durante os testes [43]. Como também observado por Gait [37], ao inserir sensores em um sistema, os processos podem executar em uma ordem distinta. Isso pode ser um problema, por exemplo, quando se está tentando localizar uma falta e a monitoração mascara essa falta. Gait se refere a esse efeito intrusivo usando o termo "efeito da monitoração" (*probe effect*) e o define como sendo a diferença em comportamento entre um objeto monitorado e sua versão não instrumentada.

Considerando tal efeito, um dos trabalhos realizados no contexto desse trabalho de doutorado foi um estudo de caso para investigar este efeito durante a execução dos testes ao se utilizar as idéias apresentadas neste documento e ver como ele pode ser minimizado com técnicas auxiliares de teste, como geradores de ruído, o que é detalhado no Capítulo 8.

2.7 Conclusões Parciais

Neste capítulo foram discutidos alguns conceitos da área de testes e de outros assuntos que serão abordados ao longo desta tese. Foram também detalhadas algumas técnicas utilizadas para espera quando se tem testes envolvendo assincronia e foi mais claramente definido o problema tratado por este trabalho. A principal observação a destacar é que o uso de técnicas inadequadas pode levar a baterias de teste que demoram muito para executar (devido a intervalos de espera exagerados) ou a falhas desnecessárias nos testes, tornando-os menos

confiáveis. A principal consequência de testes não confiáveis é o aumento do risco do desenvolvimento, em especial quando se tem um processo fortemente baseado em testes, como é o caso de vários processos ágeis.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados os principais trabalhos que se relacionam a esta tese, com o intuito de identificar em que pontos se diferenciam deste. Além disso, esse capítulo visa também dar uma idéia geral destes trabalhos uma vez que alguns deles podem ser utilizados em conjunto com o que é apresentado nesta tese.

Os trabalhos apresentados a seguir foram encontrados tanto através de buscas exaustivas utilizando principalmente o engenho de busca Google, o Google Acadêmico e bibliotecas digitais da ACM e IEEE, quanto varrendo as referências dos principais trabalhos que eram encontrados e que apresentavam maior relação com o tema desta tese. Além disso, parte dos trabalhos apresentados neste capítulo surgiram através de questionamentos e sugestões dadas em apresentações deste trabalho em congressos e na avaliação da proposta de qualificação desta tese. Outra parte dos artigos aqui apresentados foi extraída através da análise de artigos produzidos nas principais conferências da área de testes e engenharia de software.

Foram considerados mais relacionados os trabalhos que tratavam de falsos positivos em testes envolvendo assincronia, por ser este o principal problema atacado por esta tese. Além destes, foram investigados trabalhos que tratam de mecanismos de sincronização, monitoração de threads e controle durante os testes, por estarem relacionados à solução que é proposta nesta tese. Além disso, foram analisados também trabalhos que tratam de ferramentas para auxiliar na identificação de defeitos de concorrência, algo complementar ao que é proposto nesta tese, uma vez que tais trabalhos têm o objetivo complementar de aumentar a confiança nos resultados de testes tendo principalmente o foco de mais facilmente expor defeitos de concorrência e não considerando a questão dos falsos positivos por problemas com os testes

em si.

Os trabalhos selecionados foram agrupados segundo suas características principais, embora alguns possam pertencer a mais de um grupo. Em cada seção apresentada a seguir será descrito o porquê daquele grupo de trabalhos ser considerado nesta revisão bibliográfica e são dados maiores detalhes sobre os trabalhos do grupo e sua relação com os resultados obtidos neste trabalho. Alguns trabalhos apresentam relação com o problema atacado neste trabalho de doutorado e com seus objetivos, outros apresentam diferentes propósitos do aqui descrito, mas se relacionam ao teste de sistemas concorrentes e podem em alguns casos ser utilizados conjuntamente com este, sendo complementares. Alguns outros trabalhos são relacionados à solução aqui proposta de monitoração e controle das threads e por isso serão também discutidos a seguir.

De maneira geral, considerando esses critérios, os trabalhos apresentados nas seções a seguir podem ser organizados conforme apresentado na Tabela 3.1.

3.1 Uso do Padrão *Humble Object* para Evitar Testes Assíncronos

Quando se trata do problema de se testar aplicações envolvendo assincronia, um trabalho relacionado ao que é apresentado nesta tese é o livro de Meszaros [66]. Neste livro são discutidos diversos problemas detectados na implementação de testes de unidade e são propostos padrões com boas práticas para o desenvolvimento desses testes.

Alguns dos problemas apontados se referem à dificuldade em desenvolver testes que envolvem código assíncrono e o problema de testes que demoram muito devido a fases de espera baseadas em atrasos explícitos. Além de apresentar estes problemas, esse trabalho sugere que testes com operações assíncronas devem ser evitados no nível de testes de unidade e de componentes, através do uso de padrões como o *Humble Object* [66] (p. 695). A idéia desse padrão é que seja realizada no código a separação entre a lógica e os mecanismos assíncronos, permitindo assim que os testes possam ser realizados diretamente sobre a parte do código que trata da lógica, que seria acessada via testes síncronos. Dessa forma, os testes não precisarão de qualquer espera a não ser a da execução síncrona da lógica que antes era executada por uma thread diferente da do teste. Entretanto, tal estratégia pode ser onerosa

Tabela 3.1: Visão Geral dos Trabalhos Relacionados

TIPOS DE TRABALHOS RELACIONADOS	**TRABALHOS NESTA CATEGORIA**
Trabalhos com o mesmo objetivo/problema	Padrão *Humble Object* [66] Problemas com Asserções em Projeto por Contrato (DBC) [82] [73]
Trabalhos relacionados à solução proposta mas com objetivos ortogonais ou complementares	Mecanismos de Sincronização em Testes (ex: PNUnit [84] e TETware [96]) Controle de Testes [13] [62] [7] Controle de Testes utilizando Máquinas de Estado Finitas(FSMs) [98] [39] Monitoração de Threads com Monitor de Referências [70] Monitoração de Threads com Programação Orientada a Aspectos [19]
Trabalhos com objetivo complementar	Trabalhos sobre identificação de defeitos de concorrência (Seção 3.6): Detecção de Condições de Corrida Análise Estática Verificação de Modelos em Programas Concorrentes Técnicas de Re-execução (*Replay*) Técnicas para forçar determinados escalonamentos Geradores de Ruído

de ser implantada, pois há a necessidade de adicionar uma nova camada na aplicação sob teste. Além disso, testes envolvendo os componentes do sistema que chamam essa camada e que envolvem assincronia acabam não sendo cobertos por essa abordagem de testes que foca mais em testes unitários.

O trabalho apresentado nesta tese sugere uma solução para os casos em que não se pode, ou não se quer, durante certos testes, evitar o uso de operações com assincronia dos componentes, como em alguns testes de sistema ou de integração em que não se deseja criar uma camada síncrona para partes do sistema só por causa dos testes, o que pode torná-los mais complexos. Em tais situações, uma abordagem como a que será apresentada nesta tese e descrita na Seção 5.2 ajuda a lidar com os desafios dos testes com assincronia sem demandar grandes mudanças na aplicação para torná-la mais testável.

3.2 Mecanismos de Sincronização em Testes

Uma das formas de evitar que asserções de testes sejam feitas em momentos não adequados é provendo mecanismos de sincronização entre testes e a aplicação sob teste.

Uma das discussões encontradas na literatura sobre uso de mecanismos de sincronização em testes é feita no artigo de Santos e Garcia [84]. Nesse artigo é discutida a ferramenta PNUNit, que é uma extensão da ferramenta NUnit [6], que foi originalmente concebida através do porte da ferramenta JUnit [65] para .NET.

O PNUnit possibilita a execução de vários testes de unidade em paralelo em diferentes ambientes (sistemas operacionais e plataformas de hardware). Neste artigo, para sincronizar as diferentes execuções paralelas em determinados pontos, são utilizados mecanismos de sincronização baseados em barreiras (*barriers*). Especifica-se então quantos elementos devem passar pela barreira para que a execução da bateria de testes paralelos prossiga, ou então configura-se a barreira para que ela só seja liberada quando todos os testes passarem por ela. Embora usem mecanismos de sincronização baseados em barreiras, os autores também relatam que seria simples incluir também primitivas na ferramenta para o uso de outros mecanismos, como semáforos, por exemplo. A abordagem com barreiras é apresentada nesse trabalho como uma primitiva básica de sincronização e lembra os mecanismos de sincronização não explícitos baseados em primitivas de teste, como o `waitUntilStateIsReached`,

que este trabalho de tese propõe no arcabouço `ThreadControl`. A diferença é o foco desta tese em usar esses mecanismos para evitar asserções antecipadas e tardias e também o fato da barreira criada ser um mecanismo de sincronização entre a execução do teste e da aplicação sendo testada, enquanto que no PUNit busca-se uma sincronização entre os testes executados em paralelo.

Uma outra referência que aponta o uso de sincronização para coordenar testes distribuídos é um dos relatórios técnicos do TETware [94]. O TETware [96] é um arcabouço universal para gerência e geração de relatórios para testes automáticos e que provê uma API entre o código de teste e o processo geral de testes. O TETware tenta refletir a necessidade da comunidade de testes de uma interface comum para suportar funções básicas para testes, como por exemplo geração de relatórios, comunicação e ordenamento de testes. É importante destacar que ele gerencia testes locais, remotos e também testes distribuídos.

Quando um caso de teste é distribuído em várias partes, cada uma processada em um sistema diferente, e estas partes interagem umas com as outras, é importante garantir um certo ordenamento entre o que acontece nos diferentes sistemas. Para isso, o TETware se utiliza de sincronização dos casos de teste. Tal sincronização provida pelo arcabouço tanto pode ocorrer automaticamente, em pontos determinados do sistema (relacionados ao início e fim de testes), ou o usuário pode determinar os pontos em que deve haver sincronização de forma explícita, fazendo com que os componentes tenham pontos de sincronização conhecidos uns pelos outros.

O TETware se assemelha a este trabalho de tese por também tentar dar suporte aos desenvolvedores de testes, embora sob outra perspectiva complementar, a de coordenar os diferentes casos de teste que compõem um dado teste. Na abordagem que será descrita na Seção 5.2 desta tese e em especial no arcabouço de testes que dá suporte a esta abordagem, denominado *ThreadControl*, explicado na Seção 5.3, podem ser usados diferentes pontos de sincronização para definir quando uma asserção pode ou não ser feita e não só pontos dados em função do ciclo de vida do teste, como no TETware. Alguns dos pontos sugeridos ao apresentar o *ThreadControl* são pontos da aplicação referentes ao ciclo de vida de suas threads (ex: todas terminadas ou em estado de espera). Um outro diferencial do *ThreadControl* é o fato de buscar evitar que, para definir pontos pré-definidos pelo usuário para sincronização, o código da aplicação tenha de sofrer alterações através da definição explícita

desses novos pontos no código de cada componente. Isto foi feito nesse arcabouço utilizando uma abordagem de monitoração menos intrusiva, a Programação Orientada a Aspectos [54]. Uma outra diferença entre as ferramentas *ThreadControl* e TETWare é que o *ThreadControl* se utiliza das abstrações do JUnit [65] (ou de forma geral de outras ferramentas da família XUnit) na definição dos seus casos de teste, visando uma maior adoção pelos desenvolvedores, enquanto que o TETWare depende de uma nova forma para definir seus casos de teste explicitando cada componente distribuído isoladamente, estando o código de cada um dos processos distribuídos com chamadas explícitas a módulos do TETware que irão prover a sincronização [95]. Já no uso da abordagem apresentada nesta tese através do arcabouço *ThreadControl*, o uso de primitivas que permitem que o teste espere por determinados estados do sistema ficaria explicitada apenas no teste e não nos componentes sendo testados.

3.3 Controle de Testes

Como a solução apresentada nesta tese para evitar que asserções sejam feitas em momentos inadequados se baseia na monitoração e controle das threads da aplicação, é importante discutir nessa seção trabalhos que tratam de "Métodos de Controle de Testes" (*Test Controllability*).

Alguns dos trabalhos desta área visam fazer com que as threads da aplicação, ao executarem, sigam intercalações (*interleavings*) pré-definidas e controladas, de forma a prover uma maior repetibilidade dos testes (algo que não é foco deste trabalho de tese, mas complementar) e controlar o não-determinismo de tais testes. Um destes trabalhos é o de Cai e Chen [13], que trata do controle do não-determinismo no teste de sistemas *multi-threaded* distribuídos. Nesse trabalho de Cai e Chen é apresentado o arcabouço utilizado para um conjunto de ferramentas de controle de testes automatizado e que permite uma forma automática de fazer com que o sistema siga certos caminhos de execução particulares. A especificação do teste é baseada no caso de teste em si e em restrições de controle especificando a ordem parcial entre determinados eventos que se quer observar.

O trabalho desta tese se diferencia do de Cai e Chen por não focar na especificação de intercalações das threads do sistema durante a execução do teste, mas apenas em garantir que as asserções ocorrerão no momento apropriado, podendo assim serem considerados tra-

balhos complementares. A idéia do trabalho desta tese é auxiliar em abordagens de teste em que se deseja executar um mesmo teste diversas vezes para checar propriedades que devem ser verdadeiras independentemente da intercalação seguida pelas threads em cada escalonamento e uma vez que estas cheguem a um determinado estado global do sistema. O trabalho desta tese visa fazer com que caso aconteçam falhas do teste neste processo de re-execução, estas não sejam devidas ao teste, mas sim a problemas com a aplicação.

Para depurar tais problemas detectados em apenas algumas execuções, uma outra técnica também relacionada ao controle da execução é a técnica de controle de re-execução (*replay control*) [13], a qual demanda que as threads sejam controladas. Esta técnica, que pode ser usada em conjunto com o trabalho desta tese no auxílio à depuração do sistema, se caracteriza por gravar a execução do sistema e as decisões tomadas relativas ao não-determinismo deste, de forma que este possa ser re-executado com as mesmas decisões. Na Seção 3.6.3 se discutirá mais sobre essa técnica complementar a este trabalho.

Outros trabalhos ainda nessa direção são os que fazem referência aos problemas relacionados à habilidade de controle (*controllability*) e observação (*observability*) em testes com múltiplos elementos. Alguns desses trabalhos [98] [39] tratam dessa questão utilizando-se do formalismo de máquinas de estado finitas (*FSM - Finite State Machines*, uma abordagem que não é seguida nesta tese. Um desses trabalhos é o de Ural e Whittier [98]. Este artigo detalha o problema relacionado à habilidade de controle e observação afirmando que ele ocorre quando não se consegue determinar nem quando aplicar uma entrada particular ao sistema sendo testado (System Under Test - SUT) e nem se uma saída particular do SUT foi gerada em resposta a uma entrada específica. A solução apresentada baseia-se na especificação de sequências de teste a serem seguidas. Outro trabalho também baseado em FSMs é o de Gaudin e Marchand [39]. Uma das similaridades entre esse trabalho e esta tese é o fato de tratar do problema denominado SACP (State Avoidance Control Problem), ou seja, o problema de evitar que determinados estados globais ilegais sejam alcançáveis (*reachable*). O trabalho de Gaudin e Marchand ataca tais problemas através de supervisores que evitam que tais estados sejam alcançados, estados estes definidos em termos de FSMs. Neste trabalho de tese utilizam-se monitores e mecanismos de sincronização para evitar que o teste entre em estado de asserção em um momento antecipado ou tardio.

Um outro trabalho relacionado que trata do problema da habilidade de controle, mas no

nível de interações entre threads e não de testes distribuídos é o de Long e Hoffman [62]. Tal problema é atacado através de um método e do suporte de uma ferramenta, denominada ConAn (Concurrency Analyser). ConAn é uma ferramenta para gerar controladores (*drivers*) para o teste de unidade de classes Java. Para obter habilidade de controle nas interações entre as threads, o controlador gerado contém threads que são sincronizadas por um relógio. Além disso, esse controlador automaticamente executa as chamadas na sequência de testes de acordo com a ordem pré-determinada e compara as saídas com as saídas previstas para a sequência dada. Esse trabalho também se diferencia deste trabalho de tese por focar em especificações de ordenamentos específicos das threads da aplicação (sem mencionar a thread de teste), mas apresenta similaridades por basear-se em monitoração para garantir tais ordenamentos.

Também está relacionada a este trabalho de tese a ferramenta *Thread Weaver* [7], que foca no controle das threads durante os testes. O *Thread Weaver* é um arcabouço para se escrever testes unitários multi-threaded em Java. Ele provê mecanismos para criar pontos de parada no código (*breakpoints*), e para parar a execução de uma thread quando tais pontos são atingidos. O propósito desse arcabouço é permitir que testadores escrevam testes repetíveis que possam checar por condições de corrida e segurança de threads. Essa ferramenta permite a criação de múltiplas threads dentro de um teste de unidade e permite também que se controle como essas threads são paradas e reiniciadas. Dessa forma, ele permite a replicação de condições de corrida potenciais e a verificação de que um código está sendo executado corretamente. Esse arcabouço também foi publicado como ferramenta de código aberto no Google Code (`http://code.google.com/p/thread-weaver`) e também se baseia em manter controle sobre as threads durante os testes. No entanto, o propósito de controle de threads do arcabouço *ThreadControl*, discutido nesta tese, não é levar a um determinado escalonamento especial das threads do sistema (algo bem relevante quando se busca testes repetíveis), mas sim garantir que mudanças nas threads do sistema não irão acontecer enquanto asserções estão sendo executadas e também que as asserções só serão feitas quando o sistema estiver pronto para ser checado (e.g. algumas de suas threads estão esperando ou terminaram). Na abordagem apresentada neste trabalho de doutorado, a idéia é permitir que durante as re-execuções de testes as threads possam explorar diferentes escalonamentos, mas o arcabouço *ThreadControl* se importa em garantir que as asserções serão feitas quando

se chega em um certo ponto em que o sistema já pode ser verificado e que durante essa verificação ele não será perturbado.

3.4 Monitoração de Threads

Considerando a monitoração de *threads*, viu-se que para prover o controle da execução de testes, alguns trabalhos já se utilizam desse recurso. Nessa seção são apresentados outros trabalhos com foco em monitoração em si.

Um desses trabalhos é o de Moon e Chang [70]. Ele apresenta um sistema de monitoração de *threads* para programas Java *multi-threaded* que é capaz de capturar os pontos pelos quais as threads passam (*tracing*) e monitorar as threads em execução e ações de sincronização. O sistema foi implementado utilizando a técnica do monitor de referência incorporado (*Inlined Reference Monitor* - IRM) [32], através da qual se modifica a aplicação para incluir a funcionalidade de um monitor de referências. IRMs são inseridos na aplicação através de um re-escritor ou transformador que lê a aplicação destino e uma política e produz uma aplicação modificada com monitores. Uma das diferenças principais com relação ao trabalho aqui apresentado é que nesse trabalho não há um foco em testes, mas basicamente em depuração e perfilamento. Embora em capítulos posteriores desta tese se tenha explorado o uso de Programação Orientada a Aspectos para monitorar as threads da aplicação, a técnica denominada IRM é uma outra possibilidade. Segundo ressaltado por Moon e Chang, estudos mostram que tal abordagem têm-se mostrado eficiente para a monitoração de programas Java [32].

Um outro trabalho que trata de monitoração de aplicações e também de suas threads, mas focado no uso de Programação Orientada a Aspectos é o de Coelho et al. [19], que descreve o *Application Monitor Aspect Pattern*. Esse padrão suporta a definição separada de interesses relativos à monitoração através do uso de Programação Orientada a Aspectos, com o intuito de aumentar reuso e manutenabilidade do sistema. Neste trabalho de tese também se explora esse paradigma na implementação do arcabouço que dá suporte à abordagem *Thread Control for Tests*. Este arcabouço surgiu a partir da evolução de um arcabouço anterior e menos geral, denominado *ThreadServices*. O uso desse arcabouço anterior [25] é citado no trabalho de Coelho et al. [19] como um uso conhecido do padrão *Application Monitor Aspect Pattern*.

3.5 Problemas com Asserções na Área de Projeto por Contrato

A área de Projeto por Contrato (*Design by Contract* - DBC) também apresenta relação com este trabalho por também poder estar presente lá o problema das asserções antecipadas ou tardias. No entanto, as asserções de que se fala nesta área não são feitas em testes, mas na especificação de contratos de componentes de software, como será melhor detalhado a seguir.

Projeto por Contrato [69] corresponde à forma de visualizar o relacionamento entre uma classe e seus clientes como um acordo formal, em que são expressos os direitos e obrigações de cada parte. Para isso é necessária a definição precisa das responsabilidades e dos direitos de cada módulo de um sistema.

Tais direitos ou obrigações são observados utilizando-se asserções, de acordo também com a definição mostrada na Seção 2.1. Asserções são usadas em DBC para descrever a especificação do software de modo que sua corretude possa ser avaliada, onde corretude é definida como a habilidade do software de se comportar de acordo com sua especificação [69]. Uma fórmula de corretude parcial (também chamada de tripla de Hoare) é expressa da seguinte forma: $\{P\}A\{Q\}$. Tal expressão significa que qualquer execução de uma operação A, iniciando em um estado onde P é válida, irá terminar em um estado em que Q é válida, onde P e Q são propriedades das várias entidades envolvidas e são também chamadas asserções, sendo P chamada de **pré-condição** e Q de **pós-condição**. No contexto de DBC, pode-se definir uma asserção como sendo uma expressão envolvendo algumas entidades do software, e indicando alguma propriedade que essas entidades podem satisfazer em certos estágios da execução do software.

As asserções de pré-condições indicam as propriedades do sistema que devem ser válidas sempre que uma rotina é chamada; as pós-condições indicam as propriedades que a rotina deve garantir uma vez que ela retorna. Existem ainda as **invariantes** de classe, que são asserções que expressam restrições de consistência mais gerais e que se aplicam a cada instância de uma classe como um todo, e se diferenciam das pré-condições e pós-condições por estas outras caracterizarem rotinas individuais [67; 69]. Considerando este conceito, pode-se expressar o requisito de corretude de uma rotina da seguinte forma:

$\{INV\ and\ pre\} body \{INV\ and\ post\}$.

Asserções, para Projeto por Contrato, tanto podem ser tratadas puramente como comentários, como também é possível usá-las para checar que a execução está acontecendo conforme o planejado. Para este último caso, durante a execução, o ambiente irá monitorar automaticamente se as asserções estão sendo todas válidas e caso alguma não esteja, será disparada uma exceção, normalmente terminando a execução e informando claramente com uma mensagem o que aconteceu.

Um exemplo de ferramenta para Projeto por Contrato é JML (*Java Modeling Language*) [61]. A linguagem JML é uma linguagem que permite a especificação de comportamentos estáticos e comportamentais de interfaces em um código Java. JML suporta o paradigma de Projeto por Contrato, incluindo notações para pre- e pós-condições e invariantes. Um dos trabalhos que utilizam JML e que apresentam similaridades com o problema atacado por esta tese é o trabalho de Rodriguez et al. [82]. Ele apresenta uma extensão da linguagem JML para permitir a especificação de programas multi-threaded em Java já que JML só tratava anteriormente com programas sequenciais. As construções adicionadas a JML com a extensão proposta por Rodriguez et al. se baseiam na noção de não-interferência da atomicidade de métodos e permitem que desenvolvedores possam especificar trancas (*locks*) e outras propriedades de não interferência de métodos. A idéia é dar suporte à checagem de características de código multi-threaded, como atomicidade e a posse de trancas.

Considerando isto, é importante destacar que o trabalho de tese aqui apresentado não tem como objetivo a checagem de tais características via testes. Seu foco na verdade é no fato de propor um mecanismo de sincronização para evitar a mudança de dados utilizados nos testes enquanto verificações estão sendo feitas para evitar assim asserções tardias. Neste sentido, o trabalho de Rodriguez et al. apresenta semelhanças com o que está sendo apresentado aqui, em que se procura usar sincronização para evitar as asserções em momentos inadequados. O trabalho de Rodriguez, porém, se utilizado em combinação com o desta tese, seria utilizado para checar se tal sincronização está sendo provida corretamente. Uma outra semelhança desse trabalho com relação ao desta tese são as construções que este trabalho usa em suas especificações e que lembram as primitivas de teste apresentadas neste trabalho e discutidas na Seção 5.3.

Um outro artigo relacionado e que trata de DBC é o de Nienaltowski e Meyer [73].

Neste artigo eles apresentam uma extensão da linguagem Eiffel, o SCOOP, para prover suporte a programação concorrente. Esse trabalho discute o problema do "paradoxo da pré-condição concorrente" ("concurrent precondition paradox"), que se relaciona ao problema tratado neste trabalho de tese, mas aplicado no contexto de DBC. Este problema em DBC se refere ao fato de ao se estar checando uma pré-condição, a propriedade sendo verificada numa classe poder estar sendo alterada por um elemento externo. Para resolver este problema, Meyer [68] propõe uma nova semântica para cláusulas de pré-condições, tornando-as condições de espera (*wait-conditions*). Isso faz com que as entidades que invocam as rotinas esperem até que as condições de espera sejam satisfeitas. No modelo SCOOP, condições de espera aparecem como parte de uma pré-condição. Na versão estendida do SCOOP, apresentada por Nienaltowski et al. [73], aplica-se a semântica de espera a todas as asserções, incluindo invariantes de classes e utilizam-se condições de espera como condições de corretude. A idéia apresentada nesta tese de que sejam oferecidas primitivas para testes que os permitam esperar antes de fazer asserções se assemelha a esse trabalho. No entanto, como será posteriormente descrito nesta tese, tais primitivas, além de poderem ser usadas para especificar estados de espera desejados, funcionam também como mecanismos para fazer o teste esperar.

3.6 Identificação de Defeitos de Concorrência

Quando se discute testes em sistemas *multi-threaded*, pode-se citar vários trabalhos que tratam da identificação de defeitos relacionados a concorrência, como condições de corrida (*race conditions*) indesejadas ou impasses (*deadlocks*). Embora esta tese não foque na identificação de defeitos de concorrência, mas sim no suporte ao desenvolvimento de testes multi-threaded, tais trabalhos podem ser utilizados em combinação com a abordagem aqui apresentada e alguns deles podem até ajudar a evitar que com o uso desta aconteçam problemas como o efeito da monitoração. Além do mais, tem-se evidenciado cada vez mais que uma boa solução para auxiliar na construção de software multi-threaded de qualidade deve conter componentes de vários domínios de pesquisa [33] [10].

Em geral, as áreas de pesquisa na direção da identificação de defeitos de concorrência incluem: detecção de condições de corrida [72][85], análise estática [48], técnicas de re-

execução (*replay*) [83] [15], técnicas para forçar determinada ordem de escalonamento das *threads* (*thread interleavings*) [64], e técnicas para incentivar diferentes intercalações das threads, como geradores de ruído (*noise makers*), de forma a revelar falhas de concorrência [90][20].

Embora tais áreas não sejam mutuamente exclusivas, como forma de facilitar a leitura dessa seção, essas áreas foram utilizadas para a divisão das sub-seções apresentadas a seguir, em que alguns trabalhos de cada uma são discutidos.

3.6.1 Detecção de Condições de Corrida

Diz-se que há uma condição de corrida em um programa se vários processos estão acessando e manipulando os mesmos dados concorrentemente e o resultado da execução depende da ordem específica em que ocorre o acesso [87] (p. 148). Ferramentas para detecção de condições de corrida, sejam elas estáticas ou dinâmicas, são úteis para aumentar a confiabilidade de programas multi-threaded e há vários trabalhos que tratam desse tema. Um deles é o de Naik et al. [72], em que é apresentado o desafio para se reproduzir e corrigir o problema de condições de corrida em aplicações. Além de mostrar um interessante levantamento de outros trabalhos da área de detecção de condições de corrida, nesse artigo é proposta uma nova técnica para detecção de condições de corrida e que tenta satisfazer alguns critérios que aumentem a qualidade desse processo, dentre os quais estão: precisão (evitar falso-positivos) e escalabilidade (para que também trate de programas grandes). O método proposto nesse trabalho foi implementado em uma ferramenta chamada *Chord* [1]. Uma outra ferramenta com o mesmo propósito é a *Eraser* [85], uma ferramenta para detectar dinamicamente condições de corrida em programas multi-threaded baseados em trancas (*locks*). Eraser monitora cada referência à memória compartilhada e verifica se um comportamento de uso consistente de trancas (*locking*) é observado.

Embora esta tese foque no problema de evitar condições de corrida entre a thread do teste e as threads da aplicação através de testes em que se garanta a ausência de asserções antecipadas e tardias, ela se diferencia dos trabalhos acima mencionados por não ter o foco na detecção do problema de corrida em si, mas sim nas técnicas de desenvolvimento de testes para evitar que tais condições aconteçam.

3.6.2 Análise Estática e Verificação de Modelos em Programas Concorrentes

Análise estática é uma abordagem de verificação complementar aos testes formais e revisões de código. Ela corresponde ao processo de analisar código sem executá-lo, e ferramentas de auditoria de código podem analisar o código à procura de padrões de defeitos (*bug patterns*) [44]. Ferramentas de análise estática produzem uma lista de advertências que devem ser examinadas manualmente para determinar se eles representam erros de fato.

Um exemplo de ferramenta de análise estática é o FindBugs [4] [48], uma ferramenta para procurar defeitos em código Java. Dentre os detectores de padrões de defeitos do FindBugs, existem alguns diretamente relacionados a defeitos de concorrência. Dentre estes defeitos estão [44]: sincronização inconsistente, trancas não liberadas, trancamento duplamente checado (*double-checked locking*), threads esperando (ex: em invocações a `Thread.sleep`) enquanto estas mantém a posse de uma tranca.

Além de trabalhos que buscam procurar por padrões de defeitos diretamente no código, há outros trabalhos que se utilizam de análise estática tratando também de problemas relacionados a concorrência na área de Verificação de Modelos (*Model Checking*) [18] [17].

Verificação de modelos corresponde a uma família de técnicas, baseadas na exploração exaustiva do espaço de estados para verificar propriedades em sistemas concorrentes [33]. Uma das propriedades a se verificar é a ausência de impasses em possíveis caminhos de execução de um programa [21].

Algumas ferramentas bem conhecidas na área de Verificação de Modelos são o Verisoft [41], o Bandera [21] e o Java Path Finder (JPF) [99] [47].

O VeriSoft [41] é uma ferramenta para sistematicamente explorar os espaços de estado de sistemas compostos por vários processos concorrentes via execução de código. No trabalho descrito por Godefroid [41] discute-se a extensão da noção inicial de model checking para que sejam tratadas descrições de fato de sistemas concorrentes, como implementações de protocolos escritos em linguagens de programação como C ou C++. O Verisoft é conhecido como uma ferramenta para teste sistemático de software e ele automaticamente procura por problemas de coordenação (deadlocks, etc.) e violações de asserção em um sistema de software através da geração, controle e observação das execuções e interações dos seus com-

ponentes. Quando um deadlock ou violação de asserção é detectado, a busca no espaço de estados é parada e um cenário com todas as transições armazenadas na pilha é exibida para o usuário [42]. Um depurador/simulador gráfico interativo também é disponibilizado para que se re-executem (*replay*) os cenários e seguindo suas execuções no nível de instruções ou de procedimento/função. Valores de variáveis de cada processo podem ser examinados interativamente. Técnicas de verificação de modelos como estas sugeridas no Verisoft podem se beneficiar da abordagem apresentada neste trabalho de tese já que ela visa auxiliar no desenvolvimento de testes, uma tarefa que requer bastante esforço, e evitar que estes levantem falsos positivos. Segundo Godefroid [42], a abordagem para *model checking* utilizada pelo VeriSoft para testar um produto de software requer automação de testes (a habilidade de executar e avaliar testes de maneira automática), e como afirmado neste artigo, para os testadores, o desenvolvimento de uma infra-estrutura de testes que provê automação pode requerer um esforço significativo. Uma vez que se tem a automação de testes disponível, utilizar ferramentas como o VeriSoft para significativamente aumentar a cobertura de testes não parece demandar muito esforço.

O JPF é um ambiente de verificação, análise e testes para Java. Segundo Visser et al. [99], o JPF combina técnicas de verificação de modelos com técnicas para lidar com espaços de estado grandes ou infinitos, incluindo análise estática para dar suporte à redução parcial do conjunto de transições a serem exploradas pelo checador de modelos, abstração de predicados, para abstrair o espaço de estados, e também técnicas de análise dinâmica (*runtime analysis*), como detecção de condições de corrida.

O Bandera, por sua vez, é uma coleção de componentes de análise e transformação de programas, que permite a extração automática de modelos de estados finitos compactos a partir do código fonte Java. Podem ser gerados modelos em linguagens que servem de entrada para várias ferramentas de verificação [21].

3.6.3 Técnicas de Re-execução

Usar re-execução (*replay*) em testes consiste em duas fases: gravar (*record*) e mandar executar (*playback*). Na primeira fase são gravadas informações a respeito da velocidade (*timing*) e das decisões não-determinísticas feitas no programa. Na segunda fase, a de mandar executar, o teste é executado e o mecanismo de re-execução garante que as mesmas decisões

sejam tomadas [33] [83]. O uso de tal técnica permite que ao se re-executar um mesmo teste que falhou anteriormente, se possa de fato repetir as decisões não determinísticas tomadas na execução que falhou. Dessa forma, mesmo programas paralelos e não determinísticos poderiam passar por depuração cíclica (*cyclic debugging*), que é um processo de depuração em que se assume que uma execução de programa pode ser fielmente re-executada várias vezes [83].

Uma das ferramentas para re-execução é o DejaVu [15], que é uma ferramenta de *record/replay* para Java, desenvolvida como uma extensão da máquina virtual Java (JVM) e que visa re-execução determinística de programas Java concorrentes. Um outro exemplo de ferramenta é o CHESS [71], que além de reproduzir execuções em que os denominados "*Heisenbugs*" [45][1] se manifestam, também é capaz de encontrar tais bugs. O CHESS assume o escalonamento das threads durante a execução e usa técnicas eficientes para explorar o espaço das possíveis intercalações de threads. O CHESS introduz, através de instrumentação binária, uma camada fina que é uma espécie de casca (*wrapper*) entre o programa sob teste e a API de concorrência.

Como técnicas de re-execução dependem de testes e neste trabalho de tese são propostas maneiras de melhorar a qualidade dos testes desenvolvidos evitando que levem a falsos positivos, acredita-se que ele é complementar aos trabalhos que tratam da técnica de *record/-replay*.

3.6.4 Técnicas para forçar determinados escalonamentos

Devido ao comportamento não-determinístico dos programas concorrentes, os resultados de re-execuções de uma aplicação e de testes de regressão acabam sendo incertos. Considerando isso, existem alguns trabalhos focados na execução determinística para depuração e testes visando resolver este problema. Um destes trabalhos é o de Carver e Tai [14], que sugere a alteração do programa para que sua execução seja determinística. No caso de testes, esse trabalho sugere uma abordagem em que se fornece ao teste a ser executado, além de sua entrada, a sequência de sincronização que deve ser seguida.

[1] *Heisenbugs* são tipos de defeitos que simplesmente desaparecem quando se olha pra eles e que podem escapar aos responsáveis por buscar defeitos por anos. De fato, tais pessoas podem perturbar o sistema de forma a fazer esse defeito desaparecer, o que é análogo ao princípio da incerteza de Heisenberg da Física [45].

Um outro trabalho na área é o de MacDonald et al. [64], com o objetivo de prover execução determinística de programas concorrentes. Nesse trabalho são combinadas duas tecnologias: CSSAME, um compilador intermediário que identifica que pontos de concorrência foram alcançados para variáveis compartilhadas; e Programação Orientada a Aspectos [54], que permite a interceptação do acesso a campos e chamadas de métodos e também a injeção de código nesses pontos. Uma das vantagens dessa abordagem é que ela permite que se execute deterministicamente o programa cobrindo todas as ordens potenciais para uma dada condição de corrida.

O trabalho apresentado por Pugh e Ayewah [80] também trata de execuções determinísticas e apresenta algumas similaridades com o trabalho de doutorado apresentado neste documento. Ele apresenta o arcabouço MultithreadedTC, que permite a construção de testes de unidade determinísticos e repetíveis para se testar abstrações relativas a concorrência. Para isso, ele se utiliza da idéia de um relógio que avança quando todas as *threads* estão bloqueadas.

O trabalho de Pugh e Ayewah se diferencia desta tese por demandar um controle explícito das instâncias das threads durante o teste. Um outro fator que diferencia o trabalho de Pugh [80] desta tese é o seu foco em se exercitar escalonamentos específicos da aplicação, o que não é o foco da abordagem *Thread Control For Tests* que será apresentada neste documento. A idéia da abordagem *Thread Control For Tests* é que as *threads* possam executar em diferentes intercalações e o que se deve garantir é que o teste espere por um determinado estado de equilíbrio que é especificado no teste e no qual as asserções podem ser executadas. Dessa forma, espera-se que falhas só aconteçam caso exista de fato uma falta na aplicação.

3.6.5 Geradores de Ruído

O gerador de ruído (*noise maker*) é uma ferramenta que semeia um programa concorrente com primitivas de sincronização condicional (como o `yield()`) com o propósito de aumentar a probabilidade de que um defeito se manifeste [10]. A idéia básica é gerar diferentes intercalações de threads (*interleavings*) para revelar faltas relacionadas a concorrência em uma aplicação. A relação dessa área com esta pesquisa de tese está no fato de *noise makers* serem uma possível técnica para amenizar o efeito de monitoração, que pode levar uma aplicação sendo monitorada a não expor facilmente alguns de seus defeitos enquanto é testada.

Há várias técnicas para definir em que pontos induzir alternâncias entre threads (*switches*). Essas técnicas podem ser caixa branca ou caixa preta. Um exemplo de ferramenta geradora de ruído que usa uma técnica caixa branca é o rstest (*random scheduling test*) [90]. Ela implementa a abordagem de inserir chamadas a funções de escalonamento em pontos selecionados de um programa sob teste. A função de escalonamento pode não fazer nada ou causar uma mudança de contexto (*context switch*). Pode-se ter uma função de escalonamento simples, baseada em um gerador de números pseudo-aleatórios, ou uma mais sofisticada, combinando aleatoriedade com heurísticas que adicionam pesos às escolhas. O programa que sofreu a transformação é então repetidamente executado para fins de teste. Se um erro é encontrado, o programa é re-executado com a mesma semente usada para as escolhas aleatórias. Porém, não se garante a reprodutibilidade. Para isso seria necessário um mecanismo de gravar e mandar executar. No trabalho de Stoller [90] que descreve o rstest, foca-se no uso de análise estática para detectar locais em que as trocas das threads ajudariam a revelar padrões de defeitos, sendo por isso classificado como uma técnica caixa branca.

Um outro trabalho também seguindo a abordagem caixa branca é o de Edelstein et al. [30], em que se usa informação a respeito de cobertura para tomar as decisões relativas à geração de ruído.

Um exemplo de trabalho que explora a técnica caixa preta é o de Ben-Asher et al. [10]. Baseando-se nessa técnica, esse trabalho desenvolve uma teoria para localizações boas, más e neutras para inserção de ruído gerado, sem analisar o papel semântico real dessas localizações. Estudos feitos nesse trabalho mostraram que ao comparar os resultados de diferentes heurísticas para geração de ruído, a heurística caixa preta "aleatória"aumentava o número de defeitos encontrados em comparação com ruído baseado em caixa branca. O trabalho sugere que heurísticas baseadas em caixa-preta podem ser aplicadas quando informação a respeito do programa não está disponível ou é incorreta.

Um exemplo de ferramenta que dentre outras funcionalidades oferece geração de ruído é o ConTest [2][30][74][20]. O ConTest é uma ferramenta que instrumenta o bytecode da aplicação com instruções condicionais de `sleep` e `yield` heuristicamente controladas e que forçam diferentes execuções dos testes (considerando a ordem de execução das threads do sistema). O ConTest é usado para detectar faltas de sincronização em programas Java multi-threaded. Em tempo de execução, o ConTest toma decisões aleatórias ou baseadas

em cobertura para saber se as primitivas inseridas para incentivar diferentes escalonamentos devem ser executadas. Além disso, o ConTest também oferece um algoritmo de re-execução (*replay*) para auxiliar na depuração, salvando a ordem dos acessos à memória compartilhada e os eventos de sincronização.

Trabalhos baseados em escalonamento aleatório [86][90][30] durante a execução de testes estão relacionados a esta tese pelo fato de poderem se beneficiar do trabalho aqui apresentado. Isso ocorre pois tais técnicas dependem de testes confiáveis. Além disso, são técnicas adicionais à que se propõe neste documento e podem ajudar a amenizar o efeito de monitoração ao utilizar a abordagem aqui apresentada.

3.7 Conclusões parciais

Analisando os trabalhos relacionados, vê-se que alguns focam no mesmo problema desta tese, que é o das asserções antecipadas e tardias. No caso do trabalho de Meszaros [66], a solução proposta para este problema demanda alterações de código na aplicação e faz com que os testes que usem a solução proposta não involvam mais assincronia, algo que esse trabalho ainda busca atender, pois há testes de integração e funcionais em que verificações envolvendo operações assíncronas são necessárias. No caso dos trabalhos na área de DBC [82] [73], embora o problema seja bem semelhante, o contexto de asserções ali mencionado não é o de testes.

Outros trabalhos analisados acima são complementares ao desta tese, pois buscam auxiliar no teste de sistemas concorrentes, ajudando na detecção de problemas de concorrência. Porém, estes não focam em técnicas para ajudar no desenvolvimento de testes em si no intuito de evitar o problema atacado, mas principalmente nas técnicas para localizar faltas e que algumas vezes demandam testes confiáveis, podendo nesse ponto ser beneficiados por este trabalho de tese.

Há ainda os trabalhos que apresentam soluções semelhantes à deste trabalho, baseadas em técnicas como monitoração e controle das threads, mas que são utilizadas com objetivos ortogonais ou complementares ao problema tratado nesta tese.

Capítulo 4

Modelagem do Problema e da Solução Proposta

Neste capítulo será apresentado um modelo formal do problema de testes do qual trata este trabalho e da solução sendo proposta. A idéia principal do problema que se quer tratar é que na prática nem sempre são introduzidos mecanismos explícitos de sincronização entre a thread do teste e a thread da aplicação pois algumas vezes os testadores tentam evitar que a implementação do teste seja muito intrusiva com relação à aplicação sendo testada, ou às vezes não podem mesmo mudar o código da aplicação. A solução a ser modelada a seguir propõe o uso de monitores com o mínimo de intrusividade e que evitem condições de corrida entre os eventos da aplicação e os eventos de asserção do testes que checarão os resultados produzidos pela aplicação.

A idéia geral da modelagem aqui apresentada é utilizar lógica e teoria dos conjuntos para descrever execuções de testes e os eventos envolvidos neste processo e teoremas a respeito de execuções de testes corretas (que evitam o problema das asserções antecipadas e tardias). Através dessa modelagem busca-se demonstrar que sem monitoração e um certo nível de controle das threads da aplicação durante os testes não se pode garantir que as execuções de teste terão sucesso sob o ponto de vista de não apresentarem falsos positivos por asserções feitas em momentos inapropriados. Demonstra-se também que requisitos precisa ter uma solução que resolva este problema. As provas utilizadas são feitas com base em teoremas que são apresentados junto com suas demonstrações.

4.1 Modelagem do Problema

Seja $\mathcal{S}$ um teste em um sistema com múltiplos fluxos de execução (*multi-threaded*). Seja τ_0 o fluxo de execução (*thread*) principal do teste (sua **thread de controle**). Seja $B = \{\tau_1, \tau_2, ..., \tau_{|B|}\}$ o conjunto das threads iniciadas por τ_0 quando o sistema é exercitado por um teste, representando os fluxos de execução do sistema em si. Considerando isso, pode-se caracterizar tal teste do sistema em termos de suas threads pelo par:

$$\mathcal{S} = (\tau_0, B) \tag{4.1}$$

Um fluxo de execução opera sobre um conjunto de variáveis na memória e executa várias instruções. Considerando isso, pode-se dizer que em cada fluxo de execução acontecem **eventos** e, que correspondem a um conjunto de instruções atômicas que manipulam a memória. Pode-se, observando o conteúdo da memória (inclusive os registradores), através de uma função que será denominada $state$, identificar o estado $s \in Q$ em que cada thread $\tau \in \{\tau_0\} \cup B$ está[1] (como parada, executando, finalizada, etc). Após cada evento uma thread pode passar de um estado a outro ou permanecer no mesmo estado.

A execução de cada thread τ caracteriza-se pelo estado inicial da thread e por uma sequência de eventos que ocorrem no contexto da thread e os respectivos estados intermediários alcançados após cada um desses eventos. Seja $s_0^\tau \in Q$ o estado inicial da thread τ e Q o conjunto dos estados possíveis para todas as threads de $\{\tau_0\} \cup B$. Seja E^τ uma sequência de pares contendo os eventos que ocorreram durante a execução de τ e os respectivos estados de τ após cada evento. Pode-se definir a execução R^τ de cada thread τ pelo seguinte par:

$$R^\tau = (s_0^\tau, E^\tau) \tag{4.2}$$

Nesse par, a sequência E^τ, considerando uma execução infinita de eventos, é representada por:

$$E^\tau = ((e_1^\tau, s_1^\tau), (e_2^\tau, s_2^\tau), ...) \tag{4.3}$$

[1] Um exemplo de uma possível implementação da função $state$ é uma função que retorna o estado da thread de acordo com o conteúdo do contador de programa (*program counter* - PC).

O evento e_n^τ é o n-ésimo evento que acontece na execução de τ e cada estado $s_n^\tau \in Q$ é o estado da thread após o evento e_n^τ ter ocorrido. Por exemplo, o evento e_n^τ poderia ser uma chamada a um método como `wait` e o estado s_n^τ da thread τ após após tal chamada poderia ser "esperando".

Em particular, os eventos da thread de controle τ_0 do teste são basicamente de dois tipos: eventos de iniciação (ativação) do sistema (denotados por i, e durante os quais o sistema é exercitado, como uma chamada ao método `mySystem.initialize()`) e eventos de asserção (denotados por a, e durante os quais os resultados das operações exercitadas são verificadas, como, por exemplo, uma chamada a `assertTrue(isCorrectInitialized)`). Tais eventos ficam se alternando nas execuções de testes podendo em um teste haver várias subfases com sequências de iniciações seguidas por asserções. Sendo assim, considerando cada subfase do teste contendo j eventos de iniciação e k eventos de asserção, pode-se definir a seguinte subsequência de duplas de eventos-estados:

$$E_{sub}^{\tau_0} = ((i_1, s_1^{\tau_0}), (i_2, s_2^{\tau_0}), ..., (i_j, s_j^{\tau_0}), (a_1, s_{j+1}^{\tau_0}), ..., (a_k, s_{j+k}^{\tau_0})) \tag{4.4}$$

Tal subsequência apresenta os eventos de τ_0 e e os respectivos estados s^{τ_0} de τ_0 após cada evento para cada subfase de iniciação-asserção.

Considerando a definição acima, a sequência total de eventos-estados E^{τ_0} da thread de controle τ_0 de uma execução de teste será representada por uma sequência de eventos organizados em uma ou mais subfases de teste contendo eventos de iniciação e de asserção, conforme descrito por $E_{sub}^{\tau_0}$.

Tomando como base a sequência E^{τ_0}, pode-se representar a execução R^{τ_0} da thread de controle de um teste pela seguinte dupla:

$$R^{\tau_0} = (s_0^{\tau_0}, E^{\tau_0}) \tag{4.5}$$

A execução $\mathcal{R}^{\mathcal{S}}$ de um teste $\mathcal{S}$ corresponde às execuções da thread τ_0 (R^{τ_0}) e das threads $\tau \in B$ (R^τ) e ao escalonamento dos seus eventos no tempo.

Considerando a existência de um relógio discreto global para o sistema, pode-se dizer que, em cada execução, cada evento é escalonado para ocorrer em um tempo t que pode assumir qualquer valor no conjunto dos números naturais ($\mathbb{N}$).

Tomando-se a função $lastEvent$ que indica o último evento de uma thread τ de sua sequência de eventos E^{τ} e seja $t^{\tau_0}_{lastEvent(\tau_0)}$ o instante no tempo em que é escalonado o último evento da thread de controle ($lastEvent(\tau_0)$). Deseja-se definir uma matriz M que sintetize o escalonamento dos eventos da execução de um teste até o momento em que o último evento da thread de controle ocorre. Cada matriz M tem as dimensões $|B \cup \{\tau_0\}|$ por $t^{\tau_0}_{lastEvent(\tau_0)}$.

Cada linha da matriz representa uma das threads do conjunto $B \cup \{\tau_0\}$, sendo a primeira linha correspondente à thread τ_0. Cada coluna da matriz representa um instante de tempo. Cada elemento $m_{u,t}$ da matriz representa um evento da thread $\tau_u \in B \cup \{\tau_0\}$ escalonado para o instante de tempo t ou um evento de *nop*, que representa um conjunto vazio de instruções e indica que aquela thread específica não foi escalonada para executar naquele instante. Um exemplo de matriz M de escalonamento de execução de teste é ilustrado pela tabela a seguir:

Tabela 4.1: Exemplo de matriz de escalonamento M que mostra a distribuição dos eventos no tempo

M: thread $\Downarrow$ | t $\Rightarrow$	0	1	2	3	4	5	6	7
τ_0	i_1	i_2	*nop*	*nop*	*nop*	*nop*	a_1	a_2
τ_1	*nop*	*nop*	$e_1^{\tau_1}$	$e_2^{\tau_1}$	*nop*	*nop*	*nop*	*nop*
τ_2	*nop*	*nop*	*nop*	$e_1^{\tau_2}$	$e_2^{\tau_2}$	*nop*	*nop*	*nop*
τ_3	*nop*	*nop*	$e_1^{\tau_3}$	*nop*	$e_2^{\tau_3}$	*nop*	*nop*	*nop*
τ_4	*nop*	*nop*	*nop*	*nop*	*nop*	$e_1^{\tau_4}$	*nop*	*nop*

Os eventos que ocorrem nas várias threads obedecem a certas regras de causalidade. Tais regras definem as dependências entre os eventos das threads $\tau \in B \cup \{\tau_0\}$. Elas se inspiram na definição de Lamport [56] da relação *happened before*, simbolizada por $\rightarrow$. Porém, na definição aqui utilizada, representada por $\rightarrowtail$, ao invés de eventos de um processo, são considerados eventos de *threads*, e considera-se que a comunicação ocorre através de memória compartilhada, o que inclui os mecanismos de sincronização de threads existentes no computador que roda o teste.

Definição 1 (D1). A relação $\rightarrowtail$ no conjunto de eventos de um sistema é a menor relação que satisfaz as seguintes condições: (1) Se e_x^{τ} e e_y^{τ} são eventos em uma única thread $\tau \in B \cup \{\tau_0\}$ e na sequência E^{τ} a tupla (e_x^{τ}, s_x^{τ}) vem antes de (e_y^{τ}, s_y^{τ}), então $e_x^{\tau} \rightarrowtail e_y^{\tau}$. (2) Se e_x

e e_y são eventos quaisquer do sistema e o primeiro é a escrita de um dado na memória e o segundo é a leitura deste mesmo dado por uma outra thread[2], então $e_x \rightarrowtail e_y$. (3) Se $e_x \rightarrowtail e_y$ e $e_y \rightarrowtail e_z$, então $e_x \rightarrowtail e_z$.

Resumindo, seja $\mathcal{C}$ um conjunto de regras de causalidade definidas utilizando a relação $\rightarrowtail$, pode-se definir uma execução específica $\mathcal{R}^{\mathcal{S}}$ de um teste $\mathcal{S}$ de um sistema pela seguinte tupla:

$$\mathcal{R}^{\mathcal{S}} = (R^{\tau_0}, R^{\tau_1}, R^{\tau_2}, ..., R^{\tau_{|B|}}, \mathcal{C}, M) \tag{4.6}$$

Definição 2 (D2). Uma execução de teste $\mathcal{R}^{\mathcal{S}}$ é dita *possível* ($possivel(\mathcal{R}^{\mathcal{S}}) = true$) quando a sua matriz de escalonamento M respeita as regras de causalidade $\mathcal{C}$.

Definição 3 (D3). Dada uma execução *possível* $\mathcal{R}^{\mathcal{S}} = (R^{\tau_0}, R^{\tau_1}, R^{\tau_2}, ..., R^{\tau_{|B|}}, \mathcal{C}, M)$, o conjunto das execuções equivalentes a $\mathcal{R}^{\mathcal{S}}$, representado por $Equiv(\mathcal{R}^{\mathcal{S}})$, é dado por todas as execuções possíveis $\mathcal{R}^{\mathcal{S}'} = (R^{\tau_0}{}', R^{\tau_1}{}', R^{\tau_2}{}', ..., R^{\tau_{|B|}}{}', \mathcal{C}', M')$ tais que $R^{\tau} = R^{\tau}{}', \forall \tau \in B \cup \{\tau_0\}, \mathcal{C} = \mathcal{C}'$, onde $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$.

Com $\mathcal{R}^{\mathcal{S}}$, pode-se obter o estado de uma thread $\tau \in B \cup \{\tau_0\}$ em um determinado instante de tempo $t \in \mathbb{N}, 0 \leqslant t \leqslant t^{\tau_0}_{lastEvent(\tau_0)}$ pela função $state$:

$$state : B \cup \{\tau_0\} \times \mathbb{N} \rightarrow Q \tag{4.7}$$

Como um exemplo, considerando-se uma execução representada pela matriz M da tabela 4.1, pode-se dizer que $state(\tau_2, 4) = s_1^{\tau_2}$, considerando-se que $s_1^{\tau_2}$ é o estado para o qual vai a thread τ_2 após o seu primeiro evento ($e_1^{\tau_2}$), de acordo com E^{τ_2}.

Seja $G(t, \mathcal{R}^{\mathcal{S}})$ o estado global das threads do sistema (pertencentes a B) em um tempo $t \in \mathbb{N}$, para uma execução de teste $\mathcal{R}^{\mathcal{S}}$. Tal estado é dado por uma sequência com o estado de cada thread $\tau \in B$ no tempo t. Esse estado é representado pela seguinte sequência:

$$G(t, \mathcal{R}^{\mathcal{S}}) = (state(\tau_1, t), state(\tau_2, t), ..., state(\tau_{|B|}, t)) \tag{4.8}$$

Para um teste, o estado global esperado $\mathcal{E}_n$ para as threads do sistema no momento em que será feita uma asserção a_n é dado pela sequência ordenada de estados esperados para cada thread da aplicação[3]:

[2] O uso de primitivas de sincronização é um exemplo de leitura e escrita de um mesmo dado na memória.

[3] Na verdade pode haver um conjunto de estados esperados possíveis para cada thread e tal mapeamento pode ser feito de forma trivial, mas para efeito de simplificação do texto resolveu-se adotar um único estado

$$\mathcal{E}_n = (s_{\tau_1}, s_{\tau_2}, s_{\tau_3}, ..., s_{\tau_{|B|}}) \tag{4.9}$$

Para que uma execução de teste $\mathcal{R}^{\mathcal{S}}$ *possível* seja *correta* existem algumas restrições quanto ao estado global das threads de B antes de cada um dos eventos de asserção (a_n) definidos em R^{τ_0} que devem ser obedecidas. Sendo d o número total de asserções de um teste, pode-se representar tais restrições pela sequência $\mathcal{D}$ contendo estados esperados no momento de cada uma das d asserções.

$$\mathcal{D} = (\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_d) \tag{4.10}$$

Considerando tais restrições, uma primeira condição para que uma execução seja *correta* é que o estado global $G(t_{a_n}^{\tau_0})$ das threads do sistema no momento de cada asserção a_n deve corresponder ao estado esperado para o sistema ($\mathcal{E}_n$) no momento daquela asserção, ou seja:

$$\mathcal{E}_n = G(t_{a_n}^{\tau_0}, \mathcal{R}^{\mathcal{S}}), \forall n, 1 \leqslant n \leqslant d \tag{4.11}$$

Além disso, um outro requisito para uma execução *correta* é que no momento de tempo em que uma asserção a_n está ocorrendo, em $t_{a_n}^{\tau_0}$, nenhum outro evento de alguma thread de B que altere o estado global esperado para as threads de B seja escalonado. Ou seja, o estado global das threads do sistema deverá ser o mesmo no momento do tempo imediatamente após a asserção, em $t_{a_n}^{\tau_0} + 1$. Isso pode ser representado por:

$$G(t_{a_n}^{\tau_0}, \mathcal{R}^{\mathcal{S}}) = G(t_{a_n}^{\tau_0} + 1, \mathcal{R}^{\mathcal{S}}), \forall n, 1 \leqslant n \leqslant d \tag{4.12}$$

Definição 4 (D4). Considerando isto, diz-se que uma execução de teste $\mathcal{R}^{\mathcal{S}}$ é *correta* se:

$$Correta(\mathcal{R}^{\mathcal{S}}) \iff \mathcal{E}_n = G(t_{a_n}^{\tau_0}, \mathcal{R}^{\mathcal{S}}) = G(t_{a_n}^{\tau_0} + 1, \mathcal{R}^{\mathcal{S}}), \forall n, 1 \leqslant n \leqslant d \tag{4.13}$$

Observando tais restrições, tem-se o seguinte teorema a provar:

esperado possível por thread. Sendo assim, o casamento de um estado obtido a um estado esperado pode ser feito simplesmente utilizando igualdade. Caso fossem utilizados vários estados possíveis por thread, bastaria uma função de casamento (*matching*) que indicasse se uma configuração de estado esperado casa com o estado atual do sistema.

Teorema 1. *Considerando um teste $\mathcal{S}$ em um sistema e restrições $\mathcal{D}$ sobre uma execução de teste correta $\mathcal{R}^{\mathcal{S}}$, se não existe na definição das regras de causalidade $\mathcal{C}$ de $\mathcal{R}^{\mathcal{S}}$ nenhuma regra que relacione qualquer evento de asserção a_n com eventos de threads $\tau \in B$ e se existe pelo menos um estado esperado para asserção $\mathcal{E}_p$ que é diferente do estado inicial do sistema $G(0, \mathcal{R}^{\mathcal{S}})$, então não é possível garantir que todas as execuções $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$ serão corretas .*

Prova. Essa prova é feita por contradição. Considere-se a seguinte hipótese:

H1: Mesmo sem existir em $\mathcal{C}$ nenhuma regra do tipo $a_n \rightarrowtail e^{\tau}$ ou $e^{\tau} \rightarrowtail a_n$, onde e^{τ} é qualquer evento de uma thread $\tau \in B$, a seguinte propriedade é observada: $\mathcal{E}_n = G(t_{a_n}^{\tau_0}, \mathcal{R}^{\mathcal{S}'}) = G(t_{a_n}^{\tau_0} + 1, \mathcal{R}^{\mathcal{S}'}), \forall n, 1 \leqslant n \leqslant d, \forall \mathcal{R}^{\mathcal{S}'} = (R^{\tau_0}, R_1^{\tau}, R_2^{\tau}, ..., R_{|B|}^{\tau}, \mathcal{C}, M'), \mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$, onde $\mathcal{R}^{\mathcal{S}}$ é uma execução de teste correta. Ou seja, todas as execuções equivalentes a uma execução de teste correta também serão corretas.

Esta prova pode ser feita visando demonstrar a ocorrência do problema das asserções feitas em momentos inadequados sob dois ângulos: o das asserções antecipadas e o das asserções tardias.

Seja $\mathcal{E}_p$ o primeiro estado esperado definido em $\mathcal{D}$ para o qual $\mathcal{E}_p \neq G(0, \mathcal{R}^{\mathcal{S}})$ e que corresponde ao estado esperado no momento de um evento de asserção a_p da thread de controle τ_0.

Sejam, portanto, $\mathcal{R}^{\mathcal{S}'}$ e $\mathcal{R}^{\mathcal{S}''}$ duas execuções de um mesmo teste e que são *possíveis* e *equivalentes* à execução correta $\mathcal{R}^{\mathcal{S}}$: $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$ e $\mathcal{R}^{\mathcal{S}''} \in Equiv(\mathcal{R}^{\mathcal{S}})$. Pela hipótese H1, ambas são também *corretas*.

Seja $\mathcal{R}^{\mathcal{S}'}$ a execução de teste cuja matriz de escalonamento M' representa a execução de teste em que o evento de asserção a_p ocorre no menor momento de tempo t após o qual o estado global do sistema $\mathcal{E}_p$ tenha sido alcançado e os eventos da thread de controle anteriores a a_p não tenham ocorrido no momento de tempo imediatamente antes de $t_{a_p}^{\tau_0}$, ou seja em $(t_{a_p}^{\tau_0} - 1)$.

Portanto, sendo $\mathcal{R}^{\mathcal{S}'}$ correta, o estado global esperado para cada asserção a_p que foi escalonada no tempo $t_{a_p}^{\tau_0}$ foi alcançado no tempo $t_{a_p}^{\tau_0} - 1$ através de pelo menos um evento em alguma das threads de B que levou a uma transição de estado que conduziu tal thread ao seu estado esperado.

Tome-se agora a matriz M'', de uma execução de teste $\mathcal{R}^{\mathcal{S}''} \in Equiv(\mathcal{R}^{\mathcal{S}})$ com o mesmo escalonamento no tempo para as threads de B que M' provê. Como não há regra de causalidade que relacione o evento de asserção a_p a qualquer outro evento de alguma thread de B, nada impede que em M'' a asserção a_p seja escalonada para o momento no tempo $t_{a_p}^{\tau_0} - 1$. Porém, uma vez que isto ocorre, a restrição para o estado global do sistema $\mathcal{E}_p$ não será respeitada, já que o estado global esperado ainda não terá sido alcançado.

Retomando **H1**, pode-se dizer que existiu um n, aqui representado por p, para o qual $\mathcal{E}_n \neq G(t_{a_n}^{\tau_0}, \mathcal{R}^{\mathcal{S}''})$. Ou seja, existiu uma execução *equivalente* a uma execução *correta*, mas que não era *correta*, contradizendo a hipótese **H1** e provando o **Teorema 1**.

Nesta prova mostrou-se a ocorrência do problema das asserções antecipadas. Uma outra forma similar de provar o teorema é considerar o mesmo estado esperado $\mathcal{E}_p$ previamente discutido, e que, para ser alcançado, demanda a existência de pelo menos um evento e_y^{τ} em alguma thread τ de B que muda o estado inicial do sistema e que ocorre em um momento t de uma execução de teste correta. Focando-se numa prova que demonstre a ocorrência das asserções tardias pode-se utilizar uma argumentação semelhante a apresentada acima, diferenciando-se apenas por considerar a existência de um evento a_{p-1} antes de a_p e pelo fato de mostrar que como não há regra de causalidade que relacione o evento de asserção a_{p-1} a qualquer outro evento de alguma thread de B, nada impede que o evento a_{p-1}, em uma execução de teste equivalente a uma execução correta, aconteça no mesmo momento de tempo t em que foi escalonado o evento e_y^{τ}, que faz a thread τ mudar para um estado diferente do esperado para o momento da asserção a_{p-1}. Porém, uma vez que isto ocorre, a restrição relacionada ao conceito de execução *correta* de que o estado global deve se manter durante a execução da asserção não será respeitada, o que faria com o que tal execução equivalente não fosse *correta*. Chegar-se-ia novamente a uma contradição que provaria o **Teorema 1**.□

4.2 Requisitos para Resolver o Problema

Para evitar os problemas apontados anteriormente em uma execução de teste $\mathcal{R}^{\mathcal{S}}$, deve haver regras de causalidade que relacionem eventos de asserção a eventos de threads $\tau \in B$.

Seja $\mathcal{D}$ a sequência que define as restrições quanto ao estado global do sistema nos momentos de asserção, a definição de $\mathcal{D}$ apresenta um $\mathcal{E}_n, 1 \leqslant n \leqslant d$, para cada asser-

ção a_n, que corresponde ao estado esperado para o sistema no momento desta asserção e que se for respeitado para todas as asserções, será a primeira pré-condição para uma execução de teste *correta*. Conforme a definição dada para o estado global esperado, exige-se que cada thread $\tau \in B$ esteja em um estado específico s_τ da sequência definida em $\mathcal{E}_n = (s_{\tau_1}, s_{\tau_2}, s_{\tau_3}, ..., s_{\tau_{|B|}})$.

Para que haja uma execução correta, é preciso que o último evento realizado por cada thread de B que ocorreu antes da execução da asserção a_n (em $t_{a_n}^{\tau_0}$) tenha levado ao estado esperado para aquela thread, conforme a definição de eventos-estados E^τ.

Para que isso ocorra, para cada thread $\tau \in B$, ou o estado esperado para ela é o estado inicial s_0^τ e não houve nenhum evento, ou então deve existir um evento e_x^τ, correspondente ao elemento (e_x^τ, s_x^τ) em E^τ, tal que s_x^τ seja o estado esperado para τ em $\mathcal{E}_n$ e o evento e_x^τ ocorra antes do evento a_n, o que só pode ser garantido se $e_x^\tau \rightarrowtail a_n$.

Após tal evento, para cada thread τ, e enquanto a_n não tiver sido concluída, não pode ocorrer nenhum evento e_y^τ que mude o estado da thread τ. Ou seja, o par correspondente a este evento de mudança de estado em E^τ, representado por (e_y^τ, s_y^τ), é tal que $s_y^\tau \neq s_\tau$, onde s_τ é o estado esperado para a thread τ de acordo com $\mathcal{E}_n$. Portanto, se após a_n houver algum evento e_y^τ que mude o estado da thread, este deve acontecer num tempo $t > t_{a_n}^{\tau_0}$, o que só pode ser garantido se $a_n \rightarrowtail e_y^\tau$.

Considere-se, portanto, o conjunto de regras de causalidade $\mathcal{C}^{\tau_0}$, construído com o auxílio de duas funções auxiliares descritas a seguir.

Seja $lastEventBefore(a_n, \tau, M)$ a função responsável por obter em um escalonamento M o último evento relativo à thread τ ocorrido antes da asserção a_n, caso exista tal evento. Sendo assim, $lastEventBefore(a_n, \tau, M) = e_z^\tau$, onde e_z^τ é último evento diferente de nop da sequência de eventos e_b^τ, $0 \leqslant b \leqslant n$ correspondente à linha da matriz M que representa τ.

Seja $nextChangingEvent(\tau, a_n, M)$ a função responsável por obter, de acordo com um escalonamento M, o próximo evento de uma thread τ que mude o seu estado, após uma asserção a_n e caso tal evento exista. Sendo assim, $nextChangingEvent(a_n, \tau, M) = e_w^\tau$, onde e_w^τ é primeiro evento diferente de nop da sequência de eventos e_b^τ, $n+1 \leqslant b \leqslant lastEvent(\tau)$ correspondente à linha da matriz M que representa τ tal que seu par correspondente em E^τ, representado por (e_w^τ, s_w^τ), é tal que $s_w^\tau \neq s_\tau$, onde s_τ é o estado esperado para a thread τ de acordo com $\mathcal{E}_n$.

Definição 5 (D5). Pode-se construir, a partir de uma execução *correta* de teste $\mathcal{R}^{\mathcal{S}}$ o conjunto $\mathcal{C}^{\tau_0}$ de regras de causalidade da seguinte forma: Para o escalonamento M de uma execução correta $\mathcal{R}^{\mathcal{S}}$, para todo a_n e considerando aí cada thread τ, deve ser adicionada a $\mathcal{C}^{\tau_0}$ a regra $e_z^{\tau} \rightarrowtail a_n$, caso tal evento exista. Além disso, para todo a_n e para cada thread τ, deve ser considerado o próximo evento que modificaria o estado da thread e ocorrido após a_n, representado por e_w^{τ}. Caso este evento exista para a thread τ, deve também ser adicionada a $\mathcal{C}^{\tau_0}$ a regra: $a_n \rightarrowtail e_w^{\tau}$.

Sendo assim, propõe-se o seguinte teorema:

Teorema 2. *Considerando um teste $\mathcal{S}$ em um sistema e restrições $\mathcal{D}$ sobre uma execução de teste correta $\mathcal{R}^{\mathcal{S}}$, se todos os elementos do conjunto de regras de causalidade $\mathcal{C}^{\tau_0}$, construído conforme descrito em* ***D5****, fizerem parte das regras de causalidade $\mathcal{C}$ de $\mathcal{R}^{\mathcal{S}}$, ou seja, se $\mathcal{C}^{\tau_0} \subset \mathcal{C}$, então é possível garantir que todas as execuções $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$ serão corretas.*

Prova: A prova será feita por construção, assumindo as mesmas pré-condições do enunciado do Teorema 2 e adicionando-se as regras de causalidade requeridas.

Seja M' a matriz de escalonamento correspondente a execução de teste $\mathcal{R}^{\mathcal{S}'}$, que é *equivalente* a uma execução correta $\mathcal{R}^{\mathcal{S}}$ de matriz de escalonamento M. Em tais execuções são consideradas regras de causalidade $\mathcal{C}^{\tau_0}$ relacionando os eventos de asserções aos eventos das threads de B de acordo com a execução correta $\mathcal{R}^{\mathcal{S}}$ e as regras de construção de $\mathcal{C}^{\tau_0}$ descritas em D5. Segundo as pré-condições do teorema, tais regras devem estar inclusas em $\mathcal{C}$.

Sendo assim, para cada asserção a_p há um estado esperado em $\mathcal{D}$ para as threads de B, denominado $\mathcal{E}_p = (s_{\tau_1}, s_{\tau_2}, s_{\tau_3}, ..., s_{\tau_{|B|}})$ e que é alcançado para todas as asserções considerando o escalonamento descrito em M. Considerando tais estados e a matriz M, e quando houver um evento que levou a thread τ ao estado esperado, ou seja $s_0^{\tau} \neq s_{\tau}$, haverá em $\mathcal{C}$ regras de causalidade para cada thread τ entre o último evento de E^{τ} que levou ao estado esperado s_{τ} para essa thread, denominado e_z^{τ} (obtido por $lastEventBefore(a_p, \tau, M)$), e o evento de asserção a_p. Haverá também regras para o próximo evento na sequência de eventos/estados E^{τ} e que altere esse estado, denominado e_w^{τ} (obtido por $nextChangingEvent(\tau, a_p, M)$), também com relação a a_p. Se existirem pares de eventos/estados entre os que são identificados por e_z^{τ} e e_w^{τ} na sequência E^{τ}, todos os estados destes pares são correspondentes ao estado esperado para essa thread, s_{τ}.

Devido às regras de causalidade presentes em $\mathcal{C}$, na execução de qualquer $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$, de matriz de escalonamentos M', o último evento de cada thread τ escalonado antes de cada asserção a_p ou será o evento e_z^τ, ou algum outro evento entre este evento e e_w^τ da matriz M de escalonamentos de $\mathcal{R}^{\mathcal{S}}$, mas que também leva ao estado esperado para a thread τ, ou não haverá nenhum evento antes da asserção, caso o estado inicial desta thread (s_0^τ) já seja o estado esperado para esta thread. Em todos estes casos, o estado esperado para o sistema no momento de cada asserção será sempre o estado esperado $\mathcal{E}_p$. Esse fato, associado às regras de causalidade que impedem a ocorrência de eventos que mudem o estado do sistema após o evento a_p e antes de e_w^τ enquanto a asserção não tiver ocorrido, fazem com que qualquer execução $\mathcal{R}^{\mathcal{S}'}$ *possível*, *equivalente* a $\mathcal{R}^{\mathcal{S}}$ seja uma execução *correta*.□

Teorema 3. *Considerando um teste $\mathcal{S}$ em um sistema e restrições $\mathcal{D}$ sobre uma execução de teste correta $\mathcal{R}^{\mathcal{S}}$, de regras de causalidade $\mathcal{C}$, se $\mathcal{C}$ não contém todos os elementos do conjunto de regras de causalidade $\mathcal{C}^{\tau_0}$ ($\mathcal{C}^{\tau_0} \nsubseteq \mathcal{C}$), construído conforme descrito em **D5**, e se existe pelo menos um estado esperado para asserção $\mathcal{E}_p$ que é diferente do estado inicial do sistema $G(0, \mathcal{R}^{\mathcal{S}})$, então não é possível garantir que todas as execuções $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$ serão corretas.*

Prova: Essa prova pode ser feita por contradição, tomando como base a seguinte hipótese:

H2: Considerando um teste $\mathcal{S}$ em um sistema e restrições $\mathcal{D}$ sobre uma execução de teste *correta* $\mathcal{R}^{\mathcal{S}}$, de regras de causalidade $\mathcal{C}$, se $\mathcal{C}$ não contém todos os elementos do conjunto de regras de causalidade $\mathcal{C}^{\tau_0}$, construído conforme descrito em **D5**, e se existe pelo menos um estado esperado para asserção $\mathcal{E}_p$ que é diferente do estado inicial do sistema $G(0, \mathcal{R}^{\mathcal{S}})$, então é possível garantir que todas as execuções $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$ serão *corretas*.

Sejam $\mathcal{R}^{\mathcal{S}'} \in Equiv(\mathcal{R}^{\mathcal{S}})$ e $\mathcal{R}^{\mathcal{S}''} \in Equiv(\mathcal{R}^{\mathcal{S}})$ duas execuções *possíveis* e *equivalentes* à execução correta $\mathcal{R}^{\mathcal{S}}$. Pela hipótese H2, ambas são também *corretas*.

Seja a_p o primeiro evento de asserção de τ_0 tal que o estado esperado para o sistema no momento em que ele acontece, denominado $\mathcal{E}_p$ e definido pelas restrições em $\mathcal{D}$, seja diferente do estado inicial do sistema $G(0, \mathcal{R}^{\mathcal{S}})$. Considerando esta pré-condição do teorema, existe pelo menos um evento e_y^τ, de alguma thread $\tau \in B$ que fará o sistema sair do estado inicial, mudando assim de estado.

Seja $\mathcal{R}^{\mathcal{S}'}$ a execução de teste equivalente a $\mathcal{R}^{\mathcal{S}}$ cuja matriz de escalonamento M' represente uma execução de teste em que o primeiro evento de alguma thread de B que modifica o estado global do sistema, denominado e_y^τ, ocorra no momento do tempo t.

Tome-se agora a matriz M'', da execução de teste $\mathcal{R}^{\mathcal{S}''} \in Equiv(\mathcal{R}^{\mathcal{S}})$ com o mesmo escalonamento no tempo para as threads de B que M' provê. Se não existirem todas as regras obrigadas por **D5**, poderia não haver regra de causalidade que relacione o evento de asserção a_p a e_y^τ, o evento que mudaria o estado da thread τ para o estado esperado para esta thread no momento desta asserção. Sendo assim, nada impede que em M'' o evento de asserção a_p aconteça no tempo $t-1$, ou seja, antes do momento em que foi escalonado o evento e_y^τ. Porém, uma vez que isto ocorre, a restrição relacionada ao conceito de execução *correta* de que o estado global no momento da asserção deve corresponder ao estado esperado no momento dessa asserção não será respeitada, ou seja, $G(t_{a_p}^{\tau_0}, \mathcal{R}^{\mathcal{S}''}) \neq \mathcal{E}_p$, o que faria com que $\mathcal{R}^{\mathcal{S}''}$ não fosse *correta*.

Dessa forma, chegou-se a uma contradição, estando assim o **Teorema 3** provado. □

4.3 Uma Solução Baseada em Monitores

Observando-se os requisitos baseados em regras de causalidade para resolver os problemas de se fazer asserções cedo (antes de um estado esperado global) ou tarde demais (depois que um estado esperado global foi atingido e não é mais alcançável), e fazer com que se tenham apenas execuções corretas de teste, resta ainda o desafio de garantir que os eventos seguirão tais regras de causalidade.

Sabe-se que para respeitar as regras envolvendo apenas threads da aplicação ($\tau \in B$), a própria aplicação já se prepara através de mecanismos explícitos para garantir certo ordenamento entre seus eventos, como por exemplo os semáforos. Porém, ao considerar a thread de controle τ_0 do teste e seus eventos, normalmente não há mecanismos explícitos para isso, já que se deve evitar que a thread de controle do teste seja muito intrusiva com relação a aplicação em si. A solução proposta neste trabalho é o uso de monitores com o mínimo de intrusividade, que garantam que ordenamentos entre eventos de asserção e eventos das threads da aplicação sejam respeitados.

4.3.1 Descrição da Solução

Um **monitor** é uma entidade adicionada ao sistema e que tem a capacidade de alterar o escalonamento das threads da aplicação ($\tau \in B$) ou de controle (τ_0) mas apenas no sentido de poder adicionar eventos de *nop* ao escalonamento para evitar asserções antecipadas ou tardias. Isso é feito no intuito de garantir que determinadas regras de implementação (IR, *implementation rules*) serão seguidas na implementação do teste para que as restrições $\mathcal{D}$ sejam observadas durante a sua execução. Pode-se definir, portanto, uma execução de teste monitorada $\mathcal{R}^{\mathcal{S}^{mon}}$ pela seguinte tupla:

$$\mathcal{R}^{\mathcal{S}^{mon}} = (\mathcal{R}^{\mathcal{S}}, \mathcal{D}, IR) \tag{4.14}$$

Sem as regras de implementação IR, algumas execuções de teste podem não ser corretas. Para evitar que isso aconteça, é preciso considerar apenas escalonamentos em que as restrições $\mathcal{D}$ relativas aos eventos de asserção a_n da thread de controle τ_0 sejam obedecidas.

Para isso, além de considerar a causalidade entre os eventos das threads $\tau \in B$ e os eventos de asserção de τ_0, deve-se considerar também o estado global das threads do sistema a cada unidade de tempo como condição para que um evento seja escalonado no momento seguinte de tempo. Para isso, as restrições $\mathcal{D}$ quanto ao estado global das threads antes de cada evento de asserção devem ser consideradas. Portanto, as seguintes regras de implementação $IR = \{IR_1, IR_2, IR_3\}$ devem ser seguidas pelas matrizes de escalonamento:

- IR_1: A cada nova unidade de tempo t, observa-se o próximo evento a escalonar de cada thread $\tau \in B \cup \{\tau_0\}$ e o estado global atual das threads no tempo t: $G(t, \mathcal{R}^{\mathcal{S}})$. Se o próximo evento de τ_0 for alguma asserção a_n, observa-se se o estado global esperado $\mathcal{E}_n$ correspondente a tal asserção em $\mathcal{D}$. Se $G(t, \mathcal{R}^{\mathcal{S}}) = \mathcal{E}_n$, na coluna da matriz de escalonamento referente ao tempo t e na linha correspondente a τ_0 deverá ser escalonado o evento a_n de τ_0 ou um evento de *nop*.

- IR_2: Considerando o mesmo que foi considerado em IR_1, e tendo sido escalonada para o instante t a asserção a_n, não deverá ser escalonado no mesmo momento de tempo nenhum outro evento e_n^{τ} de alguma thread $\tau \in B$ que leve a alguma mudança de estado de sua thread. Ou seja, e_n^{τ} só poderá ser escalonado em t se, considerando

os eventos e estados de τ como sendo $E^\tau = (..., (e^\tau_{n-1}, s^\tau_{n-1}), (e^\tau_n, s^\tau_n), ...)$, $s^\tau_{n-1} = s^\tau_n$. Caso contrário, a thread $\tau \in B$ terá um evento de nop escalonado para o tempo t.

- IR_3: Nas mesmas condições iniciais descritas por IR_1, onde o próximo evento de τ_0 é também alguma asserção a_n. Se $G(t, \mathcal{R}^\mathcal{S}) \neq \mathcal{E}_n$, o evento a_n não pode ser escalonado em t. A thread de controle ficará executando eventos de *nop* enquanto o estado esperado global para as threads do sistema (como todas finalizadas ou esperando, por exemplo) não tiver sido alcançado. Isso garantirá que as asserções não serão feitas antes do momento esperado.

Considerando tais regras, vê-se que as matrizes de escalonamento que gerassem execuções de teste que não fossem *corretas* não poderiam existir. Se apenas IR_1 e IR_3 fossem consideradas, evitar-se-ia a geração de matrizes com asserção antecipada, mas não as com asserção tardia. Mas seguindo-se todas as regras de IR nunca haveria um evento de asserção antecipado ou tardio. Pode-se provar que isto é verdade fazendo um mapeamento entre $\mathcal{R}^{\mathcal{S}^{mon}}$ e as regras de causalidade exigidas no Teorema 2.

4.3.2 Prova

Pode-se fazer um paralelo entre as IRs e a construção do conjunto de regras de causalidade $\mathcal{C}$ que garantem um conjunto de execuções *equivalentes corretas*.

$IR1$ em conjunto com $IR3$ garantem a relação de causalidade $e^\tau_z \rightarrowtail a_n$ para uma execução de teste $\mathcal{R}^\mathcal{S}$ correta, de matriz de escalonamentos M, onde e^τ_z é obtido por $lastEventBefore(a_n, \tau, M)$.

$IR2$ é equivalente à criação de regras de causalidade com a relação $\rightarrowtail$ que fazem com que para uma execução correta, o próximo evento após uma asserção que mude o estado do sistema, denominado e^τ_z, obtido por $nextChangingEvent(\tau, a_n, M)$, só seja escalonado depois da asserção, permitindo assim que o estado do sistema no momento seguinte ao da asserção seja mantido, ou seja, para uma execução de teste $\mathcal{R}^\mathcal{S}$, $G(t^{\tau_0}_{a_n}, \mathcal{R}^\mathcal{S}) = G(t^{\tau_0}_{a_n} + 1, \mathcal{R}^\mathcal{S})$.

Sendo assim, as execuções de teste acompanhadas de monitores $\mathcal{R}^{\mathcal{S}^{mon}}$ serão *corretas*.

□

4.4 Conclusões Parciais

Neste capítulo foi apresentada uma modelagem do problema do qual trata este trabalho, as asserções antecipadas e tardias e demonstrou-se a existência deste problema através da prova do Teorema 1. Além disso, foram apresentados requisitos para resolver o problema e também uma proposta de solução para ele.

Pode-se perceber que o modelo inicial de execução de testes $\mathcal{R}^{\mathcal{S}}$, em que na geração das matrizes de escalonamento não se considera um monitor para garantir que as regras de implementação IR sejam seguidas, corresponde na prática aos testes em que se usa como método de espera a técnica de atrasos explícitos. Essa técnica caracteriza-se por chamadas de operações visando gerar atrasos antes que se iniciem os eventos de asserção (o que equivale a inserir eventos de *nop* antes dos eventos de asserção a_n na matriz de escalonamentos).

Já um modelo estendido $\mathcal{R}^{\mathcal{S}^{mon\prime}} = (\mathcal{R}^{\mathcal{S}}, \mathcal{D}, \{IR_1, IR_3\})$, considerando só as regras IR_1 e IR_3 corresponde aos testes com espera ocupada, em que além da espera, se faz verificações de tempos em tempos. Porém, esse modelo não resolve o problema da asserção tardia.

Para resolvê-lo, a regra IR_2 deve ser considerada. A solução proposta neste trabalho baseia-se no modelo de execução $\mathcal{R}^{\mathcal{S}^{mon}}$ utilizando todas as regras de IR.

Capítulo 5

A Abordagem *Thread Control for Tests*

Neste capítulo é apresentada a abordagem para testes de operações envolvendo assincronia denominada *Thread Control for Tests*. Além disso, é apresentado também o arcabouço de testes que foi desenvolvido para dar suporte à abordagem, o *ThreadControl*. Considerando isso, é mostrado um detalhamento da versão atual desse arcabouço e de sua arquitetura. Este arcabouço foi utilizado nos estudos de caso apresentados no próximo capítulo e que foram realizados com o objetivo de mostrar que a abordagem é aplicável na prática e também de verificar se com o uso do arcabouço se consegue evitar o problema das asserções que ocorrem cedo ou tarde demais.

Este capítulo está organizado da seguinte forma. A Seção 5.1 apresenta um trabalho inicial que motivou o desenvolvimento desta tese de doutorado, onde é descrito um arcabouço de testes que contribuiu para os testes do sistema OurGrid, e do qual se originou o arcabouço *ThreadControl*. A Seção 5.2 apresenta a abordagem *Thread Control for Tests*. A Seção 5.3 apresenta o arcabouço de testes *ThreadControl*, que foi desenvolvido para avaliar a abordagem apresentada demonstrando seu uso na prática. Por fim, a Seção 5.4 sintetiza este capítulo.

5.1 O Arcabouço ThreadServices

O trabalho que foi o precursor da abordagem *Thread Control for Tests* foi publicado no Jornal da Sociedade Brasileira de Computação [25]. Esse trabalho discute como a técnica de Programação Orientada a Aspectos [54] foi utilizada para auxiliar nos testes do OurGrid [16],

um *middleware* para grades computacionais de código aberto.

Este trabalho foi motivado pelo fato de que, no Laboratório de Sistemas Distribuídos (LSD) da UFCG, o problema de falhas de testes que não eram causadas por faltas na aplicação, mas por asserções antecipadas, era comumente observado. Buscando evitá-lo no desenvolvimento do OurGrid , buscou-se melhorar os seus testes oferecendo aos desenvolvedores uma operação denominada `waitUntilWorkIsDone` por meio de um arcabouço de testes chamado `ThreadServices` [25]. Essa operação fazia com que a *thread* do teste esperasse até que todas as *threads* criadas pelo teste (enquanto invoca operações no sistema sendo testado) tivessem finalizado ou estivessem em um estado de espera (ex: aguardando novas requisições, indicando assim que requisições antigas já foram processadas). Só após tal espera é que se iniciavam as asserções dos testes.

Para ilustrar o uso dessa operação, pode-se tomar como exemplo o teste do escalonador de tarefas do OurGrid. O teste envolvia threads para submeter tarefas a serem escalonadas e outras threads do próprio escalonador para distribuir as tarefas entre as máquinas do grid e condensar os resultados produzidos. Ao invés de incluir no teste e na aplicação várias barreiras de sincronização para que a verificação via asserção sobre a conclusão das tarefas fosse feita, bastava incluir antes da chamada à asserção uma chamada ao método `waitUntilWorkIsDone` oferecido pelo arcabouço `ThreadServices`.

Outros testes do OurGrid se beneficiaram dessa operação e não mais falharam devido ao problema das asserções antecipadas (em algumas execuções dos testes as asserções eram executadas antes que houvesse tempo suficiente para que as operações assíncronas sendo testadas tivessem sido concluídas). No entanto, à medida que o sistema evoluiu, percebeu-se a necessidade de se poder definir configurações mais específicas de estados de *thread* pelos quais se desejava esperar (além do estado de todas as threads da aplicação estarem esperando ou paradas, pois era interessante também, por exemplo, poder esperar só até que uma certa thread estivesse parada). Além disso, um outro problema observado foi que em alguns testes havia *threads* periódicas que poderiam acordar de tempos em tempos, no momento das asserções, podendo gerar falhas já que alteravam o estado do sistema (asserções tardias).

Tais fatos motivaram a criação de uma abordagem que estendesse esse trabalho anterior, e que melhor tratasse o problema das asserções que ocorriam em momentos inadequados, abordando não só casos de asserções antecipadas, mas também as asserções tardias. Além

disso, pretendia-se permitir também que se pudesse especificar que um teste deveria poder esperar até uma determinada fase de execução da aplicação sob teste, mapeada em um certo estado de suas threads, o qual não deveria ser apenas o estado em que estariam todas paradas ou finalizadas, como era o caso do que era fornecido pelo arcabouço *ThreadServices*. A idéia seria permitir também a definição de outros estados do sistema pelos quais se pode esperar e também de outras transições de estados para as threads. Para evitar o problema das asserções tardias, viu-se também que a abordagem a ser utilizada deveria também ser capaz de evitar mudanças de estado enquanto asserções estão sendo feitas. Ao evoluir este trabalho inicial sob a forma de uma abordagem, ao invés de apenas um novo arcabouço, buscou-se prover uma solução mais geral para o problema das asserções antecipadas e tardias e que pudesse ser utilizada em diferentes contextos de sistemas, implementados em diferentes linguagens de programação, provendo-se as bases para a criação de diferentes arcabouços de teste.

5.2 A Abordagem *Thread Control for Tests*

Visando resolver os problemas existentes em abordagens como atrasos explícitos e espera ocupada, e ao mesmo tempo prover uma solução baseada em primitivas oferecidas aos testadores, com funcionalidade semelhante ao método `waitUntilWorkIsDone` do trabalho anterior, foi proposta a abordagem *Thread Control for Tests*. Para oferecer tais primitivas e evitar o problema das asserções feitas cedo ou tarde demais, essa abordagem se baseia na monitoração e no controle das threads do sistema sendo testado. O serviço geral provido por essas primitivas é a capacidade de fazer o teste esperar até um momento apropriado onde asserções podem ser feitas com segurança (por exemplo, no momento em que todas as threads estão paradas) e também a habilidade de evitar mudanças no estado das threads do sistema uma vez que o estado esperado para asserções, especificado também através de uma primitiva de teste, foi atingido, evitando assim falhas indesejadas na execução dos testes.

Thread Control for Tests é uma abordagem geral para o teste de sistemas *multi-threaded*. Ela propõe que testes envolvendo operações assíncronas apresentem as seguintes fases:

1. **preparação do ambiente**, indicando o estado esperado no qual asserções podem ser executadas;

2. invocação das operações do sistema, **exercitando-o**;

3. **espera** até que o estado esperado para o sistema, especificado na fase 1, tenha sido alcançado, podendo a thread do teste ser notificada uma vez que isso acontece;

4. **execução das asserções**, sem temer que alguma *thread* desperte e execute operações que afetem os resultados de teste;

5. **prosseguimento do teste**, permitindo assim que o sistema e o teste continuem normalmente (retornando-se à fase 1) ou terminem. Essa fase é necessária pois threads que tentam mudar de estado enquanto asserções estão sendo feitas são bloqueadas para evitar as asserções tardias, e só nessa fase de prosseguimento é que são liberadas para prosseguir.

Esta abordagem propõe que as fases 1, 3 e 5 sejam auxiliadas por uma ferramenta de testes. A idéia básica é que tal ferramenta deve disponibilizar aos desenvolvedores de testes algumas operações simples (primitivas de teste). Para isso, a ferramenta deve ser capaz de monitorar as *threads* do sistema e notificar o teste assim que um estado esperado para o sistema é alcançado. Além disso, ela deve também evitar perturbações no sistema enquanto as asserções estão sendo executadas (fase 4).

Na abordagem *Thread Control for Tests*, um estado a ser alcançado é definido em termos de estados das *threads*, podendo tal definição ser feita de maneira geral ou específica, como explicado a seguir. Um estado pelo qual se deseja esperar pode ser definido de maneira geral, como por exemplo: *espere até que todas as threads criadas pelo teste estejam esperando ou tenham terminado* ou *espere até que o trabalho de todas as threads tenha terminado, o que representa uma certa fase na execução do sistema*. Pode-se também especificar uma condição de parada de uma forma mais específica, como no estilo: *espere até que certas threads estejam em determinados estados* ou *espere até que uma Thread B termine e todas as instâncias da Thread A estejam esperando*. Uma definição de condição mais geral pode simplificar bastante os testes já que pode ser reusada (já que vários testes podem utilizar a mesma condição de espera antes das asserções) e esta abordagem foi explorada na definição da operação `waitUntilWorkIsDone` do trabalho sobre o *ThreadServices* [25]. No entanto, quando começam a aparecer muitas exceções à regra, como threads a excluir na

monitoração ou quando ter um controle sobre o estado de todas é desnecessário ou complexo, sendo suficiente uma identificação de uma thread cujo estado é de interesse, vê-se a importância de se poder definir estados esperados mais específicos. A idéia básica de poder oferecer tal funcionalidade é aumentar as possibilidades de estados esperados que podem ser especificadas.

A Figura 5.1 ilustra a idéia geral da abordagem proposta. Ela mostra a monitoração do sistema sendo testado (SUT - *System Under Test*), considerando pontos de execução pelos quais passam as threads da aplicação (representados por círculos pequenos dentro dos componentes do SUT) e que representam suas transições de estado. Uma vez que tais pontos são alcançados, a entidade responsável por prover primitivas oferecidas aos desenvolvedores de testes, na figura representada pelo `SystemWatcher`, é notificada.

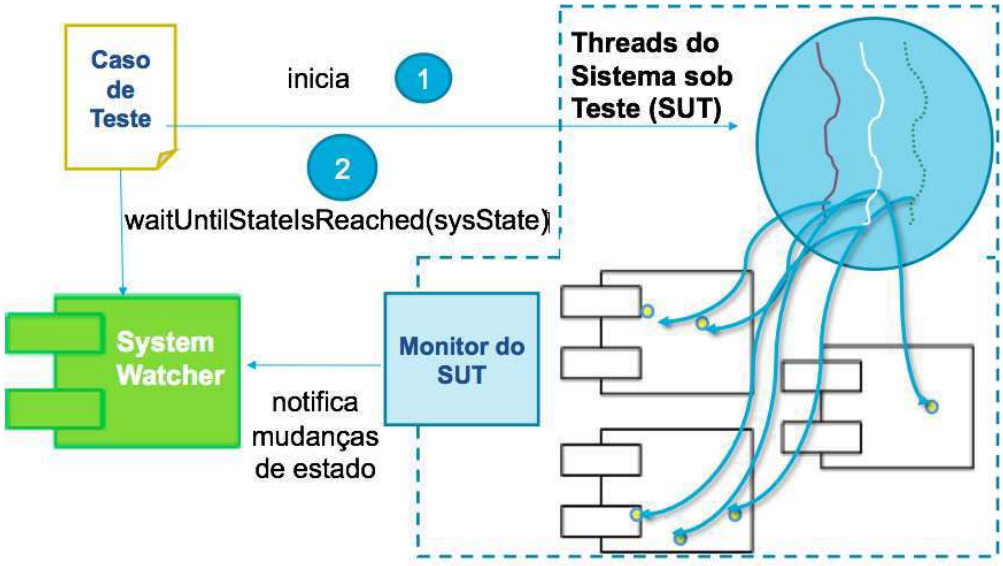


Figura 5.1: Visão Geral da Abordagem

No processo de notificação, caso seja percebido que o estado esperado para asserções já foi alcançado, a thread do teste pode prosseguir e futuras transições de estado nas threads do SUT são bloqueadas enquanto asserções não são concluídas. Um exemplo de uso dessa abordagem é um teste que utilize a primitiva `waitUntilStateIsReached(sysState)` e cujo estado esperado (`sysState`) seja o momento em que uma certa thread A está parada. Neste caso, quando a thread do teste atinge a invocação a esta primitiva de teste, ela fica esperando até ser notificada. Tomando como ponto de monitoração da aplicação a chamada

ao método `Object.wait` de Java e considerando que ao passar por este ponto a thread A é considerada "parada", uma vez que este ponto é atingido, a thread do teste é notificada e volta a executar, realizando normalmente as asserções.

Um outro serviço que a abordagem apresenta é a habilidade de evitar que outras threads perturbem o estado do sistema enquanto as asserções estão sendo realizadas e enquanto ainda não tiver havido uma chamada explicita à alguma primitiva de prosseguimento. Isto é feito bloqueando a thread que queira mudar de estado e o bloqueio ocorre no momento de notificar a mudança de estado que estaria para acontecer.

É importante destacar que **a abordagem *Thread Control for Tests* pode ser vista como um mecanismo geral de sincronização/coordenação entre o teste e a aplicação, em que se evitam condições de corrida (*race conditions*) entre estes no momento de verificação dos resultados nos testes**.

Para que a abordagem se torne menos intrusiva do que mecanismos como semáforos explícitos na aplicação e nos testes, propõe-se que os mecanismos de monitoração e controle das threads não sejam colocados diretamente no código da aplicação e que técnicas de instrumentação auxiliem nesse processo. Além disso, os desenvolvedores de testes terão como interface apenas as primitivas de teste, embora por trás, mecanismos de sincronização estejam sendo providos por ferramentas de teste de apoio à abordagem, como é o caso da ferramenta *ThreadControl*, explicada a seguir.

Como se pode observar pela descrição da abordagem, para dar suporte a ela, devem ser providos mecanismos para monitorar transições de estado, inclusive para evitar que certas transições ocorram enquanto asserções estão sendo feitas. Como forma de deixar mais concreta a forma de fazer a monitoração, a Figura 5.2 ilustra alguns exemplos de operações que podem ser monitoradas na linguagem Java. Essa figura também mostra alguns estados possíveis das *threads* do sistema e que são atingidos depois ou antes que essas operações sejam executadas.

Conforme ilustra a Figura 5.2, considera-se que uma *thread* está indo para o estado terminada (*Finished*) quando a execução de seu método `run` for concluída. Pode-se também dizer que ela está esperando (*Waiting*) uma vez que é feita uma chamada (*call*) ao método `wait` e que ela volta a estar rodando (*Running*) ao retornar dessa chamada. Transições como estas devem ser monitoradas para que se detecte o mais cedo possível quando um estado esperado

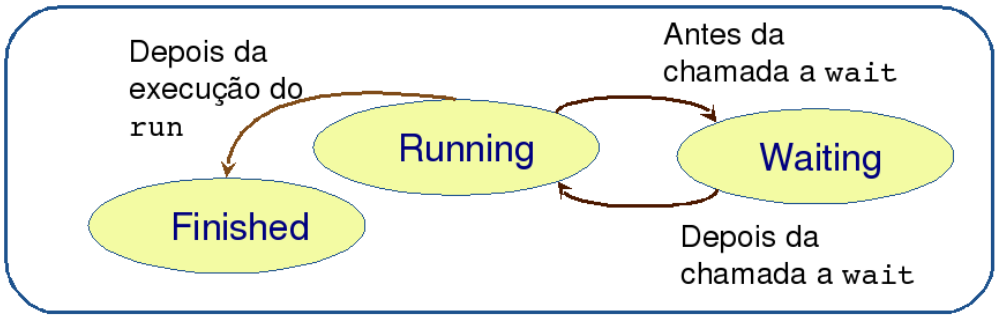


Figura 5.2: Algumas operações monitoradas e estados relacionados a estas operações.

para o sistema tiver sido alcançado e se notifique a *thread* do teste. Além disso, algumas threads monitoradas que tentam passar por transições de estado enquanto as asserções estão sendo feitas devem ser bloqueadas até o momento em que o próprio teste as faça prosseguir.

Ao definir através da abordagem uma *configuração de sistema esperada* em termos de determinadas *threads* e seus estados associados, devem também ser definidos os estados de interesse (como threads iniciadas, rodando, esperando, terminadas ou mesmo um estado específico da aplicação como processando requisições). Além disso, deve-se também identificar antes ou depois de que pontos no fluxo de execução do sistema devem ocorrer as transições de estado. Tais fatores serão considerados ao prover o suporte ferramental a essa abordagem via arcabouços de teste como o *ThreadControl*, explicado a seguir.

5.3 O Arcabouço de Testes *ThreadControl*

Para avaliar a abordagem e difundir melhor o seu uso, foi projetada e implementada, como parte deste trabalho, uma ferramenta para auxiliar no desenvolvimento de testes para sistemas assíncronos que se utilizem do arcabouço JUnit [5]. Essa ferramenta foi desenvolvida em Java e AspectJ e recebeu o nome de *ThreadControl*, sendo seu código uma evolução do arcabouço *ThreadServices*. Esse arcabouço se encontra disponível, como código aberto, no endereço `http://code.google.com/p/threadcontrol/`. O código fonte de sua versão atual (0.3) encontra-se no Apêndice G.

Nesta seção será mostrada a arquitetura desse arcabouço e serão descritos alguns detalhes a respeito de sua implementação e uso. Tais informações são apresentadas como forma de melhor difundir a abordagem proposta e de prover guias para outras implementações de

ferramentas que dêem suporte a essa abordagem e que podem ser implementadas em outras linguagens de programação.

5.3.1 Arquitetura

A arquitetura geral básica do arcabouço *ThreadControl*, que pode ser seguida por outras ferramentas que dêem suporte à abordagem *Thread Control for Tests*, está ilustrada na Figura 5.3 e consiste nos seguintes elementos:

1. **Aspectos de monitoração e controle**: Neste componente são definidos pontos de corte referentes a pontos pelos quais passam as threads da aplicação. São também definidos adendos que informam às classes de gerência do estado das threads do sistema a respeito de transições de estado que estão para ocorrer ou que ocorreram, sendo também capazes de barrar algumas transições. Estes aspectos que dão à fachada de serviços do arcabouço a capacidade de oferecer os serviços de espera por determinados estados utilizando para isso o componente de gerência de estados das threads.

2. **Gerência dos estados das threads**: Neste componente estão presentes classes que armazenam estruturas contendo os estados das threads da aplicação sendo monitoradas. Essas estruturas podem ser utilizadas para verificar se estados de interesse foram alcançados.

3. **Configuração de estados de interesse**: Neste componente estão agrupadas classes e interfaces através das quais estados de espera que sejam de interesse para testadores podem ser configurados.

4. **Fachada de serviços para testes de sistemas assíncronos**: Nessa fachada são oferecidas as primitivas de testes disponíveis aos testadores que precisam implementar testes envolvendo operações assíncronas. A fachada se utiliza de classes do componente de configuração de estados de espera para especificar estados de interesse.

Considerando os elementos da arquitetura geral apresentada acima, há algumas classes e aspectos principais para representar esses componentes na versão atual (0.3) do arcabouço *ThreadControl*. Estes estão ilustrados no diagrama apresentado na Figura 5.4 e seu código pode ser visto no Apêndice G desta tese, que apresenta o código do arcabouço.

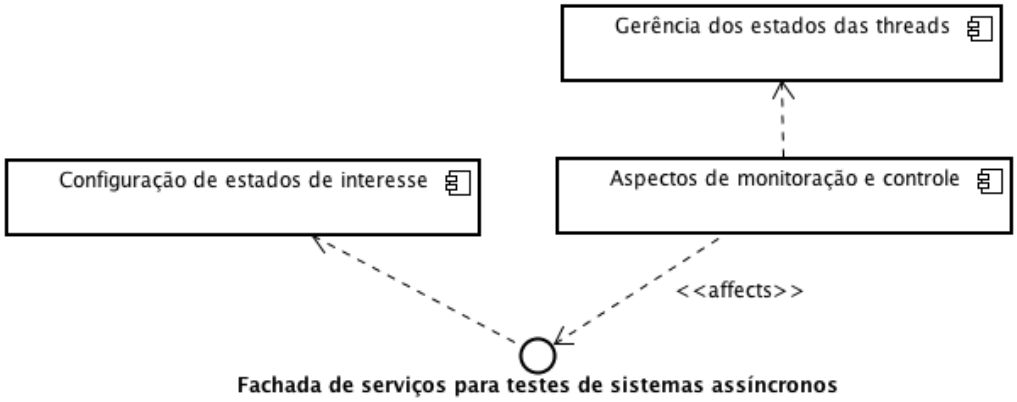


Figura 5.3: Arquitetura geral do arcabouço *ThreadControl*

O aspecto que representa os **aspectos de monitoração e controle** da arquitetura é chamado `ThreadControlAspect`. A classe `ThreadWatcher` é a classe responsável pela **gerência dos estados das threads**. Esta manipula a estrutura que guarda informações sobre o estado das threads e que foi denominada `ThreadManager`. A interface `SystemConfiguration` e suas possíveis implementações representam o componente de **configuração de estados de interesse**. Uma implementação oferecida na versão 0.3 do *ThreadControl* para essa interface é a classe `ListOfThreadConfigurations`. A **fachada de serviços para testes de sistemas assíncronos** é representada na implementação atual pela classe `ThreadControl`, que é interceptada pelo aspecto `ThreadControlAspect` para que possa prover a implementação das primitivas de teste oferecidas aos testadores e que estão detalhadas na seção a seguir.

O código do *ThreadControl* é combinado com o código da aplicação sob teste através do processo denominado *weaving*, explicado na Seção 2.4. Nos testes podem ser utilizadas primitivas oferecidas através da fachada de serviços representada pela classe `ThreadControl` com seus métodos e que vão permitir que o teste espere por determinados estados das threads do sistema (fases de sua execução) antes de fazer asserções. A seção a seguir descreve as primitivas de teste disponibilizadas atualmente pelo arcabouço *ThreadControl* através desta classe.

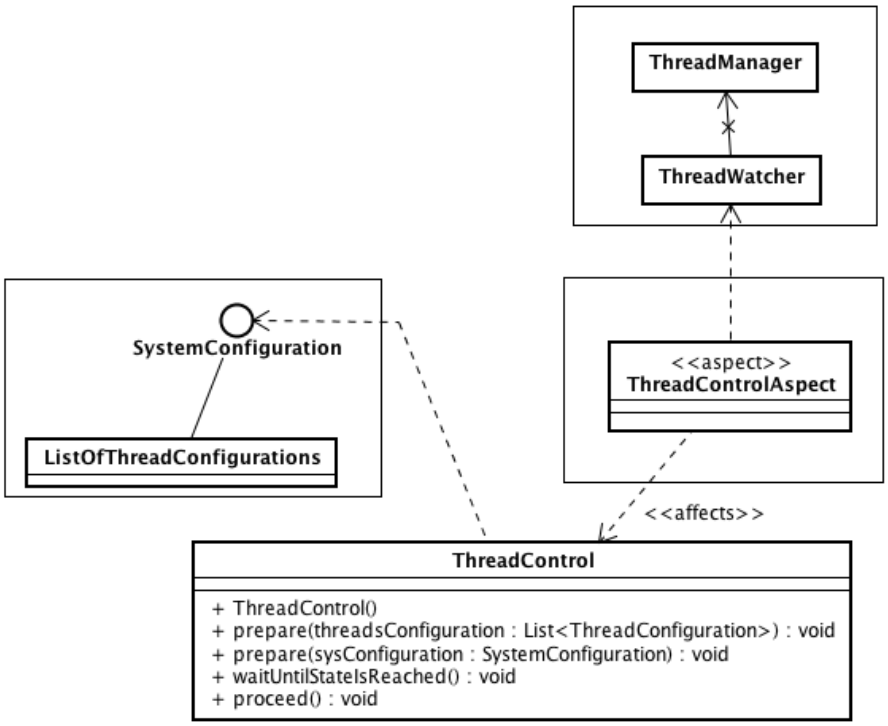


Figura 5.4: Arquitetura do arcabouço *ThreadControl* visualizando suas classes principais

5.3.2 Usando o *ThreadControl*

A idéia geral do uso do arcabouço de testes *ThreadControl* é que as seguintes operações ou primitivas sejam disponibilizadas aos desenvolvedores de testes através de uma fachada, que em seu código corresponde à classe `ThreadControl`:

- `prepare`: é utilizada para especificar configurações desejadas do sistema e pelas quais se deseja esperar antes de executar as asserções. Estas configurações representam fases na execução do sistema e que correspondem a determinados estados de suas threads e são passadas como parâmetro para essa operação;

- `waitUntilStateIsReached`: permite que as asserções sejam executadas em um momento seguro, quando o estado especificado na operação `prepare` tiver sido atingido. Se alguma *thread* tentar perturbar o sistema depois que o estado esperado tiver sido alcançado, ela será bloqueada. Quando essa operação é chamada, a thread do teste espera até que a aplicação tenha atingido o estado esperado e só então realiza as asserções;

- `proceed`: essa é a operação responsável por fazer o sistema prosseguir normalmente sua execução. Quaisquer *threads* que tenham sido bloqueadas por tentarem alterar o estado do sistema depois que o estado esperado tinha sido alcançado serão liberadas neste momento. Dessa forma o teste pode continuar normalmente ou terminar.

Como pode ser visto, essas operações correspondem às fases 1, 3 e 5 da abordagem *Thread Control for Tests*, que foi proposta. O código a seguir ilustra o uso dessas operações em um teste JUnit [5] e considerando o mesmo exemplo de iniciação do sistema que foi discutido no Capítulo 2.

```
1  public void testSystemInitialization(){
2    threadControl.prepare(expectedSystemConfiguration);
3    mySystem.initialize();
4    threadControl.waitUntilStateIsReached();
5    assertTrue(isCorrectlyInitialized(mySystem));
6    threadControl.proceed();
7  }
```

Neste exemplo, o sistema é exercitado através da operação `mySystem.initialize`, que é assíncrona. Para evitar que a asserção da linha 5 seja feita antes que o processo de inicialização tenha sido concluído, bastou incluir uma chamada à operação `waitUntilStateIsReached`, oferecida pelo arcabouço de testes *ThreadControl* (linha 4). No entanto, antes que essa operação possa ser chamada, foi necessária uma invocação à operação `prepare` (linha 2) para que fosse especificado o estado do sistema pelo qual o teste precisa esperar antes das asserções, representado pela variável `expectedSystemConfiguration`. Esta variável é do tipo `SystemConfiguration`, uma interface apresentada na Figura 5.4 e que representa configurações de estados de espera de interesse para os testadores, conforme melhor detalhado a seguir, na Seção 5.3.3. No caso deste exemplo, a variável `expectedSystemConfiguration` representa uma fase da execução da aplicação em que a inicialização do sistema foi concluída e que é mapeada em uma certa configuração de estado para algumas de suas threads principais (ex: Instâncias de `ThreadA` esperando por requisições e instâncias de `Thread B` terminadas).

5.3.3 Especificando Estados de Espera

A forma como se define no arcabouço *ThreadControl* o estado das threads pelo qual se deseja esperar é através de variáveis do tipo `SystemConfiguration`. Na versão atual do arcabouço, para definir uma configuração de threads a se esperar, basta construir uma instância de uma classe que implemente a interface `SystemConfiguration` (como a variável `expectedSystemConfiguration` mostrada no exemplo anterior), como é o caso da classe `ListOfThreadConfigurations`. Essas classes estão representadas no diagrama de classes apresentado na Figura 5.5. A exigência básica de qualquer classe que implemente a interface é a existência de um método através do qual se possa verificar se o estado esperado especificado por essa variável foi atingido (método `wasConfigurationReached`). Tal método recebe como parâmetro um objeto que guarda os estados das threads no momento corrente e que é uma instância da classe `ThreadManager` do arcabouço.

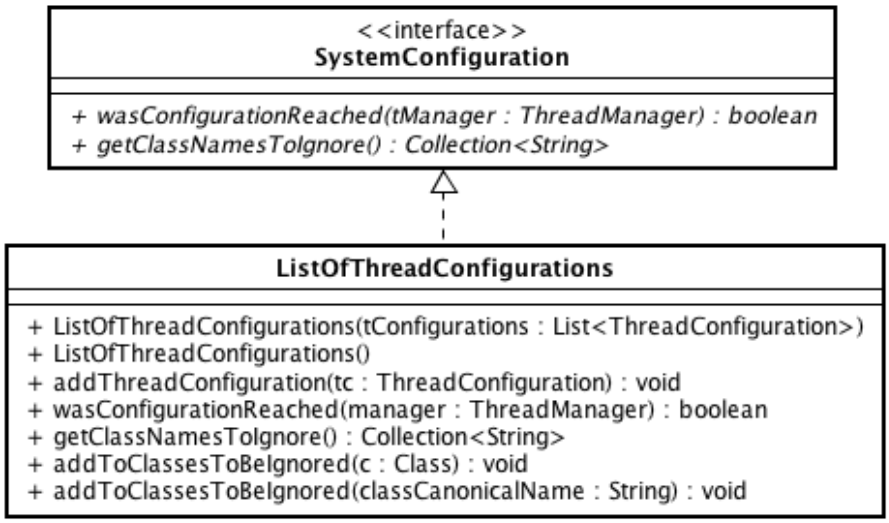


Figura 5.5: Diagrama de classes com operações da interface *SystemConfiguration* e a classe *ListOfThreadConfigurations*, que a implementa

Na versão atual do arcabouço, apenas a classe `ListOfThreadConfigurations` implementa a interface `SystemConfiguration`. Ao construir um objeto desse tipo, o desenvolvedor do teste deve definir uma lista de configurações de threads de interesse. Cada configuração de thread é definida por uma instância da classe `ThreadConfiguration`,

ilustrada no diagrama da Figura 5.6. Um objeto deste tipo pode ser definido de diversas formas, refletidas nos vários construtores desta classe. Os parâmetros básicos para construção de tais objetos são basicamente o nome da classe que identifica uma ou mais threads e um estado esperado, representado por uma instância da classe `ThreadState`. O nome que identifica as threads pode ser o nome de uma classe que estende a classe `Thread` de Java ou ainda o nome de alguma classe que implemente a interface `Runnable`, também da API (*Application Programming Interface*) padrão de Java.

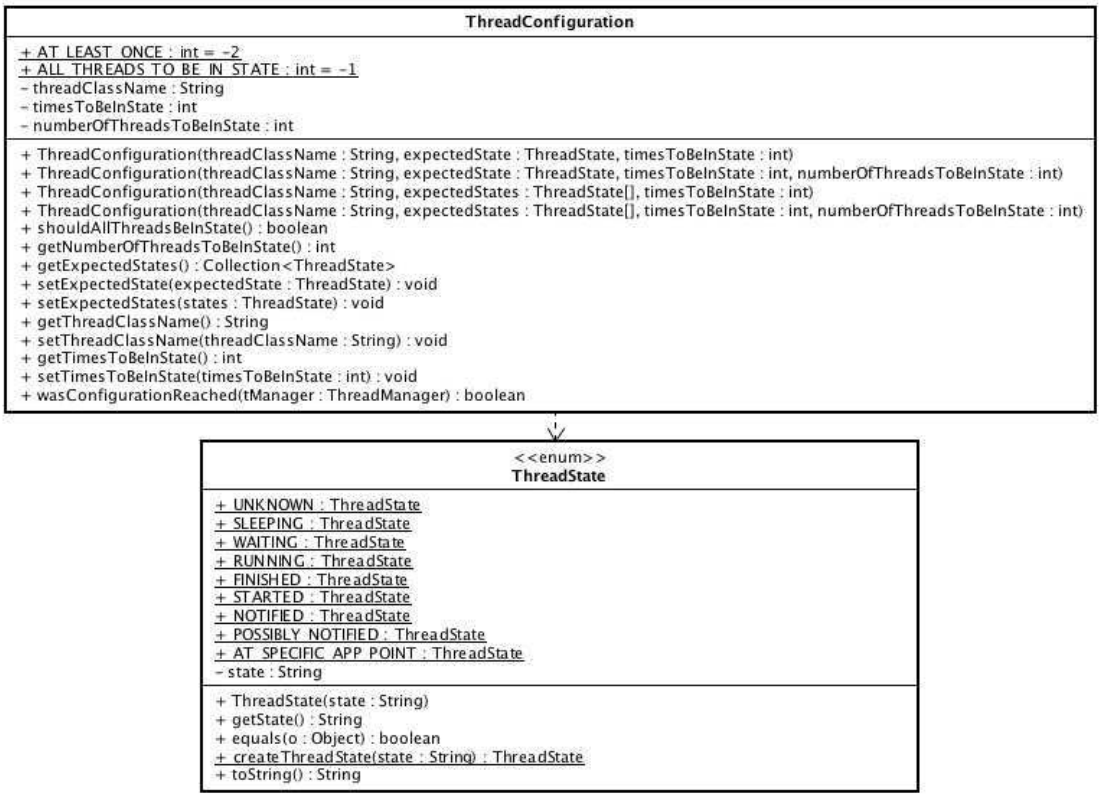


Figura 5.6: Diagrama de classes representando a classe *ThreadConfiguration* e a classe *ThreadState*

Para exemplificar a especificação de um estado esperado das threads do sistema através das classes apresentadas, a seguir é mostrado um exemplo de um método através do qual se pode construir uma variável `expectedSystemConfiguration`:

```
1  public SystemConfiguration getConfigurationToWaitFor() {
2    ListOfThreadConfigurations expectedSystemConfiguration =
3      new ListOfThreadConfigurations();
```

```
4     ThreadConfiguration conf1 = new ThreadConfiguration(
5       "AppMainThread", ThreadState.WAITING,
6       ThreadConfiguration.AT_LEAST_ONCE);
7     expectedSystemConfiguration.addThreadConfiguration(conf1);
8     return expectedSystemConfiguration;
9   }
```

Ao invocar este método, adquire-se um objeto `SystemConfiguration` que define como estado pelo qual esperar o momento em que a thread identificada pela classe `AppMainThread` deve estar esperando (estado `ThreadState.WAITING`). Neste caso, basta que qualquer thread identificada pela classe `AppMainThread` chegue ao estado `ThreadState.WAITING` para caracterizar que o estado esperado foi alcançado. Isto é especificado através do parâmetro `ThreadConfiguration.AT_LEAST_ONCE` passado para o construtor da variável `conf1`.

Porém, pode ser desejável, na prática, esperar até que threads de um certo tipo passem várias vezes por um estado antes de realizar certa asserção. Para diferenciar tal estado, ele pode ser definido pelo estado atual da thread e mais o estado de um contador. Sendo assim, além de identificadores das threads e de seus estados esperados, em uma configuração de threads no *ThreadControl* é possível especificar também um estado esperado em termos do número de vezes que um certo estado foi atingido por threads de um determinado tipo (ex: a quinta vez em que alguma thread que seja uma instância da classe A finalizou sua execução, indo para o estado `FINISHED`). Pode-se ainda dizer que se espera que a thread esteja em um certo estado e que ele tenha sido atingido não importando o número de vezes (`AT_LEAST_ONCE`).

Para exemplificar isso, pode-se tomar como sistema sob teste um middleware de grades computacionais, como o OurGrid. É possível que se queira em um teste submeter cinco tarefas a serem escalonadas na grade e antes de verificar os seus resultados, esperar até que cinco instâncias de uma determinada classe chamada `TaskExecutor` (que representa as threads que distribuem e coletam os resultados das tarefas) tenham sido criadas e terminadas. Pode-se também simplesmente especificar que se quer esperar que todas as threads da classe `TaskExecutor` estejam paradas, não importando a quantidade de vezes em que threads identificadas por esse nome passaram por esse estado naquele teste específico (utilizando

neste caso a constante `AT_LEAST_ONCE`).

Por fim, um ponto importante a acrescentar sobre a especificação de estados de espera em testes é que vários testes de um sistema podem compartilhar um mesmo estado de espera de interesse, sendo portanto importante definir métodos utilitários que podem ser reusados por vários testes em diferentes classes e através dos quais possam ser construídos objetos que retornem configurações que sejam comuns a vários testes. Um exemplo de método de tal tipo é o método `getConfigurationToWaitFor` mostrado anteriormente. Uma outra observação importante é que para definir estados de espera de interesse, é importante contar com a ajuda dos desenvolvedores do sistema sob teste para identificar que estados das threads correspondem a certas fases da execução do sistema pelas quais se deseja esperar nos testes.

5.3.4 Monitorando e Controlando as Threads

Para oferecer as operações descritas na Seção 5.3.2 como primitivas de teste (`prepare`, `waitUntilStateIsReached` e `proceed`), o arcabouço *ThreadControl* precisa monitorar e controlar as threads do sistema sendo testado e cujo código possa ser instrumentado. Para tornar possível a monitoração de forma modularizada de diversos pontos de interesse pelos quais passam essas threads e também para evitar mudanças diretas no código das aplicações sendo testadas, utilizou-se na construção do arcabouço, além de Java, a linguagem AspectJ [53], baseada no paradigma de Programação Orientada a Aspectos (*Aspect-Oriented Programming* - AOP) [54], explicado na Seção 2.4. Este paradigma foi utilizado por objetivar dar suporte à implementação de aspectos cuja implementação atravessa várias partes da aplicação, como é o caso da monitoração de threads.

Considerando AOP e a abordagem *Thread Control for Tests*, foi implementado o aspecto `ThreadControl`, cujo código é mostrado no Apêndice G.2. Neste aspecto foram identificados conjuntos de pontos de execução (*pointcuts*) que representam as mudanças principais no estado do sistema com relação às suas *threads*. Uma vez que tais pontos de execução são atingidos, há adendos (que são basicamente métodos) que atualizam o estado geral do sistema e que evitam certas mudanças de estado (caso asserções estejam em execução). Se um estado esperado, configurado através da chamada à operação `prepare`, for atingido em algum desses pontos, o teste é notificado e suas asserções podem ser executadas sem que mudanças em threads monitoradas aconteçam e perturbem o estado do sistema (ex: sem que

uma thread periódica acorde). Caso alguma thread tente mudar de estado, ela é bloqueada (utilizando a operação `wait` de Java) enquanto as asserções não foram finalizadas. Quando elas são finalizadas, este fato é sinalizado através de uma invocação à primitiva de teste `proceed` logo após as asserções.

Para que o *ThreadControl* saiba para que estados estão passando as threads sendo monitoradas, o aspecto `ThreadControlAspect` define adendos do tipo `before` e `after` que interceptam pontos antes ou depois de algumas chamadas ou execuções de métodos da aplicação representando transições de estado e estes adendos informam ao monitor o estado atual de uma thread passando por tais pontos. Este monitor corresponde à classe que gerencia os estados das threads (`ThreadWatcher`). A Figura 5.7 ilustra a ação de alguns adendos gerais para pontos de execução de Java 1.4 relativos a transições comuns de estado de threads que foram consideradas.

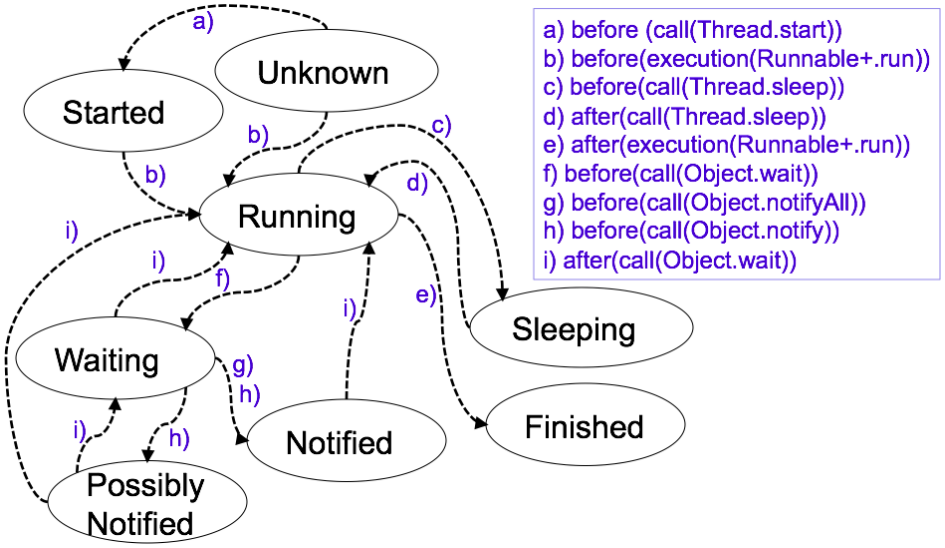


Figura 5.7: Ação de adendos do ThreadControlAspect

Para que a ferramenta *ThreadControl* saiba que uma transição de estado aconteceu, alguns pontos de execução que levam a tais transições tiveram de ser definidos. Na versão inicial da ferramenta, foi incluído suporte às seguintes operações de Java: chamadas a `Thread.start`, execuções do método `run` em classes que implementam a interface `Runnable`, chamadas ao método `Thread.sleep`, chamadas a `Object.wait`, chamadas a `Object.notifyAll` e `Object.notify`. Os estados possíveis para threads considerados

foram: *unknown, started, running, waiting, sleeping, notified, possibly notified* e *finished*. As variáveis representando cada estado no *ThreadControl* correspondem a instâncias da classe `ThreadState`. Fazendo um paralelo com a formalização apresentada no Capítulo 4, a coleção formada por esses estados possíveis corresponde ao conjunto Q.

Algumas outras operações do pacote `java.util.concurrent` (lançado com Java 5) passaram também a ser gerenciadas pelo *ThreadControl*, em sua versão atual, tais como: chamadas aos métodos bloqueantes `take` e `put` de classes que implementam a interface `BlockingQueue` (Fila bloqueante), e chamadas aos métodos `acquire`, `release` e `drainPermits` da classe `Semaphore` (Semáforo). Durante a implementação do suporte a esses métodos observou-se a necessidade de monitorar elementos internos dessas estruturas como, por exemplo, os elementos da fila (*queue*) ou o número de `permits` do semáforo. Dependendo destes elementos, é possível determinar com mais precisão em que estado cada thread está, ou atribuir-lhe como estado um estado que agrega em sua semântica mais de um estado possível, como é o caso do estado *possibly notified*.

É importante destacar que outras operações, além das atualmente suportadas pelo *ThreadControl* podem também ser gerenciadas. Para fazer isso e estender o arcabouço, basta adicionar novos pontos de corte e adendos ao aspecto `ThreadControlAspect`. Os pontos de corte devem corresponder a pontos na execução que levam a transições de estado de thread. Os adendos devem informar a classes auxiliares do arcabouço, como é o caso da classe `ThreadWatcher`, que uma transição de estado aconteceu. Além disso, tais adendos devem fazer verificações para observar se uma determinada transição de estado pode prosseguir ou não (caso em que asserções ainda estão sendo executadas), no intuito de evitar asserções tardias. Para tais verificações, o aspecto utiliza um método chamado `verifyAndBlockThreadIfNecessary`. Este método verifica se algum estado esperado está sendo aguardado, ou seja, se o método `prepare` foi chamado e ainda não houve chamada ao método `proceed`, e vê se esse estado esperado já foi alcançado. Caso tenha sido, a thread que está tentando mudar de estado é bloqueada utilizando a operação `wait` da classe `Object` de Java. Maiores detalhes sobre essa operação podem ser vistos nos Apêndices G.2 e G.8.

Toda a gerência de estados do arcabouço é feita por classes auxiliares, de acordo com notificações enviadas pelo aspecto à medida que verifica que uma determinada thread al-

cança um ponto de execução. O arcabouço não se baseia em informações de estado providas pela operação `Thread.getState` de Java já que essa informação pode não ser confiável. Observou-se em alguns experimentos iniciais que tal informação pode levar um certo tempo para ser atualizada. Tal problema foi também observado por Goetz et al [44], que aponta também que tal operação tem pouca utilidade para testes, embora possa ser usada para depuração. Um outro motivo foi que se considerou importante considerar estados como *notified* e *possibly notified* e também deixar o arcabouço aberto para a definição de novos estados, específicos para a aplicação.

Para ilustrar como o arcabouço pode ser estendido para considerar outras transições, é apresentado abaixo um trecho de código do aspecto `ThreadControlAspect` mostrando um de seus adendos e o ponto de corte referenciado neste adendo:

```
1  after(Object o): waitCalls(o) {
2    verifyAndBlockThreadIfNecessary(thisJoinPoint);
3    threadWatcher.threadFinishedToWaitOnObject(
4      Thread.currentThread(), o, false,
5      WaitType.WAIT_ON_OBJ_LOCK);
6    verifyAndBlockThreadIfNecessary(thisJoinPoint);
7  }
8
9 pointcut waitCalls(Object o):call(public void wait())&& target(o)
10 && excludedEntities();
```

O código deste adendo indica o que deve ser feito depois que a chamada ao método `Object.wait` retorna (ou seja, quando uma thread acorda, deixando de esperar). Em geral, as instâncias de classes auxiliares, como é o caso de `threadWatcher`, devem ser informadas a respeito da transição, como está ilustrado nas linhas 3-5, através da chamada ao método `threadFinishedToWaitOnObject`. No entanto, antes de informar sobre a transição de estado e depois da notificação, o arcabouço deve verificar se a transição de estado pode prosseguir ou não (linhas 2 e 6), chamando o método `verifyAndBlockThreadIfNecessary`. Se não puder, a thread corrente será bloqueada até que uma chamada à primitiva de teste `proceed` seja invocada pela thread do teste. É importante observar que operações para indicar à gerên-

cia de estados das threads (classe `ThreadWatcher`) do arcabouço as mudanças de estado (como o método `threadFinishedToWaitOnObject`) devem ser sincronizadas (`synchronized`) para evitar condições de corrida.

Como se pôde observar, a ferramenta de teste apresentada dá suporte à abordagem previamente proposta. No entanto, esta ferramenta não constitui a única implementação possível. Soluções baseadas em padrões de projeto são também possíveis. Optou-se por utilizar Programação Orientada a Aspectos como forma de evitar muitas mudanças de código na aplicação sob teste e como forma de tornar a ferramenta aplicável a diferentes sistemas sem demandar destes mudanças em seu código. Usando aspectos, é possível monitorar vários pontos de execução de forma transparente e modularizada. Se for necessário remover o código de monitoração utilizado para propósitos de teste, basta compilar a aplicação sem o aspecto que introduz tal funcionalidade.

A idéia básica é que para usar o *ThreadControl* em testes, basta implementar os testes utilizando as primitivas de teste oferecidas pela fachada do arcabouço e antes de executar esses testes, assegurar-se de que o código base da aplicação e seus testes são combinados com os aspectos e classes auxiliares do *ThreadControl* através do processo de *weaving* usando o combinador **ajc**, gerando um bytecode a ser executado nos testes que incorpora o código adicionado pelos adendos relativo à monitoração e controle das threads.

5.4 Conclusões Parciais

Neste capítulo foram apresentados a abordagem *Thread Control for Tests* e o arcabouço *ThreadControl*, que lhe dá suporte. A proposta inicial dessa abordagem e a primeira versão do arcabouço contemplando apenas operações básicas sobre *threads* da versão 1.4 de Java e uma avaliação inicial em que foram implementados testes baseados em casos de teste reais foram também publicados nos anais da primeira edição do ICST [24]. A versão mais recente da abordagem e do arcabouço estendido para Java 5 e os resultados da avaliação inicial considerando os testes de um sistema real, o *OurBackup*, gerou como resultado o artigo publicado no Simpósio Brasileiro de Engenharia de Software [26].

Embora o foco desta tese seja na abordagem, procurou-se apresentar neste capítulo detalhes sobre o *ThreadControl* como forma de divulgar formas de implementar ferramentas

que dêem suporte à abordagem que foi apresentada e para dar suporte a futuras extensões do arcabouço.

Com relação a abordagem em si, pode-se observar que é possível fazer um mapeamento entre ela e o modelo de solução baseada em monitores apresentado na Seção 4.3.

Retomando o que foi apresentado na Seção 4.3, uma possível solução para o problema das asserções feitas em momentos inadequados seria uma solução baseada em **monitores**, que correspondem a *entidades adicionadas ao sistema e que têm a capacidade de alterar o escalonamento das threads da aplicação* ($\tau \in B$) *ou de controle* (τ_0) *adicionando novos eventos de* nop *ao escalonamento*. Considerando na execução dos testes a presença de monitores, teve-se a definição de uma execução de teste monitorada pela tupla: $\mathcal{R}^{\mathcal{S}^{mon}} = (\mathcal{R}^{\mathcal{S}}, \mathcal{D}, IR)$. Nesta tupla, o $\mathcal{D}$ corresponde ao conjunto de restrições quanto ao estado esperado para as threads no momento de asserções da execução de teste $\mathcal{R}^{\mathcal{S}}$ e IR corresponde ao conjunto contendo as regras de implementação ($IR1$, $IR2$ e $IR3$) que garantirão que as restrições referentes ao estado esperado para as threads serão respeitadas.

Considerando o que foi apresentado neste capítulo a respeito da abordagem *Thread Control for Tests*, descrita na Seção 5.2, foi feito um mapeamento desta para a solução modelada e que está descrito na Tabela 5.1.

Tal mapeamento serve apenas para dar indícios de que a abordagem *Thread Control for Tests* consegue evitar o problema das asserções antecipadas ou tardias. Porém, outras avaliações foram feitas, como os estudos de caso apresentados no capítulo a seguir e a avaliação da abordagem utilizando TLA+ e que está apresentada no Capítulo 7.

Tabela 5.1: Mapeamento entre a descrição da abordagem *Thread Control for Tests* e o modelo de solução baseada em monitores proposto na Seção 4.3

Descrição da abordagem *Thread Control for Tests*	**Elemento correspondente na modelagem**
"*A abordagem se baseia na monitoração e no controle das threads do sistema sendo testado*."	monitores sobre as threads $\tau \in B \cup \{\tau_0\}$
Fase de **preparação do ambiente** (1), indicando o estado esperado para asserções	Especificação das restrições $\mathcal{D}$ em $\mathcal{R}^{\mathcal{S}^{mon}}$
Fase de **espera** (3) até que o estado esperado seja alcançado	IR3, que não permite asserções antes desse momento
Fase de **execução das asserções** (4)	IR1, que permite o escalonamento dos eventos de asserção no momento certo
Execução das asserções(4) sem temer que alguma thread desperte e afete os resultados de teste	Garantia da IR2, que evita mudanças no estado das threads enquanto asserções estão sendo feitas.

Capítulo 6

Avaliação Inicial da Abordagem *Thread Control for Tests*

Embora se tenha dado indícios de que a abordagem proposta resolve os problemas a que se propõe mapeando-a para o modelo formal descrito anteriormente, optou-se por acrescentar à avaliação deste trabalho também estudos de caso práticos pelo fato destes proporcionarem o casamento das idéias aqui propostas com a realidade [52].

O objetivo geral da avaliação inicial da abordagem *Thread Control for Tests* apresentada neste capítulo é utilizar estudos de caso para investigar seu uso na prática e verificar se ela consegue de fato evitar o problema das asserções feitas cedo ou tarde demais e que levam os testes a falharem desnecessariamente. Um outro objetivo também considerado foi comparar essa abordagem com abordagens alternativas de uso comum como atrasos explícitos e espera ocupada.

Para esta avaliação, foram planejados dois estudos de caso, um utilizando casos de teste feitos para o estudo e reproduzindo casos semelhantes aos que ocorrem na realidade, apresentado na Seção 6.1 e um outro estudo de caso, apresentado na Seção 6.2, utilizando casos de teste de um sistema real, o *OurBackup* [75], uma solução entre pares (P2P) de *backup* baseada em redes sociais. Os resultados de versões iniciais desses estudos de caso foram apresentados no ICST [24] e no SBES 2008 [26] respectivamente.

6.1 Estudo 1: Estudo de Caso sobre o Uso da Abordagem *Thread Control for Tests* em Casos de Teste Sintéticos, mas Reproduzindo Casos de Teste Reais

Nesta seção é apresentado o planejamento do primeiro estudo piloto realizado, sua execução e seus resultados.

6.1.1 Planejamento do Estudo de Caso

Para o primeiro estudo de caso realizado, foi elaborado um plano descrito nas subseções a seguir, como forma de melhor descrever os procedimentos básicos utilizados. A forma de documentação deste plano seguiu algumas das sugestões dadas por Jedlitschka e Pfahl [50] e Travassos et al. [97] e a terminologia de experimentos comumente usada na literatura [78] [55].

De maneira geral, este estudo de caso se baseia na comparação entre o uso do arcabouço de testes *ThreadControl* e o uso de outras abordagens e buscou identificar se falhas devidas a asserções feitas em momentos inadequados deixariam de ocorrer usando-se a abordagem *Thread Control for Tests*. Também são avaliadas as falhas que poderiam acontecer utilizando abordagens alternativas como atrasos explícitos e espera ocupada.

Esse primeiro estudo foi na verdade um estudo piloto e utilizou testes sintéticos que foram desenvolvidos especialmente para a avaliação.

Objetivo do Estudo

Seguindo uma convenção apresentada em [97] de acordo com a abordagem Objetivo/Questão/Métrica (*Goal/Question/Metric*) [9], o objetivo do estudo de caso apresentado nessa seção pode ser descrito da seguinte forma:

- *Analisar diferentes técnicas para espera (atrasos explícitos, espera ocupada e uso da abordagem Thread Control for Tests) em testes assíncronos com o propósito de avaliar o uso de controle e monitoração de threads por meio do arcabouço ThreadControl com respeito a conseguir evitar a ocorrência de falhas em testes devidas a asserções*

feitas cedo ou tarde demais do ponto de vista de quem executa testes automáticos e no contexto de testes que exercitam operações assíncronas.

Hipóteses

Observando-se a declaração de objetivos apresentada anteriormente, vê-se que essa declaração é focada no aspecto: ocorrência de falhas incorretas em testes (1).

Considerando o aspecto ocorrência de falhas, embora ele já fosse esperado nas versões de testes com atrasos explícitos e esperas ocupadas, concluiu-se que seria importante avaliar através de estudo de caso prático se estas ocorreriam na versão de testes utilizando monitoração e controle de threads, já que tal estudo também serviria para demonstrar o uso da abordagem aqui proposta em casos em que o problema de asserções antecipadas e tardias acontece na prática.

Observando os aspectos de ocorrência de falhas incorretas, foi formulada a seguinte hipótese nula (H_{01}) e hipótese alternativa (H_{11}) correspondentes ao objetivo apresentado:

- H_{01} Não há diferença em termos de quantidade de falhas incorretas em testes por asserções feitas em momentos inadequados entre testes usando monitoração e controle de threads através do *ThreadControl* e testes utilizando outras abordagens populares como atrasos explícitos e espera ocupada.

- H_{11} É menor (e preferencialmente nula) a quantidade de falhas incorretas em testes por asserções feitas em momentos inadequados em testes usando *ThreadControl* do que em testes que usam abordagens baseadas em atrasos explícitos e espera ocupada.

Variáveis

Há dois tipos de variáveis a serem descritas em um plano de experimentos [50]: as variáveis de resposta ou variáveis dependentes e os fatores ou variáveis independentes.

Para este estudo de caso, será utilizada a seguinte **variável de resposta**:

- Número de testes que falham para um certo número de re-execuções de testes; e

Quanto aos **fatores** (variáveis independentes), será variado apenas um, que é a **técnica de espera utilizada** nos testes com operações assíncronas. Os **tratamentos** (alternativas) para este fator que serão considerados são os seguintes:

- Uso da abordagem *Thread Control for Tests* através do arcabouço de testes *ThreadControl*;
- Uso de atrasos explícitos com diferentes valores fixos para os intervalos de espera;
- Uso da abordagem de espera ocupada;

Para os tratamentos utilizando atrasos explícitos devem ser implementadas diferentes versões variando o intervalo de espera, sendo ao menos um dos valores correspondente ao tempo médio de espera obtido ao utilizar a versão dos testes com *ThreadControl*. Isso foi feito apenas para se ter uma base mais justa de comparação e não utilizar esperas pequenas demais que provocariam maiores índices de falhas em testes por asserções antecipadas.

Sujeitos

Os sujeitos de um experimento de software são os indivíduos que utilizam um certo método ou ferramenta [55].

No caso do primeiro estudo de caso, o único sujeito foi Ayla Dantas (autora deste trabalho de doutorado), responsável por criar as diferentes versões do teste sintético para os diferentes tratamentos.

Objetos

Os objetos de um experimento são os aspectos ou características que podem impactar nos resultados de um estudo, podendo ser programas ou algoritmos sobre os quais um método ou ferramenta são aplicados [55].

No caso do estudo de caso 1, os objetos utilizados foram os **testes implementados** para o estudo. No caso foram 2 testes, sendo cada um deles utilizado em um cenário diferente do estudo de caso, como será descrito na Seção 6.1.2.

Instrumentação

A instrumentação do estudo de caso para colher os valores das variáveis de resposta foi feita através de scripts que varriam os relatórios de teste gerados pelo JUnit observando o sucesso ou falha na execução de cada teste.

6.1.2 Execução e Resultados do Primeiro Estudo de Caso

Para avaliar a ocorrência de falsos alarmes nos testes neste primeiro estudo piloto, foram criados dois casos de teste sintéticos utilizando o JUnit Framework [5] para representar dois cenários. Esses testes foram implementados baseando-se em cenários de teste comuns observados em dois sistemas desenvolvidos no LSD: o OurGrid [16] (um middleware entre pares de código aberto para computação em grade) e o OurBackup [75]. Três versões desses testes foram desenvolvidas: uma delas utilizando atrasos explícitos, uma outra utilizando espera ocupada e uma terceira utilizando a abordagem proposta neste trabalho através da ferramenta *ThreadControl*. Os dois testes exploram operações assíncronas simples e apenas o segundo explora uma situação em que pode ocorrer o problema de se poder passar do ponto ideal para verificação (*miss of opportunity to fire*). Os testes implementados foram simples e sua execução só deve falhar caso haja algum problema de asserção feita cedo ou tarde demais. Cada versão do teste foi executada 1000 vezes com o intuito de verificar:

- frequência de testes que não passam.

Cenário 1

O primeiro teste é bem simples, de forma a tornar o cenário simples de entender. Ele inicia uma thread cuja tarefa principal é simplesmente esperar um certo tempo e posteriormente, alterar para verdadeiro (`true`) o valor de uma *flag* que indica que a thread executou. A função do teste que inicia esta thread é verificar se o valor da *flag* é `true`. Em um sistema real, tal thread poderia criar várias outras que produziriam o resultado a ser verificado via asserções. Mas considerando-se apenas esta thread, utilizou-se como tempo de espera antes de configurar a *flag* para `true` o retorno de uma operação que retorna um valor de acordo com uma distribuição exponencial de parâmetro 0.6.

Usando atrasos explícitos, o código para o teste JUnit é o seguinte:

```
1  public void testSimpleExecution()
2    throws InterruptedException {
3      MonitorableThread mt = new MonitorableThread();
4      assertFalse(mt.hasExecuted);
5      mt.start();
```

```
6       Thread.sleep(TIME_TO_SLEEP);
7       assertTrue(mt.hasExecuted);
8     }
```

O valor da variável `TIME_TO_SLEEP` é dado pelos desenvolvedores considerando-se um tempo avaliado como suficiente para que o teste não falhe. Esse tempo também não pode ser muito grande para que se evite que o teste leve mais tempo que o necessário para executar, o que atrapalha o desenvolvimento. Testes que levam muito tempo para executar podem tanto atrasar o desenvolvimento se precisarem ser executados com frequência, como podem também sofrer o risco de serem executados raramente, aumentando o tempo entre o momento em que um defeito é introduzido e o momento em que este é identificado por um teste [66].

Utilizando a abordagem de espera ocupada, a linha 6 do código acima poderia ser substituída por:

```
while (!mt.hasExecuted){
  Thread.sleep(TIME_TO_SLEEP);
}
```

Utilizando a abordagem *Thread Control for Tests* através do *ThreadControl*, seria necessária uma fase antes da linha 5 para invocar a operação `prepare` usando-se uma configuração esperada de threads especificando que a thread identificada pela classe `MonitorableThread` deverá estar no estado *FINISHED* antes das asserções (ver código a seguir). Então, ter-se-ia a chamada à operação `waitUntilStateIsReached` do *ThreadControl* ao invés de uma invocação a `Thread.sleep`. Após a asserção final, seria necessário incluir uma chamada ao método `proceed` do *ThreadControl* só para informá-lo que as asserções foram concluídas. O código abaixo ilustra tais modificações no método `testSimpleExecution` já considerando a versão atual do *ThreadControl*.

```
1  public void testSimpleExecution()
2    throws InterruptedException {
3    MonitorableThread mt = new MonitorableThread();
4    assertFalse(mt.hasExecuted);
```

```
5     threadControl.prepare(getThreadConfigurationToWaitFor);
6     mt.start();
7     threadControl.waitUntilStateIsReached();
8     assertTrue(mt.hasExecuted);
9     threadControl.proceed();
10 }

11 public SystemConfiguration getConfigurationToWaitFor() {
12   ListOfThreadConfigurations expectedSystemConfiguration =
13     new ListOfThreadConfigurations();
14   ThreadConfiguration conf1 = new ThreadConfiguration(
15     "MonitorableThread", ThreadState.FINISHED,
16     ThreadConfiguration.AT_LEAST_ONCE);
17   expectedSystemConfiguration.addThreadConfiguration(conf1);
18   return expectedSystemConfiguration;
19 }
```

Após as 1000 execuções deste primeiro teste para cada versão, em uma mesma máquina dedicada ao estudo de caso, percebeu-se que nenhuma das execuções dos testes na versão com *ThreadControl* apresentou falhas.

Para o primeiro teste, observou-se que as falhas só ocorreram na versão em que atrasos explícitos foram usados. Esta versão foi implementada de forma parametrizada, com diferentes valores para o intervalo de espera (*t*, correspondente ao `TIME_TO_SLEEP`) para variações dessa versão. A frequência com que falhas ocorriam em tal versão dependia do valor do intervalo de espera *t* escolhido para se esperar. Por exemplo, utilizando 80ms, que foi o tempo médio de espera obtido para a versão *ThreadControl*, a frequência de falhas nos testes era de 99.7%, enquanto que aumentando este intervalo para 83ms, a frequência de falhas caia para 3.5% das execuções (veja a Tabela 6.1). No entanto, aumentando o intervalo de espera utilizado, gera-se um aumento no tempo de execução dos testes.

Tabela 6.1: Frequência de falhas para o cenário 1

Abordagem	**Frequência de falhas**
ThreadControl	0%
Atrasos explícitos (t=80)	99.7%
Atrasos explícitos (t=83)	3.5%
Atrasos explícitos (t=100)	2.6%
Atrasos explícitos (t=200)	0%
Espera Ocupada (t=1)	0%
Espera Ocupada (t=5)	0%
Espera Ocupada (t=80)	0%

Cenário 2

Como se pôde observar pela Tabela 6.1, não houve falhas nas execuções utilizando espera ocupada. Esta versão não apresenta falhas pois ela só deixa o laço quando a condição sendo testada também na asserção é verdadeira (mas nem sempre a realidade consiste em se utilizar como guarda da espera ocupada a mesma condição da asserção, para que se evite um laço infinito quando o teste deveria falhar, na verdade). Para ilustrar uma situação em que falhas acontecem com espera ocupada, foi implementado um teste similar ao do cenário 1 e que é detalhado a seguir.

O segundo teste representa um cenário em que se pode passar do ponto ideal para realizar a asserção. Sua diferença quanto ao código do cenário 1 é que foi adicionada uma nova asserção. Além disso, foi também alterado o código da thread utilizada no teste, a `MonitorableThread`. Foi incluído em seu método `run` um laço de 1 a 4 e que incrementa um contador e espera (usando um `wait` temporizado) por um intervalo pequeno (obtido também através de uma distribuição exponencial de parâmetro 0.6) após cada incremento. Espera-se que a asserção seja feita assim que a thread `MonitorableThread` começa a dormir. A asserção incluída no teste foi a seguinte: `assertEquals(1, mt.getCounterValue())`.

Esse teste representa um cenário comum encontrado em testes onde existem threads pe-

riódicas cujo comportamento se quer verificar no momento exato em que uma thread deste tipo executou e começou a dormir ou a esperar em uma tranca.

Para obter a frequência de falhas para as diferentes versões implementadas para os testes, foi seguida uma abordagem semelhante a que foi seguida para o primeiro cenário. Os resultados obtidos para 1000 execuções das diferentes versões estão mostrados na Tabela 6.2. O tempo médio de espera para o ThreadControl que foi calculado foi de 83ms e por isso ele foi utilizado como intervalo de espera na versão com atrasos explícitos e na versão com espera ocupada.

Tabela 6.2: Probabilidade de falhas e tempo médio de espera para cada versão ao executar o cenário 2

Abordagem	Frequência de Falhas
ThreadControl	0%
Atrasos explícitos (t=83)	16.6%
Espera Ocupada(t=83)	5.3%
Espera Ocupada (t=1)	0.1%

Utilizando a ferramenta *ThreadControl*, pode-se especificar que se quer fazer as asserções assim que a thread monitorada começa a esperar. Além de evitar que a asserção seja feita antecipadamente, pode-se também evitar que qualquer thread mude o seu estado enquanto asserções estão sendo feitas, inclusive a própria `MonitoredThread`. Dessa forma, evita-se falhas no teste devidas a asserções feitas também tarde demais. Ao usar a abordagem de atrasos explícitos com um tempo de espera de 83 ms, há ainda uma probabilidade de falha nos testes de 16.6%. Usando a abordagem com espera ocupada, observou-se neste cenário a ocorrência de falhas nas execuções feitas. Porém, a frequência dessas falhas se reduzia à medida que menores intervalos de espera eram utilizados no interior do laço da espera ocupada. Por exemplo, ao utilizar um intervalo de espera *t* de 83ms, ocorreram falhas em 5.3% das execuções de teste. Ao reduzir esse intervalo de espera para 1ms, a frequência de falhas caiu para 0.1% das execuções. A probabilidade de ocorrência destas decrescia à medida que o intervalo usado na chamada a `wait` dentro do laço era diminuído. É importante destacar que embora se tenha conseguido diminuir o percentual de falhas para 0.1% ao diminuir o tempo

de espera, até mesmo o valor 0.1% é indesejável já que a ocorrência de falhas incorretas, por mais rara que seja, leva à diminuição da confiança nos testes por parte dos desenvolvedores.

De maneira geral, os resultados dessa avaliação inicial nos dois cenários mostraram que, pelo menos até onde se analisou, o uso do *ThreadControl* conseguiu evitar falhas em testes que ocorreriam se abordagens de espera como espera ocupada e atraso explícito fossem utilizadas.

6.2 Estudo 2: Casos de Teste com Testes Reais do OurBackup

Com um intuito de investigar o uso do *ThreadControl* em testes existentes, buscou-se observar em alguns sistemas do LSD o problema das falhas de testes cujas causas não eram defeitos com o software, mas problemas com o teste. Nesta seção é descrito um outro estudo de caso que foi feito e que utilizou casos de teste já existentes do sistema de backup OurBackup [75]. O objetivo deste estudo foi avaliar o uso da abordagem *Thread Control for Tests* em um sistema real e compará-lo com o uso de abordagens alternativas comumente utilizadas, inclusive a abordagem utilizada pelos desenvolvedores anteriormente ao estudo de caso.

O sistema OurBackup foi escolhido pois ele apresenta diversos testes que exercitam operações assíncronas. Além disso, também se observou que ao re-executar os testes deste sistema por diversas vezes, ocorriam falhas em algumas dessas execuções, sendo algumas delas devidas ao problema das asserções feitas em momentos inadequados.

A abordagem que vinha sendo usada pelos testes do OurBackup para esperar antes de fazer as asserções era baseada em uma combinação de espera ocupada e atrasos explícitos e ela é referenciada neste documento como *CurrentVersion* ou simplesmente *Current*. A condição da guarda da espera ocupada se baseava na operação `Thread.getState` de Java, a qual não é sempre confiável [44], como foi previamente discutido. Por isso, nessa versão, em alguns dos testes, após a espera ocupada havia também o uso de atrasos explícitos. Uma outra característica da abordagem que vinha sendo usada é que algumas convenções de código deveriam ser seguidas pelos desenvolvedores ao criar suas threads. Por exemplo, as threads a serem monitoradas deveriam ser criadas através da classe `MonitoredThreadFactory`,

a qual fabrica threads do tipo `MonitoredThread`.

Como seguir certas convenções não é sempre possível para sistemas existentes, alguns desenvolvedores tendem a usar, em situações similares, a abordagem de atrasos explícitos [44] [66]. Analisando isso, decidiu-se neste segundo estudo de caso comparar três versões de alguns casos de teste do sistema:

- *CurrentVersion*: A versão corrente dos testes, que utiliza como abordagem de espera uma combinação de espera ocupada e atrasos explícitos através de classes utilitárias de teste que se utilizam da operação `Thread.getState` de Java para fazer o teste esperar;

- *ExplicitDelaysVersion(delayTime) ou ED(delayTime)*: Uma versão dos testes usando atrasos explícitos. Esta versão foi configurada com diferentes intervalos de espera já que os valores desses intervalos influem na frequência de falhas causadas por esperas de tempo insuficiente;

- *ThreadControlVersion ou TC*: Uma versão dos testes usando o arcabouço de testes *ThreadControl*.

Para a versão *ThreadControlVersion*, a condição de espera utilizada em alguns testes era o momento em que threads da classe `BackupExecutorImpl` estavam terminadas e threads das classes `BackupSchedulerImpl` e `BackupExecutorThread` estavam esperando. Em outros testes, a condição de espera era o momento em que threads das classes `FakeBackupExecutor` e `BackupSchedulerImpl` estavam esperando. Para a definição desses estados de espera, correspondentes a fases de execução do sistema, os desenvolvedores foram consultados.

Para escolher os casos de teste a utilizar, desenvolvedores do OurBackup também foram entrevistados para identificar alguns testes que apresentavam operações assíncronas e que poderiam ser problemáticos, tendo sido escolhidos 12 casos de teste.

Após implementar as três versões base (*CurrentVersion*, *ExplicitDelaysVersion* e *ThreadControlVersion*), esses testes foram re-executados 10.000 (dez mil) vezes. A seguir será descrito em maiores detalhes o planejamento desse segundo estudo de caso, sua execução e seus resultados.

Um importante ponto a observar é que em um estudo piloto antes deste segundo estudo de caso, e publicado no SBES 2008 [26], identificou-se falhas em execuções de teste do OurBackup mesmo com o *ThreadControl*. Porém, ao depurar essas falhas viu-se que elas se deviam a um defeito conhecido (*known bug*) no sistema associado ao uso de uma biblioteca externa. Para isso, neste segundo estudo de caso, buscou-se também classificar as falhas em testes identificando as que caracterizavam asserções antecipadas e tardias e as que caracterizavam esse defeito conhecido, como será melhor detalhado adiante ao detalhar o procedimento de análise utilizado.

6.2.1 Planejamento do Segundo Estudo de Caso

Objetivo do Estudo

O objetivo do estudo de caso apresentado nessa seção pode ser descrito da seguinte forma:

- *Analisar diferentes técnicas para espera (atrasos explícitos, abordagem corrente utilizada no sistema OurBackup e abordagem Thread Control for Tests) em testes assíncronos do sistema OurBackup com o propósito de avaliar o uso de controle e monitoração de threads por meio do arcabouço ThreadControl com respeito a conseguir evitar a ocorrência de falhas em testes devidas a asserções feitas cedo ou tarde demais do ponto de vista de quem executa testes automáticos e no contexto de testes que exercitam operações assíncronas.*

Hipóteses

As hipóteses consideradas nesse estudo são semelhantes a do estudo anterior, podendo ser descritas da seguinte forma:

- H_{01} Não há diferença em termos de quantidade de falhas incorretas em testes por asserções feitas em momentos inadequados entre testes usando monitoração e controle de threads através do *ThreadControl* e testes utilizando atrasos explícitos e a abordagem corrente de espera utilizada nos testes do sistema OurBackup.

- H_{11} É menor (e preferencialmente nula) a quantidade de falhas incorretas em testes por asserções feitas em momentos inadequados em testes usando *ThreadControl* do

que em testes que usam como abordagem os atrasos explícitos ou a versão corrente de espera utilizada nos testes do sistema OurBackup.

Variáveis

Para o segundo estudo de caso aqui apresentado, será utilizada a seguinte **variável de resposta**:

- Número de testes que falham para um certo número de re-execuções de testes.

Os tratamentos utilizados serão os seguintes:

- Uso da abordagem *Thread Control for Tests* através do arcabouço de testes *ThreadControl*;
- Uso da abordagem atualmente utilizada nos testes do OurBackup (*status quo*, chamada de *CurrentVersion*) e que mescla espera ocupada e atrasos explícitos em cada caso de teste;
- Uso de atrasos explícitos (*Explicit Delays* ou ED) com diferentes valores fixos para os intervalos de espera.

Para os tratamentos baseados em atrasos explícitos serão implementadas 3 versões, cujos intervalos de espera corresponderão a 80%, 100% e 120% dos tempos médios de espera utilizados pela versão com *ThreadControl*. Isso foi feito apenas para tornar mais justa a comparação do número de falhas entre as versões já que o uso de um tempo muito inferior ocasionaria uma frequência maior de falhas nessa versão. Para a versão *CurrentVersion*, que também se utiliza de espera ocupada em alguns de seus testes, foram utilizados os mesmos tempos de espera escolhidos pelos desenvolvedores.

Sujeitos

Os sujeitos de um experimento de software são os indivíduos que utilizam um certo método ou ferramenta [55].

No caso desse segundo estudo de caso, os sujeitos principais são Ayla Dantas e Matheus Gaudêncio (aluno de graduação na época), que trabalharam na produção de cada uma das

diferentes versões dos testes correspondentes aos tratamentos do estudo de caso referente ao uso da ferramenta *ThreadControl* e também os correspondentes a variações de uma versão utilizando atrasos explícitos. Uma característica importante desses sujeitos é que eles não participaram do desenvolvimento do sistema OurBackup. No entanto, membros do time de desenvolvimento foram consultores destes na definição de estados de espera desejáveis para os testes em termos das threads do sistema, sendo portanto também sujeitos do estudo de caso. É importante destacar que estes membros foram os responsáveis pela versão de testes já existente (*CurrentVersion*) (o *status quo*).

Objetos

No caso do estudo de caso 2, os objetos utilizados foram 12 dos **testes existentes no OurBackup**. Estes testes foram selecionados através de entrevistas com desenvolvedores sobre as principais classes de testes com assincronia que apresentavam problemas de falhas por asserções feitas em momentos não apropriados.

Instrumentação

A instrumentação do estudo de caso para colher os valores das variáveis de resposta foi feita através de scripts que varriam os relatórios de teste gerados pelo JUnit observando o sucesso ou falha na execução de cada teste.

Procedimento de coleta dos dados

Para cada um dos diferentes tratamentos (diferentes versões dos testes), os 12 testes deverão ser executados mais de 10.000 vezes observando-se a ocorrência de falhas e os tipos de falhas encontradas, classificando-as entre falhas de fato causadas por problemas de tempo (asserções antecipadas e tardias) e falhas devidas ao problema já conhecido (*known bug*) em uma biblioteca utilizada pelo OurBackup. Essa classificação foi automática baseando-se nas mensagens de erro dos logs de testes, como detalhado a seguir.

Procedimento de análise

Na análise será avaliada a hipótese nula referente ao número de falhas em testes. Considerando tal hipótese, foi feita uma comparação entre o valor encontrado para o número de falhas, para a mesma quantidade de execuções em cada variação dos testes. Porém, como em um estudo piloto anterior foi encontrada na versão de teste utilizando o ThreadControl um teste que falhou e identificou-se que tal falha era devida a um problema conhecido em uma das bibliotecas utilizadas pelo sistema, nesse novo estudo de caso as falhas foram classificadas através das mensagens de erro dos *logs* de execuções de testes para identificar quais falhas em testes estavam mais provavelmente ligadas a asserções antecipadas e tardias (**falhas por asserção antecipada ou tardia**) e quais estavam relacionados ao *known bug* (**falhas por defeitos da aplicação já conhecidos**). Como eram muito variadas as mensagens de erro e como não se conhecia todos os possíveis defeitos da aplicação que é relativamente complexa, houve casos em que não se pôde classificar as falhas nesses dois grupos e estas foram classificadas no grupo de **outras falhas**.

A análise feita buscou identificar dentre as falhas encontradas nos testes, quantas estavam em cada grupo. O objetivo principal era identificar se com a versão de testes usando `ThreadControl` havia alguma falha por asserção antecipada ou tardia e comparar tal número com a quantidade de falhas de tal tipo nas execuções utilizando as outras versões de testes.

Avaliação da Validade

Para aumentar a confiabilidade das medidas feitas, a máquina em que são executados os testes é a mesma para cada um dos diferentes tratamentos. Além disso, essa máquina ficou inacessível pela rede enquanto estava dedicada ao estudo de caso.

Configurações do Ambiente

A Tabela 6.3 mostra as configurações de hardware e software da máquina sendo utilizada no estudo de caso.

Tabela 6.3: Configuração do Ambiente para execução do estudo de caso

Versão do OurBackup	Versão corrente do repositório OurBackup em 4/4/2008
Versão de Java	Java 1.6.0_06
Sistema Operacional	Linux 2.6.24-1-686 (Apr 19) libc6-i686 (2.7-10) Debian Lenny (5.0)
Modelo da máquina	HP Compaq dc5100 SFF(ED514LA)
Processador	Intel(R) Pentium(R) 4 CPU 3.00GHz (XU1) L2 1MiB
Memória	1.5 GiB de RAM (1548220 kB) (512 MiB e 1 GiB)
HD	SAMSUNG HD080HJ/P (80 GB) SATA

6.2.2 Execução e Resultados do Segundo Estudo de Caso

O primeiro passo de execução do estudo de caso consistiu em re-executar por diversas vezes os 12 testes do OurBackup na versão utilizando o `ThreadControl` medindo seu tempo médio de espera. Os testes utilizados foram:

- **T1** - BackupSchedulerTest.testPurgeBackupWithOneVersion
- **T2** - BackupSchedulerTest.testCreateNotSoSimpleBackup
- **T3** - BackupSchedulerTest.testIncrementalOldReplicaRemove
- **T4** - BackupSchedulerTest.testCreateBackupWithNotEnoughSpace
- **T5** - BackupSchedulerTest.testCreateBackupWithoutOneFriend
- **T6** - BackupSchedulerTest.testUpdateBackupWithOneVersion
- **T7** - BackupControllsIntegrationTest.testCreateNotSoSimpleBackup
- **T8** - BackupControllsIntegrationTest.testCreateBackupWithNotEnoughSpace
- **T9** - BackupControllsIntegrationTest.testCreateBackupWithoutOneFriend
- **T10** - BackupControllsIntegrationTest.testUpdateBackupWithOneVersion
- **T11** - BackupControllsIntegrationTest.testPurgeBackupWithOneVersion

- **T12** - BackupControllsIntegrationTest.testIncrementalOldReplicaRemove

Para o cálculo do tempo médio de espera na versão com `ThreadControl` foram utilizadas 10.000 (dez mil) execuções visando obter, com 95% de confiança, um erro menor que 10% nas médias encontradas.

Após calcular esses tempos médios de espera, foram preparadas as versões de tratamento baseadas em atrasos explícitos correspondentes a 80%, 100% e 120% desses tempos. Foi também retirado da versão com `ThreadControl` o código de instrumentação que media os tempos de espera.

Cada uma dessas versões e também a versão atual dos testes (*Current Version*) foi re-executada 10.000 (dez mil) vezes em uma máquina isolada da rede e sem carga a ela submetida. A intenção das re-execuções era identificar testes que falhavam, procurando classificar tais falhas. Um trabalho futuro interessante é também investigar com essa versão as falhas que ocorriam quando a máquina era submetida a carga.

Para realizar a classificação entre os tipos de falhas em testes, depurou-se o código do OurBackup junto com a equipe de desenvolvimento, considerando os testes que falhavam e foram identificadas as mensagens características de falhas por tempo e que apareciam nos logs de teste e também as mensagens dos *logs* que eram comuns quando o problema previamente conhecido (*known bug*) se manifestava.

O objetivo principal da classificação era identificar se em alguma das execuções com `ThreadControl` (TC) havia alguma falha classificada no grupo das asserções antecipadas e tardias.

Como se propõe que a abordagem *Thread Control for Tests* seja combinada com abordagens que estimulem diferentes escalonamentos durante a execução dos testes, além das 10.000 execuções originais com `ThreadControl`, foram realizadas outras 10.000 re-execuções, só que deixando a máquina submetida a uma carga constante (rodando-se em paralelo 4 processos do aplicativo `burnP6`, que gera carga na máquina para efeito de testes de sistema).

Em resumo, foram medidas as quantidades de falhas de execução para as seguintes versões de testes:

- **TC** - Versão com `ThreadControl` em ambiente dedicado;

- **TC_carga** - Versão com `ThreadControl` em máquina rodando outros processos concorrentemente;
- **Current** - Versão corrente dos testes do OurBackup rodando em ambiente dedicado;
- **ED(100%)** - Versão utilizando atrasos explícitos (*Explicit Delays*) de espera correspondente ao tempo médio de espera da versão com `ThreadControl` e executada em ambiente dedicado;
- **ED(80%)** - Versão utilizando atrasos explícitos (*Explicit Delays*) de espera correspondente a 80% do tempo médio de espera da versão com `ThreadControl` e executada em ambiente dedicado;
- **ED(120%)** - Versão utilizando atrasos explícitos (*Explicit Delays*) de espera correspondente a 120% do tempo médio de espera da versão com `ThreadControl` e executada em ambiente dedicado.

Seria interessante realizar o estudo de caso utilizando também carga em outras versões, além da TC_carca, mas por restrições de tempo, já que esse estudo levou meses de execução, apenas as versões acima foram consideradas no estudo aqui descrito.

Análise

Os dados coletados relativos ao número de falhas de execução para cada um dos 12 testes, considerando cada um dos tratamentos estão mostrados na Tabela 6.4.

Ao considerar a classificação de falhas causadas por asserções antecipadas e tardias, foram obtidos os dados mostrados na Tabela 6.5.

Considerando as falhas dos grupos **TC** e **TC_carga**, viu-se que nenhuma de suas falhas se enquadraram no grupo das falhas por asserções antecipadas e tardias, enquanto que estas eram bem comuns nas versões com atrasos explícitos e aconteceram em alguns testes da versão **Current** mesmo sem submeter a máquina em que esta estava rodando à nenhuma carga extra. Isso mostra, considerando os dados obtidos, e a análise feita com base nas mensagens de erro, que é diferente a quantidade de falhas em testes por asserções antecipadas e tardias quando se usa o `ThreadControl` do que ao usar outras abordagens, o que invalida a hipótese nula investigada por este estudo de caso.

Tabela 6.4: Número de falhas em testes para cada tratamento considerando 10.000 execuções de cada

	TC	TC_carga	Current	ED(100%)	ED(80%)	ED(120%)
T1	0	0	0	1192	9871	2340
T2	2	3	5	5713	9548	189
T3	0	0	1	7361	9912	4333
T4	0	0	0	6295	6660	470
T5	0	0	0	3667	9715	1447
T6	0	0	0	3147	9882	3902
T7	0	0	0	318	402	336
T8	108	12	37	1393	9938	764
T9	0	0	0	250	283	243
T10	0	0	0	1745	9945	205
T11	0	0	23	2454	9992	1565
T12	0	0	1	561	9994	290

Porém, como se pôde ver, as execuções de testes com `ThreadControl` (com e sem carga) falharam nos testes T2 e T8. Ao executar o código responsável por classificar as falhas de acordo com as mensagens de erro produzidas, para todas essas execuções com falha de T2 apareciam as mensagens comuns em testes que falhavam pelo problema conhecido. Considerando T8, das 108 falhas de TC, 100 se deveram ao problema conhecido e 8 não puderam ser classificadas em nenhum dos 2 grupos. Das 12 falhas de TC_carga, 4 foram classificadas como devidas ao problema conhecido e 8 delas se devem a outros motivos não identificados (suas mensagens de erro não se enquadravam nos padrões de mensagens que caracterizavam as asserções antecipadas e tardias e nem nos padrões de mensagens para o defeito conhecido). Até onde se pôde depurar, não se sabe o motivo das falhas atribuídas no grupo de outras falhas e o mais provável é que se refiram à outros defeitos da aplicação ou de suas bibliotecas auxiliares e cuja fonte ainda não havia sido descoberta. O uso de técnicas como *replay* e de outras ferramentas de apoio ao teste de sistemas multi-threaded poderia ser combinado com o *ThreadControl* para identificar o motivo de tais falhas nessas situações, algo não trivial, e que ficou apenas como trabalho futuro, para não desviar muito o foco desta tese para a depuração de problemas no sistema e que eram de difícil reprodução.

Ao analisar esse estudo de caso, embora ele não possa provar, por falta de um controle

Tabela 6.5: Número de falhas classificadas como falhas causadas por asserções antecipadas e tardias para cada tratamento considerando 10.000 execuções de cada

	TC	TC_carga	Current	ED(100%)	ED(80%)	ED(120%)
T1	0	0	0	982	7986	2172
T2	0	0	0	318	7138	112
T3	0	0	0	3971	6333	1041
T4	0	0	0	658	3093	128
T5	0	0	0	927	2453	311
T6	0	0	0	1319	7209	848
T7	0	0	0	195	282	242
T8	0	0	0	938	7701	212
T9	0	0	0	184	208	184
T10	0	0	0	1221	6307	161
T11	0	0	23	1944	9806	1490
T12	0	0	1	377	6800	114

total das faltas na aplicação, que nenhuma das falhas encontradas se deve de fato a asserções antecipadas e tardias, ele serviu para dar indícios de que com testes reais tais tipos de falhas não puderam ser encontradas quando o arcabouço de testes *ThreadControl* foi utilizado, já que nenhuma das falhas encontradas foi classificada neste grupo considerando as mensagens de logs. No entanto, mais testes devem ser feitos com o arcabouço e experimentos semelhantes aos estudos de caso feitos devem ser repetidos em outros sistemas e em testes sintéticos onde se tenha um maior controle sobre as faltas existentes na aplicação sendo testada. Além disso, é importante também evoluir o arcabouço e utilizar ferramentas adicionais para o teste de sistemas concorrentes, como técnicas de *replay*, como forma de validá-lo.

Outra observação importante resultante do estudo é que dependendo da abordagem utilizada para esperar antes das asserções, pode-se obter falhas em um teste porque as verificações não são feitas em um tempo apropriado. Essa conclusão pôde ser tomada observando o código exercitado nos testes e também fazendo um paralelo entre o tempo de espera utilizado e o número de falhas na versão *ExplicitDelaysVersion*, que aumentava à medida que se diminuía o intervalo de espera desta versão. No entanto, a prática de aumentar o tempo de

espera não garante que os testes irão passar em máquinas diferentes e pode levar a asserções tardias, feitas depois que o estado esperado foi alcançado mas já se alterou.

Uma outra conclusão que pôde ser tirada desse segundo estudo de caso foi que até onde foi visto, a abordagem *Thread Control for Tests* foi capaz de evitar falhas nos testes do OurBackup que não eram causadas por defeitos na aplicação. Esse fato era esperado, observando-se a implementação do arcabouço, mas o estudo de caso foi importante para dar uma idéia de seu uso em casos práticos. Foi percebido que ao usar outras abordagens tais falhas aconteciam e sua frequência dependia de intervalos de espera escolhidos pelos testadores. Quando se aumenta muito os intervalos escolhidos, além de fazer com que os testes demorem mais, os desenvolvedores podem abandonar alguns dos testes ou raramente executá-los, até mesmo se estes forem os únicos a exercitar o sistema de uma maneira particular.

Observou-se também, através desse estudo de caso que o uso do arcabouço de teste proposto poderia auxiliar desenvolvedores de teste. Ao usar este arcabouço, evitou-se que os desenvolvedores tivessem que prever tempos de espera apropriados a utilizar.

Embora trazendo tais benefícios, é importante perceber que a abordagem *Thread Control for Tests* também apresenta algumas limitações. Uma delas está relacionada ao fato de que monitorar threads afeta a maneira em que as mesmas são escalonadas (*thread interleavings*). Isso pode diminuir as chances de que um determinado escalonamento problemático seja exercitado. Para evitar isso, técnicas e ferramentas para gerar diferentes escalonamentos [90][20] devem ser usadas em conjunto com a abordagem proposta neste trabalho. Isso foi investigado nesse trabalho e as conclusões parciais estão reportadas no Capítulo 8. Uma outra desvantagem da abordagem aqui proposta é que ela não é tão simples de implementar quanto atrasos explícitos. No entanto, acredita-se que este esforço é compensado pelo fato de se evitar falhas em testes devidas a asserções feitas em momentos inadequados, o que leva a testes mais confiáveis. Porém, caso se faça uma preparação da ferramenta de apoio a testes da abordagem já adicionando-lhe bibliotecas que ofereçam operações específicas para a aplicação sendo testada, com cenários de espera comuns, por exemplo, tal esforço adicional por parte dos desenvolvedores de testes é minimizado.

6.3 Conclusões Parciais

Neste capítulo foram apresentadas algumas avaliações iniciais da abordagem utilizando o arcabouço *ThreadControl* em estudos de caso com o intuito de demonstrar que a abordagem é aplicável na prática.

De maneira geral, as avaliações feitas deram indícios de que o uso da abordagem proposta através do arcabouço *ThreadControl* tem conseguido evitar falhas em testes devidas a asserções feitas cedo ou tarde demais. O segundo estudo feito reforçou a necessidade de combinar a abordagem proposta com ferramentas para auxiliar na detecção de defeitos de concorrência e para tentar evitar o efeito de monitoração causado por esta. Um estudo inicial do uso combinado da abordagem com ferramentas desta natureza é apresentado no Capítulo 8.

Como forma de fortalecer a avaliação da abordagem *Thread Control for Tests* apresentada neste capítulo, o Capítulo 7 apresenta uma outra avaliação feita só que sem utilizar o arcabouço `ThreadControl` em si, mas apenas suas idéias por meio de um modelo utilizando a linguagem TLA+, o que permite que este possa ser verificado e com que possam ser feitas simulações neste modelo.

Capítulo 7

Avaliação da Abordagem *Thread Control for Tests* usando TLA+

No Capítulo 4 foi modelado o problema das asserções antecipadas e tardias. Naquele capítulo, demonstrou-se que quando as threads do sistema sob teste não são monitoradas durante a execução dos testes, não se pode garantir que uma asserção antecipada ou tardia não irá ocorrer.

Na Seção 4.2 mostrou-se que quando certas regras de causalidade são consideradas como restrições na execução de testes de um sistema, pode-se garantir que todas as execuções de teste equivalentes a uma execução de teste correta serão também corretas, o que significa que suas asserções não serão executadas em um momento não apropriado.

No Capítulo 5 foi apresentada a abordagem *Thread Control for Tests*. Através de um mapeamento entre esta abordagem e a solução modelada formalmente para evitar testes com falsos positivos por asserções antecipadas e tardias, obteve-se alguns indícios de que ela poderia evitar tal problema, mas isso não pôde ser provado formalmente.

Neste capítulo, será apresentada a validação da abordagem previamente apresentada de maneira formal, utilizando um modelo que possa ser verificado através de suporte ferramental e mais próximo de implementações reais de testes em que não ocorrem os problemas das asserções feitas em momentos inapropriados. Para esta validação, utilizou-se a linguagem TLA+ [58] para modelar testes sem monitoração de threads e testes com monitoração e controle de threads seguindo o que é apresentado na abordagem *Thread Control for Tests* e também as idéias da sua ferramenta de suporte *ThreadControl*.

O intuito deste capítulo é demonstrar que testes que utilizem essa abordagem não apresentam o problema das asserções antecipadas e tardias.

Para isso, apresenta-se inicialmente a motivação para se utilizar a linguagem de especificação TLA+ e posteriormente são descritas especificações de execuções de teste utilizando essa linguagem. A seguir são mostradas as verificações automáticas feitas nos modelos especificados no intuito de verificar estados alcançáveis em cada um dos modelos para se identificar se no modelo que representa o uso da abordagem *Thread Control for Tests* algum desses estados caracteriza um estado de asserções antecipadas ou tardias. Por fim, investiga-se também utilizando uma ferramenta de apoio da linguagem TLA+ se são verificados comportamentos (sequências de estado das threads) caracterizando asserções antecipadas ou tardias em simulações deste modelo.

7.1 Motivação para usar TLA+

A Lógica Temporal de Ações (*Temporal Logic of Actions - TLA*) é uma lógica para especificar e raciocinar a respeito de sistemas concorrentes [57]. Esta lógica é a base para TLA+, uma linguagem completa de especificação.

A idéia principal de TLA+ é tornar prática a descrição de um sistema usando uma única fórmula. Nesse trabalho, TLA+ foi usada para preparar três especificações básicas: a) uma representando uma execução de testes sem monitoração de threads (mostrada no Apêndice A); b) uma outra representando uma execução de teste em que as threads são monitoradas de acordo com a abordagem *Thread Control for Tests* (mostrada no Apêndice B) e considerando uma invariante representando a ausência de asserções antecipadas e tardias nas possíveis execuções de teste; c) uma terceira, estendendo a primeira (a que não usa a abordagem *Thread Control for Tests*), mas também considerando a verificação da ausência de asserções antecipadas e tardias em testes.

A linguagem TLA+ foi usada porque ela se mostrou ser mais próxima do modelo geral apresentado nas seções 4.1 e 4.2 e também porque existem ferramentas de suporte para a linguagem TLA+ que permitem a verificação de especificações preparadas utilizando essa linguagem e também a checagem de modelos. Nesse trabalho, três dessas ferramentas foram utilizadas: [59]:

- *TLATeX*, um programa para composição tipográfica (*typesetting*) de especificações TLA+ (e que foi utilizada para produzir os modelos mostrados nos Apêncides A, B e C)
- O *Syntactic Analyzer*, um parser e verificador de sintaxe de especificações TLA+, e
- *TLC*, um verificador de modelos e simulador para uma subclasse de especificações TLA+ "executáveis".

Na Lógica Temporal de Ações, em que se baseia TLA+, algoritmos são representados com fórmulas [59]. Fórmulas na Lógica de Ações são construídas usando: Valores, Variáveis, Estados, Funções de Estado, e Ações. Um estado (*state*) é uma atribuição de valores a variáveis. Uma ação (*action*) é uma expressão de valor booleano consistindo de variáveis, variáveis denominadas *primed* (representam o novo valor da variável após um certo passo da execução) e símbolos constantes.

De maneira geral, a Lógica Temporal (*Temporal Logic (TL)*) é usada para descrever comportamento dinâmico (sequências infinitas de estados) de programas. TL é usada para formular propriedades de programas. TL também permite que se raciocine a respeito de uma sequência de estados. Uma fórmula temporal é construída de fórmulas elementares usando operadores booleanos e os operadores unários $\Box$ (sempre - *always*) e $\Diamond$ (eventualmente - *eventually*).

Em TLA+, especifica-se um sistema indicando-se comportamentos possíveis – aqueles representando uma execução correta do sistema. Formalmente, um comportamento (*behavior*) é definido como sendo uma sequência de estados (*states*), onde um estado é definido como sendo uma atribuição de valores a variáveis [58]. Um par de estados sucessivos é chamado de um passo (*step*). Passos que deixam uma variável sem modificação são denominados *stuttering steps* ou passos "gagos". Uma execução de um programa é representada por um comportamento consistindo na sequência de estados assumidos [59].

Um programa Π é descrito por quatro itens:

1. Uma coleção de variáveis de estado
2. Um predicado de estado *Init* especificando o estado inicial
3. Uma ação *N* especificando todas as transições permitidas

4. Uma fórmula temporal *L* especificando a condição de progresso do programa.

Depois que se prepara uma especificação de sistema usando TLA+, pode-se checá-la usando a ferramenta TLC. Quando se executa o TLC no modo de checagem de modelo, ele tenta encontrar todos os estados alcançáveis (todos os estados que podem ocorrer em comportamentos satisfazendo uma dada fórmula na forma $Init \wedge \Box[Next]_{vars}$). Quando ele é executado no modo de simulação, ele randomicamente gera comportamentos (*behaviors*), sem tentar checar todos os estados alcançáveis [58].

Especificações TLA+ são particionadas em módulos. A seguir são apresentados três módulos que foram criados para este trabalho: $TestExecution$, $MonitoredTestExecution$ e $TestExecutionCheckingAssertionInvariant$.

7.2 A Especificação *Test Execution* usando TLA+

No Apêndice A é apresentada uma especificação em TLA+ de uma execução de teste sem usar a abordagem *Thread Control for Tests*. Essa especificação foi escrita em um arquivo texto chamado `TestExecution.tla` e então formatada para impressão usando a ferramenta TLATeX. A sintaxe dessa especificação foi checada usando a ferramenta verificadora de sintaxe `tlasany`.

Para que fosse possível checar essa especificação utilizando TLC, teve-se de definir no modelo os possíveis estados de threads sendo considerados e o número de threads a serem consideradas no sistema sob teste (SUT). O número de threads foi 3 para o modelo mostrado no Apêndice A. Cada um dos estados possíveis (e.g. $running$) foi definido como uma constante, e definiu-se também uma outra constante para representar o conjunto dos estados possíveis, denominada $ThreadsPossibleStates$. A definição das constantes do Módulo *TestExecution* foi feita da seguinte forma:

CONSTANT $ThreadsPossibleStates$, $unstarted$,
$started$, $waiting$, $running$, $finished$, $verifying$

Especificou-se que o estado do sistema a ser checado seria o estado das threads que são criadas quando um teste é executado, incluindo a thread de teste. Esse estado é representado na especificação pela variável $threadsStates$, que é um registro (*record*). Registros em TLA+ são uma forma de substituir algumas variáveis por uma única variável. A definição dessa

variável foi feita na especificação da seguinte forma:

VARIABLE $threadsStates$

Os estados possíveis considerados para as threads do sistema (representando os valores assumidos por $tState1$, $tState2$ e $tState3$ da tupla $threadsStates$) são: $\{unstarted,\ started,\ running,\ waiting$ and $finished\}$ e esse conjunto de estados possíveis é representado pela constante $ThreadsPossibleStates$, cujo valor é definido em um arquivo de configuração chamado `TestExecution.cfg`. Os estados possíveis que foram definidos para a thread de teste (representando os valores assumidos por $tState0$) são: $\{unstarted,\ running,\ verifying\}$. Na especificação TLA+, a definição dos estados possíveis para as três threads do sistema e para a thread do teste estão descritos na definição da seguinte invariante:

$$
\begin{aligned}
&TypeInvariant \triangleq \\
&\quad \land\ threadsStates \in [tState0 : \{unstarted,\ running,\ verifying\}, \\
&\quad tState1 : ThreadsPossibleStates,\ tState2 : ThreadsPossibleStates, \\
&\quad tState3 : ThreadsPossibleStates]
\end{aligned}
$$

A idéia básica da especificação denominada Execução de Teste (*Test Execution*) é que uma execução de teste (TE) é representada pela fórmula:

$$TE \triangleq TEini \land \Box[TEnxt]_{\langle threadsStates \rangle}$$

Essa fórmula é apresentada na parte final da especificação e a sintetiza.

A fórmula temporal TE especifica a condição de progresso da execução do teste, a qual indica que o estado inicial deve ser verdadeiro e afirma (*asserts*) que $TEnxt$ é verdadeiro para qualquer passo no comportamento (que representa uma sequência infinita de estados).

Nessa fórmula, $TEini$ é o predicado de estado que especifica o estado inicial do programa. Esse estado indica que todas as threads, incluindo a thread de teste, devem estar no estado $unstarted$. Este predicado é definido no arquivo `TestExecution.tla` da seguinte forma:

$$
\begin{aligned}
&TEini \triangleq \\
&\quad \land\ threadsStates \in [tState0 : \{unstarted\}, \\
&\quad tState1 : \{unstarted\},\ tState2 : \{unstarted\}, \\
&\quad tState3 : \{unstarted\}]
\end{aligned}
$$

A ação $TEnxt$ representa todas as transições permitidas pela especificação. Ela é chamada de relação de próximo estado (*next state relation*) e especifica como o valor da variável

$threadsStates$ pode mudar em cada passo (sendo um passo um par de estados sucessivos). A especificação dessa ação é feita da seguinte forma:

$$
\begin{aligned}
&TEnxt \triangleq \\
&\quad \vee\ TestStarts \\
&\quad \vee\ Thread1Starts \\
&\quad \vee\ Thread1Runs \\
&\quad \vee\ Thread1FinishesRunning \\
&\quad \vee\ Thread1StartsToWait \\
&\quad \vee\ Thread2Starts \\
&\quad \vee\ Thread2Runs \\
&\quad \vee\ Thread2FinishesRunning \\
&\quad \vee\ Thread2StartsToWait \\
&\quad \vee\ Thread3Starts \\
&\quad \vee\ Thread3Runs \\
&\quad \vee\ Thread3FinishesRunning \\
&\quad \vee\ Thread3StartsToWait \\
&\quad \vee\ NothingHappens \\
&\quad \vee\ AssertionPerformed \\
&\quad \vee\ StartAssertionsBlock \\
&\quad \vee\ FinishAssertionsBlock
\end{aligned}
$$

A idéia básica da ação $TEnxt$ é que os passos permitidos em uma execução de teste (especificados na definição dessa ação) são os seguintes: o teste inicia ($TestStarts$) ou a thread1 começa ($Thread1Starts$) ou a thread1 roda ($Thread1Runs$), ou a thread1 para de rodar ($Thread1FinishesRunning$) ou a thread1 começa a esperar ($Thread1StartsToWait$) ou a thread2 começa ($Thread2Starts$) ou a thread2 roda ($Thread2Runs$) ou a thread2 termina de rodar ($Thread2FinishesRunning$), ou a thread2 começa a esperar ($Thread2StartsToWait$) ou a thread3 começa ($Thread3Starts$), ou a thread3 roda ($Thread3Runs$) ou a thread3 termina de rodar ($Thread3FinishesRunning$) ou a thread3 começa a esperar ($Thread3StartsToWait$) ou nada acontece ($NothingHappens$), ou seja, passos do tipo *stuttering*) são permitidos, ou ainda uma asserção é executada ($AssertionPerformed$), ou ainda um bloco de asserções é iniciado ($StartAssertionsBlock$) ou um bloco de asserções é ter-

minado ($FinishAssertionsBlock$). Todas essas ações possíveis são definidas no arquivo `TestExecution.tla` ilustrado no Apêndice A, indicando as pré-condições para que esses passos sejam habilitados e os novos valores de variáveis após a ocorrência desses passos, os quais são representados pela variável *primed* chamada $threadsStates'$.

Por exemplo, a ação indicando quando um teste inicia ($TestStarts$) é definida da seguinte forma:

$$
\begin{aligned}
&TestStarts \triangleq \\
&\quad \land\ threadsStates \in [tState0 : \{unstarted\}, \\
&\quad tState1 : \{unstarted\}, \\
&\quad tState2 : \{unstarted\},\ tState3 : \{unstarted\}] \\
&\quad \land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = running]
\end{aligned}
$$

Essa definição indica que esse passo só é habilitado quando todas as threads estão no estado $unstarted$, e depois que ele ocorre, o novo estado do sistema $threadsStates'$ indica que todas as threads permanecem no mesmo estado, exceto a thread do teste, cujo estado, representado por $tState0$ muda para $running$.

A mesma idéia é seguida pelas outras ações definidas nessa especificação em TLA+.

Algumas das ações são definidas como locais ($LOCAL$) para indicar que em Módulos que estendem o módulo ***TestExecution*** elas podem ser redefinidas. Um exemplo de ação nesse estilo é a que indica que a thread1 roda ($Thread1Runs$). Esta ação é definida da seguinte forma:

$$
\begin{aligned}
&\text{LOCAL } Thread1Runs \triangleq \\
&\quad \land\ (threadsStates.tState1 = started \lor threadsStates.tState1 = waiting) \\
&\quad \land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = running]
\end{aligned}
$$

Esta ação indica que o passo de fazer a thread1 rodar só é habilitado quando esta thread estava no estado $started$ ou $waiting$. Além disso, uma vez que esse passo ocorre, o novo estado da thread1 passa a ser $running$ e os estados das demais threads se mantêm.

Em TLA+, uma fórmula temporal satisfeita por todo comportamento é chamada um teorema. Na especificação *Test Execution*, o seguinte teorema é estabelecido:

$$\text{THEOREM } TE \Rightarrow \Box TypeInvariant$$

Esse teorema indica que a fórmula temporal TE implica que a definição de invariante $TypeInvariant$ é sempre verdadeira. Essa invariante indica que para cada estado possível

de uma execução de teste, o estado de cada thread do sistema (SUT thread) é o conjunto $ThreadsPossibleStates = \{unstarted, started, running, waiting$ and $finished\}$ e que o estado da thread do teste está no conjunto $\{unstarted, running, verifying\}$.

Um outro invariante é definido no arquivo `TestExecution.tla`, o $NotEarlyOrLateAssertionInvariant$. No entanto, a especificação `TestExecution.tla` não contém o teorema indicando que essa invariante deve ser sempre verdadeira. Isso é feito apenas na segunda e na terceira especificações, mostradas nos Apêndices B e C e descritas nas próximas seções. Esse invariante representa a não ocorrência de asserções antecipadas e tardias e sua especificação indica que quando a thread do teste está no estado $verifying$ (executando alguma asserção), todas as threads do SUT devem estar $waiting$ ou $finished$. A definição desse invariante é feita da seguinte forma:

$$
\begin{aligned}
&NotEarlyOrLateAssertion \triangleq \\
&\quad \lor threadsStates \in [tState0 : \{verifying\}, \\
&\quad tState1 : \{waiting, finished\}, tState2 : \{waiting, finished\}, \\
&\quad tState3 : \{waiting, finished\}] \\
&\quad \lor threadsStates \in [tState0 : \{running\}, \\
&\quad tState1 : ThreadsPossibleStates, tState2 : ThreadsPossibleStates, \\
&\quad tState3 : ThreadsPossibleStates] \\
&\quad \lor threadsStates \in [tState0 : \{unstarted\}, \\
&\quad tState1 : \{unstarted\}, tState2 : \{unstarted\}, \\
&\quad tState3 : \{unstarted\}]
\end{aligned}
$$

É importante perceber que a execução de teste descrita pela especificação do arquivo `TestExecution.tla` ilustra uma execução de teste em que as threads não são monitoradas. Isso pode ser observado na definição da ação $StartAssertionsBlock$, que está ilustrada a seguir:

$$
\begin{aligned}
&\text{LOCAL } StartAssertionsBlock \triangleq \\
&\quad \land (threadsStates.tState0 = running) \\
&\quad \land threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = verifying]
\end{aligned}
$$

A única pré-condição especificada nessa ação para que um bloco de asserções se inicie é que a thread do teste esteja no estado $running$. Depois que essa ação acontece, o novo estado da thread de teste $tState0$ será $verifying$.

7.3 A Especificação *Monitored Test Execution* utilizando TLA+

A especificação para ilustrar uma execução de teste na qual as threads são monitoradas de acordo com a abordagem *Thread Control for Tests* está apresentada no Apêndice B. Essa especificação foi escrita em um arquivo texto chamado `MonitoredTestExecution.tla` e então formatada para impressão usando o TLATeX. A sintaxe dessa especificação também foi checada pelo analisador sintático de TLA+.

O módulo definido nesse arquivo incorpora as definições do módulo $TestExecution$ previamente explicado. Isso foi feito utilizando a palavra-chave `EXTENDS`. As ações a serem redefinidas foram especificadas no módulo $TestExecution$ utilizando a palavra-chave `LOCAL`.

A idéia básica dessa especificação é que ela incorpora as idéias da abordagem *Thread Control for Tests* para evitar asserções antecipadas e tardias através de monitoração de threads durante transições de estado importantes. Isso foi representado na especificação por algumas mudanças principais em algumas ações definidas na especificação *Test Execution*:

- Algumas ações tiveram de mudar para incluir em suas pré-condições a verificação de que a thread do teste deve estar rodando: $Thread1Runs$, $Thread1StartsToWait$, $Thread1FinishesRunning$, $Thread2Runs$, $Thread2StartsToWait$, $Thread2FinishesRunning$, $Thread3Runs$, $Thread3StartsToWait$, $Thread3FinishesRunning$. Essa mudança foi feita para evitar asserções tardias, as quais acontecem quando a thread muda seu estado enquanto verificações (asserções) estão sendo feitas. Fazendo essa mudança na especificação da execução de um teste corresponde a considerar a **fase 4** de teste da abordagem *Thread Control for Tests* ("execução das asserções, sem temer que alguma *thread* desperte e execute operações que afetem os resultados de teste"). Essas ações alteradas, quando habilitadas, correspondem à **fase 2** de uma execução de teste, em que operações do sistema estão sendo invocadas, exercitando assim o sistema e fazendo as transições de estado das threads acontecerem.

- A ação $StartAssertionsBlock$ foi redefinida incluindo como pré-condição o fato de

que as threads do sistema sendo testado devem estar $waiting$ ou $finished$, o que é uma situação comum para momentos apropriados de realizar asserções (quando todas as threads estão esperando ou finalizadas). Dessa forma, asserções antecipadas podem ser evitadas já que a execução só entrará no bloco de asserções quando estiver no momento certo de realizar a asserção. A definição de um momento apropriado para realizar asserções corresponde à **fase 1** da abordagem *Thread Control for Tests*, que é a preparação do ambiente indicando o estado esperado no qual asserções podem ser executadas. O uso de tais estados como pré-condições para habilitar o passo que faz com que o estado da thread do teste se torne $verifying$ corresponde à **fase 3** da abordagem ("espera até que o estado esperado para o sistema, especificado na fase 1, tenha sido alcançado").

- A ação $AssertionPerfomed$ da especificação *Test Execution* foi substituída pela ação $AssertionNotEarly$. A ação anterior estava considerando como pré-condição o fato de que o estado da thread de teste poderia ser $running$ ou $verifying$ (dentro de um bloco de asserções) já que em uma execução de threads sem monitoração uma asserção pode ocorrer antes que o teste espere que ela ocorra. No módulo $MonitoredTestExecution$, a ação $AssertionNotEarly$ só é habilitada quando o estado da thread de teste ($threadsStates.tState0$) é $verifying$, o que indica que a execução da asserção ocorre dentro de um bloco de asserções, depois da verificação sobre se o estado esperado para o sistema foi atingido.

A ação $AssertionNotEarly$ foi definida da seguinte forma:

$$
\begin{aligned}
&AssertionNotEarly \triangleq \\
&\quad \land (threadsStates.tState0 = verifying) \\
&\quad \land \text{UNCHANGED } threadsStates
\end{aligned}
$$

Para que o estado da thread de teste esteja $verifying$ e a asserção possa ocorrer, a asserção só poderá ser feita dentro de um bloco de asserções, delimitado pelos passos $StartAssertionsBlock$ e $FinishAssertionsBlock$. A ação $StartAssertionsBlock$ foi redefinida no arquivo `MonitoredTestExecution.tla` da seguinte forma:

$$
\begin{aligned}
&StartAssertionsBlock \triangleq \\
&\quad \land (threadsStates.tState1 = finished \lor threadsStates.tState1 = waiting)
\end{aligned}
$$

$$\wedge\ (threadsStates.tState2 = finished \vee threadsStates.tState2 = waiting)$$
$$\wedge\ (threadsStates.tState3 = finished \vee threadsStates.tState3 = waiting)$$
$$\wedge\ (threadsStates.tState0 = running)$$
$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = verifying]$$

Como se vê, o sistema só consegue executar o passo de entrar em um bloco de asserções quando suas threads estiverem esperando ou finalizadas e quando isto ocorre, a thread do teste passa para o estado $verifying$.

A ação $FinishAssertionsBlock$ não foi alterada na especificação *Monitored Test Execution*. A idéia principal dessa ação é que ela é habilitada quando a thread de teste está $verifying$ e ela muda o valor da variável `threadStates` para $running$ novamente. Isso corresponde à **fase 5** de uma execução de teste utilizando a abordagem *Thread Control for Tests*, na qual a execução de teste prossegue de forma que o sistema possa ser exercitado novamente e novas asserções possam ser feitas ou o teste possa terminar. No arcabouço de testes *ThreadControl*, essa ação corresponde à primitiva de teste `proceed`. O início de um bloco de asserção é representado no arcabouço pela invocação do método `waitUntilStateIsReached`.

Considerando as ações declaradas no arquivo `MonitoredTestExecution.tla`, a relação de próximo estado $TEnxt2$ do módulo $MonitoredTestExecution$ é definida da seguinte forma:

$$TEnxt2 \triangleq$$
$$\vee\ TestStarts$$
$$\vee\ Thread1Starts$$
$$\vee\ Thread1Runs$$
$$\vee\ Thread1FinishesRunning$$
$$\vee\ Thread1StartsToWait$$
$$\vee\ Thread2Starts$$
$$\vee\ Thread2Runs$$
$$\vee\ Thread2FinishesRunning$$
$$\vee\ Thread2StartsToWait$$
$$\vee\ Thread3Starts$$
$$\vee\ Thread3Runs$$

$\lor\ Thread3FinishesRunning$

$\lor\ Thread3StartsToWait$

$\lor\ NothingHappens$

$\lor\ AssertionNotEarly$

$\lor\ StartAssertionsBlock$

$\lor\ FinishAssertionsBlock$

A especificação *Monitored Test Execution* ($TE2$) pode ser resumida pela seguinte fórmula temporal:

$$TE2 \triangleq TEini \land \Box[TEnxt2]_{\langle threadsStates \rangle}$$

Os teoremas considerados nessa especificação são os seguintes:

THEOREM $TE2 \Rightarrow \Box TypeInvariant$

THEOREM $TE2 \Rightarrow \Box NotEarlyOrLateAssertion$

Esses teoremas afirmam que quando se considera a especificação de Execução de Teste Monitorada (`TE2`), isso implica que as invariantes $TypeInvariant$ e $NotEarlyOrLateAssertion$ são sempre verdadeiras. Isso significa que asserções antecipadas ou tardias não acontecem e que todos os estados possíveis respeitam os estados válidos para as threads do sistema sob teste ($\{unstarted, started, running, waiting$ and $finished\}$) e para a thread de teste ($\{unstarted, running, verifying\}$).

7.4 A Especificação em TLA+ da Execução de Teste Checando Invariante de Asserções

Para que o TLC analise asserções antecipadas e tardias na execução do teste no qual a abordagem *Thread Control for Tests* não é usada, teve de ser implementado um outro módulo, denominado $TestExecutionCheckingAssertionInvariant$ e descrito no arquivo `TestExecutionCheckingAssertionInvariant.tla`. A definição deste módulo é bem curta, como se pode ver a seguir:

MODULE $TestExecutionCheckingAssertionInvariant$

EXTENDS $TestExecution$

THEOREM $TE \Rightarrow \Box NotEarlyOrLateAssertion$

Esse módulo incorpora as definições do módulo $TestExecution$ (usando `EXTENDS`), mas ele também define um outro teorema. Esse teorema afirma que, considerando-se a especificação TE, definida no módulo $TestExecution$, a invariante $NotEarlyOrLateAssertion$ deve ser sempre verdadeira. A ferramenta TLC foi usada para checar essa e as outras especificações previamente apresentadas, como será explicado nas seções a seguir.

7.5 Análise de Estados Alcançáveis

Para analisar os estados alcancáveis de cada modelo para tentar identificar as ocorrências de asserções antecipadas ou tardias, executou-se o TLC no modo de checagem de modelo. Nesse modo, ele tenta encontrar todos os estados alcançáveis (todos os estados satisfeitos por uma dada fórmula) [58]. Quando a opção `-dump` é usada, todos esses estados são gravados em um arquivo. Quando se executa o TLC nesse modo, e em um dado estado, as invariantes configuradas não são válidas, ele interrompe a checagem e apresenta a sequência problemática de estados (comportamento) que foi encontrado.

Para executar TLC considerando o módulo $TestExecution$, foi utilizado o seguinte comando:

```
java tlc.TLC -dump TEStates.out TestExecution.tla
```

Parte da saída mostrada é a seguinte:

```
Model checking completed. No error has been found.
1428 states generated, 251 distinct states found
```

Os 251 estados distintos encontrados foram armazenados no arquivo `TEStates.out.dump`.

O mesmo procedimento foi seguido para o arquivo `MonitoredTestExecution.tla`. Parte da saída do TLC é mostrada a seguir:

```
Model checking completed. No error has been found.
535 states generated, 134 distinct states found
```

Os 134 estados distintos encontrados foram armazenados no arquivo `TE2States.out.dump`.

Ao considerar o arquivo `TestExecutionCheckingAssertionInvariant.tla`, TLC detectou que uma invariante foi violada e apresentou uma sequência de estados (comportamento) que exemplifica isso. Parte da saída da execução do TLC com essa especificação é mostrada a seguir:

```
Error: Invariant NotEarlyOrLateAssertion is violated. The behavior
up to this point is:
STATE 1: <Initial predicate>
threadsStates = [ tState0 |-> unstarted,
  tState1 |-> unstarted,
  tState2 |-> unstarted,
  tState3 |-> unstarted ]

STATE 2: <Action line 32, col 9 to line 35,
col 68 of module TestExecution>
threadsStates = [ tState0 |-> running,
  tState1 |-> unstarted,
  tState2 |-> unstarted,
```

```
  tState3 |-> unstarted ]

STATE 3: <Action line 81, col 9 to line 82,
col 77 of module TestExecution>
threadsStates = [ tState0 |-> verifying,
  tState1 |-> unstarted,
  tState2 |-> unstarted,
  tState3 |-> unstarted ]
```

Esse comportamento exemplifica um caso de execução de teste em que a thread do teste faz a asserção (vai para o estado *verifying*) antes que as threads da aplicação sendo testada tenham iniciado sua execução (ou seja, estão ainda no estado *unstarted*). Ao rodar o TLC utilizando o modo de checagem de modelos para esse arquivo diversas vezes, o mesmo comportamento problemático é exibido. Para mostrar diferentes exemplos de comportamentos que violam essa invariante, pode-se rodar o TLC no modo de simulação, como será explicado na próxima seção.

Além de rodar o TLC e encontrar o primeiro comportamento que quebra a invariante para a terceira especificação mostrada, implementou-se um programa Java, chamado `States Analyzer`, para analisar os estados distintos encontrados ao rodar o TLC usando a especificação *Test Execution* (sem checar por asserções antecipadas e tardias) e a especificação *Monitored Test Execution*. A idéia desse programa Java era verificar quais dos estados alcançáveis encontrados era um caso de asserção antecipada ou tardia. Esse código foi implementado considerando a invariante $NotEarlyOrLateAssertion$, mostrada a seguir:

$$
\begin{aligned}
&NotEarlyOrLateAssertion \triangleq \\
&\quad \lor\ threadsStates \in [tState0 : \{verifying\}, \\
&\quad tState1 : \{waiting, finished\},\ tState2 : \{waiting, finished\}, \\
&\quad tState3 : \{waiting, finished\}] \\
&\quad \lor\ threadsStates \in [tState0 : \{running\}, \\
&\quad tState1 : ThreadsPossibleStates,\ tState2 : ThreadsPossibleStates, \\
&\quad tState3 : ThreadsPossibleStates] \\
&\quad \lor\ threadsStates \in [tState0 : \{unstarted\}, \\
&\quad tState1 : \{unstarted\},\ tState2 : \{unstarted\},
\end{aligned}
$$

$tState3 : \{unstarted\}]$

O programa Java `StatesAnalyzer` leu todos os estados dos dois arquivos gerados (`TEStates.out.dump` e `TE2States.out.dump`) e analisou os casos de asserções antecipadas ou tardias ($tState0 = verifying$ e $tState1$ ou $tState2$ ou $tState3$ em um estado diferente de *finished* ou $waiting$).

Ao rodar esse programa com a saída do TLC com a especificação do arquivo `MonitoredTestExecution.tla`, o programa não encontrou nenhum caso de estados caracterizando uma asserção antecipada ou tardia, o que mostra que nenhum dos estados alcançáveis de uma execução de teste monitorada leva a asserções antecipadas ou tardias. A saída desse programa é mostrada a seguir:

```
Number of states to analyze:134
No early/late assertion found in states
```

Ao executar o programa `StatesAnalyzer` com a saída da execução do TLC com a especificação *Test Execution*, foram encontrados alguns estados caracterizando asserções antecipadas ou tardias. No Apêndice D se mostra a saída do programa `StatesAnalyzer` usando como entrada o arquivo `TEStates.out.dump`. Essa saída mostra que 251 estados foram analisados e também mostra os estados em que asserções antecipadas ou tardias foram encontradas (117 estados). Um desses exemplos é o seguinte:

```
Early/Late assertion in state:State 243:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> RUNNING,
 tState2 |-> WAITING]
```

Este pode ser tanto um caso em que era muito cedo para realizar a asserção (porque a thread1 não está $waiting$ ou *finished* ainda, ou um caso em que uma asserção tardia acontece (asserções estavam sendo executadas e a thread1, que estava esperando, começa a executar novamente). Para exemplificar tais exemplos de comportamentos, o TLC foi executado no modo de simulação, como explicado a seguir.

7.6 Análise de Comportamentos

Quando se executa o TLC no modo de simulação, ele aleatoriamente gera comportamentos, sem tentar checar todos os estados alcançaveis [58]. Considerando isso, invocou-se o TLC nesse modo várias vezes (1000) para a especificação `TestExecutionChecking AssertionInvariant.tla` redirecionando-se a saída padrão e de erro das várias rodadas para um arquivo chamado `EarlyLateAssertions.out`. Dessa forma se pôde obter alguns diferentes exemplos da execução do modelo em que foram encontrados automaticamente comportamentos que violam a invariante $NotEarlyOrLateAssertion$.

Para diferenciar os casos de asserções antecipadas e tardias, implementou-se um programa Java chamado `BehaviorsAnalyzer`. Esse programa analisa a saída do TLC removendo repetições de comportamentos e analisando quais dos comportamentos encontrados representam asserções antecipadas e também os que representam asserções tardias. Alguns exemplos desses casos de asserções detectados são mostrados nos apêndices E e F respectivamente. Esses exemplos provam que não se pode garantir a ausência de tais tipos de asserções quando as threads não são monitoradas (em consonância com o Teorema 1 da Seção 4.1).

Como previamente apresentado, nenhum dos estados alcançáveis gerado pelo modo de checagem de modelos do TLC para a execução de teste monitorada representava um caso de asserção antecipada ou tardia. Esse mesmo resultado foi obtido estendendo a especificação para utilizar cinco threads (o que segue a mesma idéia da especificação original aumentando apenas o número de estados a serem analisados, que passou a ser 3158). Dessa forma, pôde-se mostrar que uma execução de teste seguindo a abordagem *Thread Control for Tests* não leva a comportamentos caracterizando asserções antecipadas ou tardias. Considerando isso, não é necessário executar o TLC no modo de simulação para encontrar comportamentos quebrando a invariante já que tais comportamentos não seriam encontrados. Quando se executa o TLC nesse modo com o arquivo `MonitoredTestExecution.tla`, essa execução nunca para.

7.7 Conclusões Parciais

Nesse capítulo mostrou-se um modelo de teste utilizando a abordagem *Thread Control for Tests* em TLA+ e um outro em que a monitoração das threads de teste não é considerada durante a sua execução. Demonstrou-se mais uma vez que sem monitoração e sem prover um certo nível de controle das threads do sistema evitando certas transições de estado em momentos inoportunos dos testes, não se pode garantir que falsos positivos causados por asserções antecipadas ou tardias não irão ocorrer.

Utilizando as ferramentas de suporte de TLA+ pôde-se verificar o modelo gerado e instanciado para estados comuns de threads, tentando identificar casos de asserções antecipadas ou tardias, os quais não foram identificados no modelo representando uma execução de teste seguindo a abordagem *Thread Control for Tests*.

O modelo em TLA+ demonstrou ser bem próximo da implementação da abordagem utilizando a ferramenta *ThreadControl* e apresentou a vantagem de ser executável comparado a outras formas de modelar formalmente a abordagem de testes apresentada neste trabalho.

Capítulo 8

Combinando a Abordagem *Thread Control for Tests* com *Geradores de Ruído*

Como previamente comentado, o uso de monitoração de threads durante os testes de aplicações pode afetar no escalonamento das threads, em alguns casos podendo levar ao denominado "efeito de monitoração" ou "efeito de observação" (*probe effect* ou *observer effect*). Através desse efeito, alguns defeitos no código podem se revelar com menos frequência durante a execução dos testes. Com o intuito de investigar maneiras para tentar evitar este efeito, foram buscadas ferramentas que poderiam ser utilizadas em combinação com a abordagem proposta nesta tese para ajudar na exploração de diferentes caminhos de escalonamento durante os testes de sistemas multi-threaded, como geradores de ruído.

Neste capítulo, através da apresentação de um exemplo prático de uso da abordagem em uma aplicação simples, demonstra-se como a abordagem *Thread Control for Tests* pode ser combinada com a técnica de geradores de ruído (*noise makers*) para que assim se possam incentivar diferentes escalonamentos das threads durante re-execuções dos testes. Um gerador de ruído (*noise maker*) tem o objetivo de aumentar a probabilidade de se exercitar um comportamento incorreto relacionado a algum *bug* de concorrência [10]. No exemplo prático que será apresentado foi utilizado o arcabouço de testes *ThreadControl* e a ferramenta ConTest [2]. A metodologia utilizada para o desenvolvimento desse estudo de caso está detalhada na Seção 8.4.

Com o estudo de caso apresentado neste capítulo pretende-se mostrar que é possível combinar a abordagem proposta nesta tese com ferramentas que ajudam a explorar diferentes

escalonamentos durante a execução de testes de sistemas multi-threaded. Essa combinação é algo desejável não só para tentar evitar o efeito de monitoração, mas também para auxiliar no processo de detecção de defeitos de concorrência por meio de testes que precisam ser confiáveis.

8.1 A Ferramenta ConTest

A ferramenta ConTest surgiu como parte de um método proposto para achar defeitos de concorrência de maneira efetiva [29]. ConTest é uma ferramenta da IBM para o teste de aplicações Java Multi-threaded [3]. Ela é principalmente utilizada para expor e eliminar *bugs* relacionados a concorrência em software paralelo e distribuído.

A idéia geral da ferramenta ConTest é que ela sistematicamente e de maneira transparente escalona a execução das threads de um programa de forma que cenários que são mais propensos a possuírem condições de corrida, impasses (*deadlocks*) e outros *bugs* intermitentes, ou seja, problemas de sincronização, sejam forçados a aparecer com uma maior frequência. Busca-se fazer com que o uso da ferramenta tenha pouco impacto no esforço de teste aproveitando testes existentes que podem ser re-executados.

ConTest já foi usada em grandes aplicações (como o WebSphere da IBM) e além da capacidade de gerar ruídos, pode também ser usada para medir cobertura de teste, auxiliar em depuração, permitir re-execução (*replay*) e para apresentar informação de deadlocks.

A versão da ConTest disponível para download se apresenta como uma ferramenta de uso simples para facilitar a identificação de bugs através de testes existentes. Ela trabalha instrumentando o código da aplicação com instruções condicionais de `sleep` e `yield` heuristicamente controladas. Em tempo de execução, o ConTest às vezes tenta causar trocas de contexto nesses locais instrumentados. Os locais escolhidos são aqueles cuja ordem relativa entre as threads tem maior propensão de afetar no resultado: entrada e saída de blocos sincronizados, acesso a variáveis compartilhadas, etc. As decisões são randômicas de forma que diferentes ordens de execução das threads sejam testadas a cada rodada. Heurísticas são usadas para tentar revelar bugs típicos [74].

Para utilizar o ConTest basta que o usuário configure um arquivo de propriedades indicando as classes a serem instrumentadas (arquivo `KingProperties`) e que

este execute os testes utilizando o seguinte argumento para a máquina virtual Java: `-javaagent:contestLibPath/ConTest.jar`, onde `contestLibPath` é o caminho para o diretório onde está o arquivo `ConTest.jar`.

Observando a descrição do ConTest, pode-se ver que essa ferramenta se baseia na existência de testes e estes precisam ser confiáveis. Considerando isso, acredita-se que utilizando a abordagem *Thread Control for Tests* no desenvolvimento de testes, se está também provendo as bases para ferramentas que auxiliam na descoberta de defeitos de concorrência baseadas em código executável, como é o caso do ConTest.

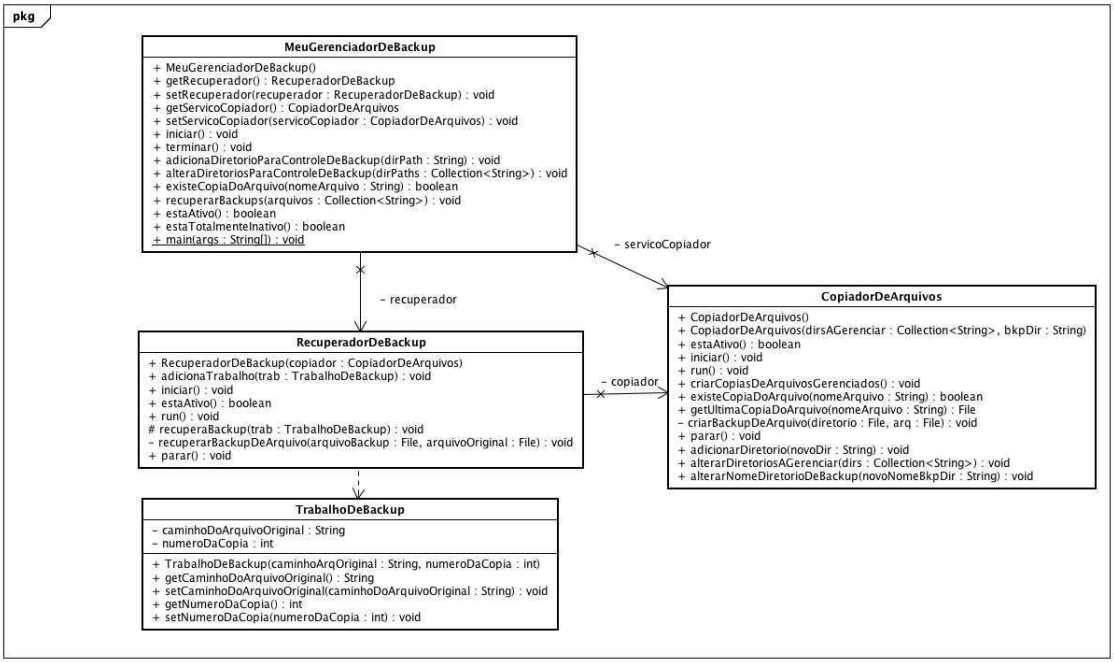
8.2 Aplicação Exemplo Utilizada: *Meu Gerenciador de Backup*

Com o objetivo de utilizar uma aplicação multi-threaded em que se tivesse um controle maior sobre o seu código do que utilizando o OurBackup e onde se pudessem injetar faltas controladas que fossem identificadas em apenas algumas das execuções dos testes, foi implementado um sistema simples de gerenciamento de backup pessoal, denominado *Meu Gerenciador de Backup*. Foram implementadas basicamente as classes que ofereceriam o serviço em si. A idéia dessa aplicação simples é que o usuário cadastre alguns diretórios a serem gerenciados e cujos arquivos devem ser copiados para um outro diretório de backup de tempos em tempos, e também que o usuário possa recuperar alguns dos arquivos que estavam em algum dos diretórios gerenciados caso queira recuperar uma versão anterior ou tenha simplesmente perdido aquele arquivo. Embora também se baseie em backup, esse era um sistema muito mais simples do que o OurBackup e que não era nem entre pares e nem se baseava em redes sociais ou usava bancos de dados.

Por simplicidade foram projetadas quatro classes, indicadas no diagrama apresentado na Figura 8.1.

A classe que condensa todos os serviços básicos é a classe `MeuGerenciadorDeBackup`. As demais classes são Threads que de tempos em tempos realizam ou o serviço de criar cópias de arquivos de diretórios gerenciados (classe `CopiadorDeArquivos`) ou o serviço de recuperar arquivos requisitados pelo usuário (classe `RecuperadorDeBackup`) e empilhados como tarefas de backup (representadas

Figura 8.1: Classes do Serviço Gerenciador de Backup Pessoal



pela classe `TrabalhoDeBackup`), que são processadas de tempos em tempos pelo recuperador, que posteriormente as remove de sua fila de tarefas.

Basicamente, do ponto de vista do usuário, o sistema oferece as seguintes funções de maneira geral:

- Adicionar um certo diretório para que seja feito seu controle de backup;
- Alterar os diretórios que estão sendo considerados no controle de backup;
- Verificar se existe uma cópia de um certo arquivo;
- Recuperar backups de uma série de arquivos;

As tarefas realizadas de fato por essas operações são executadas de maneira assíncrona, o que exige que durante os testes sejam necessárias operações que garantam que elas puderam ser feitas antes que as asserções dos testes sejam realizadas.

8.3 Testando o Gerenciador de Backup

Para testar algumas funcionalidades do serviço de backup descrito anteriormente, foi projetado um caso de teste que avalia a corretude no uso concorrente do sistema por dois usuários.

Na primeira interação com o sistema cada usuário manda adicionar um certo diretório para controle de backup e na segunda interação o usuário manda recuperar algum arquivo que está no diretório que ele previamente passou.

O caso de teste projetado realiza os seguintes passos (considerando inclusive passos de espera) com o intuito de verificar o sistema através de sua fachada representada pela classe `MeuGerenciadorDeBackup`:

1. Criar diretórios “expBkp1” e “expBkp2”.
2. Garantir que esses diretórios estão vazios, apagando seu conteúdo se for o caso.
3. Criar um arquivo “arquivoTeste1.txt"no diretório “expBkp1” e um arquivo “arquivoTeste2.txt” no diretório “expBkp2”.
4. Criar uma instância do serviço de backup (`MeuGerenciadorDeBackup`).
5. Verificar que o serviço de backup não está ativo.
6. Verificar que não foram criadas cópias de backup ainda para os arquivos “arquivoTeste1.txt” e “arquivoTeste2.txt”.
7. Iniciar o serviço de backup.
8. Iniciar uma thread representando a requisição do usuário 1 de começar a criar backup dos arquivos do diretório “expBkp1”.
9. Iniciar uma thread representando a requisição do usuário 2 de começar a criar backup dos arquivos do diretório “expBkp2”.
10. Esperar que requisições possam ser processadas e que o serviço crie uma cópia dos arquivos dos diretórios a serem gereciados (threads das requisições finalizadas e threads do copiador e recuperador paradas).
11. Verificar que foram criadas cópias de backup para os arquivos “arquivoTeste1.txt” e “arquivoTeste2.txt”.
12. Apagar arquivos de teste.

13. Iniciar uma thread representando a requisição do usuário 1 de recuperar o "arquivoTeste1.txt" que foi apagado.

14. Iniciar uma thread representando a requisição do usuário 2 de recuperar o "arquivoTeste2.txt" que foi apagado.

15. Esperar que requisições de recuperação tenham sido processadas (threads das requisições finalizadas e threads do copiador e recuperador paradas).

16. Verificar que os arquivos de teste anteriormente apagados voltaram a existir.

17. Terminar o serviço de backup.

18. Esperar até que ele tenha terminado (threads do recuperador e copiador finalizadas).

19. Verificar que o serviço de backup está totalmente inativo (incluindo seus serviços internos).

20. Apagar os diretórios criados inicialmente para o teste.

Para realizar o estudo de caso descrito a seguir e re-executar esse caso de teste várias vezes utilizando a ferramenta ConTest, preparou-se inicialmente um caso de teste que deveria passar normalmente e um outro, em que se configurou a instância do `MeuGerenciadorDeBackup` para utilizar como serviço recuperador uma outra versão do `RecuperadorDeBackup`, que foi denominada `RecuperadorDeBackupProblemático`. Esse caso de teste foi chamado `MeuGerenciadorDeBackupTestForFailure`, e ele foi utilizado para as execuções de testes reais descritas a seguir.

Como se pode ver nos passos descritos anteriormente, algumas das operações a serem verificadas são executadas de maneira assíncrona pelas threads representadas pelas classes `RecuperadorDeBackup` e `CopiadorDeArquivos`, sendo necessário que o teste espere algum tempo antes de fazer verificações sobre a cópia de arquivos ou sua recuperação. Para evitar que estes testes levassem a falsos positivos por tempos inadequados de espera e para evitar que se tenha de prover métodos para expor as instâncias do recuperador e do copiador só por causa do teste, foi utilizado o arcabouço de testes `ThreadControl`.

O código abaixo ilustra o trecho principal desse teste e serve como um exemplo de teste em que vários estados de espera diferentes são demandados, como se pôde ver pelo detalhamento dos passos do teste, sendo a configuração desses estados passada nas chamadas aos métodos `prepare` feitas nas linhas 26, 50 e 63-64, utilizando para isso a variável `sysConfig` que é do tipo `ListOfThreadConfigurations` e que é preparada passando-se identificadores de threads do sistema e o estado em que devem estar.

Código Fonte 8.1: Teste *testaBackupMultiplosUsuarios*

```
1  @Test
2  public void testaBackupMultiplosUsuarios() throws IOException {
3          File dirParaFazerBackup1 = new File("expBkp1");
4          File dirParaFazerBackup2 = new File("expBkp2");
5          cleanDir(dirParaFazerBackup1);
6          cleanDir(dirParaFazerBackup2);
7          dirParaFazerBackup1.mkdir();
8          dirParaFazerBackup2.mkdir();
9          File arq1 = new File(dirParaFazerBackup1.getAbsolutePath()
10             + File.separator + "arquivoTeste1.txt");
11         BufferedWriter writer = new BufferedWriter(new FileWriter(arq1));
12         writer.write("Arquivo de teste1");
13         writer.close();
14         File arq2 = new File(dirParaFazerBackup2.getAbsolutePath()
15             + File.separator + "arquivoTeste2.txt");
16         writer = new BufferedWriter(new FileWriter(arq2));
17         writer.write("Arquivo de teste2");
18         writer.close();
19         ThreadControl tc = new ThreadControl();
20         ListOfThreadConfigurations sysConfig =
21             buildConfigurationOfCopiadorAndRecuperadorProblematicoWaiting
                   ();
22         ThreadConfiguration tcUserThreadsFinished = new
23             ThreadConfiguration(RequisicaoDeCopia.class.getCanonicalName
                   (),
24             ThreadState.FINISHED, ThreadConfiguration.AT_LEAST_ONCE);
25         sysConfig.addThreadConfiguration(tcUserThreadsFinished);
26         tc.prepare(sysConfig);
27         MeuGerenciadorDeBackup backup = new MeuGerenciadorDeBackup();
```

```
28          backup.setRecuperador(new RecuperadorDeBackupProblematico(backup
29              .getServicoCopiador()));
30          assertFalse(backup.estaAtivo());
31          assertFalse(backup.existeCopiaDoArquivo(arq1.getAbsolutePath()));
32          assertFalse(backup.existeCopiaDoArquivo(arq2.getAbsolutePath()));
33          backup.iniciar();
34          RequisicaoDeCopia req1 = new RequisicaoDeCopia(backup,
35              dirParaFazerBackup1, arq1);
36          RequisicaoDeCopia req2 = new RequisicaoDeCopia(backup,
37              dirParaFazerBackup2, arq2);
38          req1.start();
39          req2.start();
40          tc.waitUntilStateIsReached();
41          assertTrue(backup.existeCopiaDoArquivo(arq1.getAbsolutePath()));
42          assertTrue(backup.existeCopiaDoArquivo(arq2.getAbsolutePath()));
43          sysConfig =
44              buildConfigurationOfCopiadorAndRecuperadorProblematicoWaiting
                    ();
45          ThreadConfiguration tcRequisicoesRecuperacaoFinished = new
46              ThreadConfiguration(
47              RequisicaoDeRecuperacao.class.getCanonicalName(),
48              ThreadState.FINISHED, ThreadConfiguration.AT_LEAST_ONCE);
49          sysConfig.addThreadConfiguration(tcRequisicoesRecuperacaoFinished
                );
50          tc.prepare(sysConfig);
51          tc.proceed();
52          arq1.delete();
53          arq2.delete();
54          RequisicaoDeRecuperacao req3 =
55              new RequisicaoDeRecuperacao(backup, arq1);
56          RequisicaoDeRecuperacao req4 =
57              new RequisicaoDeRecuperacao(backup, arq2);
58          req3.start();
59          req4.start();
60          tc.waitUntilStateIsReached();
61          assertTrue(arq1.exists());
62          assertTrue(arq2.exists());
```

```
63          tc.prepare(
64              buildConfigurationOfCopiadorAndRecuperadorProblematicoFinished
                    ());
65          tc.proceed();
66          backup.terminar();
67          tc.waitUntilStateIsReached();
68          assertTrue(backup.estaTotalmenteInativo());
69          tc.proceed();
70          cleanDir(dirParaFazerBackup1);
71          cleanDir(dirParaFazerBackup2);
72  }
```

8.4 Estudo de Caso para Avaliar o Uso da abordagem *Thread Control for Tests* utilizando Geradores de Ruído

Para verificar através de um estudo de caso a quantidade de falhas identificadas pelo teste quando este é implementado utilizando a abordagem *Thread Control for Tests* apenas e utilizando essa abordagem combinada com geradores de ruído (através da ferramenta ConTest) foi utilizado o caso de teste descrito anteriormente. A seguir são descritos maiores detalhes sobre esse estudo e seus resultados.

8.4.1 Objetivos

Os objetivos do estudo de caso proposto nesse capítulo de acordo com a abordagem Objetivo/Questão/Métrica (*Goal/Question/Metric*) [9] podem ser descritos da seguinte forma:

- *Analisar o número de falhas em execuções de um teste de aplicação multi-threded utilizando a abordagem Thread Control for Tests isolada e utilizando essa abordagem combinada com a técnica de geradores de ruído com o propósito de avaliar se as duas técnicas podem ser combinadas e de que forma o uso de geradores de ruído poderia melhorar a detecção de faltas em aplicações com respeito ao aumento da frequência com que estas faltas se revelavam via falhas de execuções de testes do ponto de vista de quem executa testes automáticos e no contexto de testes que exercitam operações*

assíncronas.

8.4.2 Hipóteses

O estudo de caso em que se foca esse capítulo busca investigar dois aspectos:

- É possível utilizar a abordagem *Thread Control for Tests* combinada com geradores de ruído.
- O uso da abordagem *Thread Control for Tests* combinada com geradores de ruído ajuda a evitar o problema do efeito de observação incentivando diferentes escalonamentos e aumentando a frequência com que testes que poderiam expor uma dada falta injetada falhem.

8.4.3 Variáveis

Para o estudo de caso aqui apresentado, é utilizada a seguinte **variável de resposta**:

- Número de testes que falham para um certo número de re-execuções de testes.

Quanto aos **fatores** (variáveis independentes), foram variados dois: a **forma de invocação dos testes** (se usando um gerador de ruído ou não) e a máquina utilizada, já que sua arquitetura influencia na frequência com que determinados defeitos relacionados a concorrência se manifestam. Os **tratamentos** (alternativas) considerados para este primeiro fator foram os seguintes:

- Invocação normal de execuções repetidas de um teste JUnit utilizando a abordagem *Thread Control for Tests* através do arcabouço de testes *ThreadControl*;
- Invocação utilizando a ferramenta ConTest como geradora de ruído de execuções repetidas de um teste JUnit utilizando a abordagem *Thread Control for Tests* através do arcabouço de testes *ThreadControl*;

8.4.4 Sujeitos e Objetos

Considerando como sujeitos de um experimento de software os indivíduos que utilizam um certo método ou ferramenta [55], instanciados nesse caso para as ferramentas *ThreadControl*

e *ConTest*, este estudo de caso teve como sujeito apenas a autora deste trabalho de doutorado. Esta foi responsável por criar o caso de teste utilizado, focando em um caso próximo a casos reais, e por executar diversas vezes o teste em diferentes ambientes e explorando ou não o gerador de ruído.

Os objetos do estudo de caso foram o caso de teste em si, anteriormente apresentado, a aplicação a ser testada (o serviço simples de backup) e a falta que foi injetada nessa aplicação. A falta injetada consistia em esquecer propositalmente de incluir em um bloco sincronizado parte do código da thread de recuperação de backup. Dessa forma, quando solicitações concorrentes de recuperação de backup fossem feitas, era possível que uma fosse desprezada. Para isso, uma extensão da classe `RecuperadorDeBackup` foi provida, sendo esta denominada `RecuperadorDeBackupProblematico`. A instância do serviço de backup utilizada no teste foi configurada para utilizar essa instância de recuperador problemática. Isso só foi feito para que se pudesse ter uma versão de teste do sistema funcionando e uma outra do sistema com a falta injetada, mas a idéia na prática é que o teste não precisa ter acesso direto às instâncias de objetos representando threads internas.

8.4.5 Instrumentação

A instrumentação do estudo de caso para colher os valores das variáveis de resposta foi feita através de scripts que varriam as saídas das diversas re-execuções de testes feitas, sendo, para cada máquina utilizada na execução dos testes, metade usando a ferramenta ConTest e outra metade não. Nessa instrumentação verificava-se se o teste era falho ou não e a linha em que ocorria a falha (asserção que não passou em dada execução de teste que não passou).

8.4.6 Procedimento de coleta dos dados

Para cada um dos diferentes tratamentos (execução com e sem ConTest em cada uma das máquinas utilizadas), o caso de teste foi executado através de um script automático que repetia sua execução inicialmente 1.000 vezes, número este que foi aumentado em execuções futuras para 11.000, para garantir que com o aumento do tamanho da amostra o percentual de testes falhos em cada máquina convergia. As saídas geradas por cada execução de teste eram redirecionadas para um arquivo e o script de execução tratava de numerar estes arquivos para

que uma execução não sobrepusesse os resultados da outra.

O mesmo script foi utilizado em três máquinas diferentes considerando que o ConTest utilizava um critério aleatório de geração de ruído que poderia ser dependente da máquina. Exemplo disso ocorre com a geração de ruídos através do método `yield` da classe `Thread` que em algumas implementações da máquina virtual Java apresenta pouca influência no escalonamento das threads.

8.4.7 Procedimento de análise

A primeira análise consistiu em verificar se seria possível a execução do ConTest combinado com o `ThreadControl` visto que ambos se baseiam em instrumentação. Porém, como o ConTest introduz tal instrumentação simplesmente com o acréscimo de um parâmetro para a máquina virtual Java na execução do teste, não houve nenhum impedimento para o uso combinado das duas abordagens, ao menos com as ferramentas escolhidas para esse estudo de caso.

A análise posterior consistiu em investigar o número de falhas esperadas para o teste, representadas por falhas nas linhas de código 88 e 89 da classe de teste (equivalentes às linhas 61 e 62, respectivamente, do código apresentado anteriormente), em que se verifica se os arquivos anteriormente apagados foram recuperados com sucesso. Com a falha injetada, dependendo da ordem de execução das threads representando as requisições de usuários, um ou outro arquivo poderia não ser recuperado, fazendo com que o teste mostrasse falha em uma asserção ou na outra.

8.4.8 Execução

A execução dos testes utilizando os scripts previamente preparados foi feita em três máquinas distintas, todas com a máquina virtual Java (JVM) instalada e com a mesma versão das classes da aplicação e dos testes. Um dos scripts configurava a execução dos testes utilizando um argumento da JVM informando o uso do ConTest para gerar ruído na execução e adicionando também ao *classpath* da execução um arquivo `.jar` representando essa ferramenta. No mesmo diretório em que eram executados os testes com ConTest havia um arquivo de propriedades em que estavam definidos os pacotes a serem instrumentados pelo ConTest

(ou seja, os que continham a aplicação de backup e seus testes, especificados nesse arquivo através do pacote em que estas classes estão).

As configurações de hardware e software das três máquinas utilizadas estão descritas nas tabelas apresentadas a seguir. A Tabela 8.1 mostra as configurações da primeira máquina utilizada, um MacBook. A Tabela 8.2 mostra as configurações da segunda máquina, uma máquina virtualizada oferecida através do provedor de infra-estrutura rackspacecloud.com. A Tabela 8.3 mostra as configurações da terceira máquina, um desktop comum do Laboratório de Sistemas Distribuídos, que foi isolado em parte da rede, sendo possível o seu acesso apenas por uma máquina. As duas últimas máquinas ficaram dedicadas ao estudo de caso, não havendo nenhuma outra aplicação executando nelas. A primeira máquina, embora não estivesse dedicada, estava submetida à mesma carga de processamento ao longo de toda a execução do estudo.

Tabela 8.1: Configuração do Ambiente 1 para execução do estudo de caso: Macbook

Versão de Java	Java 1.6.0_20
Sistema Operacional	Mac OS X 10.6.4 (Snow Leopard)
Modelo da máquina	MacBook
Processador	2 GHz Intel Core 2 Duo
Memória	2 GB de RAM (1067 MHz DDR3) e 3MB de L2 Cache
HD	Macintosh HD (160 GB)

Em cada uma dessas máquinas foram rodados dois scripts, um que executava o teste diversas vezes, mas sem utilizar o ConTest e outro que executava o teste várias vezes, mas utilizando o ConTest.

As saídas geradas pela execução desses scripts foram armazenadas nos diretórios `saidaNormal` e `saidaContest` respectivamente.

8.4.9 Análise

Os dados coletados relativos às várias execuções de testes foram analisados por scripts que varriam os diretórios de saída descritos acima e imprimiam o número da execução de teste

Tabela 8.2: Configuração do Ambiente 2 para execução do estudo de caso: Máquina virtualizada

Versão de Java	Java 1.6.0_21
Sistema Operacional	Linux version 2.6.32.12-rscloud
Serviço de Virtualização	Xen Hypervisor
Processador	Quad-Core AMD Opteron(tm) Processor 2374 HE
Memória	2 GB de RAM
Espaço de Disco Alocado	80GB

Tabela 8.3: Configuração do Ambiente 3 para execução do estudo de caso: Desktop LSD

Versão de Java	Java 1.6.0_20
Sistema Operacional	Linux 2.6.24-1-686 (Apr 19) libc6-i686 (2.7-10) Debian Lenny (5.0)
Modelo da máquina	HP Compaq dc5100 SFF(ED514LA)
Processador	Intel(R) Pentium(R) 4 CPU de 3.00GHz (XU1) L2 1MiB
Memória	1.5 GiB de RAM (1548220 kB) (512 MiB e 1 GiB)
HD	SAMSUNG HD080HJ/P (80 GB) SATA

que falhou, para cada falha encontrada, especificando a linha de código do teste associada à falha. Além disso, esses scripts imprimiam ao final o total de testes com falhas, o total de testes com falha na linha 88 (equivalente à asserção da linha 61 do código anteriormente mostrado) e o total de testes com falha na linha 89 (equivalente à asserção da linha 62 do código de teste mostrado), ambas falhas esperadas, e também o total de testes que passou dentre os que foram executados.

Para uma análise mais detalhada, essas informações foram passadas para um outro programa responsável por gerar a quantidade de falhas totais a cada execução e como estas iam crescendo à medida que se aumentava a amostra considerada. Esse analisador gerava também o percentual de falhas do teste com o aumento do tamanho da amostra considerada.

Considerando as 11.000 execuções de cada tipo de rodada de teste, para cada um dos três

ambientes considerados, foram encontrados os dados apresentados nas tabelas a seguir. A Tabela 8.4 descreve as execuções utilizando o MacBook, a Tabela 8.5 descreve as execuções utilizando a máquina virtualizada e a Tabela 8.6 descreve as execuções de teste utilizando o desktop.

Tabela 8.4: Resultados de Execuções de Teste no Ambiente 1 (MacBook)

MACBOOK_ConTest	Numero Execuções	Percentual das Execuções
Total ok	8736	79.42%
Total falhas88	371	3.37%
Total falhas89	1893	17.21%
Total falhas	2264	20.58%

MACBOOK_Normal	Numero Execuções	Percentual das Execuções
Total ok	10986	99.87%
Total falhas88	0	0.00%
Total falhas89	14	0.13%
Total falhas	14	0.13%

Tabela 8.5: Resultados de Execuções de Teste no Ambiente 2 (Máquina Virtualizada)

VIRTUAL_ConTest	Numero Execuções	Percentual das Execuções
Total ok	10312	93.75%
Total falhas88	60	0.55%
Total falhas89	628	5.71%
Total falhas	688	6.25%

VIRTUAL_Normal	Numero Execuções	Percentual das Execuções
Total ok	10976	99.78%
Total falhas88	0	0.00%
Total falhas89	24	0.22%
Total falhas	24	0.22%

Tabela 8.6: Resultados de Execuções de Teste no Ambiente 3 (Desktop)

DESKTOP_ConTest	Numero Execuções	Percentual das Execuções
Total ok	9980	90.73%
Total falhas88	103	0.94%
Total falhas89	917	8.34%
Total falhas	1020	9.27%

DESKTOP_Normal	Numero Execuções	Percentual das Execuções
Total ok	9987	90.79%
Total falhas88	14	0.13%
Total falhas89	999	9.08%
Total falhas	1013	9.21%

Para se verificar se os percentuais obtidos representavam um valor apropriado para o percentual médio de falhas dentre o total de execuções, foram gerados os gráficos referentes ao número de falhas à medida que aumentava o número de execuções e também do

comportamento do percentual das falhas com o aumento do número de execuções de testes consideradas. Estes gráficos estão apresentados no final deste capítulo. A Figura 8.3(a) e a Figura 8.3(b) representam os resultados para o ambiente 1, a Figura 8.4(a) e a Figura 8.4(b) representam os resultados para o ambiente 2 e, a Figura 8.5(a) e a Figura 8.5(b) representam os resultados para o ambiente 3.

Através dos percentuais de falha observados nas diferentes máquinas pode-se ver o quanto varia a probabilidade da mesma versão de teste de uma mesma aplicação poder explicitar uma falta no software através de uma falha em seu teste dependendo da máquina utilizada. Considerando a execução normal do teste, sem utilizar o ConTest, vê-se que no Ambiente 1 as falhas esperadas para os testes devido à falta injetada só ocorreram em 0.13% dos casos, já no Ambiente 2, elas ocorreram em 0.22% dos casos, enquanto no Ambiente 3, mesmo sem nenhum gerador de ruído externo, elas ocorreram em 9.21% dos casos. Percebe-se, porém, que a falha da linha 88 do teste não ocorreu nenhuma vez nos dois primeiros ambientes e ocorreu apenas 14 vezes das 11.000 execuções no terceiro ambiente.

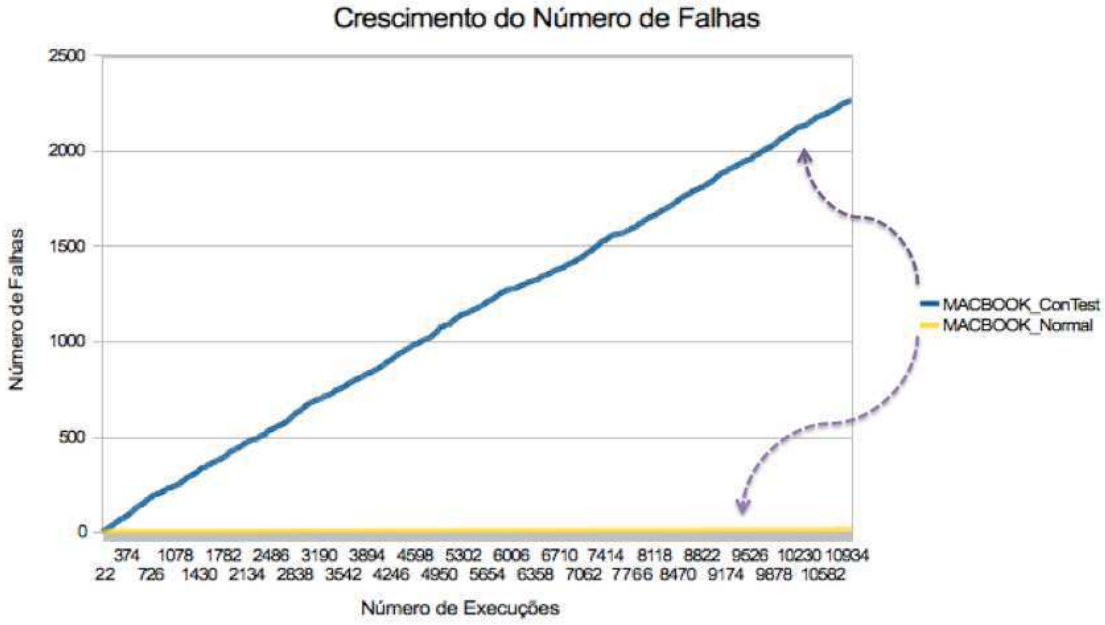
Ao utilizar o ConTest aumentou-se consideravelmente a quantidade de vezes em que essa falha 88 se revelava, passando a ocorrer 103 vezes no Ambiente 3, 371 vezes no Ambiente 1 e 60 vezes no Ambiente 2. O número de falhas em geral, nos dois primeiros ambientes foi maior nas execuções utilizando ConTest, passando para 20.58% no Ambiente 1 e para 6.25% no Ambiente 2. No Ambiente 3, em que o percentual de falhas detectadas já era considerável, o uso de ConTest não trouxe ganhos considerando-se o percentual total de execução de testes que falhava. Porém, esse uso fez com que aumentasse o índice de ocorrência de uma das falhas que dificilmente se manifestava na execução normal (a falha da linha 88 do teste, em que a falta introduzida ocorreu considerando um escalonamento diferente das threads). A diferença do Ambiente 3 e que justifica um maior índice de ocorrência das falhas mesmo sem o ConTest comparada com essa execução em outras máquinas é que esta máquina é a única dentre as consideradas que não possui um processador de múltiplos núcleos, o que diminui as diferentes possibilidades de escalonamentos possíveis. Os testes normais e com ConTest foram repetidos nessa máquina e resultados similares foram encontrados.

8.5 Conclusões Parciais

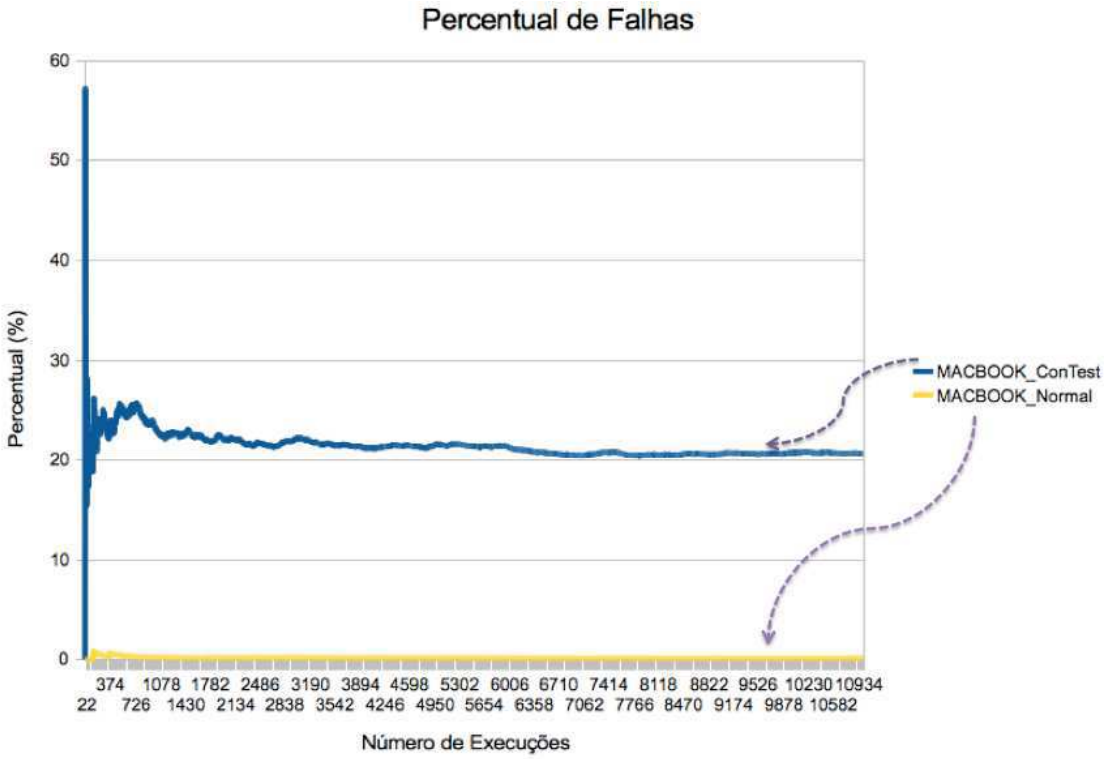
Considerando o estudo de caso apresentado nesse capítulo, conclui-se que ele contribuiu para demonstrar que é possível combinar a abordagem *Thread Control for Tests* com geradores de ruído. Isso foi demonstrado ilustrando um caso de uso em que se usou o arcabouço de testes *ThreadControl* e a ferramenta *ConTest*. Esse resultado é importante considerando que a abordagem proposta neste trabalho de tese pode dar suporte ao desenvolvimento de testes a serem usados por ferramentas como o ConTest, que têm como objetivo a re-execução de códigos aumentando a probabilidade de expor defeitos de concorrência.

O estudo apresentado neste capítulo também mostrou que o uso de geradores de ruído fez com que aumentasse a probabilidade de ocorrência de uma falha esperada no teste nos ambientes deste estudo em que a probabilidade de manifestação da falta injetada no estudo era menor que 1% (como nos dois primeiros ambientes do estudo de caso). O estudo também mostrou que dependendo do ambiente, o uso de geradores de ruído pode não influenciar na frequência com que uma dada falta se manifesta através de uma falha de teste, o que foi visto através dos resultados obtidos para o Ambiente 3. No entanto, o gerador de ruído proporcionou maiores chances de diferentes escalonamentos serem testados, já que por mais vezes a falha de teste ocorreu na linha 88. É importante destacar que estes resultados representam o que foi detectado nos ambientes estudados e para que possam ser generalizados, experimentos exaustivos e que explorem inclusive outros ambientes e mais tipos de casos de teste são necessários.

Figura 8.2: Análise do comportamento das falhas no Ambiente 1

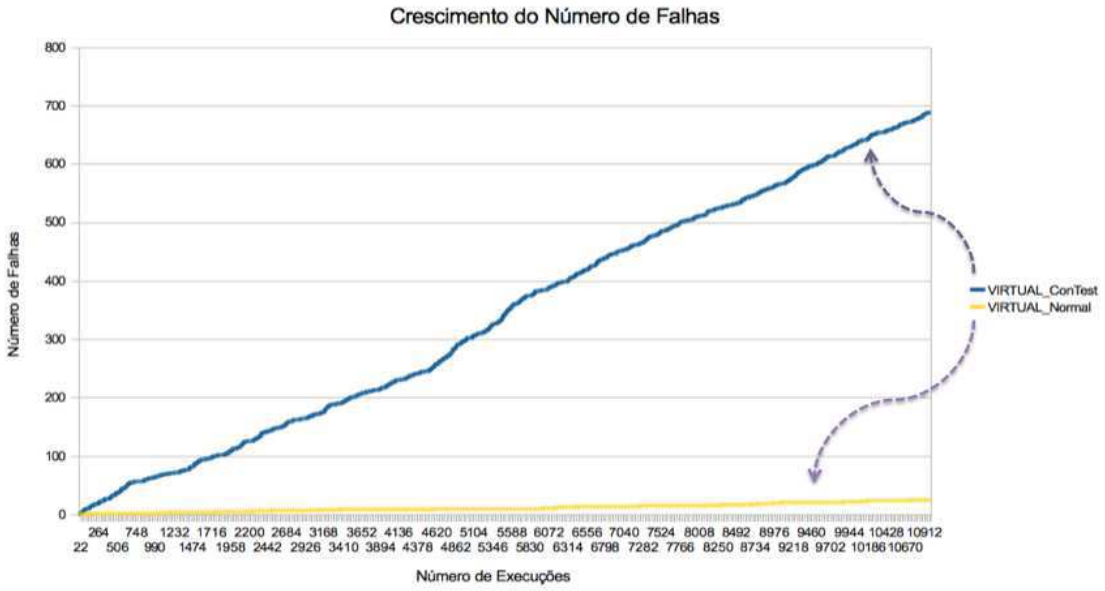


(a) Falhas em Execuções no Ambiente 1 (MacBook)

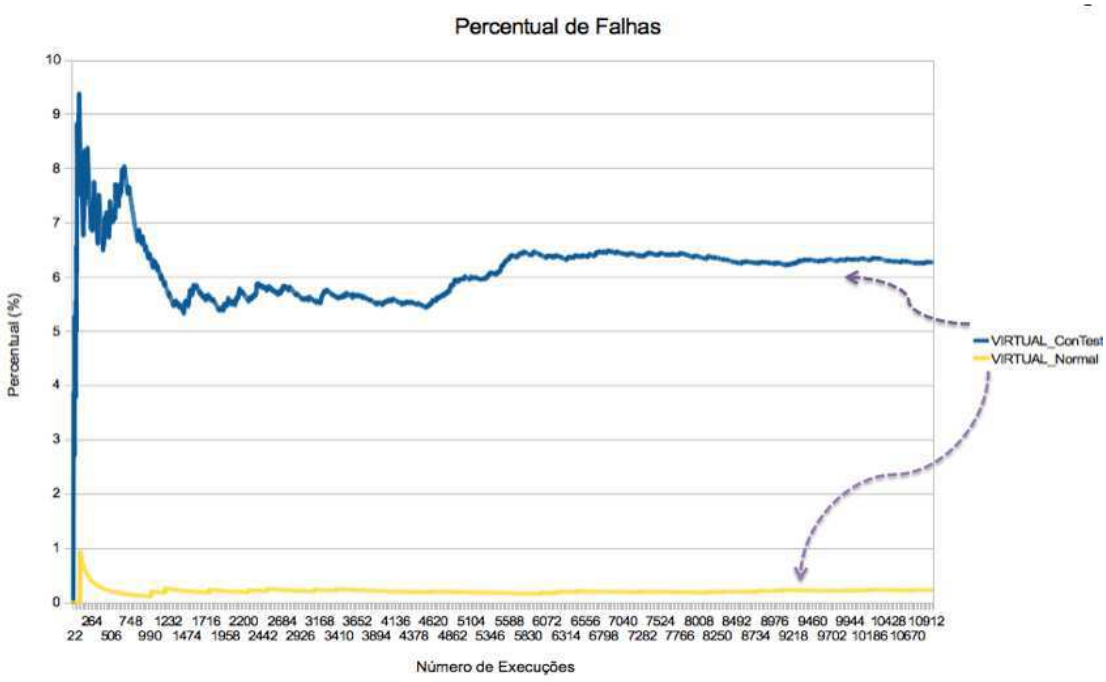


(b) Percentual de Falhas em Execuções no Ambiente 1 (MacBook)

Figura 8.3: Análise do comportamento das falhas no Ambiente 2

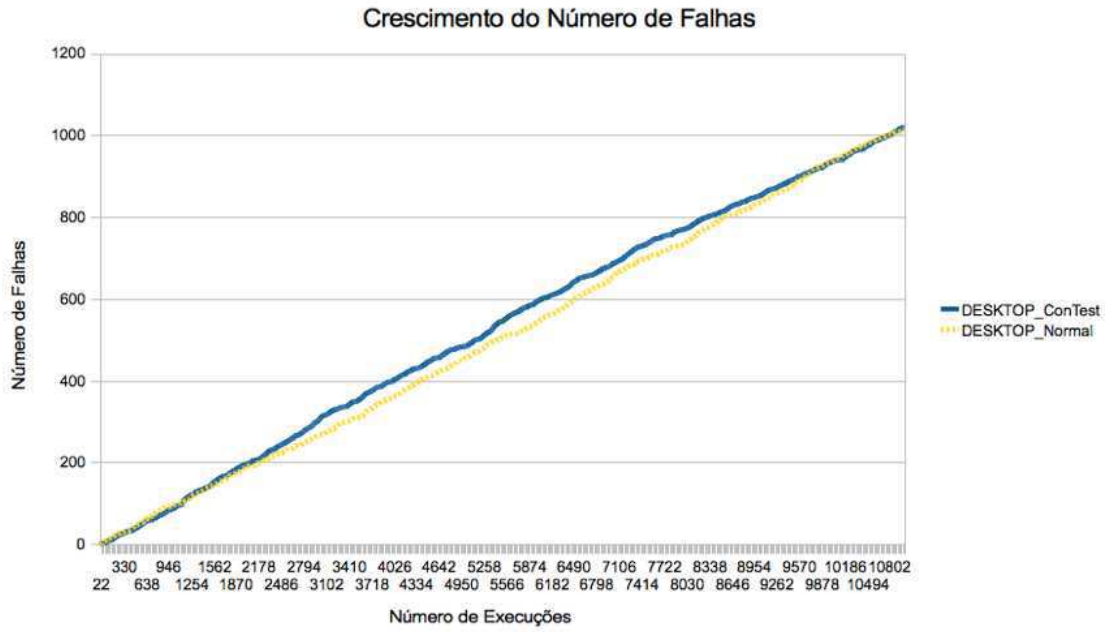


(a) Falhas em Execuções no Ambiente 2 (Virtual)

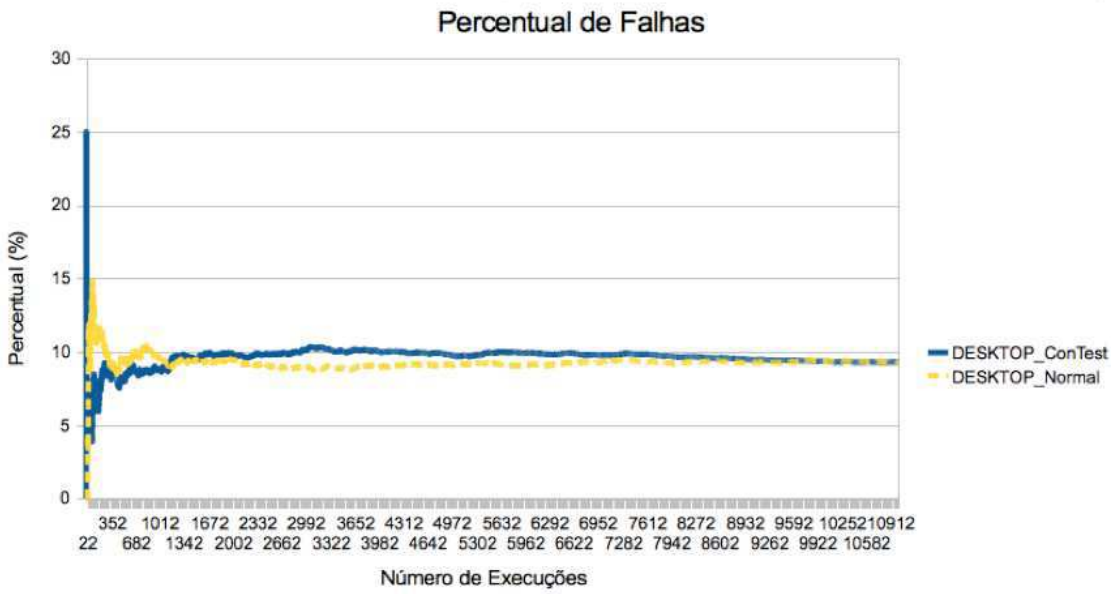


(b) Percentual de Falhas em Execuções no Ambiente 2 (Virtual)

Figura 8.4: Análise do comportamento das falhas no Ambiente 3



(a) Falhas em Execuções no Ambiente 3 (Desktop)



(b) Percentual de Falhas em Execuções no Ambiente 3 (Desktop)

Capítulo 9

Considerações Finais

Neste capítulo são sintetizados os principais resultados obtidos neste trabalho e as sugestões de trabalhos futuros.

9.1 Conclusões

O objetivo geral desta tese foi o de prover mecanismos aos testadores para evitar que sejam implementados testes que falhem por asserções feitas em momentos inapropriados, evitando-se assim testes não confiáveis (que levem a falsas suspeitas sobre o sistema sendo testado).

Através de modelagem formal e de estudos de caso práticos demonstrou-se que não é possível evitar falhas em testes com assincronia em sistemas multi-threaded causadas por asserções feitas em momentos não apropriados sem monitorar e controlar as threads do sistema sob teste. Viu-se também que através de técnicas como Programação Orientada a Aspectos ou instrumentação é possível monitorar e controlar as threads do sistema testado com pouca interferência em tal sistema sob o ponto de vista da necessidade de alteração de seu código.

A demonstração formal apresentada no Capítulo 4 teve como intuito principal deixar claro que sem monitoração e controle parcial das threads, como é o caso de testes utilizando técnicas comuns como atrasos explícitos e espera ocupada, não se pode garantir a ausência de falsos positivos por asserções antecipadas e tardias. Tal resultado tem como propósito principal desestimular o uso de tais técnicas quando se tem a exigência de testes confiáveis.

Além da demonstração da existência do problema, apresentou-se também os requisitos para uma solução que resolvesse o problema, através de teoremas que foram demonstrados

no Capítulo 4 e das regras de implementação ali apresentadas.

Levando em consideração tais requisitos, apresentou-se uma abordagem para o teste de sistemas envolvendo operações assíncronas e baseada nos requisitos formalmente modelados, a abordagem *Thread Control for Tests*, descrita no Capítulo 5. Para demonstrar que ela poderia ser utilizada na prática e sem afetar no desenvolvimento do sistema sob teste sob o ponto de vista da necessidade de alterações de código neste, foi desenvolvido o arcabouço de testes *ThreadControl*. Este arcabouço foi implementado utilizando o paradigma de Programação Orientada a Aspectos. Tal arcabouço foi utilizado nos estudos de caso apresentados nesta tese e que serviram para demonstrar o uso prático da abordagem e também para dar indícios de que seu uso através do arcabouço *ThreadControl* tem conseguido evitar falhas devidas a asserções feitas cedo ou tarde demais.

Conseguiu-se também através deste trabalho demonstrar, utilizando a Lógica Temporal de Ações através da linguagem TLA+, que testes utilizando a abordagem *Thread Control for Tests* não apresentam o problema das asserções antecipadas e tardias. A especificação formal dos testes utilizando essa linguagem pôde ser verificada com ferramentas de apoio a TLA+ que servem para fazer checagens nos modelos preparados com esta linguagem. A avaliação da abordagem utilizando TLA+ foi descrita no Capítulo 7.

Por fim, esse trabalho também apresentou uma investigação inicial do uso combinado da abordagem *Thread Control for Tests* com geradores de ruído, por meio da ferramenta ConTest [2]. Essa investigação, apresentada no Capítulo 8, consistiu em um estudo de caso em que se projetou uma aplicação concorrente simples, mas próxima a um cenário real e se injetou uma falta na aplicação que deveria se manifestar por meio de falhas em apenas algumas das execuções dos testes. Tal falta consistia em não deixar sincronizado um bloco de código que precisava ser. Este estudo teve como propósitos tanto investigar a viabilidade da aplicação conjunta da abordagem aqui apresentada com outras técnicas utilizadas em testes concorrentes, como também investigar se o efeito de observação introduzido pela monitoração de threads pode ser minimizado com o uso de técnicas como geradores de ruído. Os resultados desse estudo mostraram que o uso de geradores de ruído proporcionou maiores chances de diferentes escalonamentos serem testados, incluindo os que levavam o teste a falhar por exporem a falta introduzida.

Sendo assim, considera-se que o trabalho conseguiu alcançar seus objetivos e que a solu-

ção geral modelada formalmente e a abordagem apresentada resolvem o problema dos testes que falham devido a asserções antecipadas e tardias, auxiliando o trabalho dos testadores de sistemas multi-threaded. Além disso, através dos estudos de caso apresentados, demonstrou-se que a abordagem pode ser usada na prática para resolver problemas reais em testes. No entanto, é importante que sejam feitos outros estudos práticos de uso da abordagem para que testadores de sistemas multi-threaded possam ser mais facilmente convencidos de sua relevância na sua vida prática. É interessante também evoluir o arcabouço *ThreadControl* para que mais aplicações possam utilizá-lo caso estas demandem outros pontos de monitoração além dos atualmente considerados. É importante também que se avalie o uso da abordagem *Thread Control for Tests* com outras ferramentas de propósito semelhante ao do ConTest e que se utilizam de testes existentes para identificar problemas de concorrência em aplicações multi-threaded. Tais avaliações futuras têm o intuito de fortalecer o resultado de que ferramentas assim podem se beneficiar da abordagem aqui apresentada visto que se basearão em testes mais confiáveis.

Diante do exposto, esta tese traz as seguintes contribuições:

1. Levantamento dos principais trabalhos relacionados ao problema atacado nesta tese e que foram descritos no Capítulo 3;

2. Demonstração formal da seguinte propriedade: sem a monitoração e controle parcial das threads (evitando algumas de suas transições de estado), não é possível evitar o problema das asserções antecipadas e tardias e demonstração de que são necessários alguns requisitos mínimos para uma solução que resolva este problema, o que foi descrito no Capítulo 4;

3. Uma abordagem geral para a implementação de testes assíncronos, denominada *Thread Control For Tests*, baseada em primitivas oferecidas aos desenvolvedores de testes, e que foi descrita na Seção 5.2;

4. Demonstração utilizando a linguagem TLA+ de que testes utilizando a abordagem *Thread Control for Tests* não apresentam o problema das asserções antecipadas e tardias, conforme detalhado no Capítulo 7;

5. Uma ferramenta, o *ThreadControl*, para auxiliar no desenvolvimento de testes que

seguem a abordagem apresentada, disponibilizada em `http://code.google.com/p/threadcontrol`;

6. A descrição da arquitetura e de alguns detalhes de implementação do *ThreadControl*, como forma de auxiliar no desenvolvimento de outras ferramentas para diferentes linguagens de programação (ver Seção 5.3);

7. Resultados de estudos de caso envolvendo o uso da ferramenta *ThreadControl* (ver Capítulo 6) que mostram que nos estudos feitos os testes que utilizavam esta ferramenta não falhavam ou suas falhas de execução não se enquadravam nas que caracterizavam asserções antecipadas e tardias;

8. Um estudo de caso em que se analisou o uso combinado da ferramenta *ThreadControl* com a ferramenta *ConTest*, uma ferramenta capaz de gerar ruídos para execução de testes (ver Capítulo 8), e que demonstrou que tal uso da abordagem com outras ferramentas como o ConTest que auxiliem na detecção de defeitos de concorrência através de testes existentes é possível e que tais ferramentas serão beneficiadas com a possibilidade de poderem utilizar testes mais confiáveis. Além disso, o uso de *ConTest* no estudo em geral aumentou a frequência com que determinados escalonamentos que revelavam a falta aconteciam em re-execuções dos testes.

9.2 Trabalhos Futuros

Os trabalhos propostos como atividades a serem feitas posteriormente para dar continuidade a este trabalho são os seguintes:

- *Realizar mais avaliações práticas de uso da abordagem para incentivar seu uso por mais testadores*. Embora em um dos estudos de caso se tenha utilizado um sistema real, que no caso foi o OurBackup, é importante que mais avaliações práticas sejam feitas utilizando o arcabouço de testes *ThreadControl* de modo a demonstrar seu uso em diferentes sistemas e avaliar os custos desse uso (como dificuldade de uso, linhas de código, horas de desenvolvimento, etc) e os benefícios que pode trazer (como economia de tempo de desenvolvimento deixando-se de depurar problemas que não eram

da aplicação mas sim dos testes). Em diferentes apresentações deste trabalho, pessoas de empresas que estavam na platéia revelaram encontrar comumente o problema dos falsos positivos por asserções antecipadas e tardias e demonstraram-se interessadas em aplicar a abordagem. A idéia desses estudos é investigar a frequência com que tais falhas acontecem em testes e como o uso da abordagem *Thread Control for Tests* pode evitar tais ocorrências fazendo com que as falhas em execuções de testes realmente sinalizem defeitos nos sistemas sendo testados e não sejam falsos positivos.

- *Evoluir o arcabouço ThreadControl e melhorar a sua documentação para que possa mais facilmente ser utilizado na prática*. Atualmente a única documentação existente para o uso do arcabouço *ThreadControl* são os artigos produzidos e esta tese, o que não torna simples sua aplicação real sem o suporte de alguém que já utilizou o arcabouço. A disponibilização de uma documentação *online* com os requisitos e exemplos de uso do arcabouço pode ser de grande valia para quem o for aplicar. Além disso, para possibilitar definições mais flexíveis de estados das threads pelos quais o teste precisa esperar, é interessante que se estenda o arcabouço de forma a oferecer uma semântica mais rica para a definição desses estados. Atualmente, a única implementação existente da interface `SystemConfiguration`, utilizada para definir um estado esperado para o sistema, é a classe `ListofThreadConfigurations`. Ela não permite que sejam especificadas diferentes possibilidades de estado utilizando operadores lógicos como `AND` e `OR`, mas simplesmente que se especifiquem identificadores de threads (que atualmente são os nomes das classes que implementam a interface `Runnable` de Java), o estado em que threads identificadas dessa maneira devem estar, e a quantidade de vezes que devem passar por aquele estado (caso este seja um requisito importante, senão se esperará que todas as threads identificadas dessa forma estejam no estado indicado). Uma outra evolução proposta para o arcabouço `ThreadControl` consiste em fazer com que outros pontos de corte da aplicação sob teste possam ser monitorados e representem transições de estado. Exemplos de tais pontos são chamadas de métodos de objetos pertencentes a outras classes da biblioteca de concorrência introduzida na versão 5 da linguagem Java. Outros pontos que não estão sendo monitorados e que poderiam ser caso seja de interesse para os testadores são os momentos em que se adquire ou se libera trancas (*locks*). Para a monitoração nesse último caso pode-se

utilizar uma extensão proposta para AspectJ [12] e apresentada no ISSTA'08 e que introduziu novos pontos de corte que permitem essa monitoração: `lock()`, `unlock()` e `maybeShared()`.

- *Realizar experimentos e outros estudos de caso demonstrando o uso da abordagem combinada com outras ferramentas de apoio ao teste de sistemas concorrentes, como ferramentas com foco na detecção de problemas de concorrência e ferramentas de replay*. Alguns dos trabalhos na área de testes de sistemas concorrentes focam na detecção de problemas de concorrência em sistemas. Algumas vezes esses trabalhos se baseiam em verificação de modelos que não estão expressos em linguagens de programação. Porém, alguns dos trabalhos, como é o caso dos que utilizam o Contest [2] ou a ferramenta Verisoft [41], ou ainda outros trabalhos baseados em escalonamento aleatório [86][90][30] durante a execução de testes podem se beneficiar do trabalho aqui apresentado. Isso ocorre pois tais técnicas dependem de testes confiáveis. Mais estudos que demonstrem maior eficiência no uso dessas ferramentas quando a abordagem apresentada neste trabalho é utilizada na preparação dos testes utilizados podem fortalecer os resultados desta tese e incentivar o uso de tais resultados no dia a dia dos desenvolvedores de testes de sistemas multi-threaded.

- *Avaliar o uso da abordagem no contexto de testes distribuídos, estendendo o arcabouço inicial para uma versão denominada Distributed ThreadControl.* Parte dos sistemas concorrentes que são desenvolvidos atualmente são compostos por vários processos que podem ou não estar distribuídos em diferentes máquinas. A versão atual do arcabouço *ThreadControl* só contempla testes envolvendo um único processo, mas acredita-se que ela pode ser estendida para testes com múltiplos processos. Embora se acredite que a abordagem *Thread Control for Tests* possa ser aplicada com pequenos ajustes para sistemas distribuídos, é importante que isso seja investigado através de outros estudos. A arquitetura básica planejada para esta versão estende a arquitetura básica do *ThreadControl* ilustrada pela Figura 5.3. A idéia geral também é de uma fachada de serviços de teste oferecendo aos testadores primitivas que lhes permitam especificar estados de espera interessantes e fazer com que o teste espere até que tais estados sejam alcançados. A idéia básica do primeiro esboço de arquitetura do *Dis-*

tributed ThreadControl está ilustrada pela Figura 9.1. A diferença básica nesta nova arquitetura é que a especificação dos estados esperados seria dada em termos de estados de componentes (internamente mapeáveis em estados comuns de suas threads em momentos de espera de testes) e também o fato de que para cada processo sendo monitorado haveria uma réplica dos componentes descritos no quadrado superior da figura. Além disso, os "Aspectos de monitoração e controle" ganhariam a função extra de conhecer o "Monitor do Sistema Distribuído" informando-se junto a este, no momento da inicialização de componentes sobre os estados esperados relativos àquele componente e notificando tal monitor quando tal estado fosse alcançado. Vários desafios adicionais terão de ser tratados nessa versão do arcabouço, como a forma de lidar com problemas de atraso na comunicação, a necessidade de monitoração do envio e recebimento de mensagens, a inexistência de memória compartilhada e a existência de falhas parciais.

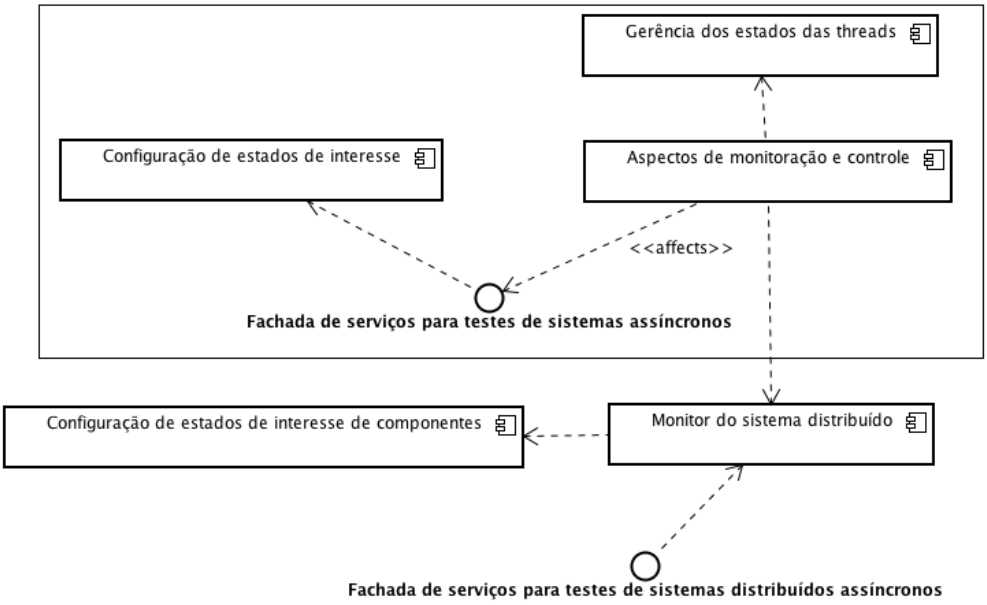


Figura 9.1: Arquitetura inicial proposta para o arcabouço *Distributed ThreadControl*

- *Investigar os ganhos em produtividade que times de desenvolvimento podem ter ao utilizar abordagens que evitam falsos positivos em testes.* A idéia de estudos dessa natureza é investigar o tempo desperdiçado na investigação de possíveis defeitos no sistema sinalizados através de testes que apresentam falsos positivos e comparar este

tempo com o tempo necessário para se aplicar a abordagem *Thread Control for Tests* no desenvolvimento de testes de sistemas existentes. Para este estudo é importante considerar times de desenvolvimento que utilizem ferramentas que coletem métricas relativas ao tempo de desenvolvimento de suas atividades (`time trackers`) para que se possam obter resultados quantitativos que demonstrem que vale a pena investir em testes mais confiáveis e que não demandam alterações no código da aplicação sob teste.

- *Submeter o artigo de jornal já escrito e escrever outros artigos referentes ao conteúdo deste trabalho ainda não publicado.* Para aumentar a visibilidade do trabalho e contribuir de fato para a diminuição do problema dos falsos positivos em testes por asserções antecipadas e tardias, é importante que os resultados aqui apresentados e os que se pretende alcançar com os trabalhos futuros acima propostos sejam mais amplamente divulgados em meios como jornais científicos e conferências na área de testes e engenharia de software. O formalismo feito para validar a abordagem ainda não foi publicado e serve para fortalecer os resultados até então publicados.

Referências Bibliográficas

[1] Chord. At http://code.google.com/p/jchord/.

[2] ConcurrentTesting - Advanced Testing for Multi-Threaded Applications. At http://www.alphaworks.ibm.com/tech/contest.

[3] ConTest - A Tool for Testing Multi-threaded Java Applications. At https://www.research.ibm.com/haifa/projects/verification/contest/.

[4] FindBugs - Find Bugs in Java Programs . At http://findbugs.sourceforge.net/.

[5] JUnit testing framework. http://www.junit.org.

[6] NUnit. http://nunit.org.

[7] Thread weaver). At http://code.google.com/p/thread-weaver/.

[8] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages –, Dec 1990.

[9] V.R. Basili, G. Caldiera, and H.D. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 1:528–532, 1994.

[10] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Noise makers need to know where to be silent-producing schedules that find bugs. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*, 2006.

[11] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[12] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *Proceedings of International Conference on Software Testing and Applications (ISSTA'08)*, pages 155–165. ACM, 2008.

[13] X. Cai and J. Chen. Control of Nondeterminism in Testing Distributed Multithreaded Programs. *apaqs*, 00:29, 2000.

[14] R. H. Carver and K.C. Tai. Replay and testing for concurrent programs. *Software, IEEE*, 8(2):66–74, 1991.

[15] J.D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, 1998.

[16] W. Cirne, F. Brasileiro, N. Andrade, L.B. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.

[17] E. M. Clarke. Model cheking. In *Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, London, UK, 1997. Springer-Verlag.

[18] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[19] R. Coelho, A. Dantas, U. Kulesza, W. Cirne, A. von Staa, and C. Lucena. The Application Monitor Aspect Pattern. In *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs*, pages 1–10, New York, NY, USA, 2006. ACM.

[20] S. Copty and S. Ur. Multi-threaded Testing with AOP is Easy, and It Finds Bugs. *Proc. 11th International Euro-Par Conference, LNCS*, 3648:740–749, 2005.

[21] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM.

[22] A. Dantas. Improving developers' confidence in test results of multi-threaded systems: avoiding early and late assertions. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 899–900, New York, NY, USA, 2008. ACM.

[23] A. Dantas. Improving the Dependability of Tests Involving Asynchronous Operations. In *Proceedings of The Student Forum at LADC 2009*, pages A8–A10, 2009.

[24] A. Dantas, F. Brasileiro, and W. Cirne. Improving automated testing of multi-threaded software. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 521–524, Washington, DC, USA, 2008. IEEE Computer Society.

[25] A. Dantas, W. Cirne, and K. Saikoski. Using AOP to Bring a Project Back in Shape: The OurGrid Case. *Journal of the Brazilian Computer Society (JBCS)*, 11:21–36, 2006.

[26] A. Dantas, M. G., F. Brasileiro, and W. Cirne. Obtaining trustworthy test results in multi-threaded systems. In *SBES 2008: Proceedings of the XXII Simpósio Brasileiro de Engenharia de Software*, October 2008.

[27] E.W. Dijkstra. *Notes on structured programming*. Technological University Eindhoven, Netherlands Department of Mathematics, [Eindhoven], 1969.

[28] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, 1999.

[29] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice & Experience*, 15(3):485–499, 2003.

[30] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[31] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing Aspects of AOP. *Communications of the ACM*, 44(10):33–38, October 2001.

[32] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004. Adviser-Schneider,, Fred B.

[33] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. *Concurrency and Computation: Practice and Experience*, 2006.

[34] The Apache Software Foundation. The apache ant project. http://ant.apache.org/.

[35] The Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/.

[36] The Apache Software Foundation. Hadoop. http://hadoop.apache.org/core/.

[37] J. Gait. A probe effect in concurrent programs. *Software—Practice & Experience*, 16(3):225–233, 1986.

[38] M. Gaudencio. Estudo sobre Técnicas de Espera Utilizadas em Testes envolvendo Assincronia. Technical report, Relatório de Estágio Integrado, DSC / UFCG, 2009.

[39] B. Gaudin and H. Marchand. Supervisory control of product and hierarchical discrete event systems. *European Journal of Control*, 10(2), 2004.

[40] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall Inc, 1991.

[41] P. Godefroid. Model checking for programming languages using verisoft. In *In Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

[42] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[43] B. Goetz. Testing Concurrent Programs, 2006. At http://www.theserverside.com/tt/articles/article.tss?l=TestingConcurrent.

[44] B. Goetz and T. Peierls. *Java concurrency in practice*. Addison-Wesley, 2006.

[45] J. Gray. Why do computers stop and what can be done about it. In *Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[46] M.J. Harrold. Testing: a roadmap. *Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, 2000.

[47] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[48] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM.

[49] JBoss. Jboss enterprise application platform features. http://www.jboss.com/products/platforms/application/features.

[50] A. Jedlitschka and D. Pfahl. Reporting guidelines for controlled experiments in software engineering. In *International Symposium on Empirical Software Engineering*, pages 92–101, 2005.

[51] P. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, 2002.

[52] N. Juristo and A.M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, 2001.

[53] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[54] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer–Verlag.

[55] B. Kitchenham, L. Pickard, and SL Pfleeger. Case studies for method and tool evaluation. *Software, IEEE*, 12(4):52–62, 1995.

[56] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[57] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[58] L. Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[59] L. Lamport, J. A. Akinyemi, and M. Menarini. Temporal logic of actions (formal logic course), January 2006. At http://www.sts.tu-harburg.de/r̃.f.moeller/lectures/lgris-ws-07-08/LGRIS-10.pdf.

[60] D. Lea. *Concurrent Programming in Java (tm): Design Principles and Patterns*. Addison-Wesley, 2000.

[61] G.T. Leavens and Y. Cheon. Design by Contract with JML, 2006. At http://jmlspecs.org.

[62] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *Software Engineering, IEEE Transactions on*, 29(6):555–566, 2003.

[63] LSD. Laboratório de sistemas distribuídos - lsd - ufcg. http://lsd.ufcg.edu.br.

[64] S. MacDonald, J. Chen, and D. Novillo. Choosing Among Alternative Futures. *Proc. Haifa Verification Conference, LNCS*, 3875:247–264, 2005.

[65] V. Massol. *JUnit in Action*. Manning, 2004.

[66] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[67] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[68] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[69] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.

[70] S. Moon and B. M. Chang. A thread monitoring system for multithreaded Java programs. *ACM SIGPLAN Notices*, 41(5):21–29, 2006.

[71] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI '08*, 2008.

[72] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM Press.

[73] P. Nienaltowski, B. Meyer, and J.S. Ostroff. Contracts for concurrency. *Formal Aspects of Computing*, pages 1–14, 2008.

[74] Y. Nir-Buchbinder and S. Ur. Multithreaded unit testing with ConTest, April 2006. At http://www.ibm.com/developerworks/java/library/j-contest.html.

[75] M. I. S. Oliveira. OurBackup: Uma Solução P2P de Backup Baseada em Redes Sociais. Master's thesis, Universidade Federal de Campina Grande, 2007.

[76] C. Otaku. Testing asynchronous code, February 2005. At http://beust.com/weblog/archives/000236.html.

[77] R. Patton. *Software Testing*. Sams. 2nd edition, 2005.

[78] S. L. Pfleeger. Experimental design and analysis in software engineering: Part 2: How to set up an experiment. *SIGSOFT Softw. Eng. Notes*, 20(1):22–26, 1995.

[79] S.L. Pfleeger and J.M. Atlee. *Software engineering: theory and practice*. Prentice hall, 2001.

[80] W. Pugh and N. Ayewah. Unit testing concurrent software. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 513–516, 2007.

[81] A.D. Rocha, A.S. Simão, J.C. Maldonado, and P.C. Masiero. Uma ferramenta baseada em aspectos para o teste funcional de programas Java. In *Simpósio Brasileiro de Engenharia de Software, SBES 2005*, Uberlândia-MG, October 2005.

[82] E. Rodríguez, M. B. Dwyer, C. F., J. Hatcliff, and G. T. Leavens. Extending jml for modular specification and verification of multi-threaded programs. In *ECOOP*, pages 551–576, Glasgow, UK, July 2005.

[83] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999.

[84] P. Santos and F. J. Garcia. Distributed Unit Testing: Supporting multiple platforms accurately and efficiently, October 2006. http://www.ddj.com/architect/193104810.

[85] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[86] K. Sen. Effective random testing of concurrent programs. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332, 2007.

[87] A. Silberschatz, P. B. Galvin, and G. Gagne. *Sistemas Operacionais com Java*. Elsevier, 2008.

[88] I. Sommerville. *Software Engineering*. Addison-Wesley, 2006.

[89] K. Stobie. Too darned big to test. *Queue*, 3(1):30–37, 2005.

[90] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

[91] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[92] P. Tarr and H. Ossher. Advanced separation of concerns in software engineering. In *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001*, 2001. At http://www.research.ibm.com/hyperspace/workshops/icse2001.

[93] The AspectJ Team. The AspectJ Programming Guide. At http://www.eclipse.org/aspectj.

[94] The Open Group. Synchronization - The Key to Distributed Testing (TETware White Paper). http://tetworks.opengroup.org/Wpapers/synchronization.htm.

[95] The Open Group. TETware User Guide. http://tetworks.opengroup.org/documents/3.7/uguide.pdf.

[96] The Open Group. TETware White Paper. http://tetworks.opengroup.org/Wpapers/TETwareWhitePaper.htm.

[97] G.H. Travassos, D. Gurov, and E.A.G. Amaral. Introdução à Engenharia de Software Experimental. Technical report, Relatório Técnico RT-ES-590/02, COPPE / UFRJ, 2002.

[98] H. Ural and D. Whittier. Distributed testing without encountering controllability and observability problems. *Information Processing Letters*, 88(3):133–141, 2003.

[99] W. Visser, K. Havelund, G. Brat, and SJ Park. Model checking programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[100] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An Initial Assessment of Aspect-Oriented Programming. In *Proceedings of the 21st International Conference on Software Engineering*, pages 120–130. IEEE Computer Society Press, 1999.

[101] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber. *The Java Tutorial: A Short Course on the Basics*. Prentice-Hall, 2006.

Apêndice A

Especificação em TLA+ de Execução de Testes sem Monitoração

MODULE *TestExecution*

This module contains the definition of the specification of a test execution regarding the threads of the System Under Test and the test thread

CONSTANT $ThreadsPossibleStates$, $unstarted$,

$started$, $waiting$, $running$, $finished$, $verifying$

VARIABLE $threadsStates$

$TypeInvariant \triangleq$

$\wedge\ threadsStates \in [tState0 : \{unstarted, running, verifying\},$

$tState1 : ThreadsPossibleStates, tState2 : ThreadsPossibleStates,$

$tState3 : ThreadsPossibleStates]$

$NotEarlyOrLateAssertion \triangleq$

$\vee\ threadsStates \in [tState0 : \{verifying\},$

$tState1 : \{waiting, finished\}, tState2 : \{waiting, finished\},$

$tState3 : \{waiting, finished\}]$

$\vee\ threadsStates \in [tState0 : \{running\},$

$tState1 : ThreadsPossibleStates, tState2 : ThreadsPossibleStates,$

$tState3 : ThreadsPossibleStates]$

$\vee\ threadsStates \in [tState0 : \{unstarted\},$

$tState1 : \{unstarted\},\ tState2 : \{unstarted\},$

$tState3 : \{unstarted\}]$

$TEini \triangleq$

$\land\ threadsStates \in [tState0 : \{unstarted\},$

$tState1 : \{unstarted\},\ tState2 : \{unstarted\},$

$tState3 : \{unstarted\}]$

$TestStarts \triangleq$

$\land\ threadsStates \in [tState0 : \{unstarted\},$

$tState1 : \{unstarted\},$

$tState2 : \{unstarted\},\ tState3 : \{unstarted\}]$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = running]$

$Thread1Starts \triangleq$

$\land\ threadsStates.tState0 = running$

$\land\ threadsStates.tState1 = unstarted$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = started]$

LOCAL $Thread1Runs \triangleq$

$\land\ (threadsStates.tState1 = started \lor threadsStates.tState1 = waiting)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = running]$

LOCAL $Thread1StartsToWait \triangleq$

$\land\ (threadsStates.tState1 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = waiting]$

LOCAL $Thread1FinishesRunning \triangleq$

$\land\ (threadsStates.tState1 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = finished]$

$Thread2Starts \triangleq$

$\land\ threadsStates.tState0 = running$

$\land\ threadsStates.tState2 = unstarted$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState2 = started]$

LOCAL $Thread2Runs \triangleq$

$\land\ (threadsStates.tState2 = started \lor threadsStates.tState2 = waiting)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState2 = running]$

LOCAL $Thread2StartsToWait \triangleq$

$\land\ (threadsStates.tState2 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState2 = waiting]$

LOCAL $Thread2FinishesRunning \triangleq$

$\land\ (threadsStates.tState2 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState2 = finished]$

$Thread3Starts \triangleq$

$\land\ threadsStates.tState0 = running$

$\land\ threadsStates.tState3 = unstarted$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState3 = started]$

LOCAL $Thread3Runs \triangleq$

$\land\ (threadsStates.tState3 = started \lor threadsStates.tState3 = waiting)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState3 = running]$

LOCAL $Thread3StartsToWait \triangleq$

$\land\ (threadsStates.tState3 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState3 = waiting]$

LOCAL $Thread3FinishesRunning \triangleq$

$\land\ (threadsStates.tState3 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState3 = finished]$

$NothingHappens \triangleq$

$\land\ \text{UNCHANGED } threadsStates$

LOCAL $StartAssertionsBlock \triangleq$

$\land\ (threadsStates.tState0 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = verifying]$

$AssertionPerformed \triangleq$

$\land\ (threadsStates.tState0 = running \lor threadsStates.tState0 = verifying)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = verifying]$

$FinishAssertionsBlock \triangleq$

$\land\ (threadsStates.tState0 = verifying)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = running]$

$$
\begin{aligned}
TEnxt \triangleq\ & \\
& \lor\ TestStarts \\
& \lor\ Thread1Starts \\
& \lor\ Thread1Runs \\
& \lor\ Thread1FinishesRunning \\
& \lor\ Thread1StartsToWait \\
& \lor\ Thread2Starts \\
& \lor\ Thread2Runs \\
& \lor\ Thread2FinishesRunning \\
& \lor\ Thread2StartsToWait \\
& \lor\ Thread3Starts \\
& \lor\ Thread3Runs \\
& \lor\ Thread3FinishesRunning \\
& \lor\ Thread3StartsToWait \\
& \lor\ NothingHappens \\
& \lor\ AssertionPerformed \\
& \lor\ StartAssertionsBlock \\
& \lor\ FinishAssertionsBlock \\
TE \triangleq\ & TEini \land \Box[TEnxt]_{\langle threadsStates \rangle}
\end{aligned}
$$

THEOREM $TE \Rightarrow \Box TypeInvariant$

Apêndice B

Especificação em TLA+ de Execução de Teste Monitorando Threads segundo Abordagem *Thread Control for Tests*

MODULE $MonitoredTestExecution$

This module contains the definition of the specification of a test execution in which the System Under Test threads are monitored in order to avoid early and late assertions. The monitoring is based on the Thread Control for Tests Approach.

EXTENDS $TestExecution$

$StartAssertionsBlock \triangleq$

$\land (threadsStates.tState1 = finished \lor threadsStates.tState1 = waiting)$

$\land (threadsStates.tState2 = finished \lor threadsStates.tState2 = waiting)$

$\land (threadsStates.tState3 = finished \lor threadsStates.tState3 = waiting)$

$\land (threadsStates.tState0 = running)$

$\land threadsStates' = [threadsStates \text{ EXCEPT } !.tState0 = verifying]$

$AssertionNotEarly \triangleq$

$\land (threadsStates.tState0 = verifying)$

$\land$ UNCHANGED $threadsStates$

$Thread1Runs \triangleq$

$\land threadsStates.tState0 = running$

$$\wedge\ (threadsStates.tState1 = started \vee threadsStates.tState1 = waiting)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = running]$$

$$Thread1StartsToWait \triangleq$$

$$\wedge\ threadsStates.tState0 = running$$

$$\wedge\ (threadsStates.tState1 = running)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = waiting]$$

$$Thread1FinishesRunning \triangleq$$

$$\wedge\ threadsStates.tState0 = running$$

$$\wedge\ (threadsStates.tState1 = running)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState1 = finished]$$

$$Thread2Runs \triangleq$$

$$\wedge\ threadsStates.tState0 = running$$

$$\wedge\ (threadsStates.tState2 = started \vee threadsStates.tState2 = waiting)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState2 = running]$$

$$Thread2StartsToWait \triangleq$$

$$\wedge\ threadsStates.tState0 = running$$

$$\wedge\ (threadsStates.tState2 = running)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState2 = waiting]$$

$$Thread2FinishesRunning \triangleq$$

$$\wedge\ threadsStates.tState0 = running$$

$$\wedge\ (threadsStates.tState2 = running)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState2 = finished]$$

$$Thread3Runs \triangleq$$

$$\wedge\ threadsStates.tState0 = running$$

$$\wedge\ (threadsStates.tState3 = started \vee threadsStates.tState3 = waiting)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState3 = running]$$

$$Thread3StartsToWait \triangleq$$

$$\wedge\ threadsStates.tState0 = running$$

$$\wedge\ (threadsStates.tState3 = running)$$

$$\wedge\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState3 = waiting]$$

$$Thread3FinishesRunning \triangleq$$

$\land\ threadsStates.tState0 = running$

$\land\ (threadsStates.tState3 = running)$

$\land\ threadsStates' = [threadsStates \text{ EXCEPT } !.tState3 = finished]$

$TEnxt2 \triangleq$

$\lor\ TestStarts$

$\lor\ Thread1Starts$

$\lor\ Thread1Runs$

$\lor\ Thread1FinishesRunning$

$\lor\ Thread1StartsToWait$

$\lor\ Thread2Starts$

$\lor\ Thread2Runs$

$\lor\ Thread2FinishesRunning$

$\lor\ Thread2StartsToWait$

$\lor\ Thread3Starts$

$\lor\ Thread3Runs$

$\lor\ Thread3FinishesRunning$

$\lor\ Thread3StartsToWait$

$\lor\ NothingHappens$

$\lor\ AssertionNotEarly$

$\lor\ StartAssertionsBlock$

$\lor\ FinishAssertionsBlock$

$TE2 \triangleq TEini \land \Box[TEnxt2]_{\langle threadsStates \rangle}$

THEOREM $TE2 \Rightarrow \Box TypeInvariant$

THEOREM $TE2 \Rightarrow \Box NotEarlyOrLateAssertion$

Apêndice C

Especificação em TLA+ de Execução de Testes sem Monitoração mas com Checagem de Asserções Antecipadas e Tardias

MODULE *TestExecutionCheckingAssertionInvariant*

EXTENDS *TestExecution*

THEOREM $TE \Rightarrow \Box NotEarlyOrLateAssertion$

Apêndice D

Saída do Programa StatesAnalyzer

```
Number of states to analyze:251
Early/Late assertion in state:State 6:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 10:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 13:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 15:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 20:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 23:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> STARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 25:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
```

```
Early/Late assertion in state:State 29:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 31:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 34:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 37:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> FINISHED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 40:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> WAITING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 43:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> RUNNING,
tState2 |-> STARTED]
Early/Late assertion in state:State 45:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 49:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> STARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 51:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 54:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 56:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> FINISHED]
```

```
Early/Late assertion in state:State 58:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 60:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 63:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 64:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 65:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 68:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> FINISHED,
tState2 |-> STARTED]
Early/Late assertion in state:State 70:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> FINISHED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 73:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> WAITING,
tState2 |-> STARTED]
Early/Late assertion in state:State 75:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> WAITING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 79:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> RUNNING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 81:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> STARTED]
```

```
Early/Late assertion in state:State 84:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 86:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> STARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 88:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> STARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 90:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 93:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 94:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 95:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 97:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 99:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 102:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> UNSTARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 103:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
```

```
Early/Late assertion in state:State 104:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 108:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> FINISHED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 110:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> FINISHED,
tState2 |-> STARTED]
Early/Late assertion in state:State 113:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> FINISHED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 117:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> WAITING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 119:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> WAITING,
tState2 |-> STARTED]
Early/Late assertion in state:State 122:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> WAITING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 124:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> RUNNING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 126:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> RUNNING,
tState2 |-> WAITING]
Early/Late assertion in state:State 128:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 131:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> STARTED]
```

```
Early/Late assertion in state:State 132:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 133:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 135:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 137:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 140:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 141:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> STARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 142:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> STARTED,
tState2 |-> STARTED]
Early/Late assertion in state:State 145:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> UNSTARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 148:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> UNSTARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 149:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> UNSTARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 150:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> UNSTARTED,
tState2 |-> RUNNING]
```

```
Early/Late assertion in state:State 152:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> FINISHED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 154:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> FINISHED,
tState2 |-> WAITING]
Early/Late assertion in state:State 156:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> FINISHED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 159:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> FINISHED,
tState2 |-> STARTED]
Early/Late assertion in state:State 160:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> FINISHED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 161:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> FINISHED,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 163:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> WAITING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 165:
[tState0 |-> VERIFYING, tState3 |-> UNSTARTED, tState1 |-> WAITING,
tState2 |-> WAITING]
Early/Late assertion in state:State 167:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> WAITING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 170:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> WAITING,
tState2 |-> STARTED]
Early/Late assertion in state:State 171:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> UNSTARTED]
```

```
Early/Late assertion in state:State 172:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> WAITING,
tState2 |-> UNSTARTED]
Early/Late assertion in state:State 174:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 176:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> WAITING]
Early/Late assertion in state:State 179:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 180:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> STARTED]
Early/Late assertion in state:State 181:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> RUNNING,
tState2 |-> STARTED]
Early/Late assertion in state:State 184:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 187:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 188:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> STARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 189:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> STARTED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 190:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> UNSTARTED,
tState2 |-> FINISHED]
```

```
Early/Late assertion in state:State 191:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> UNSTARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 192:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> UNSTARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 193:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> UNSTARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 195:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> FINISHED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 197:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> FINISHED,
tState2 |-> WAITING]
Early/Late assertion in state:State 200:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> FINISHED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 201:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> FINISHED,
tState2 |-> STARTED]
Early/Late assertion in state:State 202:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> FINISHED,
tState2 |-> STARTED]
Early/Late assertion in state:State 204:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> WAITING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 206:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> WAITING,
tState2 |-> WAITING]
Early/Late assertion in state:State 209:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> WAITING,
tState2 |-> RUNNING]
```

```
Early/Late assertion in state:State 210:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> STARTED]
Early/Late assertion in state:State 211:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> WAITING,
tState2 |-> STARTED]
Early/Late assertion in state:State 214:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 217:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> WAITING]
Early/Late assertion in state:State 218:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 219:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> RUNNING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 220:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> STARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 221:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> STARTED,
tState2 |-> FINISHED]
Early/Late assertion in state:State 222:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> STARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 223:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> STARTED,
tState2 |-> WAITING]
Early/Late assertion in state:State 226:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> FINISHED,
tState2 |-> FINISHED]
```

```
Early/Late assertion in state:State 229:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> FINISHED,
tState2 |-> WAITING]
Early/Late assertion in state:State 230:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> FINISHED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 231:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> FINISHED,
tState2 |-> RUNNING]
Early/Late assertion in state:State 234:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> WAITING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 237:
[tState0 |-> VERIFYING, tState3 |-> RUNNING, tState1 |-> WAITING,
tState2 |-> WAITING]
Early/Late assertion in state:State 238:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 239:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> WAITING,
tState2 |-> RUNNING]
Early/Late assertion in state:State 240:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 241:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> RUNNING,
tState2 |-> FINISHED]
Early/Late assertion in state:State 242:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> WAITING]

Early/Late assertion in state:State 243:
[tState0 |-> VERIFYING, tState3 |-> WAITING, tState1 |-> RUNNING,
```

```
tState2 |-> WAITING]


117 problems found
```

Apêndice E

Exemplos de Asserções Antecipadas Identificadas

```
=============EARLY ASSERTIONS======================
Early:1--->Behavior:
State 1:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 2:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 3:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 4:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 5:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 6:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> STARTED,
```

```
tState2 |-> UNSTARTED]
State 7:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
State 8:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]

-----------------------

Early:2--->Behavior:
State 1:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 2:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 3:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 4:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 5:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 6:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
State 7:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
State 8:
```

```
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
State 9:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
State 10:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
State 11:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> STARTED]
State 12:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> STARTED]
State 13:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> STARTED]

-----------------------

Early:3--->Behavior:
State 1:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 2:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 3:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 4:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
```

```
State 5:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
State 6:
[tState0 |-> VERIFYING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]

-----------------------
```

Apêndice F

Exemplos de Asserções Tardias Identificadas

```
=============LATE ASSERTIONS======================
Late:1--->Behavior:
State 1:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 2:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 3:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 4:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 5:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 6:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
```

```
State 7:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
State 8:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
State 9:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
State 10:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> UNSTARTED,
tState2 |-> STARTED]
State 11:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> STARTED]
State 12:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> STARTED]
State 13:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> STARTED]
State 14:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> STARTED]
State 15:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> RUNNING]
State 16:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
State 17:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
```

```
State 18:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
State 19:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
State 20:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> WAITING]

-----------------------

Late:2--->Behavior:
State 1:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 2:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 3:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 4:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
State 5:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> RUNNING,
tState2 |-> UNSTARTED]
State 6:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> WAITING,
tState2 |-> UNSTARTED]
State 7:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> WAITING,
```

```
tState2 |-> STARTED]
State 8:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> WAITING,
tState2 |-> STARTED]
State 9:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> WAITING,
tState2 |-> RUNNING]
State 10:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> RUNNING]
State 11:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> RUNNING,
tState2 |-> FINISHED]
State 12:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> RUNNING,
tState2 |-> FINISHED]
State 13:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> WAITING,
tState2 |-> FINISHED]
State 14:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> WAITING,
tState2 |-> FINISHED]
State 15:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> FINISHED]
State 16:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> FINISHED]
State 17:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> FINISHED]
State 18:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> RUNNING,
```

```
tState2 |-> FINISHED]

-----------------------

Late:3--->Behavior:
State 1:
[tState0 |-> UNSTARTED, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 2:
[tState0 |-> RUNNING, tState3 |-> UNSTARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 3:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> UNSTARTED,
tState2 |-> UNSTARTED]
State 4:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> UNSTARTED]
State 5:
[tState0 |-> RUNNING, tState3 |-> STARTED, tState1 |-> STARTED,
tState2 |-> STARTED]
State 6:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> STARTED]
State 7:
[tState0 |-> RUNNING, tState3 |-> RUNNING, tState1 |-> STARTED,
tState2 |-> STARTED]
State 8:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> STARTED,
tState2 |-> STARTED]
State 9:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> STARTED]
State 10:
```

```
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> RUNNING,
tState2 |-> STARTED]
State 11:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> STARTED]
State 12:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> STARTED]
State 13:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> RUNNING]
State 14:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
State 15:
[tState0 |-> RUNNING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
State 16:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
State 17:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> WAITING]
State 18:
[tState0 |-> VERIFYING, tState3 |-> FINISHED, tState1 |-> WAITING,
tState2 |-> RUNNING]

-----------------------
```

Apêndice G

Código do Arcabouço ThreadControl

Código Fonte G.1: Classe *ThreadControl.java* (fachada)

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18
19 package br.edu.ufcg.threadcontrol;
20
21 import java.util.List;
22
23 /**
```

```
24  * Services offered for tests in order to obtain a better control of
        their
25  * execution and avoid false negatives. The real services for this class
        are
26  * specified by the <code>ThreadControlAspect</code>, which should be
        weaved
27  * with this class before its operations are invoked. <br>
28  * For each state to be expected in a test, this sequence should be
        followed:
29  * <br>
30  * 1)ThreadControl.prepare(configuration)<br>
31  * 2)Stimulate the system<br>
32  * 3)ThreadControl.waitUntilStateIsReached() <br>
33  * 4)Perform assertions <br>
34  * 5) ThreadControl.proceed()
35  */
36 public class ThreadControl {
37
38         private SystemConfiguration systemConfiguration;
39
40         /**
41          * Default constructor.
42          */
43         public ThreadControl() {
44                 super();
45                 this.reset();
46         }
47
48
49         /**
50          * Indicates the expected state for the system.
51          *
52          * @param threadsConfiguration
53          *            a list of ThreadConfiguration objects indicating
                the state in
54          *            which should be some application threads
                considering the names
```

```
55          *              of Runnable classes.
56          */
57         public void prepare(List<ThreadConfiguration>
               threadsConfiguration) {
58                 throwNotUsingAspectsException();
59         }
60
61         /**
62          * Indicates the expected state for the system.
63          *
64          * @param threadsConfiguration
65          *              a list of ThreadConfiguration objects indicating
               the state in
66          *              which should be some application threads
               considering the names
67          *              of Runnable classes.
68          */
69         public void prepare(SystemConfiguration sysConfiguration) {
70                 throwNotUsingAspectsException();
71         }
72         /**
73          * Waits until the specified threads configuration is obtained
               and then
74          * stops all threads trying to modify the achieved system state.
75          *
76          */
77         public void waitUntilStateIsReached() {
78                 throwNotUsingAspectsException();
79         }
80
81         /**
82          * Throws a RuntimeException indicating that the classes that use
               this
83          * package should be compiled using AspectJ compiler.
84          */
85         private static void throwNotUsingAspectsException() {
86                 throw new RuntimeException(
```

```
87                                  "This class should be used only after
                                        compiling with aspects.");
88          }
89
90          /**
91           * Makes the system proceed normally after assertions have been
                performed.
92           * Any thread blocked because the expected system configuration
                has been
93           * reached will be unblocked when this method is invoked.
94           */
95          public void proceed() {
96                  throwNotUsingAspectsException();
97          }
98
99          /**
100          * Resets the control of threads removing any expected system
                configuration
101          * and history of previous threads state transitions.
102          */
103         public void reset() {
104                 throwNotUsingAspectsException();
105         }
106 }
```

Código Fonte G.2: Aspecto *ThreadControlAspect.aj*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18 package br.edu.ufcg.threadcontrol.aspects;
19
20 import java.util.Collection;
21 import java.util.HashSet;
22 import java.util.List;
23 import java.util.concurrent.BlockingQueue;
24 import java.util.concurrent.Semaphore;
25
26 import org.aspectj.lang.JoinPoint;
27
28 import br.edu.ufcg.threadcontrol.ListOfThreadConfigurations;
29 import br.edu.ufcg.threadcontrol.MonitoredQueue.Operation;
30 import br.edu.ufcg.threadcontrol.SystemConfiguration;
31 import br.edu.ufcg.threadcontrol.ThreadConfiguration;
32 import br.edu.ufcg.threadcontrol.ThreadControl;
33 import br.edu.ufcg.threadcontrol.ThreadState;
34 import br.edu.ufcg.threadcontrol.ThreadWatcher;
35 import br.edu.ufcg.threadcontrol.WaitType;
```

```
36
37 /**
38  * This aspect handles thread state changes and also replaces
       ThreadControl
39  * methods by operations from the ThreadWatcher class.
40  *
41  */
42 public aspect ThreadControlAspect {
43
44         private Collection<String> classesToIgnoreVerification = new
              HashSet<String>();
45
46         /**
47          * The object responsible for storing threads and their states.
48          */
49         private ThreadWatcher threadWatcher = new ThreadWatcher();
50
51         /**
52          * Replaces ThreadControl.waitUntilStateIsReached() method
               implementation
53          * using ThreadWatcher.
54          */
55         void around(ThreadControl tcs): execution(
56                         public void ThreadControl.waitUntilStateIsReached
                                ()) && this (tcs){
57                 threadWatcher.waitUntilSystemConfiguration();
58         }
59
60         /**
61          * Replaces ThreadControl.proceed() method implementation using
62          * ThreadWatcher.
63          */
64         void around(ThreadControl tcs): execution(public void
              ThreadControl.proceed())
65                 && this (tcs){
66                 threadWatcher.proceed();
67         }
```

```
68
69          /**
70           * Replaces ThreadControl.reset() method implementation using
                ThreadWatcher.
71           */
72          void around(ThreadControl tcs): execution(public void
               ThreadControl.reset())
73                  && this (tcs){
74                  threadWatcher.reset();
75          }
76
77          /**
78           * Replaces ThreadControl.prepare(List<ThreadConfiguration>)
                method
79           * implementation using ThreadWatcher.
80           */
81          void around(List<ThreadConfiguration> threadsConfiguration):
82                  execution(public void ThreadControl.prepare(List<
                        ThreadConfiguration>))
83                          && args(threadsConfiguration){
84                  threadWatcher.prepare(new ListOfThreadConfigurations(
                        threadsConfiguration));
85                  this.classesToIgnoreVerification = new HashSet<String>();
86          }
87
88          /**
89           * Replaces ThreadControl.prepare(SystemConfiguration) method
90           * implementation using ThreadWatcher.
91           */
92          void around (SystemConfiguration sysConfig):
93                  execution(public void ThreadControl.prepare(
                        SystemConfiguration))
94                          && args(sysConfig){
95                  threadWatcher.prepare(sysConfig);
96                  this.classesToIgnoreVerification = sysConfig.
                        getClassNamesToIgnore();
97          }
```

```
98
99          /*
100          * In the following we define pointcuts and advices to monitor
                the
101          * application and notify ThreadWatcher about state transitions
                so that it
102          * can provide its functionality.
103          */
104
105         /**
106          * This pointcut defines join points where thread actions should
                not be
107          * monitored such as operations inside ThreadWatcher class.
108          */
109         pointcut excludedEntities():
110         !within(br.edu.ufcg.threadcontrol.ThreadWatcher) &&
111         !within(br.edu.ufcg.threadcontrol.aspects.ThreadSleeperAspect);
112
113         /**
114          * Collects calls to Thread.start() method.
115          */
116         pointcut threadStartCalls(Thread t):call(public void start())&&
                target(t)
117                 && excludedEntities();
118
119         /**
120          * Collects calls to Object.wait() method.
121          */
122         pointcut waitCalls(Object o):call(public void wait())&& target(o)
123                 && excludedEntities();
124
125         /**
126          * Collects the execution of the run() method of a Runnable
                implementation.
127          */
128         pointcut runnableRunExecutions():execution(public void Runnable+.
                run())
```

```
129             && excludedEntities();
130
131         /**
132          * Collects the calls to Thread.sleep(long) method.
133          */
134         pointcut sleepCalls(): call (public static void Thread.sleep(long
                ))
135             && excludedEntities();
136
137         /**
138          * Collects calls to Object.wait that use a timeout.
139          */
140         pointcut timedWaitCalls(Object o): call(public void wait(long))
141             && target(o) && excludedEntities();
142
143         /**
144          * Collects calls to Object.notifyAll method.
145          */
146         pointcut notifyAllCalls(Object o): call(public void notifyAll())
147             && target(o) && excludedEntities();
148
149         /**
150          * Collects calls to Object.notify method.
151          */
152         pointcut notifyCalls(Object o): call(public void notify())
153             && target(o) && excludedEntities();
154
155         /**
156          * Collects calls to BlockingQueue+.put
157          */
158         pointcut blockingQueuePutCalls(BlockingQueue q): call(
159                     public void BlockingQueue+.put(..)) && target(q)
160                     && excludedEntities();
161
162         /**
163          * Collects calls to BlockingQueue+.take
164          */
```

```
165         pointcut blockingQueueTakeCalls(BlockingQueue q): call(
166                         public * BlockingQueue+.take()) && target(q)
167                         && excludedEntities();
168
169         /**
170          * Collects calls to Semaphore.acquire without a permits
171          * parameter
172          */
173         pointcut semaphoreAcquireCallsOnePermit(Semaphore sem): call(
174                         public void Semaphore.acquire()) && target(sem)
175                         && excludedEntities();
176
177         /**
178          * Collects calls to Semaphore.acquire with a specific
179          * permits parameter value.
180          */
181         pointcut semaphoreAcquireCallsDefiningPermits(Semaphore sem, int
               permits):
182                 call(public void Semaphore.acquire(int)) && target(sem)
183                         && args(permits) && excludedEntities();
184
185         /**
186          * Collects calls to Semaphore.release without a permits
187          * parameter
188          */
189         pointcut semaphoreReleaseCallsOnePermit(Semaphore sem): call(
190                         public void Semaphore.release()) && target(sem)
191                         && excludedEntities();
192
193         /**
194          * Collects calls to Semaphore.release with a specific
195          * permits parameter value.
196          */
197         pointcut semaphoreReleaseCallsDefiningPermits(Semaphore sem, int
               permits):
198                 call(public void Semaphore.release(int)) && target(sem)
199                         && args(permits) && excludedEntities();
```

```
200
201         /**
202          * Collects calls to Semaphore.drainPermits() method.
203          */
204         pointcut semaphoreDrainPermits(Semaphore sem):
205                 call(public int Semaphore.drainPermits()) && target(sem);
206
207         /*
208          * ADVICE
209          */
210
211         /**
212          * Before a Thread.sleep call, ThreadWatcher is notified about a
                state
213          * change: the current thread has started to sleep.
214          */
215         before(): sleepCalls() {
216                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
217                 threadWatcher.threadHadStateChange(Thread.currentThread()
                        ,
218                                 ThreadState.SLEEPING);
219                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
220         }
221
222         /**
223          * After a Thread.sleep call, when it is about to return,
                ThreadWatcher is
224          * notified about a state change: the current thread will go to
                the RUNNING
225          * state again.
226          */
227         after():sleepCalls() {
228                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
229                 threadWatcher.threadHadStateChange(Thread.currentThread()
                        ,
230                                 ThreadState.RUNNING);
231                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
```

```
232         }
233
234         /**
235          * Before a Thread.start call, ThreadWatcher is notified that a
                new thread
236          * should be monitored and that its initial state is STARTED.
237          */
238         before(Thread t): threadStartCalls(t) {
239                 threadWatcher.threadHadStateChange(t, ThreadState.STARTED
                        );
240         }
241
242         /**
243          * Before a run() method from a Runnable is to be executed,
                ThreadWatcher is
244          * notified about a state change: the current thread is now
                RUNNING,
245          * considering its association with the executing Runnable class.
246          */
247         before(): runnableRunExecutions() {
248                 Object associatedObject = thisJoinPoint.getThis();
249                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
250                 threadWatcher.threadHadStateChange(Thread.currentThread()
                        ,
251                                 ThreadState.RUNNING, associatedObject);
252                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
253         }
254
255         /**
256          * After a run() method from a Runnable has executed,
                ThreadWatcher is
257          * notified about a state change: the current thread is now
                FINISHED,
258          * considering its association with the executing Runnable class.
259          */
260         after(): runnableRunExecutions() {
261                 Object associatedObject = thisJoinPoint.getThis();
```

```
262                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
263                 threadWatcher.threadHadStateChange(Thread.currentThread()
                        ,
264                                 ThreadState.FINISHED, associatedObject);
265         }
266
267         /**
268          * Before a Thread begins to wait on a monitor, ThreadWatcher is
                notified
269          * about a state change: the current thread is now in the WAITING
                state.
270          */
271         before(Object obj):(waitCalls(obj)|| timedWaitCalls(obj)) {
272                 threadWatcher.threadStartedToWaitOnObject(Thread.
                        currentThread(), obj, WaitType.WAIT_ON_OBJ_LOCK);
273         }
274
275         /**
276          * Before a Thread possibly begins to wait on a BlockingQueue <
                code>put</code>
277          * operation, ThreadWatcher is notified about a state change: the
                current
278          * thread may be now in the WAITING state, depending on the queue
                capacity.
279          */
280         before(BlockingQueue queue) : blockingQueuePutCalls(queue) {
281                 Thread currentThread = Thread.currentThread();
282                 threadWatcher.threadPossiblyStartedToWaitOnBlockingQueue(
283                                 currentThread, queue, Operation.put);
284         }
285
286         /**
287          * After a Thread had possibly waited on a BlockingQueue <code>
                put</code>
288          * operation, ThreadWatcher is notified about a state change: the
                current
289          * thread is now in the RUNNING state. The state of Threads that
```

```
        were
290      * waiting on the <code>take</code> operation are updated
        considering the
291      * queue current capacity.
292      */
293     after(BlockingQueue queue) : blockingQueuePutCalls(queue) {
294             Thread t = Thread.currentThread();
295             verifyAndBlockThreadIfNecessary(thisJoinPoint);
296             threadWatcher.threadPossiblyFinishedToWaitOnBlockingQueue
                (
297                             t, queue, Operation.put);
298             verifyAndBlockThreadIfNecessary(thisJoinPoint);
299     }
300
301     /**
302      * Before a Thread possibly begins to wait on a BlockingQueue <
        code>take</code>
303      * operation, ThreadWatcher is notified about a state change: the
         current
304      * thread may be now in the WAITING state, if the queue is empty.
305      */
306     before(BlockingQueue queue) : blockingQueueTakeCalls(queue) {
307             Thread currentThread = Thread.currentThread();
308             threadWatcher.threadPossiblyStartedToWaitOnBlockingQueue(
309                             currentThread, queue, Operation.take);
310     }
311
312     /**
313      * After a Thread had possibly waited on a BlockingQueue <code>
        take</code>
314      * operation, ThreadWatcher is notified about a state change: the
         current
315      * thread is now in the RUNNING state. The state of Threads that
        were
316      * waiting on the <code>put</code> operation are updated
        considering the
317      * queue current capacity.
```

```
318          */
319         after(BlockingQueue queue) : blockingQueueTakeCalls(queue) {
320                 Thread t = Thread.currentThread();
321                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
322                 threadWatcher.threadPossiblyFinishedToWaitOnBlockingQueue
                        (
323                                         t, queue, Operation.take);
324                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
325         }
326
327         /**
328          * Before a Thread possibly begins to wait on a Semaphore
329          * <code>acquire</code> operation, ThreadWatcher is notified
                about
330          * a state change: the current thread may be now in the WAITING
331          * state, if the number of permits available is not enough.
332          */
333         before(Semaphore sem): semaphoreAcquireCallsOnePermit(sem) {
334                 Thread currentThread = Thread.currentThread();
335                 int permitsRequested = 1;
336                 threadWatcher.threadPossiblyStartedToWaitOnSemaphore(
337                                 currentThread, sem,permitsRequested);
338         }
339
340         /**
341          * Before a Thread possibly begins to wait on a Semaphore
342          * <code>acquire</code> operation, ThreadWatcher is notified
                about
343          * a state change: the current thread may be now in the WAITING
344          * state, if the number of permits available is not enough.
345          */
346         before(Semaphore sem, int permitsRequested):
347                 semaphoreAcquireCallsDefiningPermits(sem,
                        permitsRequested) {
348                 Thread currentThread = Thread.currentThread();
349                 threadWatcher.threadPossiblyStartedToWaitOnSemaphore(
                        currentThread, sem,
```

```
350                                         permitsRequested);
351         }
352
353         /**
354          * After a Thread had possibly waited on a Semaphore <code>
               acquire</code>
355          * operation, ThreadWatcher is notified about a state change: the
               current
356          * thread is now in the RUNNING state. The state of Threads that
               were
357          * waiting for available permits from the Semaphore are updated
               considering
358          * the number of permits available.
359          */
360         after(Semaphore sem):
361                 semaphoreAcquireCallsOnePermit(sem){
362                 int permitsRequested = 1;
363                 Thread t = Thread.currentThread();
364                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
365                 threadWatcher.threadPossiblyFinishedToWaitOnSemaphore( t,
                        sem, permitsRequested );
366                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
367         }
368
369         /**
370          * After a Thread had possibly waited on a Semaphore <code>
               acquire</code>
371          * operation, ThreadWatcher is notified about a state change: the
               current
372          * thread is now in the RUNNING state. The state of Threads that
               were
373          * waiting for available permits from the Semaphore are updated
               considering
374          * the number of permits available.
375          */
376         after(Semaphore sem, int permitsRequested):
377                 semaphoreAcquireCallsDefiningPermits(sem,
```

```
                permitsRequested){
378             verifyAndBlockThreadIfNecessary(thisJoinPoint);
379             threadWatcher.threadPossiblyFinishedToWaitOnSemaphore(
                    Thread.currentThread(), sem, permitsRequested );
380             verifyAndBlockThreadIfNecessary(thisJoinPoint);
381         }
382
383         /**
384          * Before a call to the <code>release</code> operation from
                Semaphore
385          * ThreadWatcher is notified to update the state of threads
                waiting on
386          * a Semaphore, considering the permits being released.
387          */
388         before(Semaphore sem): semaphoreReleaseCallsOnePermit(sem){
389             int numPermits = 1;
390             verifyAndBlockThreadIfNecessary(thisJoinPoint);
391             threadWatcher.updateThreadsAfterSemaphoreRelease(sem,
                    numPermits);
392             verifyAndBlockThreadIfNecessary(thisJoinPoint);
393         }
394
395
396         /**
397          * Before a call to the <code>release</code> operation from
                Semaphore
398          * ThreadWatcher is notified to update the state of threads
                waiting on
399          * a Semaphore, considering the permits being released.
400          */
401         before(Semaphore sem, int numPermits):
                semaphoreReleaseCallsDefiningPermits(sem, numPermits){
402             verifyAndBlockThreadIfNecessary(thisJoinPoint);
403             threadWatcher.updateThreadsAfterSemaphoreRelease(sem,
                    numPermits);
404             verifyAndBlockThreadIfNecessary(thisJoinPoint);
405         }
```

```
406
407         /**
408          * After a call to the <code>release</code> operation from
                Semaphore
409          * ThreadWatcher is notified to update the state of threads
                waiting on
410          * a Semaphore, considering the permits have been reset.
411          */
412         after(Semaphore sem): semaphoreDrainPermits(sem){
413                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
414                 threadWatcher.updateThreadsAfterSemaphoreDrainPermits(sem
                        );
415                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
416         }
417         /**
418          * After a Thread finishes to wait on a monitor (returning from
                the wait
419          * call that had a timeout), ThreadWatcher is notified about a
                state change:
420          * the current thread is now in the RUNNING state, and not in the
                 WAITING
421          * state anymore.
422          */
423         after(Object o):timedWaitCalls(o) {
424                 verifyAndBlockThreadIfNecessary(thisJoinPoint, o);
425                 threadWatcher.threadFinishedToWaitOnObject(Thread.
                        currentThread(), o,
426                                         true, WaitType.WAIT_ON_OBJ_LOCK);
427                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
428         }
429
430         /**
431          * After a Thread finishes to wait on a monitor (returning from
                the wait
432          * call), ThreadWatcher is notified about a state change: the
                current thread
433          * is now in the RUNNING state, and not in the WAITING state
```

```
        anymore.
434      */
435     after(Object o): waitCalls(o) {
436             verifyAndBlockThreadIfNecessary(thisJoinPoint, o);
437             threadWatcher.threadFinishedToWaitOnObject(Thread.
                    currentThread(), o,
438                             false, WaitType.WAIT_ON_OBJ_LOCK);
439             verifyAndBlockThreadIfNecessary(thisJoinPoint);
440     }
441
442     /**
443      * Before a Object.notifyAll() is called, ThreadWatcher is
            notified about a
444      * state change: all threads waiting on a given monitor should go
            to the
445      * NOTIFIED state.
446      */
447     before(Object o): notifyAllCalls(o) {
448             verifyAndBlockThreadIfNecessary(thisJoinPoint);
449             threadWatcher.notifyAllWaitingThreads(o);
450             verifyAndBlockThreadIfNecessary(thisJoinPoint);
451     }
452
453     /**
454      * Before a Object.notify() is called, ThreadWatcher is notified
            about a
455      * state change: one of the threads waiting on a given monitor
            should go to
456      * the NOTIFIED state. If more than one thread is waiting, it is
            not
457      * possible to detect at this moment which one will be notified.
            In this
458      * case, all of them go to the POSSIBLY_NOTIFIED state
459      */
460     before(Object o): notifyCalls(o) {
461             verifyAndBlockThreadIfNecessary(thisJoinPoint);
462             threadWatcher.notifyOneWaitingThread(o, WaitType.
```

```
                    WAIT_ON_OBJ_LOCK);
463                 verifyAndBlockThreadIfNecessary(thisJoinPoint);
464         }
465
466         /**
467          * Verifies if the current thread involved in a possible state
                change
468          * should or not proceed. In case the expected state has been
                reached
469          * the current thread will be blocked using an external lock
                object.
470          * @param jp The instance of the joinPoint associated with the
                state
471          * transition.
472          *
473          */
474         private void verifyAndBlockThreadIfNecessary(JoinPoint jp){
475                 String className = jp.getSourceLocation().getWithinType()
                        .getCanonicalName();
476                 if (threadWatcher.isSituationBeingExpected()
477                                 && !this.classesToIgnoreVerification.
                                        contains(className)) {
478                         threadWatcher.verifyThread();
479                 }
480         }
481
482         /**
483          * Verifies if the current thread involved in a possible state
                change
484          * should or not proceed. In case the expected state has been
                reached
485          * the current thread will be blocked. The lock used will be the
                object
486          * used in the wait call.
487          * @param jp The instance of the joinPoint associated with the
                state
488          * transition.
```

```
489          * @param o The object involved in the wait call, so that it can
                 be
490          * used as the lock object to block the thread if the expected
                 state has
491          * been reached in order to avoid state changes while assertions
                 are
492          * being performed.
493          *
494          */
495         private void verifyAndBlockThreadIfNecessary(JoinPoint jp, Object
                o){
496                 String className = jp.getSourceLocation().getWithinType()
                        .getCanonicalName();
497                 if (threadWatcher.isSituationBeingExpected()
498                                 && !this.classesToIgnoreVerification.
                                        contains(className)) {
499                         while (threadWatcher.
                                verifyIfThreadShouldBeBlocked()){
500                                 try {
501                                         long timeToWait = 200;
502                                         o.wait(timeToWait);
503                                 } catch (InterruptedException e) {
504                                         e.printStackTrace();
505                                 }
506                         }
507                 }
508         }
509
510 }
```

Código Fonte G.3: Interface *SystemConfiguration.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18 package br.edu.ufcg.threadcontrol;
19
20 import java.util.Collection;
21
22 /**
23  * An expected system configuration.
24  *
25  */
26 public interface SystemConfiguration {
27
28         /**
29          *
30          * Verifies if the expected system configuration was reached.
31          *
32          * @param tManager
33          *            A ThreadManager used for the verification process.
34          * @return true, if the system expected state has been reached,
                and false,
```

```
35          *          otherwise.
36          */
37         public boolean wasConfigurationReached(ThreadManager tManager);
38
39         /**
40          * Gets the class names to ignore.
41          * @return a Collection with class names to ignore.
42          */
43         public Collection<String> getClassNamesToIgnore();
44
45 }
```

Código Fonte G.4: Classe *ListOfThreadConfigurations.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18 package br.edu.ufcg.threadcontrol;
19
20 import java.util.Collection;
21 import java.util.HashSet;
22 import java.util.LinkedList;
23 import java.util.List;
24
25 /**
26  * A kind of SystemConfiguration expressed in terms of
        ThreadConfiguration
27  * objects.
28  *
29  */
30 public class ListOfThreadConfigurations implements SystemConfiguration {
31
32         /**
33          * List of ThreadConfigurations that should all be reached.
34          */
```

```
35          private List<ThreadConfiguration> threadConfigurations;
36
37          private Collection<String> classNamesToIgnore = new HashSet<
                String>();
38
39          /**
40           * Constructor
41           *
42           * @param tConfigurations
43           *            List of ThreadConfigurations which should all be
                achieved.
44           */
45          public ListOfThreadConfigurations(List<ThreadConfiguration>
                tConfigurations) {
46                  this.threadConfigurations = tConfigurations;
47          }
48
49          /**
50           * Default constructor.
51           */
52          public ListOfThreadConfigurations() {
53                  this.threadConfigurations = new LinkedList<
                        ThreadConfiguration>();
54          }
55
56          public void addThreadConfiguration(ThreadConfiguration tc){
57                  this.threadConfigurations.add(tc);
58          }
59          /**
60           * Verifies if for all ThreadConfiguration objects, the expected
                state was
61           * reached, considering the given ThreadManager
62           */
63          public boolean wasConfigurationReached(ThreadManager manager) {
64                  if (this.threadConfigurations == null
65                                  || this.threadConfigurations.size() == 0)
                                        {
```

```
66                      return true;
67              }
68              boolean wasReached = false;
69              for (ThreadConfiguration tc : this.threadConfigurations)
                    {
70                      if (!tc.wasConfigurationReached(manager)) {
71                              wasReached = false;
72                              break;
73                      } else {
74                              wasReached = true;
75                      }
76              }
77              return wasReached;
78      }
79
80      /**
81       * Get names of classes to ignore in the monitoring process.
82       */
83      public Collection<String> getClassNamesToIgnore() {
84              return classNamesToIgnore;
85      }
86
87      /**
88       * Add a new class to be ignored.
89       * @param c The class whose name will be ignored in the
90       * monitoring process.
91       */
92      public void addToClassesToBeIgnored(Class c){
93              this.classNamesToIgnore.add(c.getCanonicalName());
94      }
95
96      /**
97       * Add a new name of class to be ignored.
98       * @param classCanonicalName The canonical name of a class
99       * to be ignored.
100      */
101     public void addToClassesToBeIgnored(String classCanonicalName){
```

```
102                     this.classNamesToIgnore.add(classCanonicalName);
103             }
104
105     }
```

Código Fonte G.5: Classe *ThreadConfiguration.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18
19 package br.edu.ufcg.threadcontrol;
20
21 import java.util.Collection;
22 import java.util.HashSet;
23 import java.util.List;
24
25 /**
26  * Defines the expected state configuration for threads or associated
        Runnables.
27  *
28  */
29 public class ThreadConfiguration {
30
31         public static final int AT_LEAST_ONCE = -2;
32         public static final int ALL_THREADS_TO_BE_IN_STATE = -1;
33
34         private String threadClassName;
```

```
35          private HashSet<ThreadState> expectedStates;
36          private int timesToBeInState;
37          private int numberOfThreadsToBeInState;
38
39          /**
40           * Constructor that defines the state in which all classes with
41           * threadClassName should be.
42           *
43           * @param threadClassName
44           *            The Thread or Runnable canonical class name.
45           * @param expectedState
46           *            The expected state for Runnables with a given name.
47           * @param timesToBeInState
48           *            The number of times the state has been reached by
                 instances of
49           *            this class or <code>AT_LEAST_ONCE</code> if the
                 state has
50           *            been reached at least once.
51           */
52          public ThreadConfiguration(String threadClassName,
53                        ThreadState expectedState, int timesToBeInState)
                              {
54                  this(threadClassName, new ThreadState[] { expectedState
                        },
55                                    timesToBeInState,
                                          ALL_THREADS_TO_BE_IN_STATE);
56          }
57
58          /**
59           * Constructor that defines the state in which some instances of
                 classes
60           * with threadClassName should be.
61           *
62           * @param threadClassName
63           *            The Thread or Runnable canonical class name.
64           * @param expectedState
65           *            The expected state for Runnables with a given name.
```

```
66          * @param timesToBeInState
67          *            The number of times the state has been reached by
                instances of
68          *            this class or <code>AT_LEAST_ONCE</code> if the
                state has
69          *            been reached at least once.
70          * @param numberOfThreadsToBeInState
71          *            The number of instances of threadClassName that
                should be in
72          *            <code>expectedState</code>.
73          */
74         public ThreadConfiguration(String threadClassName,
75                   ThreadState expectedState, int timesToBeInState,
76                   int numberOfThreadsToBeInState) {
77             this(threadClassName, new ThreadState[] { expectedState
                  },
78                               timesToBeInState,
                                     numberOfThreadsToBeInState);
79         }
80
81         /**
82          * Constructor that defines the states in which all classes with
83          * threadClassName should be.
84          *
85          * @param threadClassName
86          *            The Thread or Runnable canonical class name.
87          * @param expectedStates
88          *            The possible states in which Runnables with a given
                name
89          *            should be.
90          * @param timesToBeInState
91          *            The number of times the state has been reached by
                instances of
92          *            this class or <code>AT_LEAST_ONCE</code> if the
                state has
93          *            been reached at least once.
94          */
```

```
95          public ThreadConfiguration(String threadClassName,
96                  ThreadState[] expectedStates, int
                        timesToBeInState) {
97              this(threadClassName, expectedStates, timesToBeInState,
98                      ALL_THREADS_TO_BE_IN_STATE);
99          }
100
101         /**
102          * Constructor that defines the states in which some classes with
103          * threadClassName should be.
104          *
105          * @param threadClassName
106          *            The Thread or Runnable canonical class name.
107          * @param expectedStates
108          *            The possible states in which Runnables with a given
                name
109          *            should be.
110          * @param timesToBeInState
111          *            The number of times the state has been reached by
                instances of
112          *            this class or <code>AT_LEAST_ONCE</code> if the
                state has
113          *            been reached at least once.
114          * @param numberOfThreadsToBeInState
115          *            The number of instances of threadClassName that
                should be in
116          *            <code>expectedState</code>.
117          */
118         public ThreadConfiguration(String threadClassName,
119                 ThreadState[] expectedStates, int
                        timesToBeInState,
120                 int numberOfThreadsToBeInState) {
121             this.threadClassName = threadClassName;
122             this.expectedStates = new HashSet<ThreadState>();
123             this.setExpectedStates(expectedStates);
124             this.timesToBeInState = timesToBeInState;
125             this.numberOfThreadsToBeInState =
```

```
                numberOfThreadsToBeInState;
126         }
127
128         /**
129          * Indicates if this configuration defines that all threads
                associated with
130          * a given class name should be in a certain state or if just
                some instances
131          * should be.
132          *
133          * @return true if all threads associated with a given class name
                 should be
134          *         in a certain state or set of states, and false,
                otherwise.
135          */
136         public boolean shouldAllThreadsBeInState() {
137                 return this.numberOfThreadsToBeInState ==
                        ALL_THREADS_TO_BE_IN_STATE;
138         }
139
140         /**
141          * Returns the number of threads associated with a given class
                name that
142          * should be in one of the expected states or in a specific state
                .
143          *
144          * @return the number of threads to be in the configured state(s)
                .
145          */
146         public int getNumberOfThreadsToBeInState() {
147                 return this.numberOfThreadsToBeInState;
148         }
149
150         /**
151          * Gets the possible state(s) for threads associated with a given
                 class
152          * name.
```

```
153          *
154          * @return a collection with one or more expected states.
155          */
156         public Collection<ThreadState> getExpectedStates() {
157                 return expectedStates;
158         }
159
160         /**
161          * Configured the expected state for a class of threads.
162          *
163          * @param expectedState
164          *             The expected state.
165          */
166         public void setExpectedState(ThreadState expectedState) {
167                 this.setExpectedStates(expectedState);
168         }
169
170         /**
171          * Configures the expected states for threads associated with the
                 class name
172          * of this configuration.
173          *
174          * @param states
175          *             The expected states.
176          */
177         public void setExpectedStates(ThreadState... states) {
178                 this.expectedStates = new HashSet<ThreadState>();
179                 for (ThreadState s : states) {
180                         this.expectedStates.add(s);
181                 }
182         }
183
184         /**
185          * The class name associated with threads. Usually corresponds to
                 a Runnable
186          * class name.
187          *
```

```
188          * @return the class name associated with some application
                 threads.
189          */
190         public String getThreadClassName() {
191                 return threadClassName;
192         }
193
194         /**
195          * Configures the class name associated with threads.
196          *
197          * @param threadClassName
198          *             a Runnable class name.
199          */
200         public void setThreadClassName(String threadClassName) {
201                 this.threadClassName = threadClassName;
202         }
203
204         /**
205          * Gets the number of times threads associated with a given class
                 name had
206          * achieved one of the expected states.
207          *
208          * @return the number of times this configuration has been
                 achieved by
209          *         threads associated with a given class name.
210          */
211         public int getTimesToBeInState() {
212                 return timesToBeInState;
213         }
214
215         /**
216          * Configures the number of times threads associated with this
                 configuration
217          * class name should have reached the expected state(s).
218          *
219          * @param timesToBeInState
220          *             the number of times the state has been reached by
```

```
            threads
221          *            associated with this configuration class name.
222          */
223         public void setTimesToBeInState(int timesToBeInState) {
224                 this.timesToBeInState = timesToBeInState;
225         }
226
227         /**
228          * Verifies if a given configuration has been reached,
                considering the
229          * overall system state given by ThreadManager.
230          *
231          * @param tManager
232          *            The ThreadManager instance.
233          * @return true if the configuration was reached, and false,
                otherwise.
234          */
235         public boolean wasConfigurationReached(ThreadManager tManager) {
236                 ThreadsAssociations associations = tManager.
                        getAssociationsForName(this
237                                 .getThreadClassName());
238                 if (associations == null) {
239                         return false;
240                 } else {
241                         boolean isInState = false;
242                         if (this.shouldAllThreadsBeInState()) {
243                                 Collection<ThreadState> states =
                                        associations
244                                                 .getAssociatedStates();
245                                 for (ThreadState st : states) {
246                                         isInState = this.
                                                getExpectedStates().contains(
                                                st);
247                                         if (!isInState) {
248                                                 return false;
249                                         }
250                                 }
```

```
251                         } else {
252                                 List<ThreadAssociation> associationsList
                                        = associations
253                                                 .
                                                        getThreadAssociationsList
                                                        ();
254                                 int numberOfThreadsInState = 0;
255                                 for (ThreadAssociation assoc :
                                        associationsList) {
256                                         if (this.getExpectedStates().
                                                contains(assoc.getState())) {
257                                                 numberOfThreadsInState++;
258                                         }
259                                 }
260                                 isInState = (this.
                                        getNumberOfThreadsToBeInState() ==
                                        numberOfThreadsInState);
261                         }
262                         if (isInState) {
263                                 if (this.getTimesToBeInState() ==
                                        ThreadConfiguration.AT_LEAST_ONCE) {
264                                         return true;
265                                 } else {
266                                         return this.getTimesToBeInState()
267                                                 == tManager
268                                                 .getNumberOfTimesInState(
269                                                 this.getThreadClassName(),
270                                                         this.getExpectedStates())
                                                                ;
271                                 }
272                         }
273                         return isInState;
274                 }
275         }
276
277 }
```

Código Fonte G.6: Classe *ThreadState.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18
19 package br.edu.ufcg.threadcontrol;
20
21 /**
22  * Possible states for threads.
23  *
24  */
25 public class ThreadState {
26         public static final ThreadState UNKNOWN = new ThreadState("
              UNKNOWN");
27         public static final ThreadState SLEEPING= new ThreadState("
              SLEEPING");
28         public static final ThreadState WAITING = new ThreadState("
              WAITING");
29         public static final ThreadState RUNNING = new ThreadState("
              RUNNING");
30         public static final ThreadState FINISHED = new ThreadState("
              FINISHED");
```

```
31          public static final ThreadState STARTED = new ThreadState("
               STARTED");
32          public static final ThreadState NOTIFIED = new ThreadState("
               NOTIFIED");
33          public static final ThreadState POSSIBLY_NOTIFIED = new
               ThreadState("POSSIBLY_NOTIFIED");
34          public static final ThreadState AT_SPECIFIC_APP_POINT = new
               ThreadState("AT_SPECIFIC_APP_POINT");
35
36          private String state;
37
38          /**
39           * Cosntructor
40           * @param state String representing a state.
41           */
42          public ThreadState(String state) {
43                  this.state = state;
44          }
45
46          /**
47           * Returns the String representing this state
48           * @return the String representing the state.
49           */
50          public String getState() {
51                  return this.state;
52          }
53
54          /**
55           * Compares this state with another one, regarding
56           * the String representation of the thread state.
57           */
58          public boolean equals(Object o){
59                  if (o instanceof ThreadState) {
60                          ThreadState ts = (ThreadState) o;
61                          return ts.getState().equals(this.getState());
62                  }
63                  return false;
```

```
64          }
65
66          /**
67           * Factory method to create ThreadState instances.
68           * @param state A String representation of a thread state.
69           * @return a ThreadState instance corresponding to the
70           * given String.
71           */
72          public static ThreadState createThreadState(String state){
73                  return new ThreadState(state);
74          }
75
76          /**
77           * Gets the string representation of this thread state.
78           */
79          public String toString(){
80                  return this.getState();
81          }
82  }
```

Código Fonte G.7: Classe *ThreadManager.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18
19 package br.edu.ufcg.threadcontrol;
20
21 import java.util.Collection;
22 import java.util.HashMap;
23 import java.util.HashSet;
24 import java.util.LinkedHashSet;
25 import java.util.LinkedList;
26 import java.util.List;
27 import java.util.Set;
28
29 /**
30  * This class manages threads states, and also association states (
        Classes being
31  * run by any system thread).
32  *
33  */
34 public class ThreadManager {
```

```
35
36          private static final boolean DEBUG = false;
37
38          private HashMap<String, ThreadsAssociations>
                threadsAssociationsByName;
39
40          private HashMap<String, ThreadAssociationHistory>
                threadsAssociationHistoryByName;
41
42          private HashMap<Thread, ThreadsAssociations>
                threadsAssociationByThread;
43
44          private ThreadsStates threadsByState;
45
46          /**
47           * Default constructor
48           */
49          public ThreadManager() {
50                  this.threadsAssociationsByName = new HashMap<String,
                        ThreadsAssociations>();
51                  this.threadsAssociationHistoryByName = new HashMap<String
                        , ThreadAssociationHistory>();
52                  this.threadsAssociationByThread = new HashMap<Thread,
                        ThreadsAssociations>();
53                  this.threadsByState = new ThreadsStates();
54          }
55
56          /**
57           * General method that prints the string parameter on the
58           * standard output.
59           *
60           * @param str
61           *              String to be printed.
62           */
63          public static void println(String str) {
64
65                  if (DEBUG) {
```

```
66                      System.out.println(str);
67              }
68      }
69
70      /**
71       * Verifies if a thread instance is in a given state.
72       *
73       * @param t
74       *              The thread instance.
75       * @param state
76       *              The state to be verified.
77       * @return true f the thread is in the specified state, and false
             ,
78       *          otherwise.
79       */
80      public boolean isThreadInState(Thread t, ThreadState state) {
81              return this.getThreadState(t).equals(state);
82      }
83
84      /**
85       * Informs that a thread, in relation to a given object has
             transitioned
86       * to a new state.
87       * @param t The executing thread
88       * @param state The new state
89       * @param associatedObject The associatedObject, usually a
             Runnable.
90       */
91      public void changeToState(Thread t, ThreadState state,
92                      Object associatedObject) {
93              println("==>changing thread:" + t + " / " + t.hashCode()
                    + " / " + state);
94              if (state.equals(ThreadState.RUNNING)) {
95                      if (!isThreadBeingManaged(t)) {
96                              this.addNewThreadToBeManaged(t,
                                    ThreadState.STARTED, t);
97                      }
```

```
98                      if (associatedObject.equals(t)) {
99                              updateThreadState(t, state);
100                     } else {
101                             updateThreadStateConsideringNewAssociation
                                    (t, state,
102                                             associatedObject);
103                     }
104             } else if (state.equals(ThreadState.FINISHED)) {
105                     updateThreadStateConsideringEndOfAssociation(t,
                            state,
106                                     associatedObject);
107             } else {
108                     throw new RuntimeException(
109                         "Unexpected state transition considering[:"
110                             + associatedObject.getClass().
                                    getCanonicalName()
111                             + "]");
112             }
113
114     }
115
116     /**
117      * Informs that a thread, and all current thread associations
                related with
118      * this thread have transitioned to a new state.
119      * @param t The executing thread
120      * @param state The new state
121      */
122     public void changeToState(Thread t, ThreadState state) {
123             println("==>changing thread:"+t  + " / " + t.hashCode() +
                        " / " + state);
124             if (!isThreadBeingManaged(t)) { // case of started thread
125                     if (state.equals(ThreadState.STARTED)) {
126                             this.addNewThreadToBeManaged(t,
                                    ThreadState.STARTED, t);
127                     } else {
128                             throw new RuntimeException(
```

```
129                                     "Unexpected state transition
                                            considering [:"
130                                     + t.getClass().getCanonicalName()
                                             + "]");
131                     }
132             } else {
133                     updateThreadState(t, state);
134             }
135     }
136
137     /**
138      * Gets the number of times a thread associated with a given
            class name has
139      * entered in either one of the possible states.
140      *
141      * @param className
142      *             The name of the class associated with a Thread. It
            is usually
143      *             a Runnable class name.
144      * @param possibleStates
145      *             collection of possible states in which this thread
            has been.
146      * @return the number of times a thread associated with a given
            class name
147      *         has entered in either one of the possible states.
148      */
149     protected int getNumberOfTimesInState(String className,
150                     Collection<ThreadState> possibleStates) {
151             ThreadAssociationHistory threadHistory = this.
                    threadsAssociationHistoryByName
152                             .get(className);
153             if (threadHistory == null) {
154                     return 0;
155             } else {
156                     int timesInStates = 0;
157                     for (ThreadState s : possibleStates) {
158                             timesInStates += threadHistory.
```

```
                                        getNumberOfTimesInState(s);
159                     }
160                     return timesInStates;
161             }
162     }
163
164     /**
165      * Updates the associations related with a given thread that are
166      * not finished.
167      * @param t The thread object.
168      * @param newState The new state to be updated.
169      */
170     private void updateThreadState(Thread t, ThreadState newState) {
171             List<ThreadAssociation> associations = this.
                        threadsAssociationByThread
172                                 .get(t).getThreadAssociationsList();
173             ThreadState previousState = this.getThreadState(t);
174             if (!previousState.equals(newState)) {
175                     this.threadsByState.changeThreadState(t,
                            previousState, newState);
176                     for (ThreadAssociation assoc : associations) {
177                             if (!assoc.getState().equals(newState)
178                                             && !assoc.getState().
179                                             equals(ThreadState.
                                                    FINISHED)) {
180                                     assoc.setState(newState);
181                                     this.
                                            updateHistoryOfAssociatedClassNames
                                            (
182                                                    newState, assoc
183                                                    .getName());
184                             }
185                     }
186             }
187     }
188
189     /**
```

```
190         * Updates associations and the thread state considering that an
191         * association will finish. For instance, a Runnable class has
                finished
192         * its run method execution. If this is the first run being
                monitored
193         * (the association defines the thread state) and therefore all
194         * associations will be considered to be finished.
195         */
196        private void updateThreadStateConsideringEndOfAssociation(
197                        Thread t, ThreadState state,
198                        Object associatedObject) {
199                if (state.equals(ThreadState.FINISHED)) {
200                        ThreadAssociation tAssociation =
201                                this.threadsAssociationByThread
202                                .get(t).getThreadAssociationBetween(
203                                t, associatedObject);
204                        if (tAssociation.definesGlobalEndOfThread()) {
205                                ThreadState previousState = this.
                                    getThreadState(t);
206                                this.threadsByState.changeThreadState(
207                                        t, previousState,
208                                        ThreadState.FINISHED);
209                                List<ThreadAssociation> associations =
210                                        this.threadsAssociationByThread
211                                        .get(t).getThreadAssociationsList
                                            ();
212                                for (ThreadAssociation assoc :
                                    associations) {
213                                        if (!assoc.getState().equals(
                                            state)) {
214                                                assoc.setState(state);
215                                                this.
                                                    updateHistoryOfAssociatedClassName
                                                    (
216                                                        state, assoc
217                                                        .getName());
218                                        }
```

```
219                                 }
220                                 this.threadsByState.changeThreadState(t,
221                                         this.getThreadState(t),
                                                state);
222                         } else {
223                                 tAssociation.setState(state);
224                                 this.updateHistoryOfAssociatedClassNames(
225                                         state, tAssociation
226                                         .getName());
227                         }
228                 }
229
230         }
231
232         /**
233          * Updates thread and its associations states considering the
                following
234          * cases: 1) a run is being executed, but the thread is already
                managed,
235          * but is not running yet (thread is considered started); 2) the
                thread
236          * is running, but inside an existing run, a run method is
                invoked.
237          * @param t The current thread.
238          * @param state The new state (in this case we only expect
                RUNNING)
239          * @param associatedObject The Runnable whose run is being
                executed.
240          */
241         private void updateThreadStateConsideringNewAssociation(Thread t,
242                         ThreadState state, Object associatedObject) {
243                 if (state.equals(ThreadState.RUNNING)) {
244                         List<ThreadAssociation> associations =
245                                 this.threadsAssociationByThread
246                                 .get(t).getThreadAssociationsList();
247                         for (ThreadAssociation assoc : associations) {
248                                 if (!assoc.getState().equals(state)
```

```
249                                 && !(assoc.getState().equals(
                                        ThreadState.FINISHED)
250                                 && !assoc.getObject().equals(
                                        assoc.getThread()))) {
251                                 assoc.setState(state);
252                                 this.
                                        updateHistoryOfAssociatedClassNames
                                        (
253                                             state, assoc.getName());
254                         }
255                 }
256                 ThreadAssociation newAssoc;
257                 ThreadState previousState = this.getThreadState(t
                    );
258                 if (!previousState.equals(ThreadState.RUNNING)) {
259                         this.threadsByState.changeThreadState(t,
                            previousState,
260                                         ThreadState.RUNNING);
261                         newAssoc = new ThreadAssociation(t,
                            ThreadState.RUNNING,
262                                         associatedObject, true);
263                 } else {
264                         newAssoc = new ThreadAssociation(t,
                            ThreadState.RUNNING,
265                                         associatedObject);
266                 }
267
268                 this.threadsAssociationByThread.get(t).
                    addAssociation(newAssoc);
269                 this.updateHistoryOfAssociatedClassNames(state,
                    newAssoc.getName());
270                 ThreadsAssociations assocsByName = this.
                    threadsAssociationsByName
271                                 .get(newAssoc.getName());
272                 if (assocsByName == null) {
273                         assocsByName = new ThreadsAssociations();
274                         assocsByName.addAssociation(newAssoc);
```

```
275                             this.threadsAssociationsByName.put(
                                    newAssoc.getName(),
276                                             assocsByName);
277                     } else {
278                             assocsByName.addAssociation(newAssoc);
279                     }
280
281             } else {
282                     throw new RuntimeException(
283                                     "Unexpected state transition to
                                            state ["+state
284                                     +"]  considering [:"
285                                     + associatedObject.getClass().
                                            getCanonicalName()
286                                     + "]");
287             }
288     }
289
290     /**
291      * Gets the thread state.
292      * @param t The thread whose state is being requested.
293      * @return the thread current state.
294      */
295     public ThreadState getThreadState(Thread t) {
296             return this.threadsByState.getThreadState(t);
297     }
298
299     /**
300      * Adds a new thread to be managed and its initial state.
301      * @param t The thread.
302      * @param initialState The initial state of the thread.
303      * @param associatedObject The object being run.
304      */
305     private void addNewThreadToBeManaged(Thread t, ThreadState
            initialState,
306                     Object associatedObject) {
307             ThreadAssociation assoc = new ThreadAssociation(t,
```

```
                initialState,
308                             associatedObject, true);
309             String assocName = assoc.getName();
310             ThreadsAssociations associationsByName = this.
                    threadsAssociationsByName
311                             .get(assocName);
312             if (associationsByName == null) {
313                     associationsByName = new ThreadsAssociations();
314                     associationsByName.addAssociation(assoc);
315                     this.threadsAssociationsByName.put(assocName,
                            associationsByName);
316             } else {
317                     associationsByName.addAssociation(assoc);
318             }
319             ThreadsAssociations threadsAssociationsForThread = new
                    ThreadsAssociations();
320             threadsAssociationsForThread.addAssociation(assoc);
321             this.threadsAssociationByThread.put(t,
                    threadsAssociationsForThread);
322             this.threadsByState.addNewThreadState(t, initialState);
323             this.updateHistoryOfAssociatedClassNames(initialState,
                    assocName);
324     }
325
326     /**
327      * Updates the history of previous state for each association
             name
328      * (class name).
329      * @param state The state to update.
330      * @param names One or more classes that had a state transition
331      * to <code>state</code>.
332      */
333     private void updateHistoryOfAssociatedClassNames(ThreadState
            state,
334                     String... names) {
335             for (String name : names) {
336                     ThreadAssociationHistory historyForName = this.
```

```
                                threadsAssociationHistoryByName
337                                     .get(name);
338                             if (historyForName == null) {
339                                     historyForName = new
                                            ThreadAssociationHistory(name);
340                                     historyForName.changedToState(state);
341                                     this.threadsAssociationHistoryByName.put(
                                            name, historyForName);
342                             } else {
343                                     historyForName.changedToState(state);
344                             }
345                     }
346             }
347
348             /**
349              * Verifies if a previous state transition has being
350              * registered for a given Thread.
351              * @param t The thread.
352              * @return true, if this thread is already known by this
353              * class, and false, otherwise.
354              */
355             private boolean isThreadBeingManaged(Thread t) {
356                     return this.threadsAssociationByThread.get(t) != null;
357             }
358
359             /**
360              * Get ThreadsAssociations related with a given class name.
361              * @param className The class name.
362              * @return the ThreadsAssociation with ThreadAssociations
363              * related with <code>className</code>.
364              */
365             protected ThreadsAssociations getAssociationsForName(String
                    className) {
366                     return this.threadsAssociationsByName.get(className);
367             }
368
369     }
```

```
370
371 /**
372  * Manages the collection of associations between threads and
373  * objects related with these threads, such as instances of Runnable
        classes.
374  *
375  */
376 class ThreadsAssociations {
377
378         private LinkedList<ThreadAssociation> threadAssociations;
379
380         /**
381          * Constructor.
382          */
383         public ThreadsAssociations() {
384                 this.threadAssociations = new LinkedList<
                        ThreadAssociation>();
385         }
386
387         /**
388          * Adds a new association for the collection managed by this
                class.
389          * @param assoc The ThreadAssociation.
390          */
391         public void addAssociation(ThreadAssociation assoc) {
392                 threadAssociations.add(assoc);
393         }
394
395         /**
396          * Gets the list of thread associations.
397          * @return the list of ThreadAssociation objects.
398          */
399         public List<ThreadAssociation> getThreadAssociationsList() {
400                 return this.threadAssociations;
401         }
402
403         /**
```

```
404             * Gets the collection of states associated with the
405             * threads of the ThreadsAssociation.
406             * @return
407             */
408            public Collection<ThreadState> getAssociatedStates() {
409                    Collection<ThreadState> states = new HashSet<ThreadState
                           >();
410                    for (ThreadAssociation assoc : threadAssociations) {
411                            states.add(assoc.getState());
412                    }
413                    return states;
414            }
415
416            /**
417             * Gets the specific ThreadAssociation that relates a thread with
                       an
418             * object.
419             * @param t The Thread.
420             * @param obj The related object.
421             * @return the corresponding ThreadAssociation.
422             */
423            public ThreadAssociation getThreadAssociationBetween(Thread t,
                       Object obj) {
424                    for (ThreadAssociation assoc : threadAssociations) {
425                            if (assoc.getThread().equals(t) && assoc.
                                   getObject().equals(obj)) {
426                                    return assoc;
427                            }
428                    }
429                    return null;
430            }
431    }
432
433    /**
434     * Represents the association between a thread and an object related with
                this
435     * thread.
```

```
436  *
437  */
438 class ThreadAssociation {
439         private Thread thread;
440         private ThreadState associatedThreadState;
441         private Object associatedObject;
442         private boolean definesGlobalEndOfThread;
443
444         /**
445          * Constructor.
446          * @param t The thread.
447          * @param state The state for this thread considering the
                 associatedObject.
448          * @param associatedObject The object associated with the thread.
                  Normally
449          * a Runnable instance.
450          */
451         public ThreadAssociation(Thread t, ThreadState state,
452                         Object associatedObject) {
453                 this.thread = t;
454                 this.associatedThreadState = state;
455                 this.associatedObject = associatedObject;
456         }
457
458         /**
459          * Constructor.
460          * @param t The thread.
461          * @param state The state for this thread considering the
                 associatedObject.
462          * @param associatedObject The object associated with the thread.
                  Normally
463          * @param definesEnd boolean that indicates if once this
                 association has
464          * gone to the finished state, the overall thread state has also
                 gone to
465          * finished.
466          */
```

```
467         public ThreadAssociation(Thread t, ThreadState state,
468                         Object associatedObject, boolean definesEnd) {
469                 this.thread = t;
470                 this.associatedThreadState = state;
471                 this.associatedObject = associatedObject;
472                 this.definesGlobalEndOfThread = definesEnd;
473         }
474
475         /**
476          * Gets the associated object.
477          * @return the associated object.
478          */
479         public Object getObject() {
480                 return this.associatedObject;
481         }
482
483         /**
484          * Gets the associated thread.
485          * @return the associated thread.
486          */
487         public Object getThread() {
488                 return this.thread;
489         }
490
491         /**
492          * Tells if this association defines the finished state of
493          * the thread.
494          * @return true if once this association has the state finished,
495          * the thread will also have this overall state.
496          */
497         public boolean definesGlobalEndOfThread() {
498                 return this.definesGlobalEndOfThread;
499         }
500
501         /**
502          * Sets the state of this association.
503          * @param newState The new state.
```

```
504             */
505             public void setState(ThreadState newState) {
506                     this.associatedThreadState = newState;
507
508             }
509
510             /**
511              * Gets a String representation of this association.
512              * @return the string representation of this association,
513              * represented by the canonical class name of the associated
                    object.
514              */
515             public String getName() {
516                     return this.associatedObject.getClass().getCanonicalName
                            ();
517             }
518
519             /**
520              * Gets the state of this association.
521              * @return the state of this association.
522              */
523             public ThreadState getState() {
524                     return associatedThreadState;
525             }
526     }
527
528     /**
529      * Represents the history with the number of times a thread
530      * has been in each state.
531      *
532      */
533     class ThreadAssociationHistory {
534             HashMap<ThreadState, Integer> threadHistory;
535             private String associationIdentifier;
536
537             /**
538              * Constructor.
```

```
539          * @param associationIdentifier The association identifier.
540          */
541         public ThreadAssociationHistory(String associationIdentifier) {
542                 this.associationIdentifier = associationIdentifier;
543                 this.threadHistory = new HashMap<ThreadState, Integer>();
544                 threadHistory.put(ThreadState.FINISHED, 0);
545                 threadHistory.put(ThreadState.RUNNING, 0);
546                 threadHistory.put(ThreadState.SLEEPING, 0);
547                 threadHistory.put(ThreadState.WAITING, 0);
548                 threadHistory.put(ThreadState.STARTED, 0);
549                 threadHistory.put(ThreadState.UNKNOWN, 0);
550                 threadHistory.put(ThreadState.NOTIFIED, 0);
551                 threadHistory.put(ThreadState.POSSIBLY_NOTIFIED, 0);
552                 threadHistory.put(ThreadState.AT_SPECIFIC_APP_POINT, 0);
553         }
554
555         /**
556          * Notifies this class that a state change has happened.
557          * @param state The new state.
558          */
559         public void changedToState(ThreadState state) {
560                 int previousNumberOfTimesInState = this.threadHistory.get
                        (state);
561                 this.threadHistory.put(state,
                        previousNumberOfTimesInState + 1);
562                 ThreadManager.println("==> Thread "+associationIdentifier
                        +" changed to state:"+state+" "+(
                        previousNumberOfTimesInState + 1)+ " times");
563         }
564
565         /**
566          * Gets the number of times a thread has been associated with a
                given
567          * state.
568          * @param state The desired state
569          * @return the number of times a thread has been in a state.
570          */
```

```
571         public int getNumberOfTimesInState(ThreadState state) {
572                 return this.threadHistory.get(state);
573         }
574
575         /**
576          * Returns the thread association identifier.
577          * @return the thread association identifier.
578          */
579         public String getAssociationIdentifier() {
580                 return this.associationIdentifier;
581         }
582
583 }
584
585 /**
586  * Classes that manages thread states by different indexes.
587  *
588  */
589 class ThreadsStates {
590         private HashMap<Thread, ThreadState> stateByThread;
591         private HashMap<ThreadState, Set<Thread>> threadsByState;
592
593         /**
594          * Constructor.
595          */
596         public ThreadsStates() {
597                 this.stateByThread = new HashMap<Thread, ThreadState>();
598                 this.threadsByState = new HashMap<ThreadState, Set<Thread
                        >>();
599                 this.threadsByState.put(ThreadState.FINISHED,
600                                 new LinkedHashSet<Thread>());
601                 this.threadsByState.put(ThreadState.RUNNING,
602                                 new LinkedHashSet<Thread>());
603                 this.threadsByState.put(ThreadState.SLEEPING,
604                                 new LinkedHashSet<Thread>());
605                 this.threadsByState.put(ThreadState.WAITING,
606                                 new LinkedHashSet<Thread>());
```

```
607                 this.threadsByState.put(ThreadState.STARTED,
608                                 new LinkedHashSet<Thread>());
609                 this.threadsByState.put(ThreadState.UNKNOWN,
610                                 new LinkedHashSet<Thread>());
611                 this.threadsByState.put(ThreadState.NOTIFIED,
612                                 new LinkedHashSet<Thread>());
613                 this.threadsByState.put(ThreadState.POSSIBLY_NOTIFIED,
614                                 new LinkedHashSet<Thread>());
615                 this.threadsByState.put(ThreadState.AT_SPECIFIC_APP_POINT
                        ,
616                                 new LinkedHashSet<Thread>());
617         }
618
619         /**
620          * Associates a thread with a given state.
621          * @param t The thread.
622          * @param initialState The initial state of the thread.
623          */
624         public void addNewThreadState(Thread t, ThreadState initialState)
                {
625                 this.stateByThread.put(t, initialState);
626                 this.threadsByState.get(initialState).add(t);
627
628         }
629
630         /**
631          * Gets the current thread state.
632          * @param t The thread.
633          * @return the current thread state.
634          */
635         public ThreadState getThreadState(Thread t) {
636                 ThreadState state = this.stateByThread.get(t);
637                 if (state == null) {
638                         return ThreadState.UNKNOWN;
639                 }
640                 return state;
641         }
```

```
642
643         /**
644          * Changes the state associated with a given thread.
645          * @param t The thread.
646          * @param fromState The previous state.
647          * @param toState The new state.
648          */
649         public void changeThreadState(Thread t, ThreadState fromState,
650                         ThreadState toState) {
651                 this.stateByThread.put(t, toState);
652                 this.threadsByState.get(fromState).remove(t);
653                 this.threadsByState.get(toState).add(t);
654         }
655
656 }
```

Código Fonte G.8: Classe *ThreadWatcher.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18 package br.edu.ufcg.threadcontrol;
19
20 import java.util.Collection;
21 import java.util.HashMap;
22 import java.util.Iterator;
23 import java.util.Map;
24 import java.util.Set;
25 import java.util.concurrent.BlockingQueue;
26 import java.util.concurrent.Semaphore;
27
28 import br.edu.ufcg.threadcontrol.MonitoredQueue.Operation;
29
30 /**
31  * This class watches operations involving threads and provides services
        that
32  * should know if certain system states have been achieved.
33  *
34  */
```

```
35 public class ThreadWatcher {
36
37         private ThreadManager threadManager;
38         /**
39          * Indicates if an expected system state has been reached.
40          */
41         private boolean stateReached = false;
42         /**
43          * Indicates if a certain system state is being expected. A
               prepare has been
44          * called, and the corresponding proceed() method has not been
               called yet.
45          */
46         private boolean isSituationBeingExpected = false;
47         /**
48          * The system expected state, which can be defined in terms of
               names of
49          * Runnable classes and the associated expected state.
50          */
51         private SystemConfiguration systemConfiguration;
52         /**
53          * Controls the matching between prepare and proceed calls.
54          */
55         private int prepareCallsWithoutProceed = 0;
56         /**
57          * Lock used to block threads trying to perform operations after
               an expected
58          * state has been reached.
59          */
60         private Object controllerLock;
61         /**
62          * The objects in which a wait was called.
63          */
64         private Map<WaitType, Map<Object, MonitoredObject>>
               monitoredObjectsByWaitType;
65
66         private Map<Semaphore,MonitoredSemaphore> monitoredSemaphores;
```

```
67
68          private Map<BlockingQueue<?>,MonitoredQueue> monitoredQueues;
69
70          /**
71           * A debug variable.
72           */
73          private static final boolean DEBUG = false;
74
75          /**
76           * The current test thread
77           */
78          private Thread currentTestThread = null;
79
80          /**
81           * Default constructor.
82           */
83          public ThreadWatcher() {
84                  this.reset();
85          }
86
87
88          /**
89           * Waits until a given system configuration specified by the
90           * <code>systemConfiguration</code> attribute configured in a
                previous
91           * prepare call.
92           */
93          public synchronized void waitUntilSystemConfiguration() {
94                  this.isSituationBeingExpected = true;
95                  while (!this.stateReached) {
96                          if (verifyIfConfigurationWasReached()) {
97                                  return;
98                          } else {
99                                  try {
100                                         this.wait();
101                                 } catch (InterruptedException e) {
102                                         e.printStackTrace();
```

```
103                                 }
104                         }
105                 }
106         }
107
108         /**
109          * Informs ThreadWatcher that a given system configuration is
                being
110          * expected.
111          *
112          * @param sysConfiguration
113          *             the system expected state. It can be expressed, for
114          *             instance,  in terms of Runnable class names and
115          *             expected states for each thread-Runnable
                association.
116          */
117         public synchronized void prepare(SystemConfiguration
            sysConfiguration) {
118                 this.prepareCallsWithoutProceed++;
119                 this.systemConfiguration = sysConfiguration;
120                 this.setSituationBeingExpected(true);
121                 this.setStateReached(false);
122                 this.currentTestThread = Thread.currentThread();
123                 if (prepareCallsWithoutProceed <=1){
124                         this.threadHadStateChange( currentTestThread,
                            ThreadState.RUNNING,
125                                 currentTestThread);
126                 }
127         }
128
129         /**
130          * Verifies if the system configuration specified by
131          * <code>systemConfiguration</code> has been reached.
132          */
133         private boolean verifyIfConfigurationWasReached() {
134                 boolean wasReached = this.systemConfiguration.
                        wasConfigurationReached(this.threadManager);
```

```
135                 this.setStateReached(wasReached);
136                 return wasReached;
137         }
138
139         /**
140          * Configures the stateReached value.
141          *
142          * @param stateReached
143          *              true if the expected state has been reached, and
                 false,
144          *              otherwise.
145          */
146         private void setStateReached(boolean stateReached) {
147                 synchronized (this) {
148                         this.stateReached = stateReached;
149                         this.notifyAll();
150                 }
151         }
152
153         /**
154          * Verifies if the current thread (probably trying to perform an
                 operation
155          * being monitored by ThreadControlAspect) should be blocked
                 because the
156          * expected state has been reached. In case it should, the
                 current
157          * thread will be blocked using the controllerLock.
158          */
159         //Obs: We should avoid to call this method when the thread used
                in the verification
160     //is holding a lock necessary for the verifications performed in the
            assertions.
161         public void verifyThread() {
162                 if (Thread.currentThread() != this.currentTestThread) {
163                         synchronized (controllerLock) {
164                                 boolean shouldBlockCurrentThread = false;
165                                 synchronized (this) {
```

```
166                                     shouldBlockCurrentThread = this.
                                            isStateReached()
167                                                 &&
                                                        isSituationBeingExpected
                                                        ();
168                                 }
169                                 while (shouldBlockCurrentThread) {
170                                     try {
171                                         this.println("Will block
                                                ..." + Thread.
                                                currentThread().
                                                getClass().
                                                getCanonicalName()+ "
                                                ==>"+Thread.
                                                currentThread() );
172                                         controllerLock.wait();
173                                         synchronized (this) {
174                                             shouldBlockCurrentThread
                                                    = this.
                                                    isStateReached
                                                    ()
175                                                             &&
                                                                isSituation
                                                                ()
                                                                ;
176                                         }
177                                     } catch (InterruptedException e)
                                            {
178                                         e.printStackTrace();
179                                     }
180                                 }
181                             }
182                         }
183                 }
184
```

```
185         /**
186          * Verifies if the current thread (probably trying to perform an
                 operation
187          * being monitored by ThreadControlAspect) should be blocked
                 because the
188          * expected state has been reached.
189          * @return true if the current thread should be blocked and false
                 , otherwise.
190          */
191         public boolean verifyIfThreadShouldBeBlocked() {
192                 if (Thread.currentThread() != this.currentTestThread) {
193                         synchronized (controllerLock) {
194                                 boolean shouldBlockCurrentThread = false;
195                                 synchronized (this) {
196                                         shouldBlockCurrentThread = this.
                                                 isStateReached()
197                                                         &&
                                                             isSituationBeingExpected
                                                             ();
198                                         return shouldBlockCurrentThread;
199                                 }
200                         }
201                 } else {
202                         return false;
203                 }
204
205         }
206
207         /**
208          * Configures if a given system state is being expected or not.
209          *
210          * @param isSituationExpected
211          *            true, if a system state (systemConfiguation) is
                 being
212          *            expected, and false, otherwise.
213          */
214         private void setSituationBeingExpected(boolean
```

```
            isSituationExpected) {
215             this.isSituationBeingExpected = isSituationExpected;
216         }
217
218         /**
219          * Returns true if a system state (systemConfiguration) is being
                expected,
220          * and false, otherwise.
221          */
222         public boolean isSituationBeingExpected() {
223             return this.isSituationBeingExpected;
224         }
225
226         /**
227          * Indicates if the expected state, which has been checked before
                has been
228          * reached.
229          *
230          * @return true if the expected system state has been reached,
                and false
231          *         otherwise.
232          */
233         public boolean isStateReached() {
234             return stateReached;
235         }
236
237         /**
238          * When several threads are possibly notified and associated with
                the same
239          * monitor, if one of them starts to run again, the other
                possibly notified
240          * are considered to be waiting.
241          *
242          * @param possiblyNotifiedThreads
243          *            Collection of threads in the state
244          *            ThreadState.POSSIBLY_NOTIFIED
245          * @return true if it was necessary to change the state of at
```

```
            least one of
246         *          the threads, and false otherwise.
247         */
248         private boolean makePossiblyNotifiedThreadsWaitAgain(
249                         Collection<Thread> possiblyNotifiedThreads) {
250                 boolean changed = false;
251                 for (Thread t : possiblyNotifiedThreads) {
252                         ThreadState state = threadManager.getThreadState(
                                t);
253                         if (state.equals(ThreadState.POSSIBLY_NOTIFIED))
                                {
254                                 changed = true;
255                                 this.threadManager.changeToState(t,
                                        ThreadState.WAITING);
256                         }
257                 }
258                 return changed;
259
260         }
261
262         /**
263          * Notifies that a state change has happened, verifies if after
                this change
264          * the expected state has been reached. A notification is sent to
                any thread
265          * waiting until the state is reached and this thread may be
                unblocked in
266          * case this has happened.
267          */
268         private synchronized void notifyThreadsStateChange() {
269                 if (this.systemConfiguration != null) {
270                         this.verifyIfConfigurationWasReached();
271                 }
272                 if (this.isStateReached()) {
273                         this.notify();
274                 }
275         }
```

```
276
277         /**
278          * Adds to the list of monitored objects a given object and the
                thread it is
279          * waiting for.
280          *
281          * @param o
282          *            The object where the wait was called.
283          * @param t
284          *            The thread being executed when the wait was called.
285          * @param waitType
286          *                          Type of wait used on this object.
287          */
288         private boolean addToMonitoredObjects(Object o, Thread t,
               WaitType waitType) {
289                 boolean added = false;
290                 Map<Object, MonitoredObject> monitoredObjects =
                        monitoredObjectsByWaitType.get(waitType);
291                 MonitoredObject mo = monitoredObjects.get(o);
292                 if (mo == null) {
293                         monitoredObjects.put(o, new MonitoredObject(o, t)
                                );
294                         added = true;
295                 } else {
296                         added = mo.addThread(t);
297                 }
298                 return added;
299         }
300
301         /**
302          * General method that prints the string parameter on the log and
                on the
303          * standard output.
304          *
305          * @param str
306          *            String to be printed.
307          */
```

```
308         private void println(String str) {
309
310                 if (DEBUG) {
311                         System.out.println(str);
312                 }
313         }
314
315         /**
316          * Unblocks any threads that were blocked while trying to perform
                 a
317          * monitored operation that is not allowed after the system
                 expected state
318          * is reached.
319          */
320         public void proceed() {
321                 synchronized (controllerLock) {
322                         prepareCallsWithoutProceed--;
323                         if (prepareCallsWithoutProceed == 0) {
324                                 this.setStateReached(false);
325                                 this.setSituationBeingExpected(false);
326                         }
327                         // if there are unmatched prepare and proceed
                             calls, a system state
328                         // (situation) is still being expected (
                             situationBeingExpected=true)
329                         controllerLock.notifyAll();
330                 }
331                 synchronized (this) {
332                         this.notify();
333                 }
334         }
335
336         /**
337          * Resets any information regarding threads monitoring and
                 expected system
338          * configurations.
339          */
```

```
340         public synchronized void reset() {
341                 this.threadManager = new ThreadManager();
342                 controllerLock = new Object();
343                 this.systemConfiguration = null;
344                 this.stateReached = false;
345                 this.isSituationBeingExpected = false;
346                 monitoredObjectsByWaitType = new HashMap<WaitType, Map<
                        Object, MonitoredObject>>();
347                 for ( WaitType waitType : WaitType.values()) {
348                         monitoredObjectsByWaitType.put(waitType, new
                                HashMap<Object, MonitoredObject>());
349                 }
350                 this.prepareCallsWithoutProceed = 0;
351                 monitoredSemaphores = new HashMap<Semaphore,
                        MonitoredSemaphore>() ;
352                 monitoredQueues = new HashMap<BlockingQueue<?>,
                        MonitoredQueue>();
353         }
354
355         /**
356          * Indicates that a state change has happened to a Thread.
357          *
358          * @param t
359          *             The Thread.
360          * @param toState
361          *             The new state in which the thread should be.
362          */
363         public synchronized void threadHadStateChange(Thread t,
                ThreadState toState) {
364                 if (isSituationBeingExpected()) {
365                         this.println("@@@@@CHANGING THREAD ["+t.getClass
                                ().getCanonicalName()+"] to state: "+toState);
366                         this.threadManager.changeToState(t, toState);
367                         this.notifyThreadsStateChange();
368                 }
369         }
370
```

```
371         /**
372          * Indicates that a state change has happened to a Thread,
                regarding a given
373          * Runnable object.
374          *
375          * @param t
376          *            The current thread.
377          * @param toState
378          *            The new state for this Thread considering the
                Runnable object.
379          * @param associatedObject
380          *            A Runnable object whose class name identifies its
                relation
381          *            with the current thread.
382          */
383         public synchronized void threadHadStateChange(Thread t,
384                         ThreadState toState, Object associatedObject) {
385                 if (isSituationBeingExpected()) {
386                         this.threadManager.changeToState(t, toState,
                                associatedObject);
387                         this.notifyThreadsStateChange();
388                 }
389         }
390
391         /**
392          * Indicates that Thread <code>t</code> is now in the WAITING
                state due to
393          * a monitor.
394          *
395          * @param t
396          *            The waiting thread.
397          * @param monitoredObject
398          *            The object on which the wait was called.
399          * @param waitType
400          *            Wait type.
401          */
402         public synchronized void threadStartedToWaitOnObject(Thread t,
```

```
403                     Object monitoredObject, WaitType waitType) {
404             if (isSituationBeingExpected()) {
405                 this.addToMonitoredObjects(monitoredObject, t,
                        waitType);
406                 this.threadManager.changeToState(t, ThreadState.
                        WAITING);
407                 this.notifyThreadsStateChange();
408             }
409         }
410
411         /**
412          * Indicates that Thread <code>t</code> is now in the RUNNING
                state due
413          * because it is returning from a wait call.
414          *
415          * @param t
416          *            The waiting thread.
417          * @param monitoredObject
418          *            The object on which the wait was called.
419          * @param timedWait
420          *            true, if a the wait call that has finished had a
                timeout, and
421          *            false, otherwise.
422          */
423         public synchronized void threadFinishedToWaitOnObject(Thread t,
                Object o,
424                 boolean timedWait, WaitType waitType) {
425             if (isSituationBeingExpected()) {
426                 if ((this.threadManager.isThreadInState(t,
                        ThreadState.WAITING)
427                                 || this.threadManager.
                                    isThreadInState(t, ThreadState
                                    .NOTIFIED) || this.
                                    threadManager
428                                 .isThreadInState(t, ThreadState.
                                    POSSIBLY_NOTIFIED))) {
429                     makeThreadRunAndAnalyzeNotifications(t, o
```

```
                                        , timedWait, waitType);
430                     } else if (this.threadManager.isThreadInState(t,
                            ThreadState.RUNNING)) {
431                         this.println("==>Thread was already
                                running");
432                     } else {
433                         this.println("==>STRANGE:"+this.
                                threadManager.getThreadState(t));
434                         this.threadManager.changeToState(t,
                                ThreadState.RUNNING);
435                         this.notifyThreadsStateChange();
436                     }
437                 }
438             }
439
440             /**
441              * Makes the thread go to the RUNNING state and analyzes if
442              * POSSIBLY_NOTIFIED threads that were waiting on the same lock
                     should
443              * transition to the WAITING state.
444              *
445              * @param t
446              *            The thread that will start to run.
447              * @param o
448              *            The object on which this thread was waiting.
449              * @param timedWait
450              *            a boolean indicating if the Object.wait call had a
                         timeout
451              *            (true) or not (false).
452              */
453             private void makeThreadRunAndAnalyzeNotifications(Thread t,
                        Object o,
454                             boolean timedWait, WaitType waitType) {
455                 ThreadState threadPreviousState = threadManager.
                            getThreadState(t);
456                 synchronized (monitoredObjectsByWaitType) {
457                     Map<Object, MonitoredObject> monitoredObjects =
```

```
                monitoredObjectsByWaitType.get(waitType);
458             MonitoredObject mo = monitoredObjects.get(o);
459             threadManager.changeToState(t, ThreadState.
                    RUNNING);
460             if (mo != null) {
461                 if (mo.getMonitoringThreads().size() == 1
462                         && mo.
                                getMonitoringThreads()
                                .contains(t)) {
463                     monitoredObjects.remove(o);
464                 } else {
465                     mo.removeMonitoringThread(t);
466                     if (threadPreviousState
467                             .equals(
                                ThreadState.
                                POSSIBLY_NOTIFIED
                                )) {
468                         if (!timedWait) {
469                             makePossiblyNotifiedThreadsWa
                                (mo
470                                                 .
                                                    getMonitor
                                                    ()
                                                    )
                                                    ;

471                         } else {
472                             // FIXME: Verify
                                what to do
                                when we do not
                                 know if
473                             // the thread has
                                 waked up due
                                to a notify or
                                 due to
474                             // elapsed time.
                                This is not so
```

```
                                                                critical
                                                               because
475                                                          // using
476                                                          // only notify
                                                               when more than
                                                                one thread
                                                               can be
477                                                          // waiting is not
478                                                          // a good
                                                               programming
                                                               practice.
479                                                      }
480                                                  }
481                                              }
482                                          }
483                                          this.notifyThreadsStateChange();
484                                  }
485                          }
486
487                          /**
488                           * Notifies all threads waiting for an object o, making them go
                                to the
489                           * NOTIFIED state.
490                           *
491                           * @param o
492                           *             The object where a notifyAll was called.
493                           */
494                          public synchronized void notifyAllWaitingThreads(Object o) {
495                                  if (isSituationBeingExpected()) {
496                                          synchronized (monitoredObjectsByWaitType) {
497                                                  MonitoredObject mo =
                                                        monitoredObjectsByWaitType.get(
                                                        WaitType.WAIT_ON_OBJ_LOCK).get(o);
498                                                  if (mo != null) {
499                                                          Iterator<Thread> it = mo.
                                                                getMonitoringThreadsIterator()
                                                                ;
```

```
500                                     while (it.hasNext()) {
501                                         Thread t = it.next();
502                                         this.threadHadStateChange
                                                (t, ThreadState.
                                                NOTIFIED);
503                                     }
504                                 } else {
505                                     this.println("NOTIFY LOST: No
                                            application thread was "
506                                                 + "waiting on
                                                    object #" + o
507                                                 + "# and
                                                    notifyAll was
                                                    called");
508                                 }
509                             }
510                         }
511                 }
512
513                 /**
514                  * Notifies a thread that is waiting for an object o
515                  *
516                  * @param o
517                  *            The object where a notify was called.
518                  */
519                 public synchronized void notifyOneWaitingThread(Object o,
                        WaitType waitType) {
520                     if (this.isSituationBeingExpected()) {
521                         synchronized (monitoredObjectsByWaitType) {
522                             Map<Object, MonitoredObject>
                                    monitoredObjects =
                                    monitoredObjectsByWaitType.get(
                                    waitType);
523                             MonitoredObject mo = monitoredObjects.get
                                    (o);
524                             if (mo != null) {
525                                 Iterator<Thread> it = mo.
```

```
                getMonitoringThreadsIterator()
                ;
526             boolean
                moreThanOneThreadWaitingNotification
                 = false;
527             if (mo.getMonitoringThreads().
                size() > 1) {
528                 System.err.println("
                        notify() method was
                        called on object="
529                                     + o
530                                     + " , but
                                         more
                                        than
                                        one
                                        thread

                                        should
                                         be
                                        notified
                                        ");
531                 moreThanOneThreadWaitingNotification
                        = true;
532             }
533             while (it.hasNext()) {
534                 Thread t = it.next();
535                 ThreadState toState;
536                 if (
                        moreThanOneThreadWaitingNotificati
                        ) {
537                         toState =
                                ThreadState.
                                POSSIBLY_NOTIFIED
                                ;
538                 } else {
539                         toState =
                                ThreadState.
```

```
                                                                NOTIFIED;
540                                                         }
541
542                                                         threadHadStateChange(t,
                                                                toState);
543                                                 }
544                                         } else {
545                                                 this.println("NOTIFY LOST: No
                                                        application thread was waiting
                                                        "
546                                                                 +" on object #" +
                                                                     o + "# and
                                                                    notify was
                                                                    called");
547                                         }
548                                 }
549                         }
550         }
551
552         /**
553          * Notifies this class that a thread has possibly started to wait
                 on
554          * a Semaphore. This would depend on the permits available and on
                 the
555          * number of permits requested.
556          * @param currentThread The current thread.
557          * @param sem The Semaphore.
558          * @param permitsRequested The number of permits requested.
559          */
560         public synchronized void threadPossiblyStartedToWaitOnSemaphore(
                Thread currentThread,
561                         Semaphore sem, int permitsRequested) {
562                 if (isSituationBeingExpected()) {
563                         MonitoredSemaphore monSem = this.
                                addToMonitoredObjectsWithPermits(sem,
                                currentThread,
564                                         permitsRequested);
```

```
565                         ThreadState state = monSem.
                                getThreadStateForAvailablePermits(
                                currentThread);
566                         this.threadManager.changeToState(currentThread,
                                state);
567                         this.notifyThreadsStateChange();
568                 }
569
570         }
571
572         /**
573          * Starts to monitor a thread that is waiting for permits.
574          * @param sem The Semaphore.
575          * @param t The thread.
576          * @param permitsRequested The number of permits requested.
577          * @return the MonitoredSemaphore related with the thread.
578          */
579         private MonitoredSemaphore addToMonitoredObjectsWithPermits(
                Semaphore sem, Thread t, int permitsRequested) {
580                 MonitoredSemaphore monSemaphore = this.
                        monitoredSemaphores.get(sem);
581                 if (monSemaphore == null){
582                         monSemaphore = new MonitoredSemaphore(sem, t,
                                permitsRequested);
583                         this.monitoredSemaphores.put(sem, monSemaphore);
584                 } else {
585                         monSemaphore.addThreadWithPermits(t,
                                permitsRequested);
586                 }
587                 return monSemaphore;
588         }
589
590         /**
591          * Notifies this class that a thread will start to run again and
                updates
592          * the current number of permits available.
593          * @param t The thread.
```

```
594             * @param sem The Semaphore.
595             * @param permitsRequested The number of permits this thread has
                    requested.
596             */
597            public synchronized void threadPossiblyFinishedToWaitOnSemaphore(
                    Thread t, Semaphore sem, int permitsRequested) {
598                    if (isSituationBeingExpected()) {
599                            if (isWaitingNotifiedOrPossiblyNotified(t)) {
600                                    makeThreadRunAndAnalyzeNotificationsForSemaphore
                                            (t, sem, permitsRequested);
601                            } else if (this.threadManager.isThreadInState(t,
                                    ThreadState.RUNNING)) {
602                                    this.println("Thread was already running"
                                            );
603                            } else {
604                                    this.threadManager.changeToState(t,
                                            ThreadState.RUNNING);
605                                    this.notifyThreadsStateChange();
606                            }
607                    }
608            }
609
610            /**
611             * Moves a thread to the running state and updates the state of
                    the
612             * other threads waiting on the same Semaphore.
613             * @param t The thread.
614             * @param sem The Semaphore.
615             * @param permitsRequested The number of permits this thread has
                    requested.
616             */
617            private void makeThreadRunAndAnalyzeNotificationsForSemaphore(
                    Thread t,
618                            Semaphore sem, int permitsRequested) {
619                    synchronized (monitoredSemaphores) {
620                            MonitoredSemaphore monSem = this.
                                    monitoredSemaphores.get(sem);
```

```
621                         threadManager.changeToState(t, ThreadState.
                                RUNNING);
622                         if (monSem != null) {
623                                 monSem.removeAvailablePermits(
                                        permitsRequested);
624                                 if (monSem.getNumberOfMonitoringThreads()
                                        == 1) {
625                                         monSem.removeMonitoringThread(t);
626                                 } else {
627                                         monSem.removeMonitoringThread(t);
628                                         updateStateOfThreadsAssociatedWithSemaphore
                                                (monSem);
629                                 }
630                         }
631                         this.notifyThreadsStateChange();
632                 }
633         }
634
635
636         /**
637          * Updates the state of threads after a release call on a
                Semaphore.
638          * @param sem The Semaphore.
639          * @param numPermits The number of permits released.
640          */
641         public synchronized void updateThreadsAfterSemaphoreRelease(
                Semaphore sem, int numPermits) {
642                 MonitoredSemaphore monSem = this.monitoredSemaphores.get(
                        sem);
643                 if (monSem != null){
644                         monSem.addAvailablePermits(numPermits);
645                         updateStateOfThreadsAssociatedWithSemaphore(
                                monSem);
646                         this.notifyThreadsStateChange();
647                 } else {
648                         monSem = new MonitoredSemaphore(sem);
649                         this.monitoredSemaphores.put(sem, monSem);
```

```
650                     monSem.addAvailablePermits(numPermits);
651                 }
652             }
653
654             /**
655              * Updates the states of threads waiting on a semaphore when
656              * the drainPermits method is called.
657              * @param sem The Semaphore.
658              */
659             public void updateThreadsAfterSemaphoreDrainPermits( Semaphore
                    sem) {
660                 MonitoredSemaphore monSem = this.monitoredSemaphores.get(
                        sem);
661                 if (monSem != null){
662                     monSem.resetAvailablePermits();
663                     updateStateOfThreadsAssociatedWithSemaphore(
                            monSem);
664                     this.notifyThreadsStateChange();
665                 }
666             }
667
668             /**
669              * Updates the state of threads associated with a Semaphore.
670              * @param monSem The semaphore being monitored.
671              */
672             private void updateStateOfThreadsAssociatedWithSemaphore(
673                     MonitoredSemaphore monSem) {
674                 Set <Thread> threads = monSem.getMonitoringThreads();
675                 for (Thread thread: threads){
676                     ThreadState state = this.threadManager.
                            getThreadState(thread);
677                     ThreadState newState = monSem.
                            getThreadStateForAvailablePermits(thread);
678                     if (isWaitingNotifiedOrPossiblyNotified(thread)
                            && !state.equals(newState)){
679                         threadManager.changeToState(thread,
                                newState);
```

```
680                         }
681                 }
682         }
683
684         /**
685          * Verifies if the state of a given thread is WAITING, NOTIFIED
                or
686          * POSSIBLY_NOTIFIED.
687          * @param t The thread.
688          * @return true if the thread is in one of this sates: WAITING,
689          * NOTIFIED or POSSIBLY_NOTIFIED; and false, otherwise.
690          *
691          */
692         private boolean isWaitingNotifiedOrPossiblyNotified(Thread t){
693                 return (this.threadManager.isThreadInState(t, ThreadState
                        .WAITING)
694                                 || this.threadManager.isThreadInState(t,
                                        ThreadState.NOTIFIED)
695                                 || this.threadManager
696                                 .isThreadInState(t, ThreadState.
                                        POSSIBLY_NOTIFIED));
697         }
698
699         /**
700          * Adds a queue to the collection of queues being monitored.
701          * @param queue The queue to be monitored.
702          * @return The MonitoredQueue object associated with a queue.
703          */
704         private MonitoredQueue addToMonitoredObjectsWithQueue(
                BlockingQueue queue) {
705                 MonitoredQueue monQueue = this.monitoredQueues.get(queue)
                        ;
706                 if (monQueue == null){
707                         monQueue = new MonitoredQueue(queue);
708                         this.monitoredQueues.put(queue, monQueue);
709                 }
710                 return monQueue;
```

```
711         }
712
713         /**
714          * Thread has possibly started to wait on a BlockingQueue.
715          * @param currentThread The current thread.
716          * @param queue The queue.
717          * @param oper The operation being called on the BlockingQueue.
718          */
719         public synchronized void
                threadPossiblyStartedToWaitOnBlockingQueue(
720                         Thread currentThread, BlockingQueue<?> queue,
                                Operation oper) {
721                 if (isSituationBeingExpected()) {
722                         synchronized (monitoredQueues) {
723                                 MonitoredQueue monQueue = this.
                                        addToMonitoredObjectsWithQueue(queue);
724                                 monQueue.addThreadBlockingQueue(
                                        currentThread, oper);
725                                 ThreadState state = monQueue.
                                        getThreadStateForQueueSize(
                                        currentThread);
726                                 this.threadManager.changeToState(
                                        currentThread, state);
727                                 this.notifyThreadsStateChange();
728                         }
729                 }
730         }
731
732         /**
733          * Informs that a thread is not waiting on a BlockingQueue
                anymore.
734          * @param t The thread.
735          * @param queue The BlockingQueue.
736          * @param oper The operation called on the BlockingQueue.
737          */
738         public synchronized void
                threadPossiblyFinishedToWaitOnBlockingQueue(Thread t,
```

```
739                     BlockingQueue<?> queue, Operation oper) {
740             if (isSituationBeingExpected()) {
741                     if (isWaitingNotifiedOrPossiblyNotified(t)) {
742                             makeThreadRunAndAnalyzeNotificationsForQueue
                                    (t, queue, oper);
743                     } else if (this.threadManager.isThreadInState(t,
                            ThreadState.RUNNING)) {
744                             this.println("Thread was already running"
                                    );
745                     } else {
746                             this.threadManager.changeToState(t,
                                    ThreadState.RUNNING);
747                             this.notifyThreadsStateChange();
748                     }
749             }
750     }
751
752     /**
753      * Updates the state of a thread to RUNNING and updates the state
                of other
754      * threads waiting on the same queue.
755      * @param t The thread.
756      * @param queue The BlockingQueue instance.
757      * @param operation The operation invoked on the BlockingQueue.
758      */
759     private void makeThreadRunAndAnalyzeNotificationsForQueue(Thread
            t,
760                     BlockingQueue<?> queue, Operation operation) {
761             synchronized (monitoredQueues) {
762                     MonitoredQueue monQueue = this.monitoredQueues.
                            get(queue);
763                     threadManager.changeToState(t, ThreadState.
                            RUNNING);
764                     if (monQueue != null) {
765                             monQueue.removeThreadBlockingQueue(t,
                                    operation);
766                             updateStateOfThreadsAssociatedWithQueue(
```

```
                                    monQueue, operation);
767                                 Operation symetricOper;
768                                 if (Operation.put.equals(operation)) {
769                                         symetricOper = Operation.take;
770                                 } else {
771                                         symetricOper = Operation.put;
772                                 }
773                                 updateStateOfThreadsAssociatedWithQueue(
                                        monQueue, symetricOper );
774                         }
775                         this.notifyThreadsStateChange();
776                 }
777         }
778
779         /**
780          * Updates the states of threads associated with a queue.
781          * @param monQueue The MonitoredQueue object that associates a
                thread
782          * with a queue.
783          * @param oper The operation called on a BlockingQueue.
784          */
785         private void updateStateOfThreadsAssociatedWithQueue(
786                         MonitoredQueue monQueue, Operation oper) {
787                 Set <Thread> threads = monQueue.getMonitoringThreads(oper
                    );
788                 for (Thread thread : threads){
789                         ThreadState state = this.threadManager.
                            getThreadState(thread);
790                         ThreadState newState = monQueue.
                            getThreadStateForQueueSize(thread);
791                         if (isWaitingNotifiedOrPossiblyNotified(thread)
                            && !state.equals(newState)){
792                                 threadManager.changeToState(thread,
                                    newState);
793                         }
794                 }
795         }
```

```
796 }
```

Código Fonte G.9: Classe *MonitoredObject.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18 package br.edu.ufcg.threadcontrol;
19
20 import java.util.Collections;
21 import java.util.HashSet;
22 import java.util.Iterator;
23 import java.util.Set;
24
25 /**
26  * This class represents an object and the threads waiting for it.
27  */
28 public class MonitoredObject {
29
30         /**
31          * The object being monitored (an object that has received a wait
                call).
32          */
33         private Object monitored;
34
```

```
35          /**
36           * The threads monitoring the object.
37           */
38          private Set<Thread> monitoringThreads = Collections
39                          .synchronizedSet(new HashSet<Thread>());
40
41          /**
42           * Constructor.
43           *
44           * @param o
45           *              The object being monitored (where a wait was called
                 ).
46           * @param firstThread
47           *              The thread waiting for the object.
48           */
49          public MonitoredObject(Object o, Thread firstThread) {
50                  this.monitored = o;
51                  this.monitoringThreads.add(firstThread);
52          }
53
54          /**
55           * Verifies if an object is monitored by a certain Thread.
56           *
57           * @param t
58           *              The Thread we want to know if it is monitored.
59           * @return true if while t was running, a wait on a certain
                 object was
60           *          called.
61           */
62          public boolean isMonitoredBy(Thread t) {
63                  return this.monitoringThreads.contains(t);
64          }
65
66          /**
67           * Adds a certain Thread to the list of threads waiting for an
                 object.
68           *
```

```
69          * @param t
70          *              The Thread.
71          * @return true, if t was not monitoring this object, and false
                otherwise.
72          */
73         public boolean addThread(Thread t) {
74                 Object o = this.monitoringThreads.add(t);
75                 return o != null;
76         }
77
78         /**
79          * Gets the list of threads waiting for an object notify or
                notifyAll.
80          *
81          * @return the list of threads waiting for an object notify or
                notifyAll.
82          */
83         public Iterator<Thread> getMonitoringThreadsIterator() {
84
85                 return this.monitoringThreads.iterator();
86         }
87
88         /**
89          * Gets the Map storing threads waiting for an object notify or
                notifyAll.
90          *
91          * @return the Map storing threads waiting for an object notify
                or
92          *          notifyAll.
93          */
94         public Set<Thread> getMonitoringThreads() {
95
96                 return this.monitoringThreads;
97         }
98
99         /**
100         * Gets a String that represents this object.
```

```
101             *
102             * @return the textual representation of this object.
103             */
104            public String toString() {
105
106                    return "Obj:" + this.monitored + " Threads.size:"
107                                    + this.monitoringThreads.size();
108            }
109
110            /**
111             * Removes a thread that was being monitored.
112             * @param t The thread.
113             */
114            public void removeMonitoringThread(Thread t) {
115                    this.monitoringThreads.remove(t);
116
117            }
118    }
```

Código Fonte G.10: Classe *MonitoredQueue.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18 package br.edu.ufcg.threadcontrol;
19
20 import java.util.HashMap;
21 import java.util.HashSet;
22 import java.util.Set;
23 import java.util.concurrent.BlockingQueue;
24
25 /**
26  * This class monitors the threads related with a BlockingQueue.
27  */
28 public class MonitoredQueue {
29
30         /**
31          * Blocking queue operations being monitored.
32          *
33          */
34         public enum Operation {
35                 take, put;
```

```
36          }
37
38          /**
39           * The queue size.
40           */
41          private int size;
42
43          /**
44           * The queue capacity.
45           */
46          private int capacity;
47
48          /**
49           * Map with the threads being monitored regarding a blocking
                 queue operation
50           * and also the queue size.
51           */
52          private HashMap<Operation, HashSet<Thread>>
                monitoringThreadsAndQueueSize;
53
54          /**
55           * Constructor
56           * @param queue the queue being monitored.
57           */
58          public MonitoredQueue(BlockingQueue<?> queue) {
59                  this.size = queue.size();
60                  this.capacity = queue.size() + queue.remainingCapacity();
61                  this.monitoringThreadsAndQueueSize = new HashMap<
                        Operation,
62                          HashSet<Thread>>();
63                  for (Operation oper : Operation.values()) {
64                          this.monitoringThreadsAndQueueSize.put(oper,
65                                          new HashSet<Thread>());
66                  }
67          }
68
69          /**
```

```
70          * Gets the number of threads waiting on a blocking queue
                operation.
71          * @param oper The queue operation.
72          * @return the number of threads waiting on a blocking queue
                operation.
73          */
74         public int getNumberOfMonitoringThreads(Operation oper) {
75                 return this.monitoringThreadsAndQueueSize.get(oper).size
                        ();
76         }
77
78         /**
79          * Adds a new thread to be monitored because it is waiting on
80          * a certain queue operation.
81          * @param t The new thread.
82          * @param oper The BlockingQueue operation called.
83          * @return true if the collection of monitored threads has
84          * changed when the thread was added.
85          */
86         public boolean addThreadBlockingQueue(Thread t, Operation oper) {
87                 return this.monitoringThreadsAndQueueSize.get(oper).add(t
                        );
88         }
89
90         /**
91          * Removes a thred from the collection of monitored threads
                waiting
92          * on this operation.
93          * @param t The Thread instance.
94          * @param oper The BlockingQueue operation.
95          */
96         public void removeThreadBlockingQueue(Thread t, Operation oper) {
97                 if (Operation.put.equals(oper)) {
98                         size++;
99                 } else {
100                        size--;
101                }
```

```
102                 this.monitoringThreadsAndQueueSize.get(oper).remove(t);
103         }
104
105         /**
106          * Get the state in which the thread should be considering the
107          * evaluation of the capacity and size values being managed by
                this
108          * class.
109          * @param t The thread.
110          * @return the state of this thread considering the evaluation
111          * of the queue capacity and size.
112          */
113         public ThreadState getThreadStateForQueueSize(Thread t) {
114                 if (this.monitoringThreadsAndQueueSize.get(Operation.put)
                        .contains(t)) {
115                         if (this.capacity > this.size) {
116                                 return ThreadState.POSSIBLY_NOTIFIED;
117                         } else {
118                                 return ThreadState.WAITING;
119                         }
120                 } else if (this.monitoringThreadsAndQueueSize.get(
                        Operation.take).contains(t)){
121                         if (this.size > 0) {
122                                 return ThreadState.POSSIBLY_NOTIFIED;
123                         } else {
124                                 return ThreadState.WAITING;
125                         }
126                 } else {
127                         return ThreadState.UNKNOWN;
128                 }
129         }
130
131         /**
132          * Gets the set of threads being monitored regarding a
                BlockingQueue
133          * operation.
134          * @param oper The BlockingQueue operation.
```

```
135             * @return the set of threads being monitored regarding a
                    BlockingQueue
136             * operation.
137             */
138            public Set<Thread> getMonitoringThreads(Operation oper) {
139                    return this.monitoringThreadsAndQueueSize.get(oper);
140            }
141
142    }
```

Código Fonte G.11: Classe *WaitType.java*

```
1  /*
2   * Copyright (c) 2010 Universidade Federal de Campina Grande
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation; either version 2 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program; if not, write to the Free Software
16  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
        USA
17  */
18 package br.edu.ufcg.threadcontrol;
19
20 /**
21  * The possible wait types.
22  */
23 public enum WaitType {
24         WAIT_ON_OBJ_LOCK, WAIT_ON_QUEUE_EMPTY, WAIT_ON_QUEUE_FULL;
25 }
```