UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

DETECÇÃO AUTOMÁTICA DE VIOLAÇÕES DE PROPRIEDADES DE SISTEMAS CONCORRENTES EM TEMPO DE EXECUÇÃO

ANA EMÍLIA VICTOR BARBOSA

CAMPINA GRANDE – PB
ABRIL DE 2007

Detecção Automática de Violações de Propriedades de Sistemas Concorrentes em Tempo de Execução

Ana Emília Victor Barbosa

Dissertação submetida 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 Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Dalton Dario Serey Guerrero

(Orientador)

Jorge César Abrantes de Figueiredo

(Orientador)

Campina Grande, Paraíba, Brasil

B238d

2007 Barbosa, Ana Emília Victor

Detecção Automática de Violações de Propriedades de Sistemas Concorrentes em Tempo de Execução / Ana Emília Victor Barbosa. - Campina Grande, 2007.

109f: il..

Dissertação (Mestrado em Ciência da Computação), Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências.

Orientadores: Dalton Dario Serey Guerrero, Jorge César Abrantes de Figueiredo.

1. Monitoração. 2. Detecção de Violações. 3. Sistemas Concorrentes. 4. Propriedades Comportamentais. I. Título.

CDU – 004.4’233:004.415.2

"DETECÇÃO AUTOMÁTICA DE VIOLAÇÕES DE PROPRIEDADES DE SISTEMAS CONCORRENTES EM TEMPO DE EXECUÇÃO"

ANA EMÍLIA VICTOR BARBOSA

DISSERTAÇÃO APROVADA EM 20.04.2007

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO , D.Sc
Orientador

PROFº PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Examinadora

PROF. ADENILSO DA SILVA SIMÃO, D.Sc
Examinador

CAMPINA GRANDE – PB

Resumo

Neste trabalho propomos uma técnica que visa detectar violações de propriedades comportamentais automaticamente durante a execução de sistema de software concorrentes. A técnica foi inspirada na metodologia de desenvolvimento *Design by Contract* (DbC). DbC permite que os desenvolvedores adicionem aos programas asserções para que sejam verificadas em tempo de execução.

O uso de asserções para expressar propriedades de programas concorrentes (*multi-threaded*) e paralelos, entretanto, não é suficiente. Nesses sistemas, muitas das propriedades comportamentais de interesse, como vivacidade e segurança, não podem ser expressas apenas com asserções. Essas propriedades requerem o uso de operadores temporais. Neste trabalho, utilizamos Lógica Linear Temporal (Linear Time Logic - LTL) para expressar o comportamento desejado.

Para dar suporte a checagem do comportamento dos programas em tempo de execução, propomos uma técnica baseada em Programação Orientada a Aspectos, que permite que o programa seja continuamente monitorado (o comportamento é checado através do uso de autômatos que permite a deteção de comportamentos inesperados). Associada a cada propriedade comportamental existe um conjunto de pontos de interesse do código-fonte que devem obedece-lá. Esse pontos são então monitorados durante a execução do sistema através do uso de aspectos. Entre outros benefícios, a técnica permite que o sistema de software alvo seja instrumentado de maneira não intrusiva, sem alterar o código-fonte — particulamente, nenhum código do software alvo deve ser modificado para execução da monitoração.

Para validar este trabalho, desenvolvemos como prova de conceitos um protótipo que implementa a técnica e permite a monitoração de programas Java *multi-threaded*, chamado *DesignMonitor*. Essa ferramenta é apresentada e discutida através de um estudo de caso para demonstrar a aplicação da técnica.

Abstract

In this work we propose and develop a technique that allows to detect the violation of behavior properties of concurrent systems. The technique was inspired by the Design by Contract (DbC) programming methodology, which proposes the use of assertions and their evaluation at runtime to check programs behavior.

The use of simple assertions to express properties of concurrent and parallel programs, however, is not sufficient. Many of the relevant properties of those systems, such as liveness and security, can not be expressed with simple assertions. These properties require the use of temporal operators. In our work, we used Linear Time Logic (LTL) to specify the expected behavior.

To support the runtime checking of the program against the expected behavior, we propose a technique, based on Aspect-Oriented Programming, that allows the program to be continuously monitored (behavior is checked against automata that allows the detection of unexpected behaviors). Each property is mapped to a set of points of interest in the target program. Those points are then monitored during the system execution through aspects. Among other benefits, the technique allows the instrumentation of the target software to be performed automatically and in a non-intrusive way — in particular, no code must be changed to turn monitoring on or off.

To validate the work, we developed a proof of concept prototype tool that implements the technique and allows the monitoring of multi-threaded Java programs, called DesignMonitor. The tool was used in case study that has allowed the evaluation and the discussion of practical issues related with the technique.

Agradecimentos

Primeiramente, todo meu sentimento de gratidão a Deus pelo apoio incondicional em todos os momentos da minha vida.

Aos meus pais Benedito Glauco e Maria Solange, agradeço por todo o amor, cuidados, carinho, dedicação, compreensão e ensinamentos a mim sempre dispensados. A minha irmã e amiga, Ana Esther, pelo seu carinho, companheirismo, atenção e amor, não medindo esforços para que eu conquistasse mais esta etapa de minha vida. Ao meu irmão Gustavo e meu sobrinho Léo, agradeço pela ternura e pelos momentos de alegria. Enfim, agradeço à toda minha família, pessoas importantes sempre presentes em todos os momentos de uma forma toda especial de incentivar, mesmo sem estarem presentes.

A Loreno Oliveira, meu namorado, amigo e companheiro; agradeço por todos os momentos importantes que vivenciamos juntos, pela sua dedicação e sinceridade nas palavras. O carinho, o amor e a atenção que encontro em você são as razões da minha energia, motivação e persistência.

Meus sinceros agradecimentos:

Aos professores Dalton Serey e Jorge Abrantes pela orientação, dedicação e incentivo.

Aos professores Adenilso Simão, Patrícia Machado e Joseana Fechine pelas importantes contribuições. Fica registrada a gratidão ao professor Franklin Ramalho pelo apoio dado.

Aos professores Walfredo Cirne e Francisco Brasileiro, pelas idéias e sugestões que motivaram a realização deste trabalho.

A João Arthur, Ayla, Érica, Rogério, Amanda e William, pessoas sem as quais grande parte desse trabalho não seria possível.

Aos meus amigos do GMF: Afrânio, Amâncio, Amanda, André, Cássio, Daniel Aguiar, Elthon, Emanuela, Emerson, Fábio, Fabrício, Flávio, Francisco Neto, Helton, Jaime, Jairson, João Arthur, Laísa, Lile, Makelli, Paulo Eduardo, Roberto, Rogério, Talita, Taciano, Vinícius e Wilkerson. Um grupo que é exemplo de trabalho, companherismo e motivação, que com sua alegria e entusiasmo contribuíram para a realização deste trabalho. Em especial a Paulo Eduardo, um amigo que sempre compartilhou entusiasticamente de várias idéias e esteve presente diariamente na construção desse trabalho, tornando essa caminhada bem mais suave com sua amizade e companheirismo.

A CAPES pelo apoio financeiro.

A esta Universidade e seus Professores, principais responsáveis pela minha formação.

Por fim, tenho muito o que e a quem agradecer. Muitas foram as pessoas que diretamente e/ou indiretamente colaboraram até onde já cheguei. A essas pessoas meus agradecimentos especiais: amigos, funcionários da UFCG, colegas de curso (graduação e pós-graduação).

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

Confiabilidade é uma das mais importantes características para qualquer sistema de software. Contudo, a inerente complexidade dos sistemas de software torna difícil a tarefa de assegurar automaticamente que a implementação está conforme as propriedades desejáveis. Esse quadro é ainda mais complexo para programas concorrentes (*multithreaded*). Neste trabalho apresentamos uma técnica para especificar e detectar violações de propriedades comportamentais de programas concorrentes, inspirada na junção de especificações formais com a monitoração da execução de sistemas de software.

1.1 Verificação de sistemas concorrentes

O uso de técnicas para verificação de software, apropriadas ao contexto dos sistemas de software, permite detectar faltas que dificilmente seriam descobertas pelos desenvolvedores [Col87]. Com isso, a utilização de técnicas de verificação contribui para o desenvolvimento de sistemas mais robustos e confiáveis. De maneira geral, existem duas maneiras de classificar as técnicas de verificação de software: estáticas e dinâmicas.

No caso da verificação estática a análise do sistema de software é realizada sem a necessidade de que o código-fonte seja executado, ou seja, de forma estática. Dentre as técnicas de verificação de software estática podemos citar Verificação de Modelos (*model checking*) [Kat99]. Nessa técnica, a verificação é realizada a partir de um modelo abstrato do comportamento do sistema de software alvo, sem que o código-fonte do mesmo seja executado. Uma linguagem de modelagem comportamental é utilizada para construção do modelo

comportamental e a especificação das propriedades a serem verificadas nesse modelo são expressas através de alguma lógica temporal. Contudo, os modelos abstratos podem não representar corretamente o comportamento desejado dos sistemas de software devido à distância, tanto sintática quanto semântica, existente entre a linguagem de modelagem e a codificação. Além disso, essa técnica não assegura que a implementação dos sistemas satisfaz as mesmas propriedades do modelo especificado. Um dos motivos para tal inconsistência são as faltas, falhas ou defeitos introduzidas pelo ser humano no processo de codificação do software. Desta maneira, as técnicas de verificação de software devem também ser aplicadas sobre a implementação dos sistemas com o objetivo de detectar possíveis violações de propriedades comportamentais.

Já dentre as técnicas de verificação de software dinâmicas, àquelas aplicadas à implementação, testes de software se destaca como a mais popular. Teste de software é uma técnica de verificação realizada durante a execução do código-fonte. Dentre as diversas abordagens de teste de software temos os testes baseados em modelos, conhecidos como testes de conformidade. Estes são utilizados para verificar se o comportamento do código está conforme o modelo comportamental especificado [FMP04]. No entanto, a aplicação de testes de conformidade como técnica de verificação para sistemas concorrentes não é adequado devido ao não-determinismo inerente a estes sistemas. O problema na aplicação dessa técnica é a viabilidade de definir os casos de testes a serem verificados. Para uma dada entrada podem existir inúmeros cenários de execução de sistemas concorrentes. Com isso, os casos de testes podem cobrir apenas algumas das possíveis execuções dos sistemas concorrentes, deixando de verificar grande parte do comportamento dos sistemas de software.

Por outro lado, o interesse pela monitoração em tempo de execução de sistemas de software tem aumentado. O principal objetivo da monitoração é o de observar a execução do sistema de software para determinar esse comportamento observado está de acordo com o comportamento esperado. A vantagem dessa abordagem com relação a testes de software em tempo de produção é o de permitir exercitar cenários reais e inéditos ao processo de teste. A monitoração em tempo de execução pode ser utilizada para diversos fins, como *profiling*, análise de desempenho, otimização do software, assim como para a detecção, diagnóstico e recuperação de faltas [DGR04]. A detecção de faltas a partir da monitoração em tempo de execução permite observar o comportamento durante a execução dos sistemas e confrontar

com o comportamento esperado. Nesse caso, a monitoração do sistema de software alvo parte da especificação das propriedades comportamentais de interesse. Com base nessas especificações o sistema de monitoração pode ser dividido em dois módulos distintos: um para observar e outro para analisar o comportamento durante a execução dos sistemas. Contudo, a especificação e verificação de sistemas concorrentes é uma tarefa complexa, exigindo grande esforço na definição e especificação das propriedades comportamentais.

1.2 *Design by Contract*

Os sistemas de software têm tido uma importância cada vez maior na sociedade devido à crescente disseminação do uso destes para diversos fins. Concomitantemente, enquanto os sistemas de software crescem, tanto em tamanho quanto em complexidade, também aumenta a necessidade de verificar automaticamente se o comportamento dos sistemas desenvolvidos corresponde ao comportamento esperado. Neste contexto, *Design by Contract* (DbC) [Mey86] surge como uma metodologia de implementação que provê mecanismos para detecção de violações de sua especificação.

A idéia central de DbC é estabelecer um mecanismo para a definição precisa de contratos entre cada classe e seus clientes, que são definidos explicitamente no código-fonte como anotações. A especificação dos contratos em DbC fundamenta-se na verificação formal, na especificação formal e na lógica de Hoare. Dessa maneira, temos que um contrato é uma documentação formal, que define sem ambigüidades o comportamento esperado do sistema. As condições que devem reger a especificação do contrato são chamadas de asserções, especificadas através de pré-condições, pós-condições e invariantes. Os contratos são especificações de propriedades do software ligadas diretamente ao código. Uma das maneiras de confrontar os contratos com o código-fonte do programa é em tempo de execução. Assim, estando o código-fonte devidamente anotado e com o adequado suporte ferramental, torna-se possível confrontar os contratos com o código-fonte dos programas para a detecção de violações das especificações durante a execução dos sistemas.

O foco de DbC não é o de atestar corretude, mas evendiciar violações dos contratos sempre que possível. Essa metodologia é tipicamente adotada em programas seqüênciais. Contudo, é fato conhecido que o uso de DbC não é o bastante para verificação de sistemas de

software concorrentes. Uma vez que, DbC permite verificar apenas o comportamento para um determinado ponto da execução no tempo através das asserções (pré e pós-condições). Porém, mecanismos como concorrência implicam no não-determinismo, em que a ordem das ações leva à resultados diferentes. Além disso, sistemas concorrentes interagem em qualquer ponto da execução. As propriedades comportamentais de interesse em sistemas concorrentes, tais como vivacidade e segurança, não podem ser expressas utilizando somente pré e pós-condições. Uma vez que, não basta apenas dizer o estado antes e depois em um ponto da execução para verificar essas propriedades, é necessário expressar propriedades de estados intermediários ao longo do tempo.

1.3 *DesignMonitor*

Visando detectar automaticamente violações de propriedades comportamentais de um sistema de software concorrente, apresentamos a técnica *DesignMonitor*. Essa técnica foi inspirada nos conceitos associados a DbC, que permitem expressar o comportamento desejado através de contratos. Desse modo, é possível especificar parcialmente as propriedades comportamentais de interesse do sistema alvo. Além do mais, a especificação não possui um alto nível de abstração, no sentido de que as propriedades de interesse são expressas bem próximo ao código-fonte. Contudo, a especificação do comportamento desejado para sistemas concorrentes é expresso como ocorre na verificação de modelos através de alguma lógica temporal.

Para detectar violações das propriedades especificadas relacionadas com o comportamento (mudanças de estados) dos objetos ao longo do tempo do sistema é necessário observar o comportamento do sistema durante sua execução. Desta maneira, é realizada a monitoração em tempo de execução de sistemas de software alvo. A monitoração deve ser não intrusiva, sem haver a necessidade de alterar o código-fonte do sistema alvo. Além disso, a monitoração é realizada apenas em pontos de interesse do código expressos nas especificações das propriedades comportamentais. Contudo, o uso dessa técnica não deve demandar esforço adicional da equipe de desenvolvimento com a construção do monitor. A idéia é que o desenvolvedor deva se preocupar em especificar as propriedades comportamentais, as demais atividades de monitoração devem ser geradas automaticamente, como por exemplo a

geração automática de código.

O comportamento monitorado nos pontos de interesse são confrontados com o comportamento especificado através da máquina de estados referente a uma propriedade comportamental específica. Essas máquinas de estados são baseadas em autômatos de Büchi (LBA) [Büc62]. Eles são uma extensão de um autômato de estados finito para entradas infinitas, úteis para expressar o comportamento de sistemas não-determinísticos, como sistemas concorrentes.

O uso da técnica *DesignMonitor* deve permitir detectar violações das propriedades comportamentais especificadas de maneira silenciosa. Uma vez especificadas as propriedades comportamentais utilizar a técnica *DesignMonitor* deve ser transparente. Contudo, o custo, lógico e de atividade, extra associado ao utilizar essa técnica deve ser minímo.

A Figura 1.1 ilustra uma visão geral da técnica *DesignMonitor*. As propriedades comportamentais especificadas são a entrada da técnica *DesignMonitor*. Cada propriedade comportamental do sistema de software alvo é composta pelo comportamento desejado e os pontos de interesse no código que devem obdecer tal propriedade. O comportamento desejado é expresso como uma fórmula temporal através da lógica proposicional temporal linear (Lógica Linear Temporal - LTL) [Kat99]. Para cada fórmula LTL especificada são definidos os pontos de interesse no código-fonte que devem obedecer essa propriedade comportamental. O *DesignMonitor* é composto por dois módulos: observador e analisador. O módulo observador baseia-se nos pontos de interesse para gerar o código de monitoração. A monitoração do comportamento do sistema alvo é através da Programação Orientada a Aspectos (POA). Essa abordagem permite que a instrumentação seja de maneira não intrusiva (sem alterar o código-fonte de origem). O comportamento observado é encaminhado para o módulo analisador, que confronta com o comportamento desejado expresso através de fórmulas LTL. A análise do comportamento observado em face ao comportamento especificado numa máquina de estados que baseia-se no autômato de Büchi equivalente a fórmula LTL realizado pelo módulo analisador. Desta maneira, o comportamento atual observado em um ponto de interesse é confrontado com o comportamento desejado considerando as mudanças de estados ocorridas ao longo do tempo durante a execução do sistema de software alvo. Caso seja detectada alguma violação do comportamento desejado o usuário é informado da mesma.

Diferentemente de outras técnicas de verificação de sistemas de software - como testes,

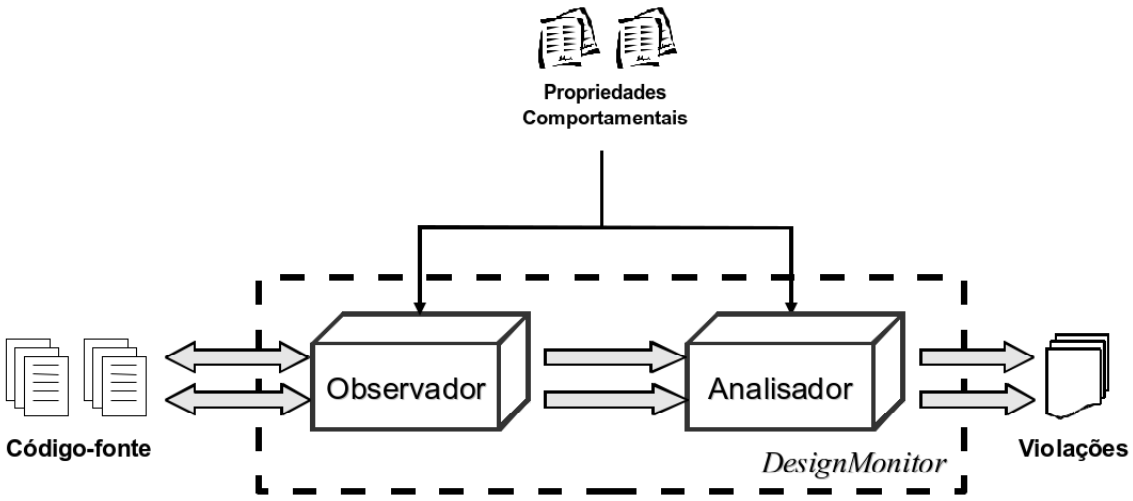

Figura 1.1: Visão geral da técnica *DesignMonitor*.

verificação de modelos e prova de teoremas, que visam assegurar a corretude dos programas - a técnica *DesignMonitor* avalia se o comportamento da execução corrente do programa viola alguma propriedade comportamental especificada para cenários reais e inéditos durante a execução do sistema alvo.

Como prova de conceito e para permitir a automatização desta técnica foi desenvolvido um protótipo da ferramenta, também chamada de *DesignMonitor*, que permite detectar violações de propriedades comportamentais através da monitoração do software alvo em tempo de execução.

1.4 Estrutura da dissertação

No Capítulo 2 apresentamos os conceitos necessários para a melhor compreensão deste trabalho. Inicialmente serão apresentados os conceitos básicos que compõem DbC e a monitoração de sistemas de software. Posteriormente, descreveremos os formalismos utilizados para a especificação das propriedades comportamentais, para sistemas de software concorrentes e paralelos, a serem confrontados com o comportamento monitorado. Essas propriedades são expressas como propriedades temporais, baseadas na lógica proposicional temporal linear (Lógica Linear Temporal - LTL). Em seguida, apresentaremos os principais conceitos da Programação Orientada a Aspectos, abordagem utilizada para realizar a monitoração dos

sistemas de software durante sua execução.

O Capítulo 3 apresenta a técnica proposta para verificação da conformidade comportamental do código desenvolvido com o comportamento esperado. Em seguida, no Capítulo 4 apresentamos detalhes da arquitetura e da implementação do protótipo da ferramenta, denominada *DesignMonitor*. Neste capítulo, discutimos detalhadamente os módulos que compõem a arquitetura da ferramenta *DesignMonitor*, apresentando uma visão geral da implementação de cada um desses módulos. Para demonstrar a aplicação da técnica e o uso da ferramenta *DesignMonitor* em um sistema de software real utilizamos um estudo de caso, apresentado no Capítulo 5.

Em seguida, no Capítulo 6 apresentamos uma breve descrição de alguns trabalhos relacionados identificados a partir de uma revisão bibliográfica. Por fim, baseado nos resultados obtidos, apresentamos nossas conclusões e as perspectivas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo, abordamos os fundamentos teóricos sobre os quais este trabalho foi desenvolvido. Inicialmente, apresentamos os conceitos básicos das abordagens DbC e monitoração de sistemas de software. A técnica proposta neste trabalho baseia-se nessas duas abordagens para permitir a verificação da conformidade entre o comportamento observado e o comportamento esperado de sistemas de software concorrentes e paralelos. Em seguida, apresentamos os formalismos utilizados neste trabalho para a especificação e verificação do comportamento monitorado, especificamente Lógica Temporal Linear (LTL) e autômatos de Büchi. Também, descrevemos como se dá a transformação de fórmulas LTL em autômatos de Büchi. Por fim, apresentamos os principais conceitos relacionados a Programação Orientada a Aspectos que é utilizada para instrumentar o sistema de software com o código de monitoração.

2.1 *Design by Contract*

2.1.1 Introdução

O termo *Design by Contract* (DbC) foi inicialmente apresentado por Bertrand Meyer em 1986 [Mey86]. Baseia-se nos trabalhos seminais de Floyd, Hoare, e Dijkstra sobre a construção (e análise) de programas. DbC é um método que promove o desenvolvimento de sistemas de software mais confiáveis, na medida em que provê mecanismos para verificar a consistência entre o código-fonte e sua especificação.

A idéia central de DbC é estabelecer contratos entre os fornecedores e seus clientes através de anotações no código-fonte. Os contratos expressam condições que o cliente deve garantir ao invocar os fornecedores, conhecidas como pré-condições. Após a execução, por outro lado, o fornecedor que deve garantir outras condições, chamadas de pós-condições. A Figura 2.1 ilustra o esquema de contratos entre cliente e fornecedor em DbC.

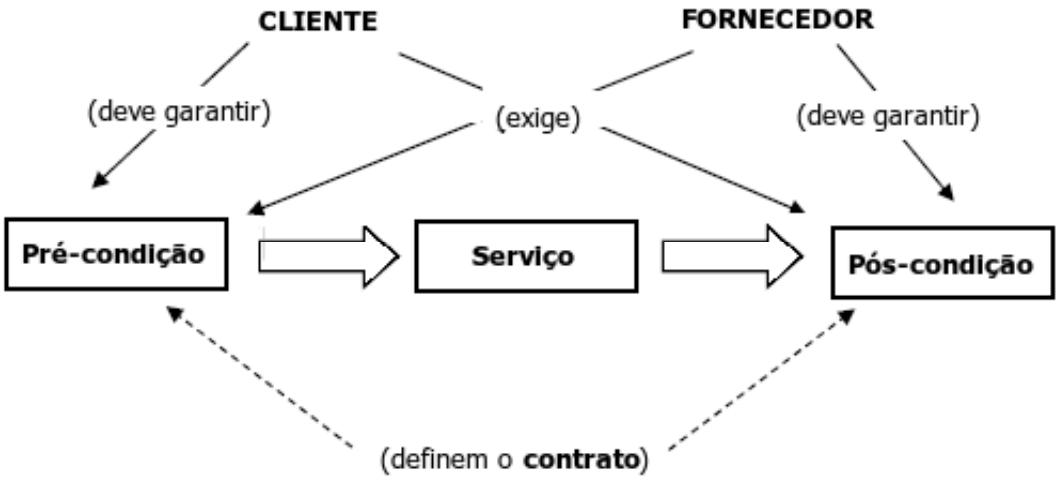

Figura 2.1: Conceitos básicos que constituem DbC.

2.1.2 Especificação de contratos

Em DbC, considerando o contexto de sistemas de software, o cliente pode ser visto como o código que invoca o método de uma classe e o fornecedor como o método invocado. As cláusulas que regem os contratos são chamadas de asserções. Pré-condições e pós-condições são asserções que definem, respectivamente, os benefícios e as obrigações que devem ser garantidos antes e após a invocação de cada método individualmente.

Exemplo 2.1 *Considere uma pilha de tamanho limitado, fazendo uso de um array como estrutura de dados. Temos que uma instância da classe deve ser capaz de empilhar e desempilhar um objeto não-nulo qualquer. A pilha pode armazenar qualquer tipo de objeto. Porém, caso a pilha não esteja vazia, todos os elementos devem ter o mesmo tipo do primeiro objeto empilhado. Considerando a operação de desempilhar um objeto da pilha. O contrato desse método pode ser documentado como ilustrado no Código 2.1. A pré-condição para o método desempilhar é que a pilha não estaja vazia. As obrigações do método, nesse caso,*

é passar o objeto que está no topo da pilha e o tamanho da pilha diminuir em um. A pré-condição especifica o que deve ser verdadeiro antes do método ser invocado (linha 3). As pós-condições estabelecem os compromissos que devem ser atendidos pelo método após ter sido executado (linhas 4 e 5).

```
1 ...
2 pré-condição pilha.length != 0;
3 pós-condição resultado == \old(pilha.get(\old(pilha.length) - 1));
4 pós-condição pilha.length == \old(pilha.length) - 1;
5 ...
```

Código 2.1: Pré e pós-condições para desempilhar um objeto de uma pilha.

No entanto, é necessário fazer uso de algum mecanismo para expressar propriedades globais sobre instâncias de uma classe. Para isto, um outro tipo de asserção em um contrato é a invariante, que como o próprio nome denota, são propriedades que devem sempre ser válidas ao longo do ciclo de um objeto. Uma invariante está associada a uma classe (ou estrutura de dados). A invariante deve ser satisfeita logo após a criação de toda instância da classe e todo método não-estático deve garantir que a invariante seja preservada após sua execução, se esta for imediatamente antes de sua invocação.

Exemplo 2.2 *Considere o Exemplo 2.1, além das condições descritas, uma pilha nunca deve ser nula e todos os seus objetos nunca devem ser nulos. No Código 2.2, podemos observar a descrição das invariantes que estabelecem tais condições*

```
1 ...
2 invariante pilha != null;
3 invariante (\forall int i; 0 <= i && i < pilha.length;
4             pilha[i] != null);
5 ...
```

Código 2.2: Invariantes na implementação de uma pilha.

2.1.3 Verificação de contratos

O papel das asserções não é descrever casos especiais, mas expressar situações esperadas no uso de um determinado método. Quando essas situações esperadas não são atendidas ocorre uma violação. A violação de uma especificação deve ser considerada como o resultado de

um erro no software. Em outras palavras, a violação de uma pré-condição indica um erro de quem invocou o método. Enquanto, a violação de uma pós-condição representa um erro no próprio método.

Embora os conceitos de pré e pós-condições no desenvolvimento de software tenham sido introduzidos desde a década de 60 por Hoare, Floyd e Dijkstra, o surgimento de novas técnicas de verificação que confrontam os contratos com o código-fonte tem aumentado interesse acerca de DbC. A verificação de contratos pode ser realizada de maneira dinâmica ou estática. Na verificação dinâmica, realizada em tempo de execução, as chances de alguma violação ser detectada depende diretamente da qualidade dos cenários de uso escolhidos. A verificação estática é realizada em tempo de compilação, fazendo uso de uma técnica que vai além da simples verificação de tipos realizada pelos compiladores convencionais. Dessa maneira, a verificação estática é muito mais audaciosa, pois tenta estabelecer a correção do código para todos os possíveis caminhos de execução, o que não acontece na verificação dinâmica.

2.1.4 Suporte a DbC

Para fazer uso adequado de DbC é necessário utilizar um suporte ferramental apropriado. As ferramentas devem automatizar ou no mínimo facilitar as tarefas de leitura, escrita e verificação da conformidade da sintaxe e semântica das anotações.

Os contratos em DbC são definidos usando a própria linguagem de programação ou uma meta-linguagem, como exemplo *Java Modeling Language* (JML) [LBR06a]. Os contratos são convertidos em código pelo compilador da linguagem de programação ou por um pré-compilador no caso das meta-linguagens. JML é uma linguagem que permite a especificação formal e detalhada de programas Java através de anotações no código-fonte. A ferramenta básica para o uso de JML é a *JMLChecker*, que tem como objetivo checar a sintaxe e manter a especificação minimamente consistente.

Atualmente, existem diversas ferramentas de suporte ao uso da abordagem DbC para verificação da conformidade entre as anotações e o código-fonte. Uma das maneiras de checar as anotações é verificar os contratos dinamicamente. Dessa maneira, caso uma violação da especificação seja detectada, o desenvolvedor é notificado em tempo de execução. Considerando os contratos especificados em JML, as principais ferramentas para verificação

dinâmica são JMLRAC (*JML Runtime Assertion Checker*) e *JMLUnit*. O objetivo da ferramenta *JMLRAC* é detectar e notificar violações entre a especificação e o código através da execução das asserções, enquanto o software está sendo executado. No caso da ferramenta *JMLUnit*, utiliza as anotações JML para gerar automaticamente testes de unidade do sistema para capturar exceções derivadas da violação de algum contrato. Em outras palavras, os testes de unidade gerados são usados para exercitar as classes sob teste e decidir se estas cumprem ou não o que sua especificação estabelece.

No caso da verificação estática, podemos destacar a ferramenta *Esc/Java2* [Kin]. A ferramenta *ESC/Java2* suporta grande parte da linguagem JML, e para este subconjunto verifica a consistência entre o código e as anotações. Diferentemente das demais ferramentas apresentadas, *ESC/Java2* não depende das anotações JML para ser utilizada - apesar destas contribuírem bastante para a sua eficácia.

2.2 Monitoração de sistemas concorrentes

2.2.1 Introdução

Com o aumento da complexidade e da natureza ambígua e os altos custos de testar os sistemas de software, o interesse pela monitoração em tempo de execução de sistemas de software tem sido renovado. A monitoração pode ser utilizada para diversos fins, como i)*profiling*, ii) análise de desempenho, iii) otimização do software, iv)detecção, v) diagnóstico e vi) recuperação de faltas [DGR04]. O objetivo da monitoração em tempo de execução na detecção de faltas é observar o comportamento do software para determinar se condiz com o comportamento esperado.

Segundo [DGR04], um sistema de monitoração é composto basicamente por dois módulos:

1. **módulo observador** monitora o caminho (*trace*) de execução;
2. **módulo analisador** avalia as propriedades observadas.

A Figura 2.2 [DGR04] ilustra uma visão geral em alto-nível de um monitor. Temos que a partir da especificação dos requisitos, o monitor observa a execução do sistema de software

alvo nos pontos de interesse (*módulo observador*) e verifica se o comportamento observado condiz com o comportamento especificado (*módulo analisador*).

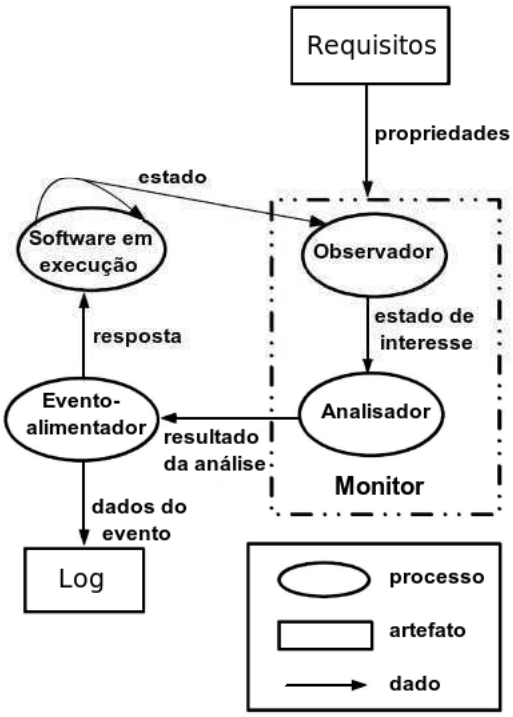

Figura 2.2: Visão de alto-nível de um Monitor em tempo de execução.

2.2.2 Observadores e analisadores

Como visto, o monitor é composto por dois módulos: observador e analisador. A entrada desses módulos são os requisitos do sistema de software alvo. Ambos os módulos trabalham em conjunto para verificar se comportamento do código desenvolvido obedece os requisitos desejados.

Um requisito está diretamente relacionado a determinados pontos de interesse dentro do código-fonte do sistema de software. Durante a execução do sistema, o módulo observador acompanha o comportamento nesses pontos de interesse de cada requisito. Esse comportamento observado é capturado e enviado para o módulo analisador, que o confronta com o comportamento esperado. Caso o comportamento observado não esteja de acordo com o comportamento esperado, o analisador dispara o evento-alimentador. O evento-alimentador é o mecanismo responsável por capturar e comunicar o resultado da monitoração ao usuá-

rio. Essa comunucação da detecção de uma violação pode requerer do sistema de software o início de uma ação, tal como parar o programa, incorporar uma rotina de recuperação no sistema de software alvo, ou a emissão de dados da violação em um arquivo de *log*.

Exemplo 2.3 *Tomemos como exemplo o seguinte requisito de um sistema de software concorrente: durante a execução de um determinado sistema de software nunca dois processos podem acessar uma região crítica simultaneamente. Em outras palavras:*

"Quando um processo entrar numa região crítica, o número de processos na região crítica deve ser exatamente um".

Durante a execução do sistema de software o módulo observador verifica o estado do programa e determina se está ou não numa região crítica. Quando é detectado que o programa está numa região crítica, o analisador checa se o número de processos na região crítica é apenas um. Caso o número de processos existentes na região crítica seja maior que um, temos que a propriedade foi violada. Então, o monitor deve responder ao usuário de alguma forma, como a emissão de dados desse evento em um arquivo de log.

2.2.3 Especificação de requisitos

Segundo Delgado [DGR04] a linguagem de especificação de requisitos em um sistema de monitoração define as propriedades monitoradas, o nível de abstração da especificação, e a expressividade da linguagem (tipo da propriedade e nível de monitoração), conforme ilustrado na Figura 2.3.

Tipo da Linguagem - a linguagem utilizada para especificar uma propriedade pode ser baseada em autômatos ou em HL-VHL. A categoria HL-VHL denota que a linguagem de especificação é funcional, orientada a objetos, imperativa ou uma extensão da linguagem do código de origem.

Nível de Abstração - se refere ao suporte que a linguagem provê para as propriedades especificadas e o conhecimento sobre o domínio, o *design*, ou implementação.

Tipo da Propriedade - são considerados dois tipos de propriedades: segurança ou temporal. Uma propriedade de segurança expressa que algo ruim nunca ocorre. As propriedades de segurança incluem, por exemplo, invariantes, as propriedades que definem uma seqüência de eventos, as propriedades que verificam valores das variáveis, e as propriedades que tratam

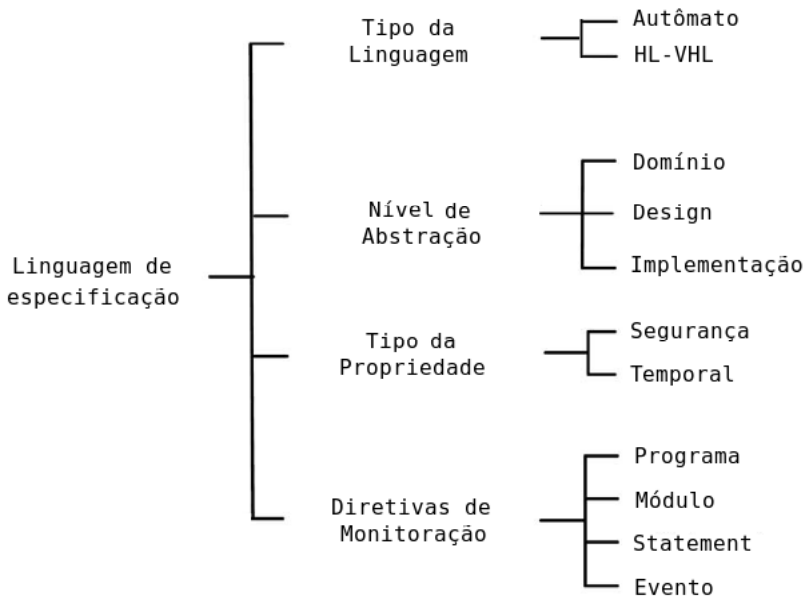

Figura 2.3: Classificação das linguagens de especificação de um monitor.

do alocamento de recursos. A categoria temporal inclui propriedades tais como vivacidade assim como as propriedades de sincronização.

Diretivas de Monitoração - uma propriedade pode ser avaliada em diferentes níveis: programa, módulo, *statement*, e evento. As propriedades a nível de programa são aquelas especificadas nas *threads* ou nas relações entre linhas de execução. As propriedades a nível de módulo são aquelas especificadas nas funções, procedimentos, métodos ou componentes, tais como tipos de dados ou classes abstratas. As propriedades do nível *statement* são aquelas especificadas para um *statement* em particular. As propriedades a nível de evento são aquelas que são definidas baseadas na mudança de estado ou em uma seqüência de mudanças de estados.

2.3 Especificação de sistemas concorrentes

Nesta seção apresentamos os formalismos utilizados na especificação e verificação de sistemas concorrentes. Inicialmente, descrevemos os paradigmas que podem ser utilizados na especificação do comportamento dos sistemas de software. Dentre estes, temos que as especificações para sistemas concorrentes podem ser escritas através de lógicas temporais, ou

seja, como fórmulas lógicas. Em seguida, apresentamos uma possível especificação de um comportamento para um sistema concorrente através da Lógica Temporal Linear (*Linear Temporal Logic* - LTL). Além disso, mostramos que cada fórmula LTL pode ser expressa como um autômato de Büchi (modelo que expressa o comportamento descrito em LTL). E por fim, apresentamos como uma fórmula LTL pode ser transformada em um autômato de Büchi.

2.3.1 Estilos de especificação

As especificações formais são expressas a partir de uma linguagem formal, que possui uma sintaxe (forma) e semântica (significado) bem definidas, baseadas geralmente em conceitos formais utilizados na matemática, como conjuntos e funções [Mey85].

As técnicas para especificação formal sob o comportamento se diferenciam principalmente com relação ao paradigma de especificação utilizado. A sua escolha é bastante importante de acordo com as características do sistema de software a ser especificado. Segundo [vL00] os paradigmas de especificação do comportamento de um sistema podem ser classificados como:

- **especificação baseada em história** - baseia-se no princípio da especificação de um sistema através de um conjunto de propriedades admissíveis ao longo do tempo. Essas propriedades são especificadas em lógica temporal e são interpretadas sobre uma estrutura de estados. Exemplos: LTL, CTL.

- **especificação baseada em estados** - o sistema é caracterizado a partir dos estados admissíveis. As propriedades de interesse são especificados em termos de asserções (invariantes, pré e pós-condições). Exemplos: linguagens como Z, B ou VDM.

- **especificação baseada em transições** - a caracterização de um sistema se dá com a definição das regras de evolução na transição de um estado para outro. As propriedades são especificadas através de conjunto de funções de transições. Dado um estado e um evento qualquer, o conjunto de funções de transições deve fornecer o estado resultante. Exemplos: *Statecharts*, PROMELA.

- **especificação funcional** - um sistema é especificado como um conjunto estruturado de funções matemáticas. Podendo ser classificada como *especificação algébrica* (exemplos: OBJ, ASL, PLUSS, LARCH) ou *funções de alta-ordem* (exemplos: PVS, HOL).

- **especificação operacional** - especificar o sistema como uma coleção estruturada de processos que podem ser executadas por uma máquina abstrata. Exemplos: Redes de Petri, Álgebra de Processos (CSP, π-calculus).

O nosso interesse está voltado para o comportamento ao longo do tempo dos sistemas de software. Contudo, sistemas de software que requerem o uso de características como paralelismo e concorrência são, geralmente, compostos por múltiplas *threads* (linhas de execução), também conhecido como sistemas *multithreaded*. Nesse tipo de sistema, um processo pode ter diferentes partes do seu código sendo executadas concorrentemente ou simultaneamente, ou seja, em um único processo é possível ter mais de uma tarefa sendo efetuada ao mesmo tempo. Dessa maneira, problemas como *deadlocks*, *livelocks* e *starvations* são comuns em sistemas de software com tais características. Com isso, uma das principais propriedades de interesse nesses sistemas de software é com relação ao comportamento das *threads*. Um exemplo disso, é um programa *multithread* que necessita isolar as *threads* em módulos para evitar *deadlocks*[1]. Assim, é preciso observar o comportamento das *threads* nos objetos de cada módulo que compõem o sistema de software. Desta maneira, para descrever o comportamento ao longo do tempo optamos pelo paradigma de especificação baseado em história, com as propriedades expressas através da linguagem temporal LTL.

2.3.2 Propriedades temporais

A especificação de um sistema de software é composto por um conjunto de propriedades temporais que o definem. Essas propriedades temporais descrevem as funcionalidades esperadas que o sistema de software deve prover na resolução de problemas.

Para sistemas concorrentes, as propriedades temporais são geralmente classificadas em: segurança (*safety*) e vivacidade (*liveness*).

[1] Propriedade (indesejável) de um conjunto de processos que estão bloqueados indefinidamente aguardando a liberação de um recurso por um outro processo que, por sua vez, aguarda a liberação de outro recurso alocado ou dependente do primeiro processo.

1. propriedades de segurança (*safety*) - expressam "comportamentos que não queremos que ocorram", em outras palavras "comportamentos indesejáveis não acontecem". Essa propriedade se relaciona com a invariância de uma determinada característica, como por exemplo, a ausência de *deadlock*;

2. propriedades de vivacidade (*liveness*) - expressam "comportamentos que queremos que (sempre) ocorram", em outras palavras essa propriedade estabelece a necessidade ou a possibilidade de que "comportamentos desejáveis que queremos que aconteçam".

Essa classificação foi originalmente proposta por Lamport [Lam77], a partir do qual também surgiu os termos da ocorrência de *coisas desejáveis* ou *indesejáveis*. Se interpretarmos essas *coisas* como fatos passíveis de verificação em tempo finito, podemos dizer que as propriedades temporais se distinguem pela ocorrência de *coisas desejáveis*.

Em sistemas concorrentes a sua especificação inclui, em regra geral, uma propriedade de segurança e uma de vivacidade. Além disso, qualquer propriedade temporal sobre o comportamento de um sistema pode ser expressa como uma combinação de uma propriedade de cada um dos dois tipos referidos [Lam77].

2.3.3 Lógica temporal linear

LTL é um formalismo utilizado para a especificação de propriedades de sistemas concorrentes e reativos, que foi introduzida por Pnueli em [Pnu77]. LTL considera uma computação linear, ou seja, apenas um estado sucessor pode ocorrer durante um processamento. As fórmulas em LTL expressam relações de ordem, sem a necessidade de recorrer à noção explícita de tempo. A sintaxe e a semântica de LTL são apresentadas a seguir.

Sintaxe

Proposições atômicas são os elementos básicos que constituem a sintaxe LTL. Uma proposição atômica p é uma setença que informa algo a respeito de um determinado estado do sistema, que pode ser interpretada como verdadeira ou falsa. Alguns exemplos de proposições atômicas são *"x é igual a zero"*, *"x é menor que 200"*, *"a linha de execução t está em execução"*, ou *"o recurso r está alocado"*.

Definição 2.1 (Sintaxe de LTL [Kat99]) *Seja p uma proposição atômica e, ϕ e ψ fórmulas. São fórmula básicas LTL:*

1. *p é uma fórmula;*
2. *$\neg\phi$ é uma fórmula;*
3. *$\phi \vee \psi$ é uma fórmula;*
4. *$X\phi$ é uma fórmula (lê-se "próximo ϕ");*
5. *$\phi \cup \psi$ é uma fórmula (lê-se "ϕ até que ψ");*
6. *nada mais é uma fórmula.*

Temos que $\neg$ denota negação e $\vee$ denota disjunção. Os operadores booleanos $\wedge$ (conjunção), $\Rightarrow$ (implicação) e $\Leftrightarrow$ (equivalência), $true$ e $false$ são definidos como:

- $\phi \wedge \psi \equiv \neg(\neg\ \phi \vee \neg\ \psi)$
- $\phi \Rightarrow \psi \equiv \neg\ \phi \vee \psi$
- $\phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$
- $true \equiv \phi \vee \neg\phi$
- $false \equiv \neg true$

Os operadores temporais G (lê-se "*sempre*", ou "*globalmente*", ou ainda "*invariavelmente*") e F (lê-se "*futuramente*") são definidos a partir dos operadores introduzidos anteriormente da seguinte forma:

$$G\phi \ \equiv \ \neg F\neg\phi$$

$$F\phi \ \equiv \ true \cup \phi$$

Exemplo 2.4 *Considere a seguinte propriedade para um determinado sistema concorrente: "quando um processo entrar numa região crítica, o número de processos nessa região crítica deve ser exatamente um". Assim temos que os processos nunca estão simultaneamente na região crítica. Então, seja os processos $p1$ e $p2$, logo:*

$$G\neg(p1 \wedge p2)$$

Semântica

A sintaxe nos fornece a maneira correta para a construção das fórmulas LTL, mas não dá uma interpretação aos operadores. Formalmente, uma fórmula LTL é interpretada como uma seqüência inifinita de estados. Intuitivamente, temos que:

- $X\phi$ significa que a fórmula ϕ é válida no próximo estado;
- $F\phi$ significa que a fórmula será válida em algum momento futuro;
- $G\phi$ significa que a fórmula é sempre válida;
- $\phi \cup \psi$ expressa que ϕ é válida ao longo de toda uma seqüência de estados consecutivos até a ocorrência de ψ.

As fórmulas LTL também podem ser representadas através de uma espécie de grafo de acessibilidade, denominado Estrutura de Kripke [Kat99].

Definição 2.2 (Estrutura de Kripke) *Uma estrutura de Kripke $\mathcal{M}$ é uma tupla $(S, I, R, Label)$, onde:*

1. *S é um conjunto finito de estados;*
2. *$I \subseteq S$ é um conjunto de estados iniciais;*
3. *$R \subseteq S \times S$ é uma relação de transição satisfazendo $\forall s \in S.(\exists s' \in S.(s, s') \in R)$;*
4. *$Label : S \rightarrow 2^{AP}$, associando a cada estado s de S, proposições atômicas $Label(s)$ que são válidas em s.*

Uma estrutura de Kripke é uma máquina de estados finita que representa o comportamento de um sistema. Cada estado do sistema é rotulado com proposições atômicas que são verdadeiras no estado correspondente.

Exemplo 2.5 *Considere um sistema tolerante a falhas, ilustrado na Figura 2.4, formado por três processadores que geram resultados para um quarto que é capaz de eleger majoritariamente qual resposta utilizar [Kat99].*

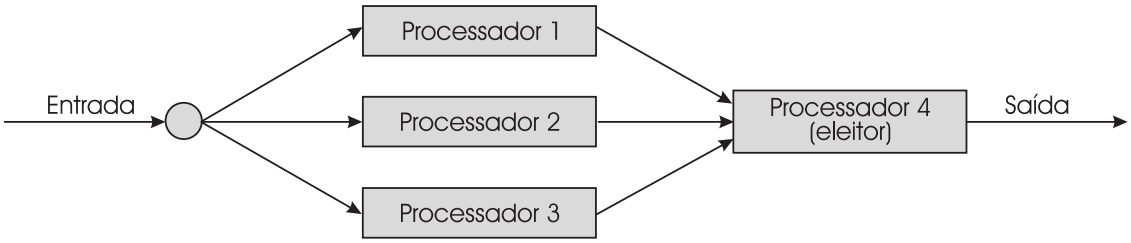

Figura 2.4: Um sistema tolerante a falhas.

Inicialmente todos os componentes estão operacionais, porém sujeitos a falhas durante uma execução. Assim, o estado $S_{i,j}$ modela que i processadores ($0 \leq i < 4$) e j eleitores majoritários ($0 \leq j \leq 1$) estão operacionais. Quando um componente falha ele pode ser reparado e voltar a funcionar. Considere que apenas um componente pode ser reparado por vez. Quando o eleitor falha, todo o sistema pára de funcionar. O conjunto de proposições atômicas deste problema é $AP = \{up_i | 0 \leq i < 4\} \cup \{down\}$. A proposição $\{up_0$ denota que apenas o processador eleitor está operacional, $\{up_1$ denota que além do processador eleitor, um outro também está operacional e assim por diante. A proposição $down$ denota que todo o sistema não está funcionando.

Uma estrutura de Kripke para este sistema tem os seguintes componentes:

- $S = \{S_{i,1} | 0 \leq i < 4\} \cup \{S_{0,0}\}$;
- $I = \{S_{3,1}\}$;
- $R = \{(S_{i,1}, S_{0,0} | 0 \leq i < 4)\} \cup \{(S_{0,0}, S_{3,1})\} \cup \{(S_{i,1}, S_{i,1} | 0 \leq i < 4)\} \cup \{(S_{i,1}, S_{i+1,1} | 0 \leq i < 3)\} \cup \{(S_{i+1,1}, S_{i,1} | 0 \leq i < 3)\}$;
- $Label(S_{0,0}) = \{down\}$ *e* $Label(S_{i,1}) = \{up_i\}$, *para* $0 \leq i < 4$.

Graficamente, a estrutura de Kripke para esse problema é ilustrado pela Figura 2.5.

Para se definir formalmente a semântica de LTL, o conceito de caminho também deve ser formalizado.

Definição 2.3 (Caminho) *Um caminho em $\mathcal{M}$ é uma seqüência infinita de estados $s_0,\ s_1,\ s_2,\ \ldots$ tal que $s_0 \in I$ e $(s_i, s_{i+1}) \in R$ para todo $i \geq 0$.*

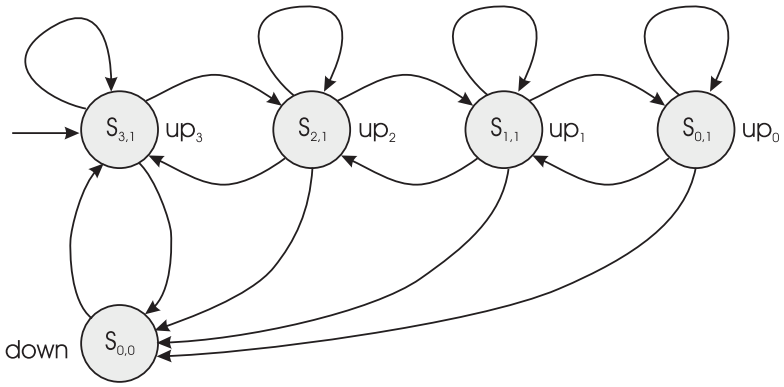

Figura 2.5: Estrutura de Kripke do Exemplo 2.5.

Portanto, um caminho é uma seqüência infinita de estados que representa uma possível execução do sistema a partir do seu estado inicial. $\sigma[i]$ denota o $(i+1)$-ésimo estado de σ e σ^1 representa o sufixo de σ obtido pela remoção do(s) i-primeiro(s) estados de σ. A função $Caminhos(s)$ determina todos os possíveis caminhos da estrutura $\mathcal{M}$ que se iniciam no estado s.

Uma vez definida a estrutura na qual uma fórmula LTL é interpretada, sua semântica pode ser então formalmente definida através da relação de satisfação, denotada por $\models$, e definida formalmente a seguir.

Definição 2.4 (Semântica de LTL) *Sejam $p \in AP$ uma proposição atômica, σ caminho infinito e ϕ, ψ fórmulas LTL, a relação de satisfação, denotada por $\models$, é definida por:*

- $\sigma \models p \Leftrightarrow p \in Label(\sigma[0])$
- $\sigma \models \neg\phi \Leftrightarrow not(\sigma \models \phi)$
- $\sigma \models \phi \wedge \psi \Leftrightarrow (\sigma \models \phi)\ e\ (\sigma \models \psi)$
- $\sigma \models X\phi \Leftrightarrow \sigma^1 \models \phi$
- $\sigma \models \phi \cup \psi \Leftrightarrow \exists j \geq 0, (\sigma^j \models \psi\ e\ (\forall 0 \leq k < j, \sigma^k \models \phi))$

2.3.4 Autômatos de Büchi

Um autômato de Büchi (LBA) [Büc62] é uma extensão de um autômato de estados finito para entradas infinitas. Os autômatos finitos podem ser vistos como reconhecedores de pa-

lavras. As palavras são definidas como seqüências finitas de elementos de um alfabeto Σ. Denotamos por Σ^* como o conjunto de todas as palavras finitas formadas a partir do alfabeto Σ. Já os autômatos de Büchi são reconhecedores de palavras infinitas. O conjunto das palavras infinitas formadas a partir de elementos de Σ é denotado por Σ^ω.

Autômatos que aceitam palavras infinitas são úteis para especificar o comportamento de sistemas não-determinísticos, tais como sistemas concorrentes e reativos.

Definição 2.5 (Autômato de Büchi) *Um autômato de Büchi A é uma 6-tupla $(\Sigma, S, S^0, \rho, F, \ell)$, onde:*

1. Σ *é um conjunto finito e não-vazio de símbolos;*
2. S *é um conjunto finito e não-vazio de estados;*
3. $S^0 \subseteq S$ *é um conjunto não-vazio de estados iniciais;*
4. $\rho : S \rightarrow 2^S$ *uma função de transição;*
5. $F \subseteq S$ *conjunto de estados de aceitação;*
6. $\ell : S \rightarrow \Sigma$ *função de rotulação.*

$\rho(s)$ é um conjunto de estados do autômato A que podem ser alcançados a partir de s, ou seja, $s \rightarrow s^{'}$ se e somente se $s^{'} \in \rho(s)$.

Definição 2.6 (Execução de um autômato de Büchi rotulado) *Seja o autômato de Büchi rotulado A, temos que uma execução π é uma seqüência de estados $\pi = s_0 s_1 ...$ tal que $s_0 \in S^0$ e $s_i \rightarrow s_{i+1}$ para todo $i \geq 0$. Seja $\lim(\pi)$ o conjunto de estados que ocorrem em σ freqüentemente infinita vezes. Uma execução π é chamada aceita, se e somente se, $\lim(\pi) \cap F \neq \varnothing$. Uma palavra $\omega = a_0 a_1 ... \in \Sigma^\omega$ é aceita se existe uma execução aceita $\pi = s_0 s_1 ... s_n$tal que $\ell(s_i = a_i)$ para todo $i \geq 0$.*

A idéia do autômato de Büchi é que uma palavra seja aceita, se e somente se, ao ser processada o autômato passa infinitas vezes por algum estado de aceitação. Observe que como o conjunto de estados é finito, então em qualquer seqüência infinita de estados deve haver pelo menos um estado que se repetirá infinitamente.

Formalmente, definimos $lim(\pi)$ como o conjunto dos estados que se repete infitamente em uma execução π do autômato. Dizemos que a execução π é aceitável se e somente se $lim(\pi) \cap F \neq \emptyset$.

Dizemos que uma palavra $\omega = a_0 a_1 ... \in \Sigma^\omega$ é reconhecida por um autômato de Büchi A se existe alguma execução aceitável do autômato $s_0, s_1, ...$ tal que $l(s_i) = a_i$ para todo $i \geq 0$.

Como as seqüências são infinitas não podemos definir aceitação em função de um estado final. De acordo com o critério de aceitação de Büchi, uma execução é aceita quando alguns estados de aceitação são freqüentemente visitados infinitas vezes. A linguagem aceita pelo autômato de Büchi A é denotada da seguinte forma:

$$\mathcal{L}_\omega(A) = \{w \in \Sigma \parallel w \text{ aceito por } A\}, \text{ se } F \text{ é vazio, então } \mathcal{L}_\omega(A) \text{ também é vazio.}$$

Exemplo 2.6 *Considere o seguinte autômato de Büchi A:*

1. $\Sigma = a, b$;
2. $S = q0, q1$;
3. $S^0 = q0$;
4. $\rho : (q0, a, q0), (q0, b, q1), (q1, a, q0), (q1, b, q0)$;
5. $F = q1$;

A representação gráfica para este autômato descrito acima é ilustrado na Figura 2.6.

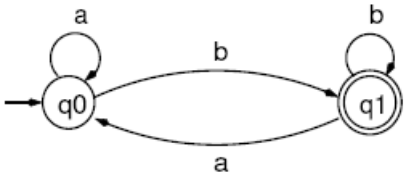

Figura 2.6: Representação gráfica do autômato do Exemplo 2.6.

O autômato A aceita a seguinte linguagem de palavras infinitas:

$$\mathcal{L}_\omega(A) = (a^* b)^\omega$$

2.3.5 Transformando fórmulas LTL em autômatos de Büchi

Temos que para cada fórmula LTL (em proposições atômica AP) existe um autômato de Büchi correspondente.

Teorema 2.1 *Para uma fórmula LTL ψ existe um autômato de Büchi A que pode ser construído com o alfabeto $\Sigma = 2^{AP}$ tal que $\mathcal{L}_w(A)$ é igual as sucessões de conjunto de proposições atômicas que satisfazem Σ.*

O algoritmo que trata da associação de uma fórmula LTL com um autômato de Büchi foi definido por Wolper, Vardi e Sistla (1983) e consiste nos passos ilustrados na Figura 2.7 [Kat99]. O passo principal nesta transformação é a construção do grafo a partir da fórmula na forma-normal.

Figura 2.7: Visão semântica do algoritmo de transformação de fórmulas LTL para autômatos de Büchi.

Fórmulas na forma-normal

O primeiro passo executado pelo algoritmo, ilustrado na Figura 2.7, dada uma fórmula LTL ϕ converter para *fórmula normal* equivalente. Para isto, consideramos inicialmente que ϕ não contém F e G (que podem ser transformados utilizando as seguintes definições: $F\psi \equiv true \cup \psi$ e $G\psi \equiv \neg F \neg \psi$), e todas as negações $\neg\phi$ são adjacentes as proposições atômicas. Considere também que $true$ e $false$ são substituídos por suas definições. A fim de permitir transformar a negação da fórmula até (*until*), um operador temporal auxiliar $\overline{\cup}$ é introduzido e definido como:

$$(\neg\phi)\ \overline{\cup}\ (\neg\psi) \equiv \neg(\phi \cup \psi).$$

Definição 2.7 (Fórmula LTL na forma-normal) *Para $p \in AP$, uma proposição atômica, o conjunto de fórmulas LTL na forma-normal é definido por:*

$$\phi := p \mid \neg p \mid \phi \vee \psi \mid \phi \wedge \psi \mid X\phi \mid \phi \cup \phi \mid \phi \,\overline{\cup}\, \phi.$$

As seguintes equações são usadas na transformação da fórmula LTL ϕ na forma-normal:

- $\neg(\phi \vee \psi) \equiv (\neg\phi) \wedge (\neg\psi)$
- $\neg(\phi \wedge \psi) \equiv (\neg\phi) \vee (\neg\psi)$
- $\neg X\phi \equiv X(\neg\phi)$
- $\neg(\phi \cup \psi) \equiv (\neg\phi) \,\overline{\cup}\, (\neg\psi)$
- $\neg(\phi \,\overline{\cup}\, \psi) \equiv (\neg\phi) \cup (\neg\psi)$

Construção do grafo

A partir das fórmulas na forma-normal obtemos o grafo com a aplicação do algoritmo *CreateGraph*, descrito em [Kat99] ilustrado na Figura 2.8. A saída da apliucação do algoritmo *CreateGraph* é o grafo $\mathcal{G}_\phi = (V, E)$, onde V é o conjunto de vértices e E o conjunto de arestas, tal que $E \subseteq V \times V$.

```
function CreateGraph (φ : Formula): set of Vertex;
(* pre-condition: φ is a PLTL-formula in normal form *)
begin var S : sequence of Vertex,
          Z : set of Vertex, (* already explored vertices *)
          v, w1, w2 : Vertex;
      S, Z := ⟨({ init }, { φ }, ∅, ∅)⟩, ∅;
      do S ≠ ⟨ ⟩ ⟶ (* let S = ⟨v⟩⌢S' *)
         if N(v) = ∅ ⟶ (* all proof obligations of v have been checked *)
            if (∃ w ∈ Z. Sc(v) = Sc(w) ∧ O(v) = O(w)) ⟶
               P(w), S := P(w) ∪ P(v), S'  (* w is a copy of v *)
            [] ¬(∃ w ∈ Z. ...) ⟶
               S, Z := ⟨({ v }, Sc(v), ∅, ∅)⟩⌢S', Z ∪ { v };
            fi
         [] N(v) ≠ ∅ ⟶ (* some proof obligations of v are left *)
            let ψ in N(v);
            N(v) := N(v) \ { ψ };
            if ψ ∈ AP ∨ (¬ψ) ∈ AP ⟶
               if (¬ψ) ∈ O(v) ⟶ S := S' (* discard v *)
               [] ¬ψ ∉ O(v) ⟶ skip
               fi
            [] ψ = (ψ1 ∧ ψ2) ⟶ N(v) := N(v) ∪ ({ ψ1, ψ2 } \ O(v))
            [] ψ = X φ ⟶ Sc(v) := Sc(v) ∪ { φ }
            [] ψ ∈ { ψ1 U ψ2, ψ1 R ψ2, ψ1 ∨ ψ2 } ⟶ (* split v *)
               w1, w2 := v, v;
               N(w1) := N(w1) ∪ (F1(ψ) \ O(w1));
               N(w2) := N(w2) ∪ (F2(ψ) \ O(w2));
               O(w1), O(w2) := O(w1) ∪ { ψ }, O(w2) ∪ { ψ };
               S := ⟨w1⟩⌢(⟨w2⟩⌢S')
            fi
            O(v) := O(v) ∪ { ψ }
         fi
      od;
      return Z;
(* post-condition: Z is the set of vertices of the graph G_φ *)
(* where the initial vertices are vertices in Z with init ∈ P *)
(* and the edges are given by the P-components of vertices in Z *)
end
```

Figura 2.8: Algoritmo de construção de um grafo a partir de uma fórmula LTL ϕ.

Exemplo 2.7 *O grafo resultante da aplicação do algoritmo CreateGraph para a fórmula* $\phi = p \cup q$ *é ilustrado na Figura 2.9 [Kat99].*

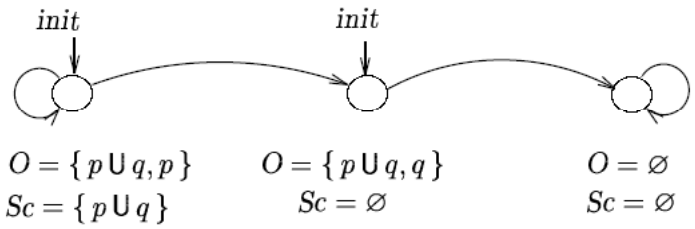

Figura 2.9: Resultado da aplicação do algoritmo na fórmula $p \cup q$.

Transformando grafo em autômato de Büchi generalizado

Definição 2.8 (Autômato de Büchi generalizado) *Um autômato de Büchi generalizado (GLBA)* A *é uma tupla* $(\Sigma, S, S^0, \rho, \mathcal{F}, \ell)$ *onde todos os componentes são os mesmos para o LBA, exceto* $\mathcal{F}$ *que é o conjunto dos conjuntos de aceitação* $\{F_1, \ldots, F_k\}$ *para* $k \geq 0$ *com* $F_i \subseteq S$*, isto é* $\mathcal{F} \subseteq 2^S$.

A conversão uma fórmula LTL ϕ na forma-normal para o GLBA $A = (\Sigma, S, S^0, \rho, \mathcal{F}, \ell)$ é definida por:

- $\Sigma = 2^{AP}$
- $S = CreateGraph(\phi)$
- $S^0 = \{s \in S | init \in P(s)\}$
- $s \longrightarrow s'$ sss $s \in P(s')$ e $s \neq init$
- $\mathcal{F} = \{\{s \in S | \phi_1 \cup \phi_2 \notin O(s) \vee \phi_2 \in O(s)\} | \phi_1 \cup \phi_2 \in Sub(\phi)\}$
- $\ell(s) = \{\mathcal{P} \subseteq AP | Pos(s) \subseteq \mathcal{P} \wedge \mathcal{P} \cap Neg(s) = \oslash$

$Sub(\phi)$ denota o conjunto de sub-fórmulas de ϕ. $Pos(s) = O(s) \cap AP$, as proposições atômicas válidas em s , e $Neg(s) = \{p \in AP | \neg p \in O(s)\}$, o conjunto das proposições atômicas negativas que são válidas em s.

Exemplo 2.8 *O GLBA que correspondente ao grafo do Exemplo 2.7 é ilustrado na Figura 2.10 [Kat99].*

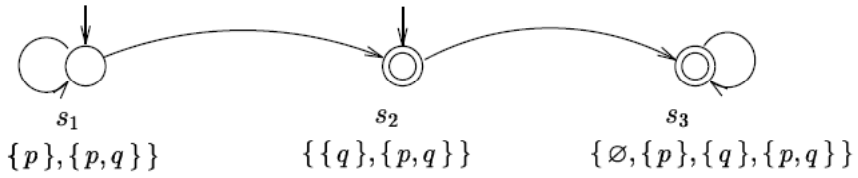

Figura 2.10: GLBA para o grafo do Exemplo 2.7.

Transformando autômato de Büchi generalizado em um autômato de Büchi

Definição 2.9 (GLBA para um LBA) *Seja* $A = (\Sigma, S, S^0, \rho, \mathcal{F}, \ell)$ *um autômato de Büchi generalizado (GLBA) com* $\mathcal{F} = \{F_1, \ldots, F_k\}$. *O autômato de Büchi equivalente* $A' = (\Sigma, S', S^0{}', \rho', \mathcal{F}', \ell')$ *tal que* $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A')$ *é obtido da seguinte maneira:*

- $S' = S \times \{i | 0 < i \leq k\}$
- $S^0{}' = S^0 \times \{i\}$ *para algum* $0 < i \leq k$
- $(s, i) \longrightarrow' (s', i)$ *sss* $s \longrightarrow s'$ *e* $s \notin F_i$
- $(s, i) \longrightarrow' (s', (i mod k) + 1)$ *sss* $s \longrightarrow s'$ *e* $s \in F_i$
- $F' = F_i \times \{i\}$ *para algum* $0 < i \leq k$
- $l'(s, i) = l(s)$.

Exemplo 2.9 *Considere o seguinte autômato de Büchi generalizado (Figura 2.11 [Kat99]):*

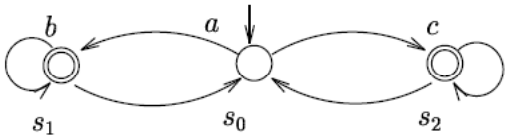

Figura 2.11: GLBA para o grafo do Exemplo 2.9.

Este autômato contém dois conjuntos de aceitação $F_1 = \{s_1\}$ *e* $F_2 = \{s_2\}$. *Os estados que correspondem ao autômato de Büchi simples são*

$$\{s_0, s_1, s_2\} \times \{1, 2\}$$

Algumas das transições, por exemplo, são:

- $(s_0, 1) \longrightarrow (s_1, 1)$ *desde que* $s_0 \longrightarrow s_1$ *e* $s_0 \notin F_1$
- $(s_0, 1) \longrightarrow (s_2, 1)$ *desde que* $s_0 \longrightarrow s_2$ *e* $s_0 \notin F_1$
- $(s_1, 1) \longrightarrow (s_0, 2)$ *desde que* $s_1 \longrightarrow s_0$ *e* $s_1 \in F_1$
- $(s_1, 2) \longrightarrow (s_1, 2)$ *desde que* $s_1 \longrightarrow s_1$ *e* $s_1 \notin F_2$
- $(s_2, 2) \longrightarrow (s_2, 1)$ *desde que* $s_2 \longrightarrow s_2$ *e* $s_2 \in F_2$

O autômato de Büchi simples equivalente ao grafo da Figura 2.11 é ilustrado na Figura 2.12 [Kat99].

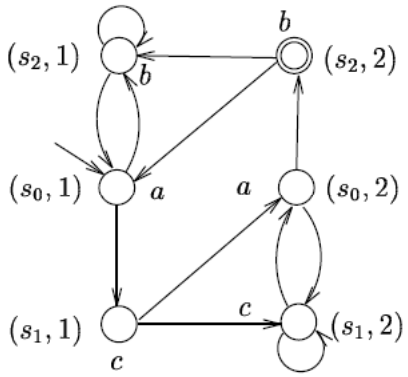

Figura 2.12: LBA para o grafo da Figura 2.11.

2.4 Programação orientada a aspectos com AspectJ

2.4.1 Introdução

A Programação Orientada a Aspectos (POA) [KLM+97] foi proposta com o objetivo de facilitar a modularização dos interesses transversais, complementando a Programação Orientada a Objetos (POO). A POA não tem o intuito de ser um novo paradigma de programação, mas uma nova técnica que deve ser utilizada em conjunto com linguagens de programação para

o desenvolvimento de sistemas de software, auxiliando na manutenção dos vários interesses e a compreensão do software. Entretanto, não é um antídoto para um *design* ruim ou insuficiente [EAK+01].

Em um sistema de software os interesses são implementados em blocos de código, que manipulam dados. Os interesses que podem ser encapsulados de forma clara em uma unidade de função são chamados de componentes. Em POO esses interesses são modularizados em objetos, compostos por métodos que contém a implementação do interesse, e os atributos composto pelos dados manipulados pelos métodos. Em POA é introduzido um novo mecanismo para abstração e composição, que facilita a modularização dos interesses transversais, o aspecto (*aspect*). Desta forma, os sistemas de software são decompostos em componentes e aspectos. Assim, os requisitos funcionais normalmente são organizados em componentes através de uma linguagem POO, como Java, e os requisitos não funcionais como aspectos relacionados as propriedades que afetam o comportamento do sistema [KLM+97].

O desenvolvimento de software orientado a aspectos envolve basicamente três etapas distintas de desenvolvimento:

- **decompor os interesses (*aspectual decomposition*)** - identificar e separar os interesses transversais dos interesses do negócio;
- **implementar os interesses (*concern implementation*)** - implementar cada um dos interesses identificados separadamente;
- **recomposição aspectual (*aspectual recomposition*)** - nesta etapa, temos integrador de aspectos que especifica regras de recomposição para criação de unidades de modularização - aspectos. A esse processo de junção da codificação dos componentes e dos aspectos é denominada combinação (*weaving*).

Na Figura 2.13 [Lad03], ilustramos as etapas de desenvolvimento da POA.

2.4.2 Conceitos básicos de aspectos

A linguagem AspectJ [KHH+01] é um dos meios mais difundidos para implementação de sistemas orientados a aspectos. Ela foi criada pela *Xerox Palo Alto Research Center* em 1997 e posteriormente agregada ao projeto *Eclipse* da IBM em 2002. AspectJ é uma extensão da

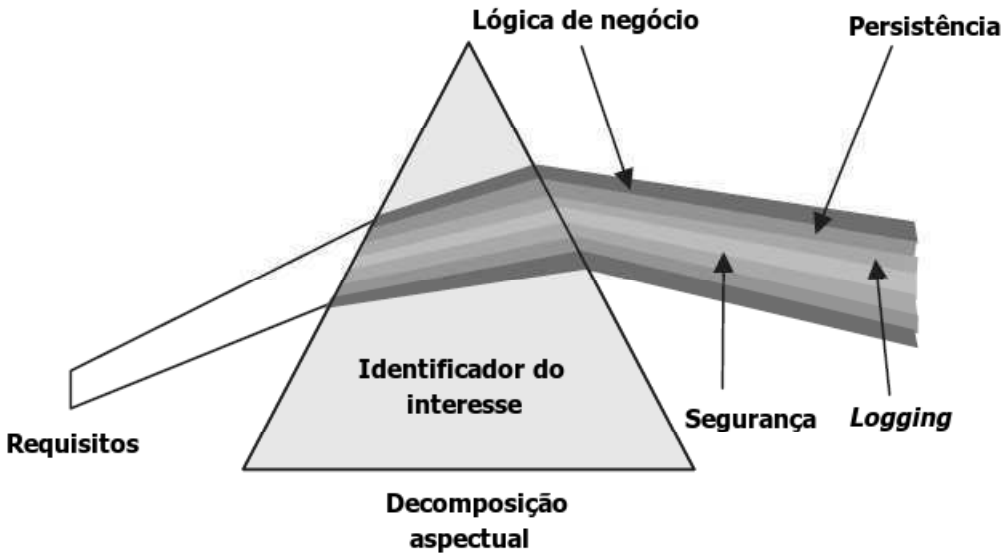

Figura 2.13: Etapas de desenvolvimento de software orientado a aspectos.

linguagem Java que dá suporte à implementação modular de interesses transversais, que podem ser dinâmicos ou estáticos. Os interesses dinâmicos definem a implementação adicional que deve ser executada em pontos pré-determinados. Por outro lado, os interesses estáticos visam modificar a estrutura do programa, o que torna possível definir novas operações em tipos já existentes de uma maneira não intrusiva.

Além dos elementos oferecidos pela POO como classes, métodos, atributos e etc, são acrescentados novos conceitos e construções ao AspectJ, tais como: aspecto (aspect), conjunto de junção (*pointcut*), ponto de junção (*joinpoint*), adendo (*advice*) e declaração intertipos (*inter-type declaration*)[2].

Aspectos são os elementos básicos dessa abordagem, pois podem alterar a estrutura estática ou dinâmica de um programa. A estrutura estática é alterada adicionando, por meio das declarações intertipos, membros (atributos, métodos ou construtores) a uma classe, modificando assim a hierarquia do sistema. Já a alteração numa estrutura dinâmica de um programa ocorre em tempo de execução por meio do conjunto de junção (compostos por diversos pontos de junção), através da adição de comportamentos (adendos) antes ou depois de cada

[2]As traduções utilizadas neste trabalho seguem as recomendações definidas no WASP 2004 - 1º Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, disponíveis em http://twiki.im.ufba.br/bin/view/AOSDbr/TermosEmPortugues

ponto de junção [Kis02].

Ponto de junção (*joinpoint*)

Para o entendimento do AspectJ é de fundamental importância o conceito de ponto de junção. Pontos de junção são pontos na execução de um programa de componentes aonde os aspectos serão aplicados. O AspectJ pode detectar e operar sobre os seguintes tipos de pontos de junção [GLG03]:

- chamada e execução de métodos;
- chamada e execução de construtores;
- execução de inicialização;
- execução de contrutores;
- execução de inicialização estática;
- pré-inicialização de objetos;
- inicialização de objetos;
- referência a campos;
- execução de tratamento de exceções.

Na Figura 2.14, demonstramos um exemplo apresentado em [SB02], de um fluxo de execução entre dois objetos, identificando alguns pontos de junção.

O primeiro ponto de junção é a invocação de um método de um objeto A, o qual pode retornar sucesso ou lançar uma exceção. O próximo ponto de junção é a execução deste método, que por sua vez também pode retornar sucesso ou lançar uma exceção. Durante a execução do método do objeto A é invocado um método de um objeto B, podendo retornar sucesso ou lançar uma exceção. A invocação e execução destes métodos são pontos de junção.

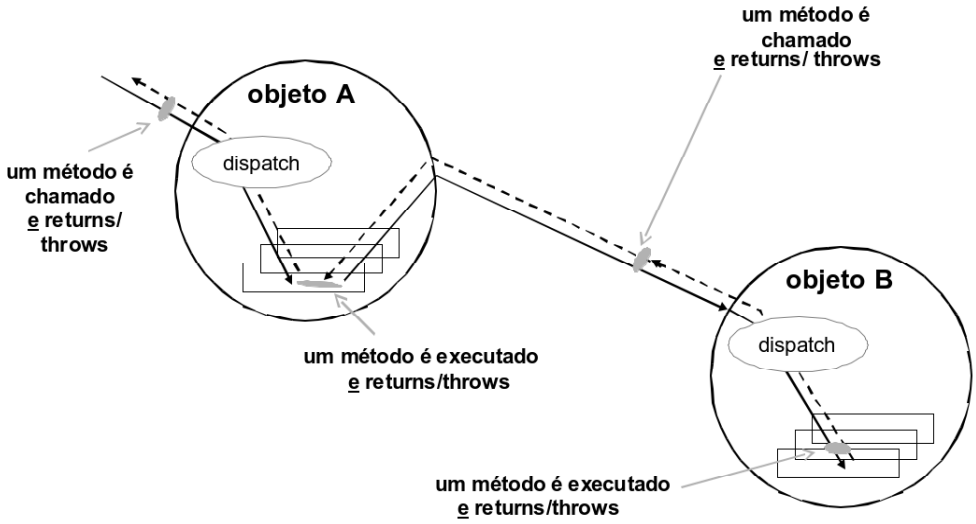

Figura 2.14: Ponto de junção de um fluxo de execução.

Conjunto de junção (*pointcut*)

Um aspecto no AspectJ geralmente define conjuntos de junção, que são aninhados por pontos de junção através de operadores lógicos `&&`, `||`, e `!`. Eles são responsáveis por selecionar pontos de junção, ou seja, eles detectam em que ponto do programa os aspectos deverão interceptar.

Podemos declarar um conjunto de junção semelhante a uma classes em Java, podendo da mesma maneira que atributos e métodos dessas classes, especificar um quantificador de acesso aos conjuntos de junção , podendo ser públicos, privados ou final, mas não podem ser sobrecarregados. Também podem ser declarados abstratos, mas somente dentro de aspectos abstratos, e ainda podem ser nomeados ou anônimos [Kis02]. A declaração de um conjunto de junção deve seguir a seguinte sintaxe:

```
pointcut <Nome> (Argumentos):  <corpo>;
```

Para definir um conjunto de junção utiliza-se construtores de AspectJ nomeados de designadores, os principais estão listados na Tabela 2.1:

Em AspectJ elementos *wildcards* são utilizados, estes permitem que em especificações de assinatura (*signature*) sejam definidos o número de caracteres (`*`) e o número dos argumentos (`..`). Por exemplo: `public void set*(.., String)`, isto irá refletir sobre todos os métodos que iniciam com a palavra set e que tenham zero ou mais argumentos como parâmetro. E em padrão tipo (*type pattern*) utilizam-se os seguintes *wildcards*:

Tabela 2.1: Listagem dos designadores em AspectJ.

Designador	**Características**
`call(Signature)`	Invocação do método/construtor identificado pela assinatura
`execution(Signature)`	Execução do método/construtor identificado pela assinaturas
`get(Signature)`	Acesso ao atributo identificado pela assinatura
`set(Signature)`	Atribuição ao atributo identificado pela assinatura
`this(Type pattern)`	Objeto em execução é a instância do tipo
`target(Type pattern)`	Objeto de destino é a instância do tipo
`args(Type pattern)`	Os argumentos são instâncias do tipo
`within(Type pattern)`	Limita o escopo do conjunto de junção para determinados tipos

- `*` - qualquer seqüência de caracteres não contendo pontos;
- `..` - qualquer seqüência de caracteres, inclusive contendo pontos;
- + - qualquer subclasse de uma classe.

Adendo (*advice*)

Adendo é o código para ser executado em um ponto de junção que está sendo referenciado pelo conjunto de junção. Os adendos podem ser executados antes, durante e depois (*before*, *around* e *after*). Portanto, de acordo com seus nomes, before executa antes do ponto de junção, *around* executa antes e depois e, *after* executa depois.

O adendo pode modificar a execução do código no ponto de junção, pode substituir ou passar por ele. Usando o adendo pode-se "logar" as mensagens antes de executar o código de determinados pontos de junção que estão espalhados em diferentes módulos. O corpo de um adendo é muito semelhante ao de qualquer método, encapsulando a lógica a ser executada quando um ponto de junção é alcançado [GLG03].

Declaração intertipos (*inter-type declaration*)

O AspectJ provê uma maneira de alterar a estrutura estática de uma aplicação, isto ocorre por meio das declarações inter-tipos que são descritas como interesses estáticos (*static cross-scutting*). Estas declarações provêm uma construção chamada *Introduction*.

Introduction é um interesse estático que introduz alterações nas classes, interfaces e aspectos do sistema. Alterações estáticas em módulos não tem efeito direto no comportamento. Por exemplo, pode ser adicionado um método ou um atributo na classe [GLG03].

Aspecto (*Aspect*)

Da mesma maneira que a classe é a unidade central em Java, aspecto é a unidade central do AspectJ. Aspectos encapsulam conjuntos de junção (*point cuts*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*) em uma unidade modular de implementação. Assim como as classes em Java, os aspectos podem conter atributos, métodos e classes internas.

Aspectos podem alterar a estrutura estática de um sistema adicionando membros (atributos, métodos e construtores) a uma classe, alterando a hierarquia do sistema, e convertendo uma exceção checada por uma não checada (exceção de *runtime*). Esta característica de alterar a estrutura estática de um programa é chamada *static crosscutting*. Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível através da interceptação de pontos de junção, e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção de total controle sobre o ponto de execução [SB02].

2.4.3 *Logging* com AspectJ

Logging é uma das técnicas mais comumente utilizada para entender o comportamento dos sistemas [Lad03]. Em sua forma mais simples, *logging* imprime as mensagens que descrevem as operações executadas.

Atualmente, os mecanismos utilizados para *logging* estão junto a lógica de negócio. Realizar mudanças na estratégia de *logging* requer freqüentemente mudanças em muitos módulos da aplicação. Como *logging* é um interesse transversal, POA e AspectJ podem ajudar a modularizar o uso dessa técnica. Com AspectJ, podemos fazer o *logging* independentemente da lógica de negócio.

A Figura 2.15 [Lad03] ilustra como a técnica de *logging* é utilizado sem o uso de POA. Podemos observar que em cada ponto que necessita de se registrar um evento temos a invo-

cação explícita do método de registro (`log()`) de um registrador (*logger*) apropriado.

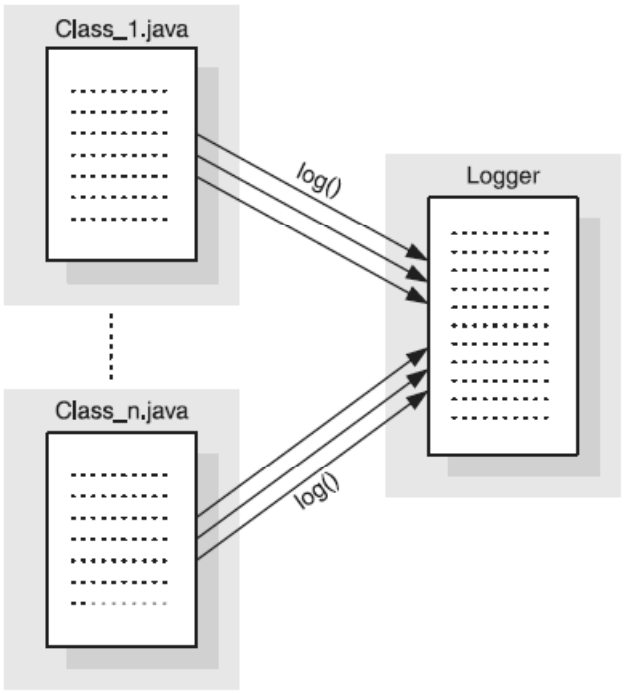

Figura 2.15: Solução convencional de *logging* sem o uso de POA.

Uma forma especial do uso da técnica de *logging* é registrar o caminho de execução (*trace*) do sistema de software. Nesse caminho são registradas a entrada e/ou a saída de métodos selecionados. O caminho de execução é útil durante a fase do desenvolvimento para compreender o comportamento do sistema.

Embora o uso de APIs de *logging* resolva alguns problemas; contudo, existem limitações. Essas limitações não são um resultado das APIs de *logging* ou de suas execuções; mas, limitações fundamentais da POO, como por exemplo, a necessidade de inserir invocações de *logging* dentro do código-fonte. O uso de AspectJ provê meios para superar tais limitações. AspectJ permite desenvolver aspectos que seguem o caminho de execução de todo o sistema, sem a necessidade de nenhuma mudança no código-fonte. Assim, o interesse de *logging* fica separado da lógica de negócio. Na Figura 2.16 [Lad03] ilustra a visão geral de *logging* baseado em AspectJ.

O Código 2.3 [Lad03] ilustra o código aspecto `TraceAspect`, que captura o caminho de execução do sistema de software. Na linha é declarado um conjunto de junção abstrato, `loggedOperations()`, que define as operações que devem ser capturadas. Nesse caso,

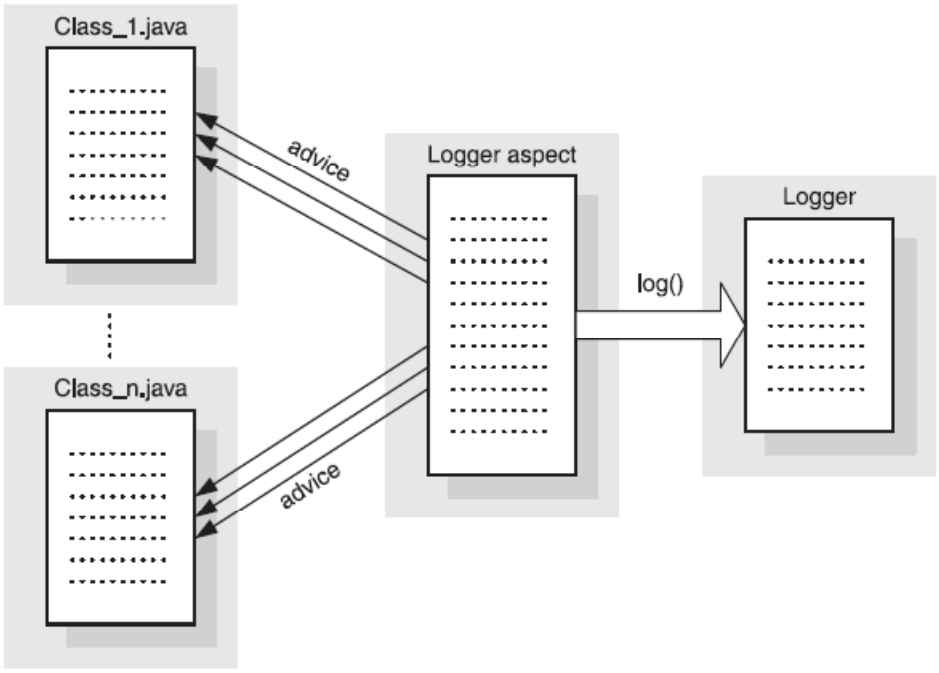

Figura 2.16: Visão geral da solução de *logging* baseada em AspectJ.

iremos capturar toda execução de métodos e construtores durante a execução do sistema de software. Antes desse conjunto de junção é registrado no *log* o tipo de objeto e o método instanciado.

2.5 Considerações Finais

Os fundamentos teóricos apresentados nesse capítulo fornecem os meios necessários para o entendimento deste trabalho. A técnica desenvolvida tem como base a junção das idéias principais das abordagens DBC e monitoração de sistemas de software em tempo de execução. DBC permite estabelecer a especificação de propriedades que os sistemas de software devem obedecer. Já a monitoração permite observar em pontos específicos o comportamento dos sistemas de software durante sua execução. A idéia é que a partir das propriedades especificadas e o comportamento observado possamos verificar se os sistemas de software estão se comportando conforme o esperado. A especificação do comportamento esperado será descrito através de LTL, por esse formalismo permitir descrever propriedades de sistemas concorrentes. Para representar esse comportamento expresso através de uma fórmula LTL utilizamos autômatos de Büchi. Desta forma, verificamos se o estado dos sistemas de soft-

```
1  import org.aspectj.lang.*;
2  import logging.*;

3  public aspect TraceAspect {

4    protected pointcut loggedOperations()
5      : (execution(* *.*(..))
6          || execution(*.new(..)));

7    before() : loggedOperations() {
8      Signature sig = thisJoinPointStaticPart.getSignature();
9      System.out.println("Entering ["
10        + sig.getDeclaringType().getName() + "."
11        + sig.getName() + "]");
12   }
13 }
```

Código 2.3: Código aspecto de *logging* do caminho de execução.

ware durante a sua execução condiz com o estado atual no autômato de Büchi equivalente a fórmula LTL. A monitoração da execução dos sistemas de software é realizada com o uso de POA, especificamente a linguagem AspectJ. A escolha de POA deve-se ao fato de ser uma estratégia de instrumentação pouco intrusiva.

Capítulo 3

Detecções de Violações Comportamentais

Neste capítulo, apresentamos a técnica *DesignMonitor*. Essa técnica tem como objetivo detectar violações de propriedades comportamentais em sistemas de software durante sua execução. Inicialmente, apresentamos o formato para a especificação do comportamento esperado, descrevendo como mapear as propriedades comportamentais que compõem os modelos do projeto em especificações formais. Logo em seguida, apresentamos como se dá a monitoração do comportamento dos sistemas de software alvo. Por fim, discutimos como é realizada a verificação da conformidade entre o comportamento monitorado e o comportamento especificado.

3.1 Conformidade de código

Um sistema de software é composto por um conjunto de requisitos. A sua implementação é o resultado da conversão da especificação desse conjunto de requisitos do software em um sistema executável [Som04]. Para desenvolver ou manter um sistema de software é necessário que a equipe de desenvolvimento possua um entendimento íntegro e consistente a respeito de sua estrutura e do seu comportamento.

O projeto (*design*) de software baseia-se nos requisitos tipicamente estabelecidos em termos relevantes ao domínio do problema. O projeto deve prover uma descrição da estrutura e do comportamento de uma solução que implementa os requisitos identificados [ABDM01]. Desta forma, o projeto de software pode ser visto como a principal ponte entre os requisitos para o sistema e a implementação do sistema de software.

Em sistemas orientado a objetos o projeto de software é composto por classes de objetos e pelas relações entre essas classes, onde os objetos são criados dinamicamente durante a execução do sistema de software a partir das definições de classe. Segundo [Som04], há dois tipos de modelos de projeto que, normalmente, são utilizados para descrever um projeto de software orientado a objetos:

- **modelos estáticos** - descrevem a estrutura estática do sistema em termos das classes de objetos do sistema e de seus relacionamentos;
- **modelos dinâmicos** - descrevem a estrutura dinâmica do sistema e mostram as interações entre os objetos do sistema.

Os modelos de projeto são essencialmente o próprio projeto de software. Cada modelo visa descrever um conjunto de propriedades, estruturais e comportamentais, que a implementação do sistema de software deve possuir (quer esteja ou não documentado). A concordância entre o código de programas e os modelos abstratos que o descrevem é comumente denominada de *conformidade de código*.

A idéia geral da técnica *DesignMonitor* é avaliar automaticamente a conformidade de código dos sistemas desenvolvidos. Em outras palavras, permitir que os desenvolvedores possam detectar automaticamente se as decisões de projeto comportamentais por eles tomadas não violam nenhuma das propriedades do projeto do sistema de software especificadas. Mas, não garante que não existem violações dessas propriedades comportamentais (a não ser em situações bastante restritas).

3.2 Caracterização da solução

Diante da idéia geral da técnica *DesignMonitor*, alguns requisitos foram considerados na definição da solução. A especificação das propriedades comportamentais foram inspiradas em contratos, visando expressar condições que devem ser obedecidas em pontos específicos no código ao longo do tempo. Além disso, essa técnica permite a especificação parcial de propriedades comportamentais. A partir das propriedades comportamentais especificadas o sistema de software alvo é instrumentado, de modo a permitir que durante sua execução o

comportamento nos pontos de interesse sejam monitorados. Essa instrumentação deve ser não intrusiva, ou seja, o código do sistema de software alvo não deve ser alterado.

Além disso, o uso dessa técnica visa não demandar esforço adicional de desenvolvimento. Com base na especificação das propriedades comportamentais devem ser gerados automaticamente os demais artefatos necessários para detecção de violações da conformidade de código. Dessa maneira, a detecção das violações deve ser realizada de maneira transparente para o usuário da técnica *DesignMonitor*.

As atividades desenvolvidas em *DesignMonitor* são divididas em duas fases distintas:

- **pré-execução** - nesta fase são gerados os artefatos necessários para prover a análise comportamental em tempo de execução;
- **em execução** - durante a execução do sistema de software alvo o comportamento observado é confrontado com o comportamento desejado a partir dos artefatos gerados na fase de pré-execução.

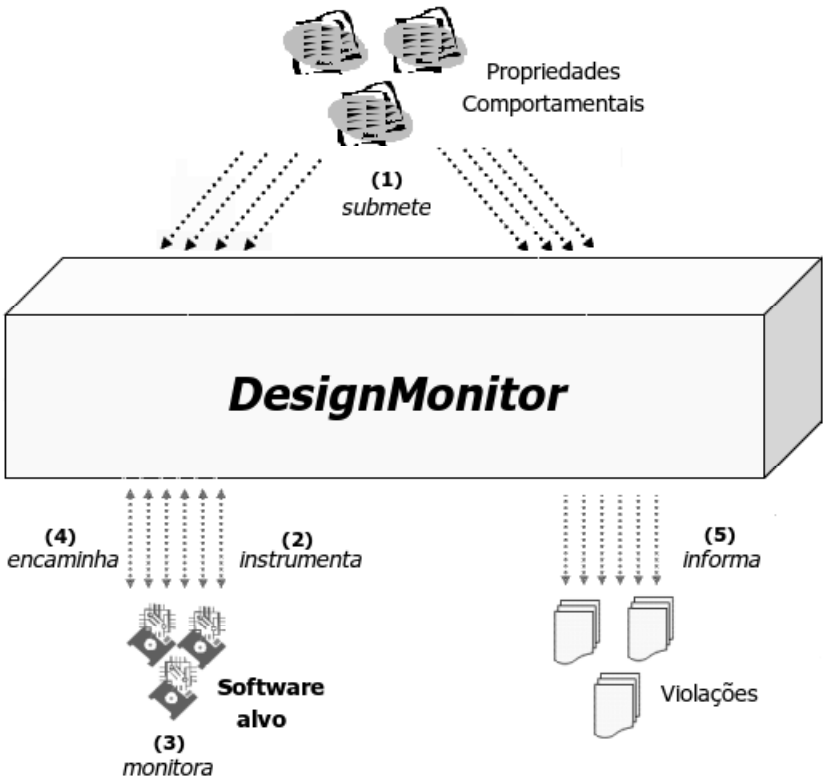

Figura 3.1: Visão geral da arquitetura *DesignMonitor*.

A Figura 3.1 ilustra uma visão geral da técnica *DesignMonitor*. Inicialmente, na fase de pré-execução, o usuário especifica as propriedades comportamentais e as submentem ao *DesignMonitor* (1). Com base nessas propriedades é realizada a instrumentação do sistema de software alvo e gera os artefatos para análise comportamental (2). Posteriormente, durante a fase de execução, o comportamento nos pontos de interesse é monitorado (3). Quando esses pontos de interesse são acessados, as informações relevantes desse comportamento observado são capturadas e encaminhadas para o *DesignMonitor* (4). Este por sua vez, os confronta com o comportamento desejado. Caso o comportamento observado não esteja de acordo com o comportamento desejado, o *DesignMonitor* informa que ocorreu violação de uma determinada propriedade comportamental (5).

3.3 Especificação comportamental

O projeto de software é composto por um conjunto de propriedades comportamentais. Em *DesignMonitor*, cada propriedade comportamental contém uma identificação da propriedade comportamental (nome), o comportamento e os pontos de interesse no código. Formalmente, a especificação comportamental S de um sistema de software pode ser expressa como uma 3-tupla:

$S = \{P, C, R\}$

onde:

$P =$ conjunto finito dos pontos de interesse;

$C =$ conjunto finito de expressões do comportamento desejado;

$R \subseteq \{P \text{ X } C\}$;

A definição de uma propriedade comportamental é formada pela relação entre comportamento desejado e os pontos de interesse do sistema de software alvo. O conjunto C é composto por expressões do comportamento desejado. Geralmente, a expressão do comportamento desejado das propriedades de projeto partem de uma descrição em linguagem natural.

Entretanto, linguagens naturais são ambíguas, o que pode levar a uma interpretação errônea das propriedades do projeto de software e, conseqüentemente, a uma implementação

também errônea. Sendo assim, o uso de especificações formais torna-se interessante devido as suas características como completude, consistência, precisão e concisão. Como o propósito da técnica *DesignMonitor* é realizar a monitoração do comportamento ao longo do tempo, de sistemas concorrentes e paralelos, a linguagem escolhida para a especificação do comportamento desejado (propriedade comportamental) foi a linguagem LTL, seguindo a sintaxe e a semântica apresentadas na Seção 2.3.3.

Para cada expressão comportamental (fórmula LTL) de C é preciso definir quais os pontos no código do software alvo que devem obdecer tal comportamento. Esses pontos de interesse formam o conjunto P, cuja sintaxe é a mesma suportada por AspectJ[1] [Prob]. Assim, ao invés de listar nominalmente cada um dos pontos de interesse, é possível usar símbolos para representar um conjunto de tipos. Temos os seguintes símbolos:

- `*` - representa um conjunto quaisquer de caracteres, para designar parte de um método, classe, interface ou pacote;
- `..` - denota todos os sub-pacotes indiretos e diretos de um determinado pacote. Para métodos é usado para denotar qualquer tipo e quantidade de argumentos do método;
- `+` - denota qualquer sub-classe ou sub-interface de um determinado tipo.

Além disso, caso os modificadores de acesso do método - tais como `public`, `private`, `static` e `final` - não sejam especificados, eles serão ignorados pelo casamento de padrões. Por exemplo, se o padrão não contiver o modificador `final`, tanto os métodos que são e os que não são `final` serão considerados.

Os modificadores podem ser usados também com o operador de negação `!` para especificar métodos que não possuam tal modificador. No padrão de assinatura de método, onde é especificado o tipo de retorno, dos parâmetros ou das exceções pode-se utilizar os padrões de tipo. Na Tabela 3.1 são apresentados alguns exemplos simples do uso de tais símbolos para esecificação dos pontos de interesse.

[1] Uma extensão que possibilita a programação orientada a aspectos [KLM+97] de propósito geral da linguagem Java

Tabela 3.1: Exemplos de padrões para especificação de pontos de interesse.

Padrão de Tipo	**Significado**
`public void List.clear()`	Método público `clear()` da classe `List`, que retorna `void` e não recebe nenhum parâmetro.
`public void List.clear() throws UnsupportedOperationException`	Método público `clear()` da classe `List`, que retorna `void`, não recebe nenhum parâmetro e tenha declarado como exceção `UnsupportedOperationException`.
`public boolean List.add*(*)`	Todos os métodos públicos da classe `List` que têm seu nome iniciado com `add`, que retornam `boolean` e recebem apenas um único parâmetro de entrada de qualquer tipo.
`public void List.*()`	Todos os métodos públicos da classe `List` que retornam `void` e não recebem nenhum parâmetro.
`public * List.*()`	Todos os métodos públicos da classe `List` que retornam qualquer tipo e não recebem nenhum parâmetro.
`public * List.*(..)`	Todos os métodos públicos da classe `List` que retornam qualquer tipo e recebem qualquer número e tipos de parâmetros.
`* List.*(..)`	Todos os métodos da classe `List` que retornam qualquer tipo e recebem qualquer número e tipos de parâmetros.
`!public * List.*(..)`	Todos os métodos não-públicos da classe `List` que retornam qualquer tipo e recebem qualquer número e tipos de parâmetros.
`* List+.*(..)`	Todos os métodos da classe e sub-classes de `List` que retornam qualquer tipo e recebem qualquer número e tipos de parâmetros.
`* List.add*(int,..)`	Todos os métodos da classe `List` que retornam qualquer tipo e recebem qualquer número e tipos de parâmetros, mas cujo o primeiro parâmetro deve ser do tipo `int`.
`public List.new(Collection)`	Todos os construtores públicos da classe `List` cujo parâmetro é do tipo `Collection`.

Exemplo 3.1 *Suponha que um sistema `S` é composto por dois módulos `A` e `B`, onde cada módulo possue as threads `ThreadA` e `ThreadB`, respectivamente. O módulo `A` é formado por objetos do tipo `ObjetoA`, que possuem a seguinte característica: durante a execução do sistema `S` os objetos do tipo `ObjetoA` a partir do momento em que forem acessados pela `ThreadA`, apenas essa thread é que quem deve acessar esse objeto. Da mesma forma para a `ThreadB`. Especificando essa característica como uma propriedade comportamental para a técnica DesignMonitor, temos:*

```
propertyName := BehaviorPropertyModule;
points := * ObjectA.*(..);
behaviorProperty := (!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB);
```

3.4 Processo de análise comportamental

Após a especificação das propriedades comportamentais do projeto de software, o próximo passo é prover os artefatos necessários para permitir a avaliação da conformidade de código. Contudo, avaliar se a implementação do sistema de software obedece certas propriedades comportamentais não é uma tarefa trivial. Para isto, utilizamos uma abordagem baseada na análise dinâmica através da monitoração da execução do sistema de software alvo. A técnica *DesignMonitor* tem como objetivo monitorar apenas pontos de interesse do código de cada propriedade comportamental. À medida que o sistema alvo está sendo executado os pontos de interesse são monitorados e o comportamento observado é confrontado com a especificação do comportamento desejado em tempo de execução.

Inicialmente, apresentamos a arquitetura de monitoração do comportamento. Em seguida, descrevemos como se dá a análise do comportamento observado com relação ao comportamento desejado.

3.4.1 Monitoração

A Figura 3.2 ilustra a arquitetura de monitoração do comportamento do sistema de software durante sua execução. A arquitetura de monitoração foi definida de modo a permitir estender facilmente a técnica *DesignMonitor* para a análise de sistemas distribuídos. Deste modo, temos uma monitoração distribuída.

A idéia é que cada nó que compõe o sistema distribuído tenha o seu comportamento monitorado durante a execução do sistema de software. Os eventos (comportamentos observados) capturados são encaminhados para um ponto central, denominado *monitor*. O monitor é responsável por analisar esses eventos capturados, determinando se violam alguma propriedade comportamental (detalhado na Seção 3.4.2).

No entanto, por questões de escopo da solução, a técnica *DesignMonitor* atualmente considera apenas a monitoração de um dos nós que compõem o sistema distribuído. Um nó é

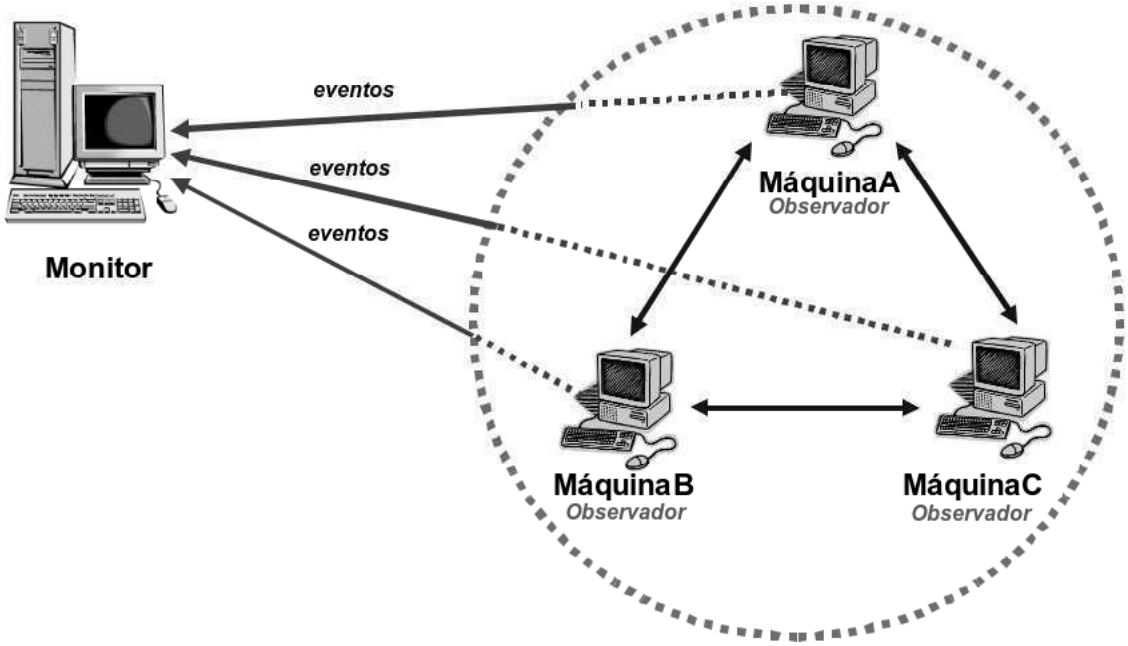

Figura 3.2: Visão geral da monitoração distribuída.

visto como uma máquina (*hardware*) que pode durante a execução do software ser composta por diversas máquinas virtuais que compartilhe o mesmo *clock*. Considerando a execução de um programa *multithread*, temos que essa execução é composta por um conjunto de *threads* que podem ter uma execução seqüencial, intercalada ou paralela [Tan92]. Dessa maneira, a observação do comportamento das *threads* é uma tarefa importante para a análise desses sistemas de software.

Assim, em *DesignMonitor* é monitorado o comportamento das *threads* apenas nos pontos de interesse que compõem cada propriedade comportamental durante a execução do software alvo. As informações sobre o comportamento observado são enviados para o monitor para a análise comportamental ao mesmo tempo que o software alvo está sendo executado.

3.4.2 Análise comportamental em tempo de execução

No monitor, para cada propriedade comportamental existe um código de análise comportamental baseado em autômatos de Büchi. Esse código de análise comportamental é obtido a partir da fórmula LTL que especifica o comportamento esperado das *threads* para determinados pontos de interesse.

Como as fórmulas LTL expressam o comportamento desejado, o autômato de Büchi

obtido a partir dessa fórmula contém apenas os estados esperados durante a execução do sistema de software alvo. Essa transformação segue os passos descritos na Seção 2.3.5. Cada estado do autômato de Büchi deve caracterizar o status atual válido do sistema alvo. Já as transições determinam quais *threads* que podem acessar àquele ponto de interesse, levando a um outro estado válido. Adicionalmente a esses estados possíveis é que tenhamos um novo estado no autômato de Büchi que represente um estado inválido. Esse estado inválido irá identificar quando for observado um comportamento não desejado durante a execução dos sistemas de software. Os eventos capturados para determinada propriedade comportamental são verificados pelo código de verificação equivalente.

Exemplo 3.2 *Considere o Exemplo 3.1. A representação da propriedade comportamental descrita como uma fórmula LTL é a seguinte:* `(!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB)`*, onde* `ThreadA` *e* `ThreadB` *são nomes de threads existes no sistema de software alvo.*

Na Figura 3.3 ilustramos através de uma representação gráfica o autômato de Büchi obtido a partir da transformação da fórmula LTL `(!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB)` *e a máquina de estado que se baseia nesse autômato de Büchi que é utilizada para acompanhar o comportamento das threads* `ThreadA` *e* `ThreadB` *durante a execução do sistema .*

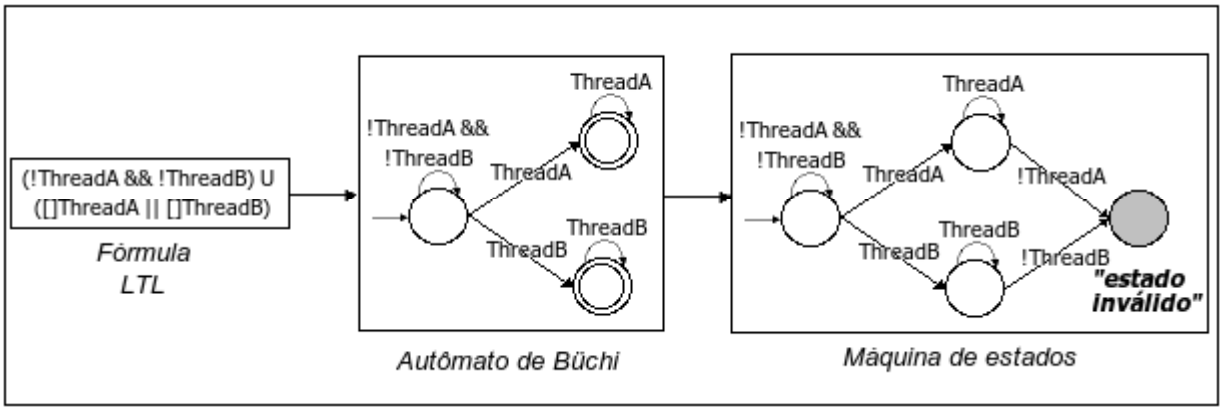

Figura 3.3: Representação gráfica do processo de conversão no *DesignMonitor* para uma propriedade comportamental.

A Figura 3.4 apresenta uma visão geral da arquitetura para verificação de propriedades comportamentais. O comportamento do sistema de software alvo deve ser monitorado à

medida que o mesmo está sendo executado. A idéia é que para cada propriedade comportamental especificada, conforme descrito na Seção 3.3, exista um código de monitoração responsável em observar os pontos de interesse. Nesses pontos é observado o comportamento das *threads* durante a execução do software. Os eventos capturados nestes pontos de interesse são encaminhados para o `Monitor` que é responsável por verificar se o comportamento observado condiz com o comportamento esperado.

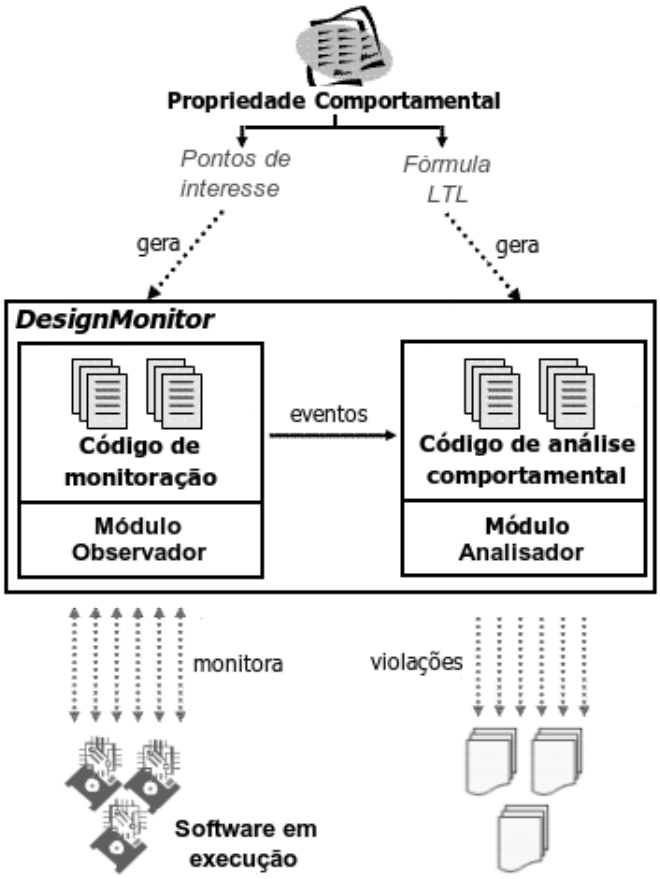

Figura 3.4: Visão geral do processo para instrumentação e análise comportamental em tempo de execução.

3.5 Considerações finais

Neste capítulo, apresentamos os pontos fundamentais para a aplicação da técnica *DesignMonitor*. Essa técnica visa verificar se o sistema de software desenvolvido obedece a certas propriedades comportamentais do projeto de software. Temos que as propriedades de projeto

podem ser definidas como estruturais ou comportamentais.

As propriedades estruturais podem ser analisadas a partir da ferramenta denominada *DesignWizard*. Essa ferramenta permite que as propriedades estruturais sejam especificadas de maneira semelhante a testes de unidade, o qual chamamos de testes estruturais, utilizando a API *DesignWizard*. A partir da extração de fatos do sistema de software alvo, estes são armazenados e manipulados pela API, permitindo verificar se o código condiz com certas propriedades estruturais especificadas através dos testes estruturais. Detalhes do seu uso estão descritos no Apêndice A.

Por outro lado, as propriedades comportamentais são analisadas a partir da monitoração da execução do sistema de software. Essa monitoração é realizada nos pontos de interesse específicos do software. O comportamento observado é então confrontado com o comportamento desejado, expresso através de fórmulas LTL. As fórmulas LTL são convertidas em autômatos de Büchi, aonde os estados do sistema de software são avaliados. Sendo assim possível a detecção da violação de alguma propriedade comportamental com relação ao comportamento observado. No Capítulo 4 apresentamos a ferramenta *DesignMonitor*, para suporte ferramental a técnica de análise comportamental apresentada neste capítulo.

Capítulo 4

DesignMonitor

Neste capítulo, apresentamos o protótipo da ferramenta denominada *DesignMonitor*. Esse protótipo foi desenvolvido visando dar suporte ferramental para à técnica de análise comportamental apresentada no Capítulo 3. Inicialmente, apresentamos uma visão geral da arquitetura definida para a ferramenta *DesignMonitor* que é subdividida em dois módulos. Em seguida, apresentamos as caracteríticas gerais do projeto e da implementação de cada um dos módulos que compõe a ferramenta. Para tanto, iremos considerar a monitoração com relação ao comportamento das *threads* durante a execução de um sistema de software *multithreads*. Com o intuito de ilustrar a aplicação do protótipo em um estudo de caso real, discutiremos sua validação utilizando o software *OurGrid* no Capítulo 5.

4.1 Visão geral

Com base na arquitetura apresentada na Seção 3.4.1 a ferramenta *DesignMonitor* foi dividida em dois módulos: o módulo de monitoração e o módulo de análise comportamental, conforme ilustramos na Figura 4.1. Como visto no Capítulo 3, a técnica *DesignMonitor* têm suas atividades desenvolvidas em duas fases distintas (pré-execução e execução), nas quais cada módulo possui as seguintes atribuições:

- **módulo de monitoração** - na fase de pré-execução é responsável por instrumentar o código-fonte do software alvo. Durante a fase de execução este módulo monitora e captura as informações relevantes do comportamento nos pontos de interesse;

- **módulo de análise comportamental** - na fase de pré-execução é responsável por gerar o código analisador para cada propriedade comportamental. Durante a fase de execução este módulo recebe os eventos referentes ao comportamento observado nos pontos de interesse de uma determinada propriedade comportamental e confronta com o comportamento desejado através do código analisador dessa propriedade.

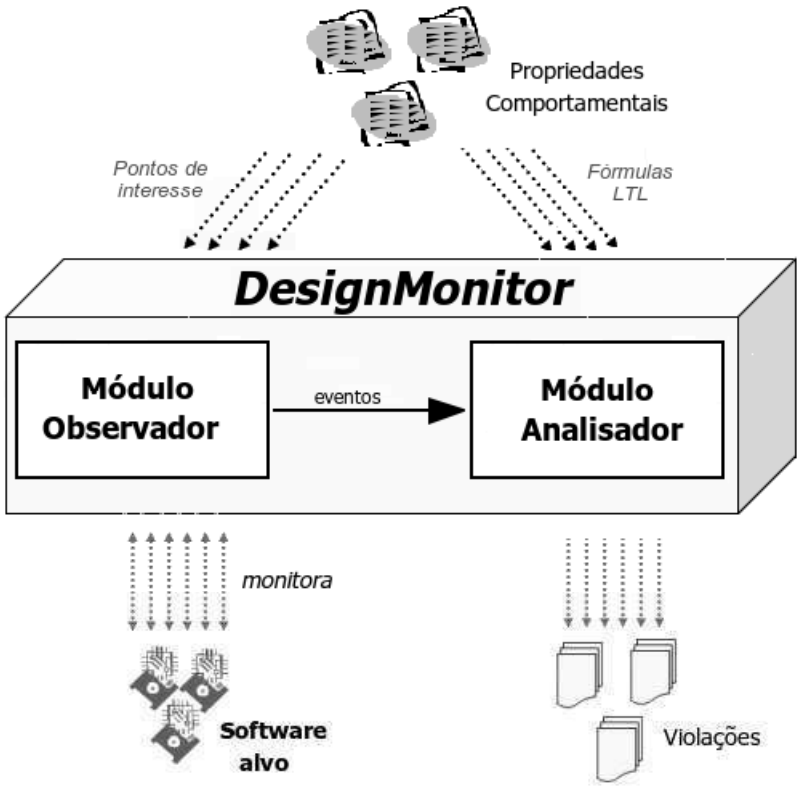

Figura 4.1: Visão geral da ferramenta *DesignMonitor*.

4.1.1 Módulo de monitoração

A Figura 4.2 ilustra uma visão geral do processo de geração do código de monitoração. Como podemos observar o módulo de monitoração, na fase de pré-execução, é responsável pela instrumentação automática do sistema de software alvo. Essa instrumentação é realizada através dos códigos de monitoração gerados automaticamente pelo gerador do código de monitoração a partir dos pontos de interesse, onde para cada propriedade comportamental existe um código de monitoração associado.

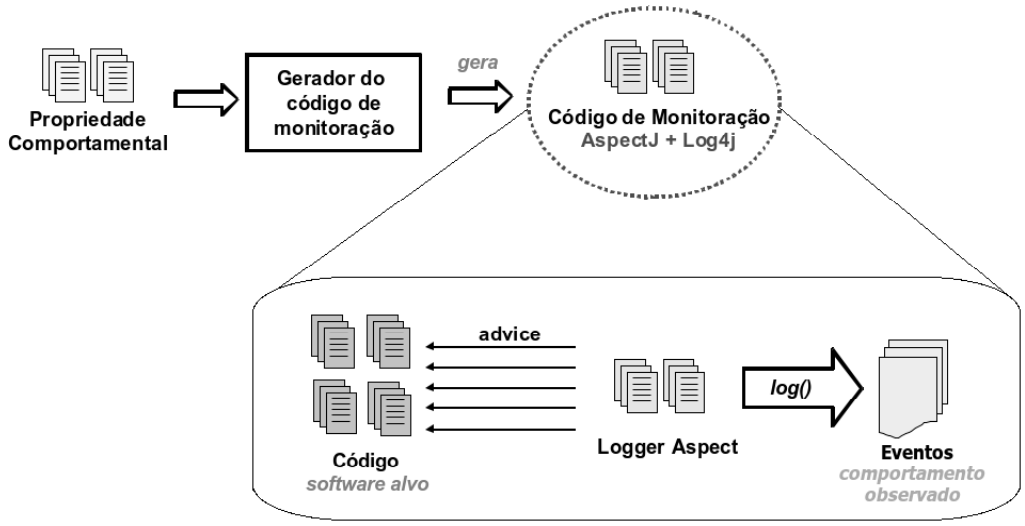

Figura 4.2: Processo de geração do código de monitoração para uma propriedade comportamental.

Na fase de execução, o código de monitoração de uma dada propriedade comportamental é responsável por monitorar o comportamento do sistema de software alvo nos pontos de interesse dessa propriedade durante sua execução. O meio utilizado para realizar a monitoração do comportamento do sistema é a partir da Programação Orientada a Aspectos (POA). Para tanto, como estamos considerando sistemas desenvolvidos em Java optamos pelo uso de AspectJ[1] na implementação do código de monitoração. A escolha pelo uso de aspectos se deve ao fato destes permitirem que o sistema de software alvo seja instrumentado de maneira não intrusiva, ou seja, sem alterar o código fonte de origem. Além disso, permite a parametrização com a escolha das classes e/ou métodos a serem monitorados.

Informações relevantes do comportamento observado nos pontos de interesse são capturadas e encaminhadas como eventos para o módulo de análise comportamental via Log4J [Prod]. Log4J é uma API que faz parte do projeto Jakarta de código aberto (*Open Source*) que tem como objetivo permitir ao desenvolvedor fazer *logging* em suas aplicações). Ela permite ao desenvolvedor controlar, de maneira flexível, cada saída de *log* e ativar o *log* em tempo de execução sem modificar os binários da aplicação, esse comportamento pode ser controlado apenas editando um arquivo de configuração. Como características deste *framework* podemos citar a flexibilidade e rapidez de geração de *logging* em tempo de execução,

[1] Uma extensão da linguagem Java para POA.

sem inserir custos de performance para a aplicação. Além disso, uma das características importantes desse *framework* é permitir o envio de *log* remotamente, viabilizando a arquitetura de monitoração distribuída apresentada na Seção 3.4.1.

4.1.2 Módulo de análise comportamental

O módulo de análise comportamental é constituído pelo código de análise gerado automaticamente a partir das fórmulas LTL, como ilustramos na Figura 4.3. Assim como no módulo de monitoração, para cada propriedade comportamental existe um código de análise comportamental associado. Esse código de análise é baseado em autômatos de Büchi, que são obtidos a partir da transformação das fórmulas LTL pela ferramenta LTL2BA4J [Too]. O comportamento observado nos pontos de interesse para uma determinada propriedade comportamental é enviado pelo módulo de monitoração para o módulo de análise comportamental na forma de um *log* do comportamento observado. No módulo de análise comportamental esse comportamento é analisado pelo código analisador específico para àquela propriedade comportamental. Caso o comportamento observado viole a propriedade comportamental o *DesignMonitor* comunica ao usuário tal violação.

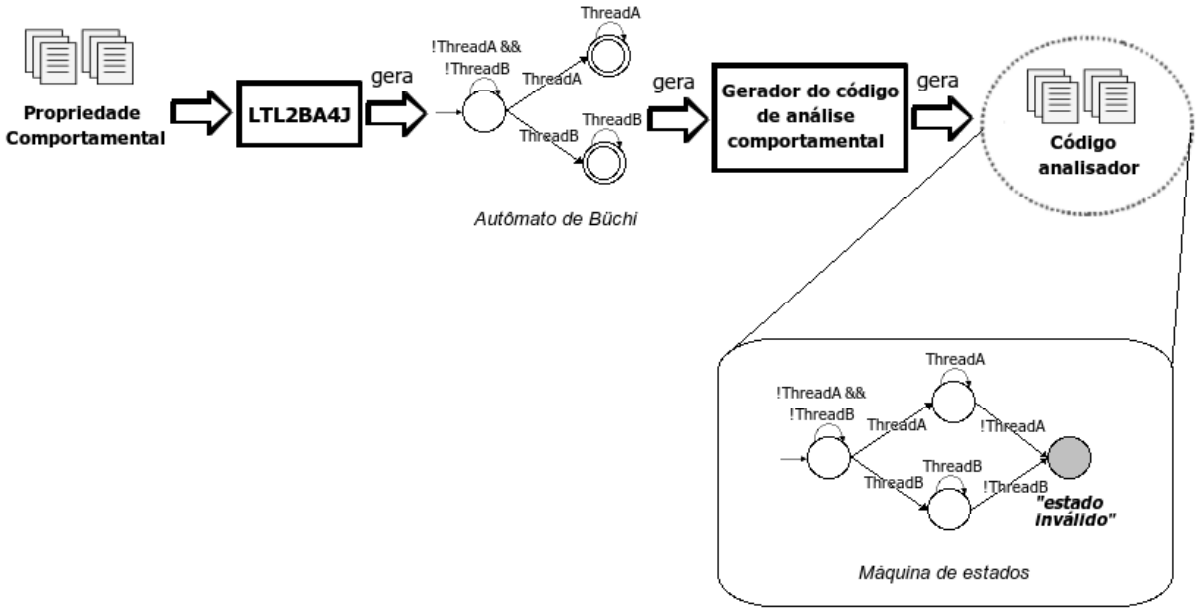

Figura 4.3: Processo de geração do código analisador.

4.2 Implementação

O principal objetivo da ferramenta *DesignMonitor* é permitir que durante o desenvolvimento de sistemas de software os desenvolvedores detectem automaticamente se o código por eles implementado violam alguma das propriedades comportamentais especificadas do projeto de software. Com isso, o protótipo da ferramenta *DesignMonitor* foi desenvolvido de modo integrado ao ambiente de desenvolvimento. Para tanto, foi considerado como ambiente a plataforma de desenvolvimento Eclipse [Proc]. Ao fazer uso da ferramenta *DesignMonitor*, na plataforma Eclipse, temos o projeto do sistema de software a ser monitorado e o projeto da ferramenta *DesignMonitor*. No projeto do sistema de software alvo é inserido o código aspecto de monitoração, que foi anteriormente gerado pelo módulo de monitoração do *DesignMonitor*. O projeto *DesignMonitor* contém o código de análise comportamental que também foi gerado automaticamente no módulo analisador. Ambos devem ser executados em paralelo, assim à medida que o sistema de software alvo for sendo executado os pontos de interesse são monitorados e os eventos capturados são analisados.

4.2.1 Monitorador

O módulo de monitoração é responsável por gerar automaticamente o código de monitoração a partir da especificação das propriedades comportamentais. O código aspecto de monitoração contém essencialmente os pontos de interesse. Para cada propriedade comportamental existe um código aspecto que herda funcionalidades do aspecto `DesignMonitorAspect`. O código aspecto `DesignMonitorAspect` define quais as informações do comportamento observado serão encaminhados para o módulo analisador. Considerando a propriedade comportamental do Exemplo 3.1, o código aspecto de monitoração gerado é conforme o apresentado no Código 4.1.

Quando o sistema de software alvo está sendo executado, se um dos pontos de interesse de alguma propriedade for acessado, o código aspecto de monitoração correspondente a essa propriedade captura as informações relevantes desse comportamento observado. Considerando o Código 4.1 será monitorado o acesso a qualquer método das instâncias do tipo `ObjectA` (linha 12). As informações do comportamento são então organizadas num formato padrão e encaminhadas para o módulo analisador pelo `DesignMonitorAspect`,

```
1  package example.aspect;
2  import br.edu.ufcg.designmonitor.aspects.DesignMonitorAspect;
3
4  public aspect BehaviorPropertyModule extends DesignMonitorAspect{
5       public BehaviorPropertyModule() throws Exception{
6         super("BehaviorPropertyModule");
7       }
8
9       public pointcut monitoringPoints()
10      : execution(* ObjectA.*(..))
11        && !cflow(execution(int *.hashCode()))
12        && !cflow(execution(java.lang.String *.toString()));
13 }
```

Código 4.1: Código de monitoração da propriedade comportamental do Exemplo 3.1.

como podemos observar no Código 4.2 na linha 20. As informações enviadas ao módulo analisador são as seguintes: o nome da propriedade comportamental cujo ponto de interesse foi acessado; o nome do tipo e a instância (*hashCode*) do objeto acessado, com o nome do método acessado (chamado), a linha do método no código de sua classe; o nome da *thread* com seu respectivo *hashCode*; e o *host* aonde a informação foi capturada.

```
1  package br.edu.ufcg.designmonitor.client.aspects;
2  import org.aspectj.lang.JoinPoint;
3  import br.edu.ufcg.designmonitor.client.log.DesignMonitorLogger;
4
5  public abstract aspect DesignMonitorAspect {
6     protected final static String SEP = "#";
7     protected static String nameAspect;
8     public DesignMonitorAspect(String nameAspect) {
9        this.nameAspect = nameAspect;
10    }
11
12    protected static void trace_record(JoinPoint joinPoint) {
13      // captura as informações relevantes do ``joinPoint''
14      ...
15      DesignMonitorLogger.getInstance().sendMsg(trace);
16   }
17
18   public abstract pointcut monitoringPoints();
19
20   before(): monitoringPoints() {
21     trace_record( thisJoinPoint );
22   }
23 }
```

Código 4.2: Trecho do código do aspecto abstrato `DesignMonitorAspect`.

4.2.2 Analisador

As propriedades comportamentais descrevem o comportamento esperado sob os pontos de interesse expressos como fórmulas LTL são convertidos para autômatos de Büchi equivalentes através da ferramenta LTL2BA4J. Essa ferramenta é a versão em Java da ferramenta LTL2BA em ANSI C, que permite a conversão das fórmulas escritas em lógica temporal LTL para autômatos de Büchi conforme descrito em [GO01].

As fórmulas LTL contêm símbolos proposicionais, operadores booleanos, operadores temporais, e parênteses (utilizando espaço entre os símbolos). Elas são expressas utilizando a mesma sintaxe do verificador de modelos Spin [Hol97].

- Símbolos proposicionais: `true`, `false` ou alguma palavra (*string*) (exceto palavras reservadas);
- Operadores booleanos: `!` (negação), `->` (implicação), `<->` (eqüivalência), `&&` (e), `||` (ou);
- Operadores temporais: `[]` (sempre), `<>` (eventualmente), `U` (enquanto), `X` (próximo);

Considerando que cada fórmula LTL expressa o comportamento das *threads* sob determinados pontos de interesse, temos que os símbolos proposicionais (proposições atômicas) que compõem essas fórmulas são os nomes das *threads*. Suponha que a fórmula LTL `(!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB)`, onde `ThreadA` e `ThreadB` são nomes de *threads* existentes no sistema de software alvo, represente uma propriedade comportamental que desejamos verificar. Seguindo a sintaxe descrita anteriormente, essa fórmula LTL será reescrita da seguinte maneira `(!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB)`.

Na Figura 4.4 temos o diagrama de classe do Analisador dos eventos monitorados. Como podemos observar, a classe `Verifier` possui as relações entre as propriedades comportamentais e os seus respectivos autômatos. Para cada propriedade comportamental existe uma fórmula LTL, um nome e um conjunto de pontos de interesse (classe `LTLRule`).

Como visto na Seção 4.1.2, o módulo analisador transforma cada fórmula LTL em um autômato de Büchi equivalente para cada `LTLRule`, através da ferramenta LTL2BA4J. A saída da ferramenta LTL2BA4J é uma representação do autômato de Büchi num formato

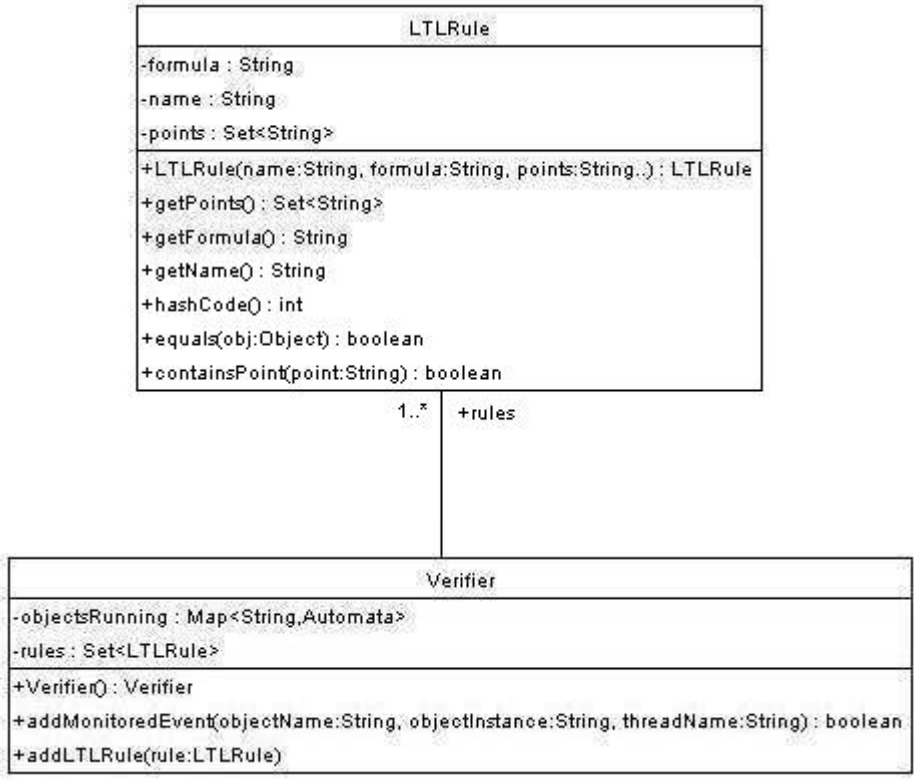

Figura 4.4: Diagrama de classe do analisador dos eventos monitorados.

de texto, considerando a fórmula `(!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB)` temos que a saída da ferramenta LTL2BA4J é a seguinte:

```
<state1_3(final)> --[ThreadB]--> <state1_3(final)>
<state1_2(final)> --[ThreadA]--> <state1_2(final)>
<state0_-1(initial)> --[!ThreadB, !ThreadA]--> <state0_-1(initial)>
<state0_-1(initial)> --[ThreadA]--> <state1_2(final)>
<state0_-1(initial)> --[ThreadB]--> <state1_3(final)>
```

Com base na saída da ferramenta LTL2BA4J, o *DesignMonitor* converte o autômato de Büchi numa máquina de estados, cujo código da mesma constitui o código de análise comportamental. Essa máquina de estados simula o comportamento das *threads* nos pontos de interesse de sua respectiva propriedade comportamental.

Essa conversão considera que uma fórmula LTL expressa o comportamento desejado, assim todos os estados existentes no autômato de Büchi são tidos como *"estados válidos"* durante a execução do sistema, desta maneira para a máquina de estados do *DesignMonitor* não existe mais o conceito de *"estado final"*. Além disso, um novo estado é adicionado, denominado *"estado inválido"*, e para cada *"estado válido"* acrescentamos uma nova tran-

sição entre esse estado e o *"estado inválido"*. O *"estado inválido"* representa que houve um status indesejável, ou seja, um comportamento inesperado durante a execução do software alvo. A função de transição para as novas transições é um evento diferente de todas as demais transições existentes para àquele *"estado válido"*.

Na Figura 4.5 ilustramos através de uma representação gráfica o autômato de Büchi obtido a partir da transformação da fórmula LTL `(!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB)`, onde `ThreadA` e `ThreadB` são nomes de *threads* existes no sistema de software alvo, que em seguida é convertido na máquina de estado utilizada para acompanhar o comportamento das *threads* `ThreadA` e `ThreadB` durante a execução do sistema.

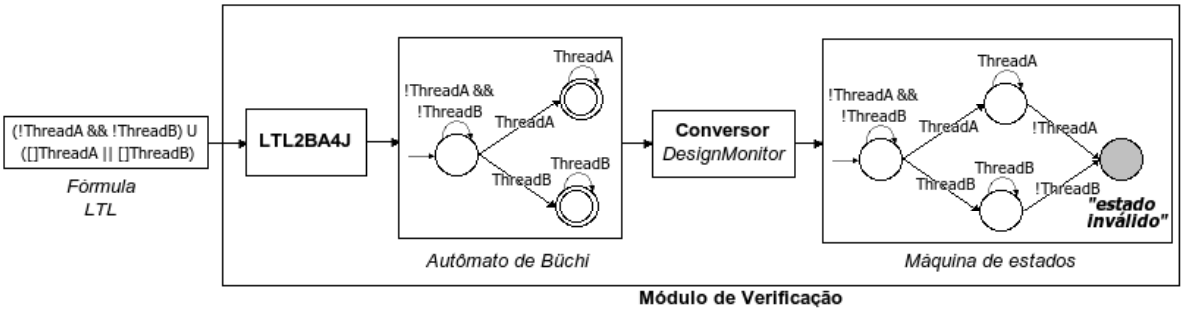

Figura 4.5: Representação gráfica do processo de conversão na ferramenta *DesignMonitor*.

Como pode ser visto no diagrama de classe ilustrado na Figura 4.6 essa conversão é implementada pela classe `Automata`, que recebe como parâmetro em seu construtor uma fórmula LTL. Durante a execução do sistema de software alvo, para cada propriedade comportamental é criada um objeto do tipo `Automata`. Suponha a propriedade comportamental *RuleX*, quando um ponto de interesse de *RuleX* é acessado, o módulo de verificação recebe um evento informando o comportamento observado. Então, a máquina de estados referente a propriedade comportamental *RuleX* executa a operação `run` que recebe como paramêtro o nome da *thread* desse evento. Se esse evento levar ao *"estado inválido"* retornamos `false`, informando que o comportamento observado viola a propriedade comportamental especificada. Caso contrário, retornamos `true` indicando que o comportamento observado é um comportamento esperado.

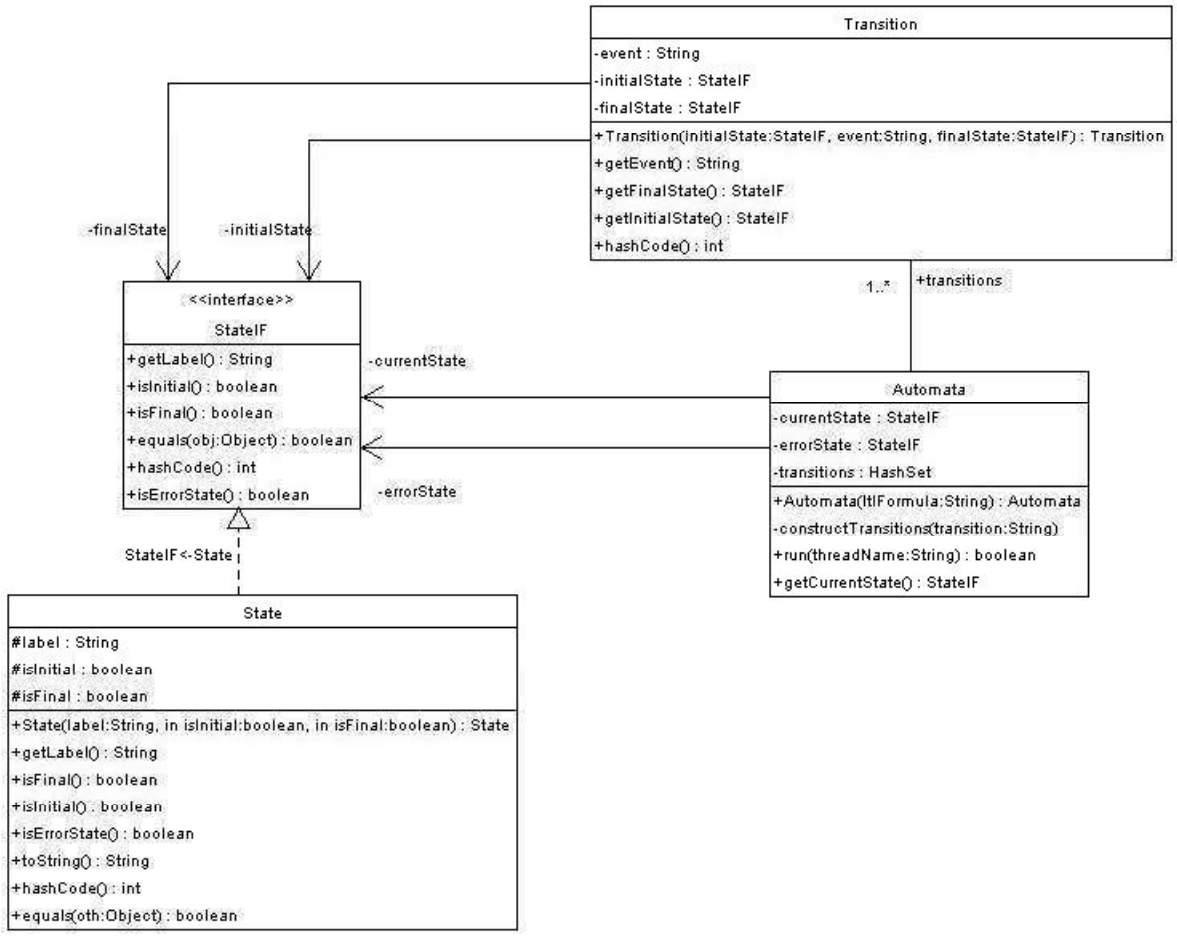

Figura 4.6: Diagrama de classe do conversor do *DesignMonitor* para máquinas de estados.

4.3 Considerações finais

Neste capítulo, apresentamos a arquitetura e as principais características da implementação do protótipo da ferramenta *DesignMonitor*. Vimos como sua arquitetura foi definida a partir da técnica apresentada no Capítulo 3. A entrada dos módulos de verificação e análise são as propriedades comportamentais especificadas conforme descrito na Seção 3.3. Como visto, cada propriedade comportamental é formada pela relação entre os pontos de interesse e a especificação do comportamento. A Figura 4.7 ilustra uma visão geral do processo de geração dos códigos de monitoração e análise comportamental. Os pontos de interesse dão origem ao código de monitoração e a especificação do comportamento através de uma fórmula LTL dá origem ao código de análise comportamental.

Além disso, discutimos qual é o papel de cada um dos módulos (monitoração e análise comportamental) que compõem a ferramenta *DesignMonitor*. Além disso, foram apre-

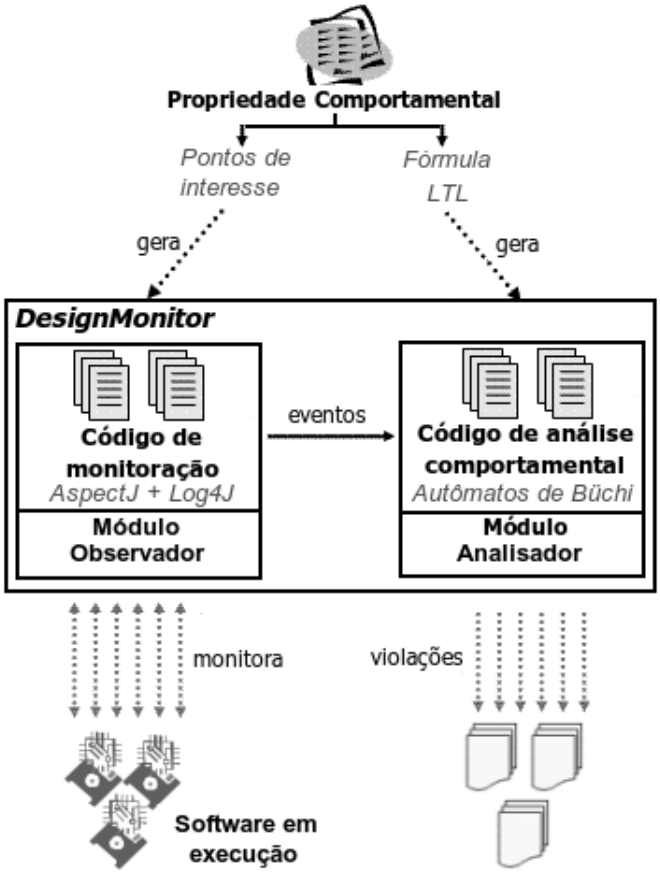

Figura 4.7: Visão geral do processo de geração de códigos no *DesignMonitor*.

sentadas as principais características de implementação desses módulos, mostrando os sub-módulos que os compõem.

Capítulo 5

Estudo de Caso

Este capítulo tem como finalidade demonstrar a aplicação da técnica *DesignMonitor*, apresentada no Capítulo 3, junto a um sistema de software real. Para isto, utilizamos o sistema *OurGrid*. O objetivo desse sistema é o desenvolvimento de um *middleware* para a execução de aplicações *Bag-Of-Task* (BoT)[1] em grades computacionais [FK99]. Inicialmente, iremos descrever os problemas que motivaram a realização deste trabalho e contextualizamos as soluções propostas pelo projeto *OurGrid* para esses problemas. Em seguida, a partir das soluções propostas apresentamos as propriedades comportamentais que compõem o projeto de software *OurGrid*. Por último, discutimos como essas propriedades são especificadas e analisadas.

5.1 Projeto *OurGrid*

O software *OurGrid* é uma plataforma para a execução de aplicações *Bag-Of-Task* (BoT). Aplicações BoT são aquelas que podem ser divididas em sub-tarefas independentes, que podem ser executadas paralelamente em computadores diferentes. Com isso, o tempo de processamento da aplicação é reduzido, aumentando conseqüentemente o poder computacional. Esses tipos de aplicações são utilizadas em diversos cenários, incluindo mineração de dados, processamento de imagens, buscas exaustivas, biologia computacional, dentre outros. O objetivo principal do software *OuGrid* é oferecer para o usuário de sua comunidade uma grade computacional aberta. Essa grade é baseada na troca de favores entre seus integran-

[1] Conjunto de tarefas que podem ser executadas de forma independente, em paralelo.

tes, com o compartilhamento de recursos computacionais no processamento de aplicações de grande escala que podem ser subdivididas em tarefas [CBC+04]. O *OurGrid* está sendo desenvolvido em Java desde 2001 no Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande (UFCG), em parceria com a empresa *Hewlett Packard* (HP). A versão do *OurGrid* considerada neste trabalho é a 3.3.1 [2].

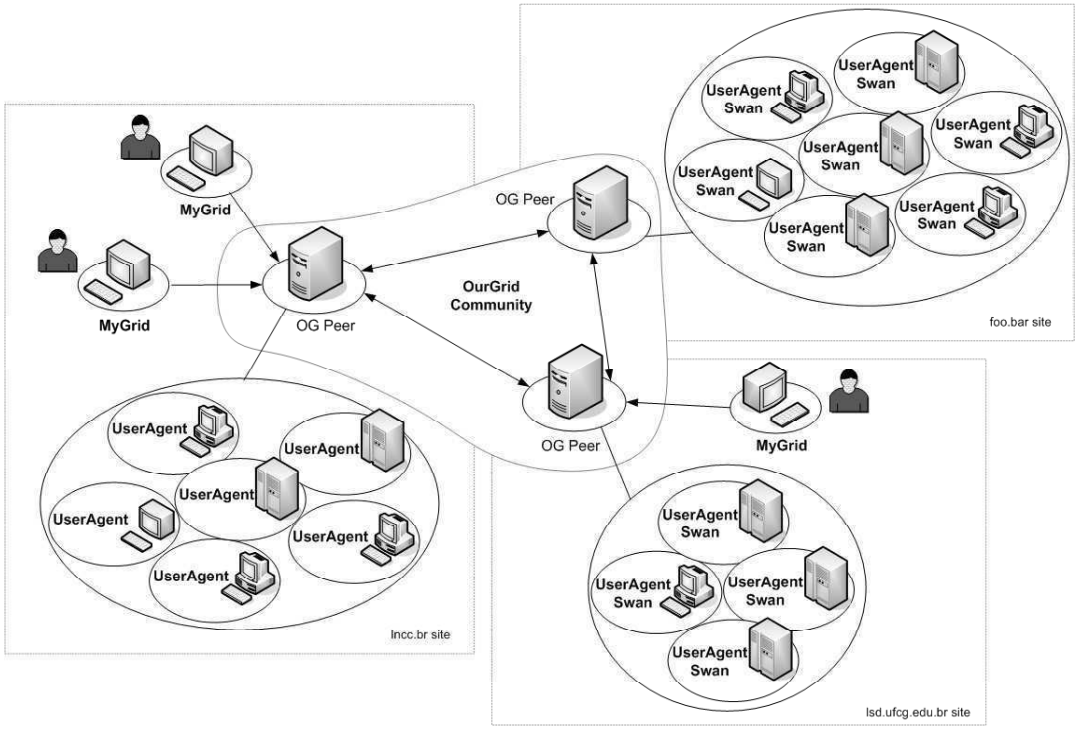

Figura 5.1: Visão geral da solução *OurGrid*.

Uma visão geral da solução *OurGrid* é ilustrada na Figura 5.1 [Proe]. O *OurGrid* é composto por três componentes principais: *MyGrid*, *Peer*, e *UserAgent*:

- ***MyGrid*** - interface entre o usuário e a grade. O *MyGrid* e o componente responsável por receber dos usuários as solicitações para execução das tarefas, solicitar recursos disponíveis na grade e gerenciar o escalonamento das tarefas entre as unidades de processamento distribuídas pela rede. A máquina onde o *MyGrid* é executado é chamada de *home machine*, que é o ponto central de uma grade. O *MyGrid* é o responsável por distribuir as tarefas para as máquinas disponíveis na grade e, por fim, unir os resultados obtidos pela execução das tarefas e apresentar o resultado final ao usuário solicitante;

[2] http://www.ourgrid.org/

- ***Peer*** - é o componente que identifica quais são as máquinas (que pertencem ao seu domínio administrativo) disponíveis e como elas poderão ser utilizadas pelo *MyGrid* para executar tarefas;

- ***UserAgent*** - componente responsável por executar as tarefas na grade. As máquinas que possuem o *UserAgent* instalado são chamadas de *gums*, ou seja, são as máquinas reais que executam as tarefas no *OurGrid*. Elas fornecem as funcionalidades necessárias para a comunicação entre a *home machine* e as *gums*.

Ao longo da evolução do software *OurGrid* chegou-se a um ponto em que entender, modificar, manter e evoluir tornavam-se tarefam cada vez mais difíceis. Os usuários e desenvolvedores queixavam-se de falhas no funcionamento do *OurGrid*, as quais não se conseguia identificar a causa. Após avaliação, se constatou que a implementação havia divergido do projeto (*design*) esperado. Diante desse quadro, a solução adotada foi a de se refazer o projeto do *OurGrid*, mas mantendo a idéia da solução *OurGrid* original. Adicionalmente, foram definidas algumas propriedades estruturais e comportamentais que os desenvoldores devem obedecer durante a implementação do software. Entretanto, a atividade de detecção de violações dessas propriedades devem interferir o mínimo possível no ambiente de produção. Dessa maneira, surgiu a necessidade de um mecanismo automático capaz de detectar violações de propriedades do projeto de software.

5.2 Propriedades comportamentais

As propriedades comportamentais do sistema *OurGrid* estão diretamente relacionadas com o comportamento das *threads* durante a sua execução. Dentre os componentes que compõem o *OurGrid*, o*MyGrid* e *Peer* são componentes modularizados. Cada módulo que compõem esses componentes possuem múltiplas *threads*, dentre as quais existe uma "*thread principal*" que possui características especiais, como de ser a única *thread* que pode acessar determinadas instâncias de objetos em um dado momento. Assim, para detectar violações dessa propriedade comportamental durante a execução do software é necessário observar o comportamento das *threads* do sistema *OurGrid*.

Para exemplificar a especificação e como é feita a detectção de violações das proprieda-

des comportamentais na aplicação da técnica *DesignMonitor*, iremos utilizar o componente *MyGrid*. Este componente é composto por dois módulos:

- ***Scheduler*** - módulo responsável por escalonar de maneira eficiente a alocação de processadores disponíveis na grade para a execução das tarefas submetidas e, por fim, unir os resultados obtidos pela execução das tarefas e apresentar o resultado final ao usuário solicitante;
- ***ReplicaExecutor***- módulo responsável por efetivamente executar as tarefas.

5.2.1 Especificação das propriedades comportamentais

Considerando o componente *MyGrid*, uma das propriedades comportamentais que o compõem possui as seguintes características:

- cada módulo que compõem o componente *MyGrid* é *multithreaded*;
- dentre as múltiplas *threads* de um módulo, existe uma "*thread principal*" que possui características especiais.

Uma maneira de exemplificar essa propriedade comportamental é caracterizar as "*threads principais*" com a definição de uma cor específica para essas *threads*, o que irá identificar cada módulo. Dessa forma, temos que:

- os objetos são inicialmente criados sem cor;
- um objeto sem cor ao ser acessado por uma "*thread principal*" passa a ter a cor da *thread* que o acessou;
- caso o objeto já possua uma cor, este só pode ser acessado pela "*thread principal*" de mesma cor, ou seja, pela *thread* do módulo ao qual o objeto pertence.

Para a técnica *DesignMonitor* o que irá representar essa cor é o nome das *threads*. Como visto anteriormente, o componente *MyGrid* é composto pelos módulos *Scheduler* e *ReplicaExecutor*, que possuem respectivamente as seguintes "*threads principais*": *SchedulerEventProcessorThread* e *ReplicaExecutorEventProcessorThread*, conforme ilustramos na Figura 5.2.

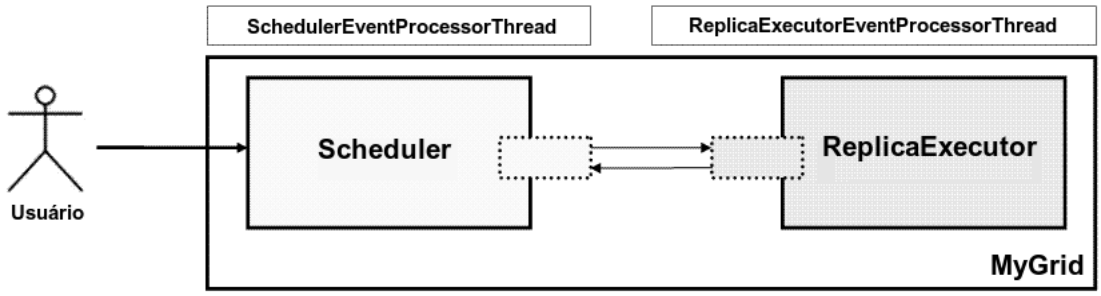

Figura 5.2: Componente *MyGrid*: propriedade comportamental.

```
ruleName := MyGridThreadsBehaviorProperty;

points := (execution(* org.ourgrid.mygrid..*.*(..)))
       && !within(org.ourgrid.mygrid..*Facade)
       && !within(org.ourgrid.mygrid.scheduler.BlackListEntry)
       && !within(org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager)
       && !within(org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager)
       && !within(org.ourgrid.mygrid..*Configuration)
       && !within(org.ourgrid.mygrid..*Impl)
       && !within(org.ourgrid.common.spec.PeerSpec)
       && !within(org.ourgrid.mygrid..*Test)
       && !within(org.ourgrid.mygrid..ui..*)
       && !within(org.ourgrid.mygrid..test..*)
       && !within(org.ourgrid.common.id.*)
       && !within(org.ourgrid.threadServicesAspects..*);

rule := (!schedulereventprocessorthread
         && !replicaexecutoreventprocessorthread)
            U ([]schedulereventprocessorthread
               || []replicaexecutoreventprocessorthread);
```

Código 5.1: Especificação da propriedade comportamental `MyGridThreadsBehaviorProperty`.

Essa propriedade comportamental é então especificada da seguinte maneira:

No Código 5.2 é apresentado o código aspecto de monitoração gerado automaticamente a partir dos pontos de interesse descritos na especificação da propriedades `MyGridThreadsBehaviorProperty` descrita no Código 5.1. Este código deve ser adicionado ao código-fonte do software alvo, no caso o *OurGrid*. Além disso, outras modificações são necessárias no projeto *OurGrid* com respeito a configuração. Neste, foram adicionamos o arquivo `designmonitor.jar` ao projeto *OurGrid*, alterando o arquivo de configuração do Ant [3], especificamente os arquivos `build.xml` e `build.properties` [Proa].

Na Figura 5.3 temos a representação gráfica do autômato de Büchi equivalente a fórmula LTL para especificação do comportamento das *threads*, que é gerado pela ferramenta LTL2BA4J [Too]. Por restrições da ferramenta LTL2BA4J a especificação do comportamento em LTL deve ser em letras minúsculas. Baseada no autômato de Büchi gerado é criada uma máquina de estados, definida pelo código de análise comportamental gerado automaticamente, aonde é realizada a verificação do comportamento observado.

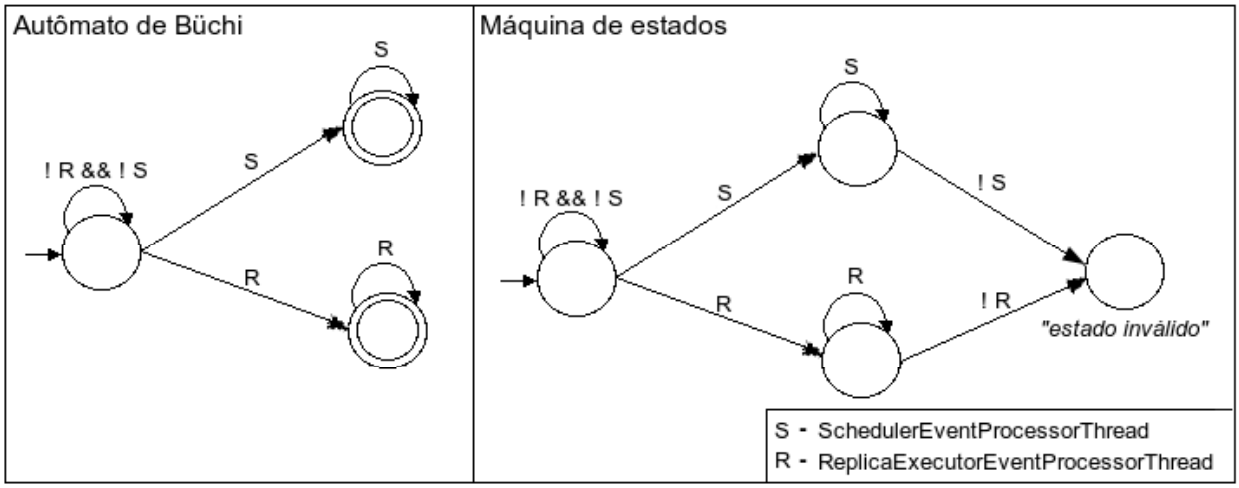

Figura 5.3: Representação gráfica da máquina de estados que verifica o comportamento observado.

Durante a execução do *OurGrid* para cada instância de objetos pertencentes aos pontos de interesse especificados é gerada uma máquina de estados e, em seguida, adicionada

[3] É uma ferramenta escrita em Java que auxilia no desenvolvimento, testes e *deployment* de aplicações Java, permitindo executar automaticamente tarefas rotineiras.

```
1  package org.ourgrid.designmonitor.mygrid;
2  import br.edu.ufcg.designmonitor.aspects.DesignMonitorAspect;
3  import java.net.InetAddress;
4  import java.net.UnknownHostException;
5  public aspect MyGridThreadsBehaviorProperty extends DesignMonitorAspect{
6     public MyGridThreadsBehaviorProperty() throws Exception{
7        super("MyGridThreadsBehaviorProperty");
8     }
9     public pointcut monitoringPoints()
10     : (execution(* org.ourgrid.mygrid..*.*(..)))
11       && !within(org.ourgrid.mygrid..*Facade)
12       && !within(org.ourgrid.mygrid.scheduler.BlackListEntry)
13       && !within(org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager)
14       && !within(org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager)
15       && !within(org.ourgrid.mygrid..*Configuration)
16       && !within(org.ourgrid.mygrid..*Impl)
17       && !within(org.ourgrid.common.spec.PeerSpec)
18       && !within(org.ourgrid.mygrid..*Test)
19       && !within(org.ourgrid.mygrid..ui..*)
20       && !within(org.ourgrid.mygrid..test..*)
21       && !within(org.ourgrid.common.id.*)
22       && !within(org.ourgrid.designmonitor..*)
23       && !within(org.ourgrid.threadServicesAspects..*)
24       && !cflow(execution(int *.hashCode()))
25       && !cflow(execution(java.lang.String *.toString()));
26 }
```

Código 5.2: Código de monitoração da propriedade comportamental `MyGridThreadsBehaviorProperty`.

ao módulo analisador do *DesignMonitor*. O módulo observador monitora cada instância de objetos pertencentes aos pontos de interesse especificados e captura o comportamento das *threads* nesses pontos à medida que o *OurGrid* é executado. Esse comportamento capturado é enviado para o módulo analisador que paralelamente analisa na máquina de estados equivalente àquele objeto observado se é um comportamento esperado ou não.

No contexto do estudo de caso, de posse das propriedades comportamentais e dos artefatos gerados a partir destas, para analisar o comportamento durante a execução do sistema *OurGrid* foi necessário definir os cenários de monitoração, que serão descritos na Seção 5.2.2.

5.2.2 Cenários de monitoração

O projeto *OurGrid* possui uma equipe de desenvolvedores que trabalham simultaneamente no desenvolvimento do sistema. Isto é possível graças ao uso do *Concurrent Version System* (CVS), um repositório central que permite que várias pessoas trabalhem simultaneamente em um mesmo projeto, além de guardar todo o histórico de alterações. Quando um desenvolvedor realiza alterações no código-fonte do *OurGrid* antes de enviar estas mudanças para o repositório central é necessário executar um conjunto de testes de unidade, testes de integração e de testes funcionais, para garantir que nenhuma "besteira" foi feita.

Para utilizarmos a ferramenta *DesignMonitor* em um sistema de software é preciso definir quais são os cenários de monitoração para que os desenvolvedores possam analisar se a execução do software está obedecendo as propriedades comportamentais especificadas. Desta maneira, os testes do *OurGrid* foram utilizados como cenários de monitoração. Os testes são formados por um conjunto de métodos. O ciclo de vida de um método é ilustrado na Figura 5.4, aonde podemos observar que a execução de um método de teste considera recursos ou dados distintos.

Figura 5.4: Ciclo de vida de um método de teste.

Assim, cada método de teste é visto como um cenário de monitoração, isto significa um novo ponto de partida na monitoração e análise do comportamento do sistema. Para isso, se faz necessário também observar a execução desse conjunto de testes. Para tanto, além do código aspecto para monitoração dos pontos de interesse adicionamos ao software *OurGrid* outro código aspecto para monitorar a execução dos testes. Esse aspecto informa ao módulo analisador do *DesignMonitor*, o início e o fim da execução de cada método de teste. No geral, a versão 3.3.1 do projeto *OurGrid* contém 568 métodos de testes (cenários de monitoração) que possuem sua execução monitoradas pelo *DesigMonitor*. A idéia é que ao executar o conjunto de testes os desevolvedores possam analisar ao mesmo tempo a conformidade entre o comportamento do software e as propriedades comportamentais especificadas.

5.2.3 Análise das propriedades comportamentais

À medida que os testes do *OurGrid* são executados, o *DesignMonitor* analisa cada cenário monitorado. Quando algum dos pontos de interesse é acessado o código aspecto de monitoração captura as informações relevantes no módulo de monitoração e envia para o módulo analisador. Este por sua vez, identifica qual a máquina de estado referente a este comportamento, então executa a função de transição considerando o estado corrente e o comportamento observado. Caso seja o primeiro acesso o estado corrente da máquina de transição é o estado inicial da mesma. Essa rotina se repete até que o cenário de monitoração seja completamente executado ou alguma violação seja detectada. Sendo assim, diante da não existência de violações pode-se dizer que o software, para aquela execução, está correto em relação as propriedades comportamentais especificadas. Contudo, a ocorrência de violações podem também gerar falsos positivos, caso a própria especificação das propriedades não esteja correta.

Ao executarmos os testes do *OurGrid*, juntamente com o *DesignMonitor*, algumas violações comportamentais foram detectadas. Essas foram então avaliadas com relação a sua veracidade, como por exemplo, se o ponto de interesse onde a violação ocorreu faz mesmo parte da propriedade comportamental violada. A notificação das violações é feita através de um arquivo de *log*, que identifica o início e o fim de cada cenário de monitoração, bem como as informações das violações detectadas, conforme mostrado na Figura 5.5.

Para avaliação do impacto no uso da técnica *DesignMonitor* no projeto *OurGrid* foram

```
...

START org.ourgrid.mygrid.scheduler.test.JobTest;

...
      RULE: MyGridThreadsRule;
      OBJECT: org.ourgrid.mygrid.scheduler.JobMonitor@5484187;
      THREAD VALID: SchedulerEventProcessorThread@487964;
      THREAD INVALID: ReplicaExecutorEventProcessorThread@987464;
...

FINISH org.ourgrid.mygrid.scheduler.test.JobTest;

...
```

Figura 5.5: Trecho de notificação de violação a partir de um arquivo de *log*.

realizados alguns experimentos. Inicialmente, executamos o conjunto de testes (cenários de monitoração) do *OurGrid* cinco vezes sem utilizar a ferramenta *DesignMonitor*. Para cada execução foi coletado o tempo dispensado para execução de cada método de teste (cenário de monitoração). Na Tabela 5.1 apresentamos o tempo total e a média na execução de cada conjunto de testes.

Tabela 5.1: Tempo de execução do conjunto de testes do *OurGrid* sem o uso da ferramenta *DesignMonitor*.

Experimento	**Tempo total**
01	13 minutos e 54 segundos
02	13 minutos e 08 segundos
03	13 minutos e 50 segundos
04	14 minutos e 39 segundos
05	13 minutos e 12 segundos
Média	***13 minutos e 44 segundos***

Em seguida, adicionamos ao projeto *OurGrid* os artefatos gerados a partir da propriedade comportamental especificada no Código 5.1. Posteriormente, executamos novamente cinco vezes o conjunto de testes, coletando o tempo dispensado na execução de cada método de teste (cenário de monitoração), apresentados na Tabela 5.2.

O que podemos observar é que o uso da ferramenta *DesignMonitor* dá, em média, um acréscimo de 4 minutos e 20 segundos no tempo de execução do conjunto de testes do *OurGrid*. Uma outra questão analisada foi o número de violações detectadas para cada execução

Tabela 5.2: Tempo de execução do conjunto de testes do *OurGrid* sem o uso da ferramenta *DesignMonitor*.

Experimento	**Tempo total**
01	18 minutos e 04 segundos
02	17 minutos e 58 segundos
03	17 minutos e 50 segundos
04	18 minutos e 19 segundos
05	18 minutos e 12 segundos
Média	***18 minutos e 04 segundos***

do conjunto de testes que são apresentadas na Tabela 5.3.

Tabela 5.3: Violação detectadas no software *OurGrid* pelo *DesignMonitor*.

Experimento	**Número de violações**
01	195 violações
02	190 violações
03	187 violações
04	193 violações
05	185 violações
Média	***190 violações***

No momento em que alguma violação comportamental é detectada o *DesignMonitor* pára de analisar aquele cenário em execução. Com isso, analisando os número de violações detectadas o que podemos concluir é que dos 568 testes em média 190 dos testes violaram alguma propriedade comportamental.

5.3 Considerações finais

Neste capítulo apresentamos a aplicação da técnica *DesignMonitor*, focando principalmente o uso da ferramenta, na especificação e análise das propriedades comportamentais do sistema de software real. Como dito anteriormente, a metodologia de desenvolvimento adotada neste trabalho foi orientada a um estudo de caso junto ao projeto *OurGrid*. Com isso, a técnica foi desenvolvida baseada nos requisitos e a partir dos resultados obtidos em experimentos realizados no software *OurGrid*.

Inicialmente, foi realizado um levantamento das propriedades de projeto existentes no software *OurGrid*. De acordo com o tipo de propriedade, estrutural ou comportamental, estas foram tratadas separadamente. Primeiramente, foram especificadas as propriedades estruturais com o auxílio da ferramenta *DesignWizard*, apresentada no Apêndice B. Após a especificação das propriedades estruturais, estas foram analisadas a partir da execução dos testes estruturais. Em seguida, foram feitos estudos para definição da técnica de detecção de violações das propriedades comportamentais.

Avaliando as propriedades que compõem uma propriedade comportamental, foi definido o modo como as mesmas deveriam ser especificadas, conforme visto no Capítulo 3. De posse da especificação das propriedades comportamentais do *OurGrid* foi visto que essas propriedades de interesse só podiam ser analisadas durante a execução do sistema. Para isso, realizamos a monitoração da execução do sistema de software por meio do uso de aspectos. Ao definirmos a arquitetura de monitoração, optamos por uma que permita facilmente a evolução da técnica para uso em sistemas distribuídos. Uma vez definido o modo como o eventos de interesse seriam observados e capturados, partimos para análise desse comportamento monitorado durante a execução do sistema de software. Com a representação através de uma máquina de estados do comportamento desejado foi implementado uma estrutura de dados que simula o comportamento observado nos pontos de interesse. A captura de um comportamento inesperado gera uma função de transição na máquina de estados que leva a um estado de falha (*"estado inválido"*), o que caracteriza uma violação da respectiva propriedade. Por fim, foi definido para o estudo de caso os cenários de monitoração das propriedades de comportamentais de modo a facilitar o trabalho do desenvolvedor na execução e análise dessas propriedades, escolhendo assim o conjunto de testes já existente no projeto *OurGrid*.

No entanto, observamos que a análise das propriedades comportamentais incluem um acréscimo no tempo de execução nos cenários de monitoração (teste do projeto *OurGrid*). Na análise das propriedades comportamentais, durante a execução do sistema, quando um ponto de interesse é acessado o aspecto interrompe a execução. O aspecto captura as informações relevantes e só então o sistema de software retorna a execução do ponto de onde parou. Paralelamente, o módulo analisador detecta se esse comportamento observado viola ou não propriedade comportamental associada ao mesmo.

Capítulo 6

Trabalhos Relacionados

Atualmente, alguns trabalhos semelhantes ao *DesignMonitor* vêm sendo desenvolvidos. Este capítulo tem como propósito apresentar alguns desses trabalhos, com a realização de uma análise crítica e comparativa com realação ao *DesignMonitor*. Inicialmente, apresentamos a abordagem *Odyssey-Tracer* que monitora o comportamento dos sistemas de software de maneira semelhante ao *DesignMonitor*. Em seguida, vemos uma outra abordagem baseada no uso de DbC para especificação de propriedades do software. Por fim, apresentamos um conjunto de ferramentas que monitoram e verificam o comportamento durante a execução de programas Java.

6.1 Odyssey-Tracer

O trabalho proposto em [CVW06] consiste num conjunto de ferramentas para extração de modelos dinâmicos a partir do código-fonte de sistemas orientados a objetos. Os modelos dinâmicos são representados através de diagramas de seqüência UML. A extração de modelos é dividida em duas etapas: coleta do caminho (*trace*) de execução e reconstrução dos diagramas de seqüência no ambiente *Odyssey*. Para isso duas ferramentas foram desenvolvidas para apoiar cada etapa do processo: o *Tracer* e a *Phoenix*, apresentada na Figura 6.1.

Tracer [CVW06] é uma ferramenta cujo objetivo é observar e coletar o caminho de execução de um sistema de software desenvolvidos em Java. Ela faz uso da tecnologia AspectJ para instrumentar o código binário (*bytecodes*) de aplicações Java. O caminho de execução observado é representado através de arquivo XML gerado contém as execuções de métodos,

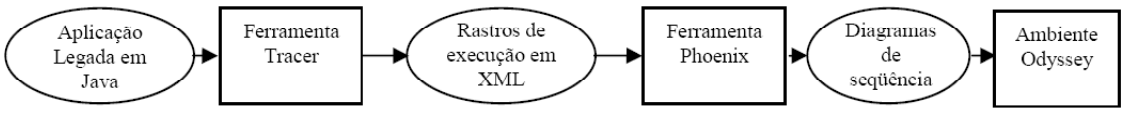

Figura 6.1: Conjunto de ferramentas para a extração de modelos dinâmicos.

indentados de acordo com a hierarquia de execução. Cada execução de método é acompanhada por informação contextual, envolvendo classe do método, nome, *thread* (linha de execução), instância e *timestamp* (Figura 6.2).

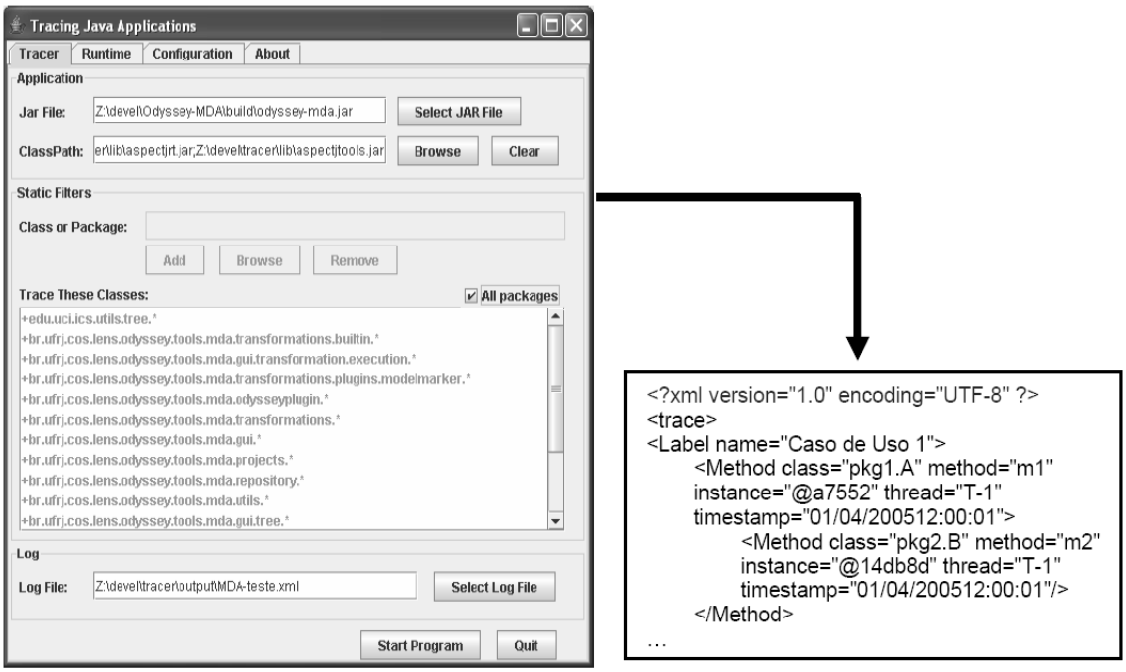

Figura 6.2: A ferramenta Tracer e o arquivo XML de saída.

O uso da abordagem *Tracer* é geral para qualquer aplicação Java, e o usuário apenas precisa informar o arquivo jar e o classpath da aplicação. Durante a execução da aplicação através da ferramenta *Tracer*, é possível habilitar e desabilitar a execução da coleta dos caminhos de execução. A ferramenta *Tracer* pode ser utilizada em conjunto com a ferramenta *Phoenix* [CVW06], um extrator de diagramas de seqüência, que está disponível como *plugin* do ambiente *Odyssey Light*[1].

[1]Ambiente de desenvolvimento que visa prover mecanismos, baseados em reutilização, para o desenvolvimento de software, servindo como um arcabouço onde modelos conceituais, arquiteturas de software, e modelos implementacionais são especificados para domínios de aplicação previamente selecionados.

A ferramenta *Phoenix* recebe como entrada os rastros de execução gerados pela ferramenta *Tracer*, realizando uma busca em profundidade e em largura na árvore de execuções de métodos, e reconstrói os diagramas de seqüência correspondentes no ambiente *Odyssey*, onde são visualizados. *Phoenix* também é capaz de reconstruir diagramas de seqüência em diferentes níveis de abstração. Na Figura 6.3 [CVW06] é apresentado um trecho de diagrama de seqüência extraído pela ferramenta *Phoenix*.

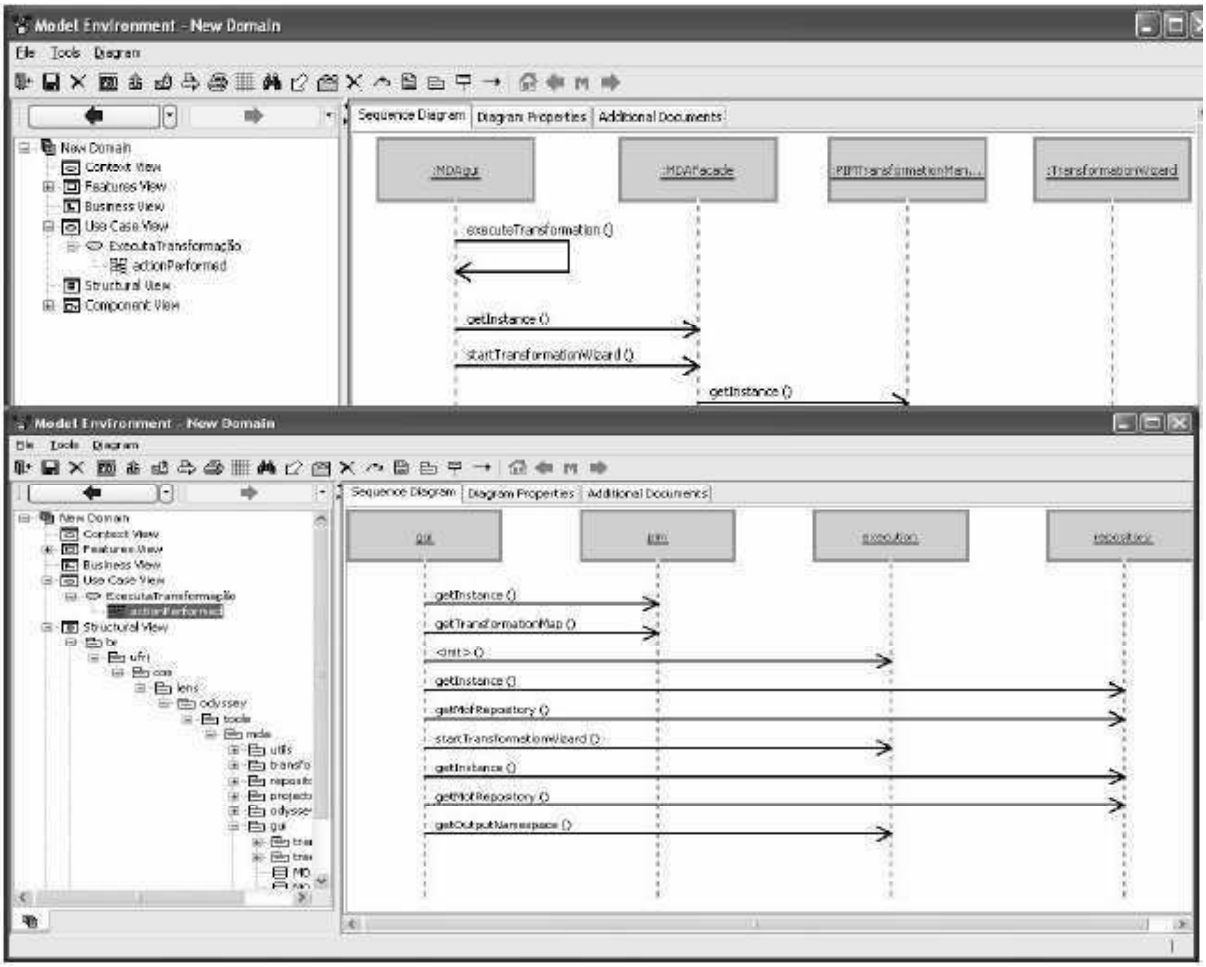

Figura 6.3: Trecho de diagrama de seqüência extraído pela ferramenta Phoenix.

A abordagem utilizada nesse trabalho para observar e capturar o comportamento dos sistemas de software durante sua execução assemelha-se a realizada pelo *DesignMonitor*. O comportamento observado é utilizado para a construção de diagramas de seqüência. Contudo, esse comportamento observado não é confrontado com o comportamento esperado, já que essa abordagem não visa a verificação do comportamento dos sistemas de software.

6.2 Esc/Java2

Esc/Java2 [Kin] é um trabalho desenvolvido pelo grupo de pesquisa de sistemas na Universidade de Dublin, Irlanda. Esc/Java2 consiste de uma ferramenta para verificação de programas Java, que procura encontrar erros comuns durante a execução através da análise estática a partir de anotações no código. A abordagem utilizada é a de verificação modular, que consiste em operar sobre cada método e cada classe especificadas separadamente.

Os requisitos são especificados como anotações no código Java em JML (*Java Modeling Language*) [LBR06b], conforme o Exemplo 6.1. Esc/Java2 permite verificar as inconsistências existentes entre os requisitos especificados e código implementado. Além disso, adverte os desenvolvedores sobre erros potenciais presentes na implementação que podem gerar falhas em tempo de execução, tais como: referência a ponteiros nulos, erros de indexação de *arrays*, erros de *cast* de tipos, dentre outros.

Exemplo 6.1 *Considere a implementação da classe* `Pessoa` *(Código 6.1). A esse código é inserida a especificação JML para o método* `equals`*. Esse método verifica se um determinado objeto igual a outro. A especificação do método* `equals` *define que o objeto* `p`*, passado como parâmetro, deve ser diferente de* `null` *e que os atributos* `nome` *de ambos os objetos devem ter valores iguais.*

```
1 public class Pessoa{
2   ...
3   //@ requires p!=null;
4   //@ ensures \result == nome.equals(p.getNome());
5   boolean equals(Pessoa p){
6      return this == p;
7   }
8   ...
9 }
```

Código 6.1: Exemplo de especificação JML.

Esc/Java2 permite que os requisitos sejam especificados fazendo uso de técnicas formais semelhante ao *DesignMonitor*. Contudo, não fornece meios de especificar propriedades relacionadas ao comportamento dos sistemas ao longo do tempo. Dessa forma, Esc/Java2 não trata de programas com múltiplas *threads*. Além disso, essa ferramenta não dá suporte a análise dinâmica para a verificação de propriedades comportamentais.

6.3 Jass

Jass (*Java with assertions*) [BFMW01] é uma abordagem de monitoração para sistemas de software escritos em Java. Asserções são escritas como anotações no código Java, expressando as propriedades que devem ser verdadeiras em determinados pontos da execução do programa. Estas são especificadas na forma de pré-condições e pós-condições de métodos, invariantes (de classe e de laço), variantes (de laço), e verificações adicionais que podem ser introduzidas em qualquer parte do código. Em adição às expressões booleanas, Jass permite que os desenvolvedores utilizem quantificadores universais e existenciais. Um pré-compilador traduz essas asserções escritas nos programas Java para código Java. A verificação da conformidade das asserções especificadas é realizada dinamicamente em tempo de execução. Exceções indicam violação das condições descritas nas asserções.

Os comandos para expressar as propriedades de interesse são:

- ***require* -** pré-condição;
- ***ensure* -** pós-condição;
- ***invariant* -** de classe ou laço;
- ***variant* -** de laço (deve ser positivo e decrescente com a execução do laço);
- ***check* -** verifica assertions em qualquer parte do código;
- ***rescue* -** bloco executado caso assertion seja false;
- ***retry* -** executa novamente o método (só pode ser utilizado dentro do bloco *rescue*).

Em adição a esses comandos Jass possui dois construtores especiais:

- ***Old* (pós-condição) -** representa o estado inicial do objeto antes da execução do método. O método *clone* deve ser implementado;
- ***changeonly* (pós-condição) -** limita a alteração de atributos da classe. Os atributos do objeto são comparados com os de *Old* pelo método *equals*, o qual deve ser rescrito.

Exemplo 6.2 *Considere a classe* `Buffer` *(Código 6.2). O* `Buffer` *armazena objetos (método* `add`*) e pode-se recuperar depois (método* `remove`*), em uma estrutura de dados do tipo fila (FIFO). O* `Buffer` *tem capacidade limitada, desse modo um novo objeto só pode ser adicionado caso o* `Buffer` *não esteja cheio. Naturalmente, nenhum objeto pode ser retornado caso o* `Buffer` *esteja vazio.*

```
1  public class Buffer implements Cloneable {
2    private int in,out,count;
3    private Object[] store;

4    public Buffer (int capacity) { ... }
5    public boolean empty() { ... }
6    public boolean full() { ... }
7    public int capacity() { ... }
8    private boolean inRange(int i) { ... }
9    public void add(Object o) { ... }
10   public Object remove() { ... }
11   public boolean contains(Object o) { ... }
12   ...
13 }
```

Código 6.2: Código da classe `Buffer`.

No Código 6.3 temos a inserção de comandos Jass no método `add` *da classe* `Buffer`*. Em Jass, as propriedades de interesse são escritas como comentários no código-fonte num formato especial.*

```
1  ...
2  public void add (Object object) {
3     /** require !isFull(); **/
4        if (o==null)
5           buffer[in % buffer.length] = new Default();
6        else
7           buffer[in % buffer.length] = object;
8        in++;
9     /** ensure changeonly{in,buffer};
10        Old.in == in - 1;
11        object!=null ? contains(object) : true; **/
12 }
13 ...
```

Código 6.3: Asserções do método `add` da classe `Buffer`.

Como visto anteriormente, para adicionar um novo objeto o `Buffer` *não pode estar com sua capacidade máxima. Essa propriedade é expressa como* `require !isFull()`*. Ao executar esse método as seguintes propriedades devem ser satisfeitas:*

- *o valor do atributo* `in` *(refere-se ao número de objetos contidos no* `Buffer`*) deve ser o valor atual menos um (*`Old.in == in - 1`*);*
- *se o objeto a ser adicionado for diferente de* `null` *então o* `Buffer` *deve possuir esse objeto (*`object != null ? contains(object) : true`*).*

Jass é uma proposta para o uso da metodologia DbC em programas Java, com a verificação das asserções em tempo de execução. Contudo, não permite especificar e verificar propriedades existentes em sistemas concorrentes (*multithreaded*) e reativos.

6.4 JavaMaC

JavaMaC [KKL+01; KKL+02; GRG+05] é um *framework* para a monitoração em tempo de execução de sistemas escritos em Java. A Figura 6.4 [KKL+01] ilustra a arquitetura da ferramenta JavaMaC. A monitoração e a verificação são executadas com base em uma especificação formal dos requisitos do sistema. As especificações das propriedades são escritas em MEDL (*Meta-Event Definition Language*). MEDL permite que os usuários definam variáveis auxiliares que armazenam valores que podem ser utilizadas para identificar eventos e condições. Essas especificações são utilizadas para criar automaticamente o verificador.

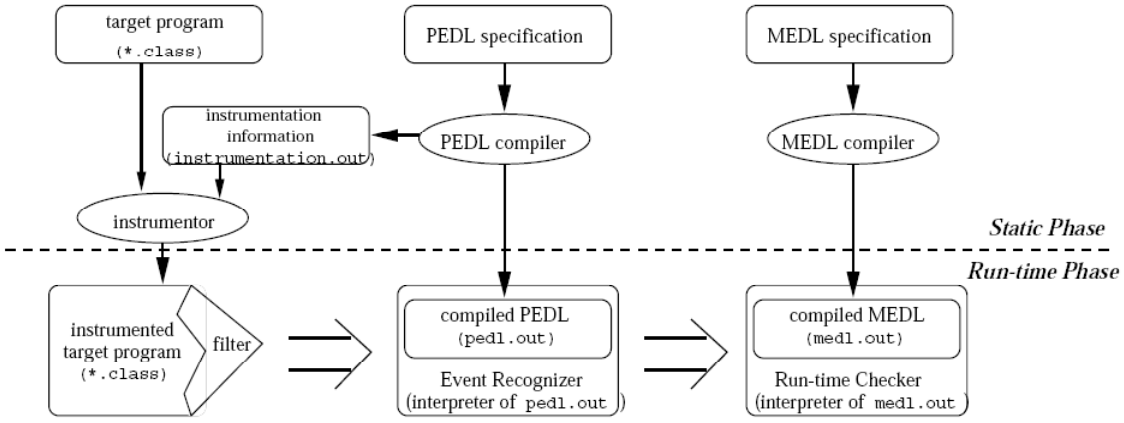

Figura 6.4: Arquitetura JavaMaC.

Um *script* de monitoração, escrito pelo usuário em PEDL (*Primitive-Event Definition Language*), é usado para monitorar objetos e métodos. PEDL pode observar variáveis locais e globais, e pode detectar nomes de variáveis falsas. Um filtro mantém uma tabela que

contém os nomes das variávies monitoradas e os endereços dos objetos correspondentes. Age como um observador que comunica a informação que deve ser verificada pelo monitor em tempo de execução. Os pontos de monitoração são introduzidos automaticamente, desde que o *script* de monitoração especifique qual informação necessita ser extraída, não onde na extração do código deve ocorrer. Faz uso da análise estática para determinar os pontos de monitoração e da análise dinâmica para verificar o comportamento do programa. O módulos de monitoração e de verificação não são acoplados, assim a monitoração e a verificação podem ser realizadas simultaneamente ou não. O monitor pode armazenar o caminho de execução em um arquivo de saída e o verificador pode analisar esse arquivo mais tarde. Existem trabalhos sendo realizados visando permitir a monitoração de aplicações executadas em grades computacionas através da ferramenta JavaMaC.

6.5 Pip

Pip [RKW+06] é proposta como uma infra-estrutura para detectar automaticamente erros estruturais e de desempenho em sistemas distribuídos, comparando o comportamento real com o comportamento previsto, auxiliando os desenvolvedores a identificarem as possíveis causas do comportamento inesperado em sistemas de software. Muitos destes erros refletem a discrepância entre comportamento do sistema de software e as suposições do desenvolvedor com relação a esse comportamento.

Pip permite que os desenvolvedores expressem através de uma linguagem declarativa as suas expectativas com relação à estrutura de comunicação, sincronismo e o consumo de recursos do sistema. Ela inclui ferramentas de instrumentação e anotação para registro do comportamento real do sistema e ferramentas para visualização e consulta do comportamento previsto, explorando o comportamento inesperado.

O comportamento de sistemas de software consiste de caminhos. Pip verifica apenas o caminho que realmente ocorreu durante o funcionamento do software. À medida que o programa é executado o comportamento do software é armazenado em arquivos com a representação desse comportamento a partir de eventos, representados como tarefas, observações e mensagens. Após a execução do sistema de software alvo, esse comportamento observado é confrontado com o comportamento esperado.

6.6 JavaMOP

Programação Orientada a Monitoração (*Monitoring-Oriented Programming* - MOP) [CR03] é uma técnica para o desenvolvimento e a análise de sistemas de software proposta pelo Laboratório de Sistemas Formais da Universidade de Illinois, Estados Unidos. A idéia geral dessa técnica é fazer uso métodos formais na verificação da conformidade entre a implementação e a especificação de sistemas de software em tempo de execução (*runtime*).

Os requisitos do sistema são expressos formalmante, através de anotações pontos específicos no código-fonte dos programas. Com base nessas anotações é gerado automaticamente o código de monitoração, que funciona como um verificador lógico dos requisitos. Na arquitetura de MOP, apresentada na Figura 6.5 [CR06], a linguagem lógica utilizada para especificar os requisitos é independente da linguagem do código de monitoração. Dessa maneira, MOP pode ser estendida para diversas linguagens lógicas e de codificação.

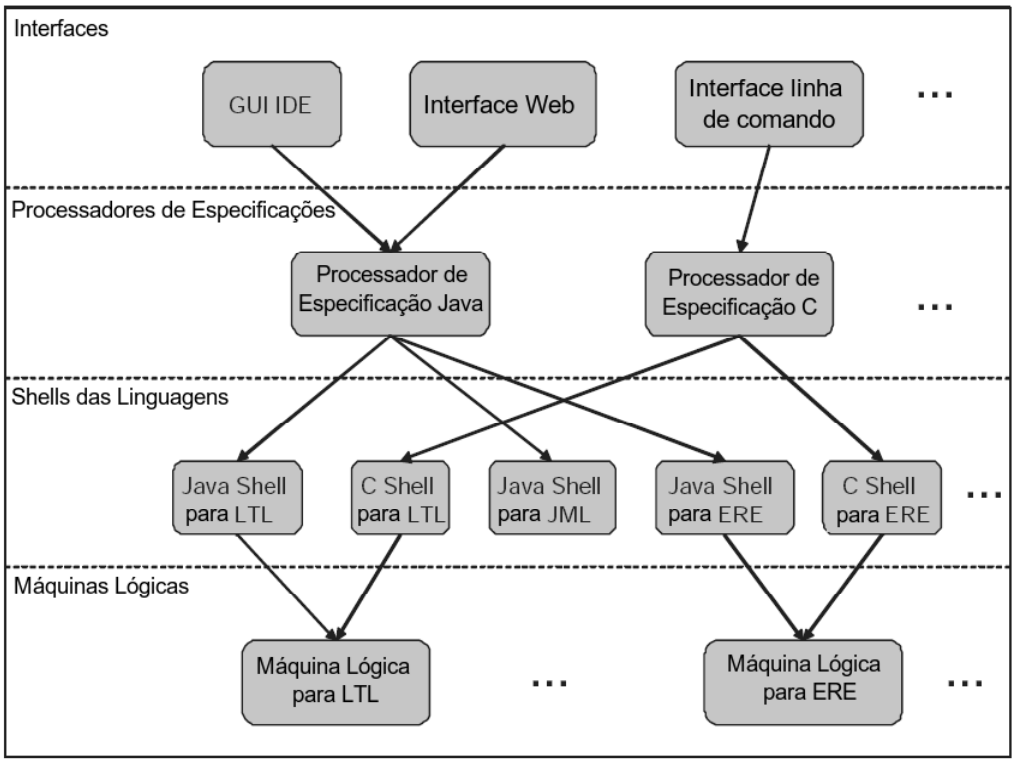

Figura 6.5: Arquitetura MOP.

Para oferecer suporte ferramental a essa técnica foi desenvolvida uma ferramenta, denominada JavaMOP [CdR06; CR05; CR06], para sistemas de software desenvolvidos em Java. A arquitetura da ferramenta JavaMOP, apresentada na Figura 6.6 [CR06], é baseada na arquitetura cliente-servidor. O cliente possui os módulos de interface e o processador de especificações JavaMOP, enquanto o servidor contém o módulo de mensagens de comunicação entre cliente-servidor e o plugins de lógica (*logic-plugin*) para linguagem Java.

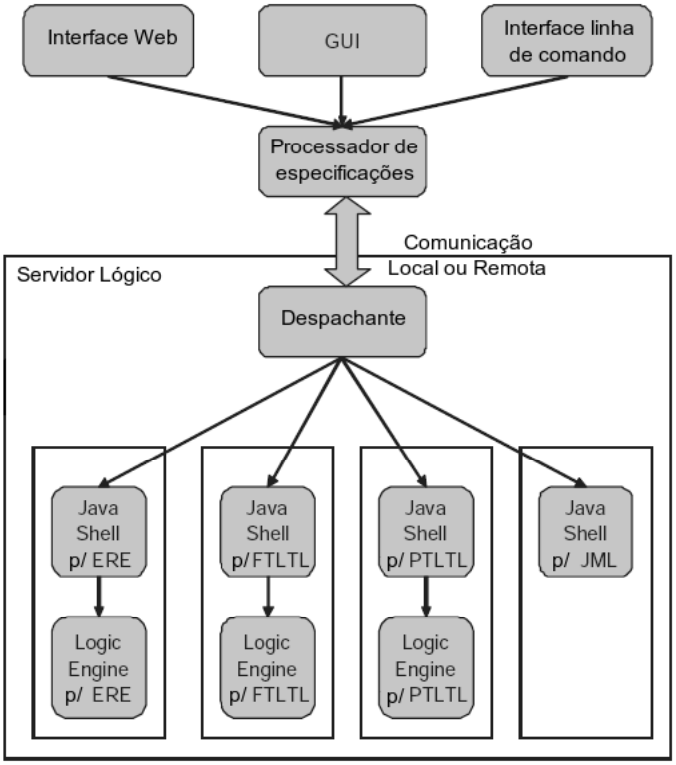

Figura 6.6: A arquitetura JavaMOP.

Três tipos de interfaces com o usuário são fornecidos pelo cliente:linha de comando, interface gráfica e web. JavaMOP oferece suporte para requisitos especificados em lógica temporal linear de tempo-passado e tempo-futuro (PTLTL e FTLTL, respectivamente), extensão de expressões regulares (ERE) e linguagem de modelagem Java (*Java Modeling Language* - JML[2]). A estrutura de especificação de MOP é sub-dividida em duas partes. Primeiramente, é especificado o que deve ser monitorado, declarando os predicados e os eventos que constroem o caminho de execução (*trace*), juntamente com algumas variáveis assistentes. Em seguida, tem-se a fórmula que especifica a propriedade de interesse (comportamento dese-

[2]Linguagem formal de especificação comportamental para Java.

jado), cuja sintaxe é específica para a lógica formal utilizada.

No Código 6.4, apresentamos um exemplo de especificação em MOP. Essa especificação caracteriza uma propriedade de segurança de um sinal de trânsito: *"sempre após o sinal verde é o sinal amarelo"*, que é expressa na linha 7 do Código 6.4 como uma fórmula de lógica temporal linear de tempo-futuro (FTLTL) [CDR04].

```
1  ... (Java code A) ...
2  /*@ FTLTL
3     Predicate vermelho : tlc.state.getColor() == 1;
4     Predicate verde : tlc.state.getColor() == 2;
5     Predicate amarelo : tlc.state.getColor() == 3;
6     // amarelo após o verde
7     Formula : [](verde -> (! vermelho U amarelo));
8     Violation handler : ... (Java "recovery" code) ...
9  @*/
10 ... (Java code B) ...
```

Código 6.4: Especificação JavaMOP em FTLTL.

Baseada nessa especificação, JavaMOP utiliza o plugin de lógica FTLTL para gerar o código de monitoração, conforme apresentado no Código 6.5.

```
1  ... (Java code A) ...
2  switch(estado) {
3  case 1:
4     estado = (tlc.state.getColor() == 3) ? 1 :
5       (tlc.state.getColor() == 2)
6        ? (tlc.state.getColor() == 1)
7        ? -2 : 2 : 1; break;
8  case 2:
9     estado = (tlc.state.getColor() == 3) ? 1 :
10      (tlc.state.getColor() == 1) ? -2 : 2; break ;
11 }
12 if (estado == -2) { ...(Violation Handler)... }
13 // Validation Handler is empty
14 ... (Java code B) ...
```

Código 6.5: Código de monitoração para especificação descrita no Código 6.4.

Os pontos de interesse no código são observados por meio do uso de AspectJ. O código aspecto captura informações sobre os pontos de interesse durante a execução dos sistemas de software e encaminha para o código de monitoração. A implementação do código de monitoração baseia-se na estrutura de dados chamada árvore de transição máquina - binária de estado finito (BTT-FSM), que são geradas a partir da especificação. BTT-FSM é uma

máquina de estado finito em que as transições são habilitadas para execução de uma série de condições organizadas como árvores de transição binária. A BTT-FSM para o Código 6.5 é apresentada na Figura 6.7.

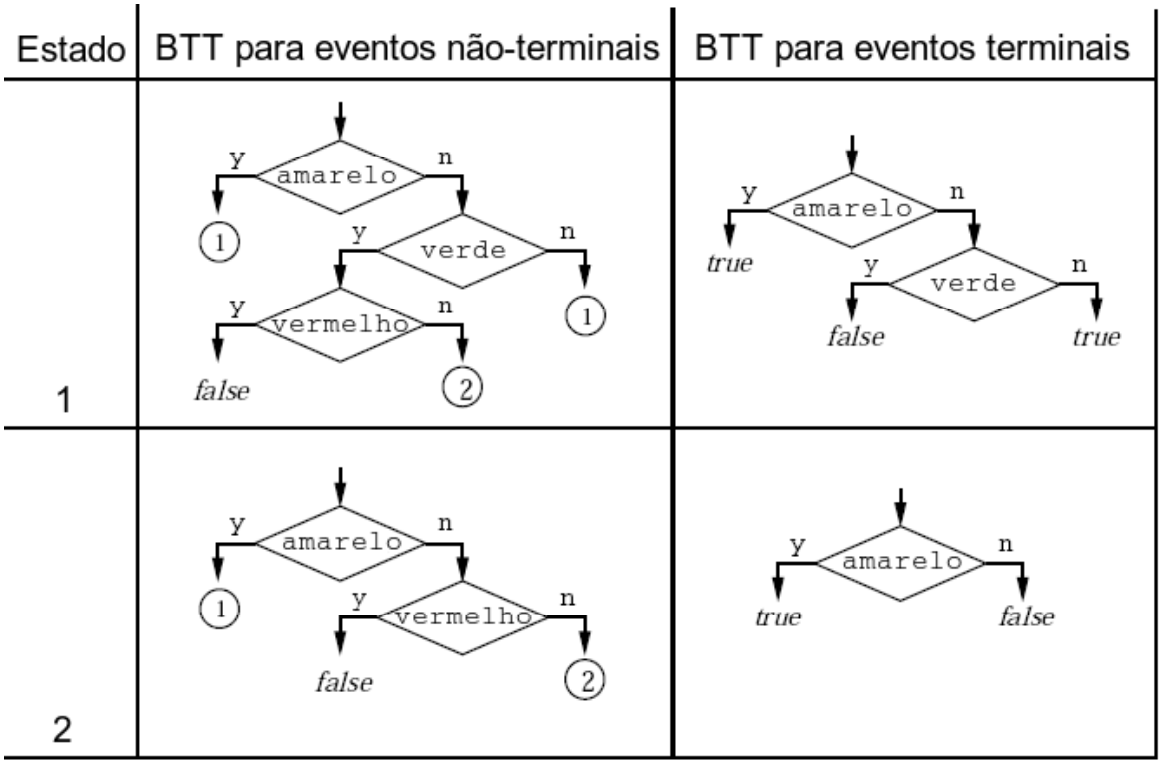

Figura 6.7: A BTT-FSM para a fórmula `[](green -> (!red U yellow))`.

JavaMOP permite expressar e verificar em tempo de execução se o comportamento dos sistemas está de acordo com o comportamento especificado. Contudo, para verificação de algumas propriedades de sistemas concorrentes essa técnica é insuficiente. A especificação e verificação dos requisitos em JavaMOP não permite descrever o comportamento das *threads* ao longo do tempo sob determinadas instâncias de objetos durante a execução dos sistemas de software.

6.7 Considerações finais

Como visto, muitos trabalhos têm sido propostos com o objetivo de monitorar o comportamento de programas Java à medida que estes são executados. Enquanto isso, a análise sobre o comportamento monitorado varia de acordo com o propósito alvo desses trabalhos.

Contudo, nenhum dos sistemas apresentados permitem, especificamente, a especificação e a análise do comportamento das *threads* durante a execução dos sistemas de software que possuem múltiplas *threads*. Com o uso da técnica *DesignMonitor* temos a possibilidade de especificar e verificar o comportamento das *threads* sob determinadas instâncias de objetos durante a execução dos sistemas de software automaticamente.

Capítulo 7

Considerações Finais

Neste trabalho de dissertação, apresentamos uma técnica automática para detecção de violações de propriedades referentes ao projeto de software. A idéia de seu desenvolvimento surgiu da necessidade de analisar se o código desenvolvido durante a execução do sistema obedece a certas propriedades, estruturais e comportamentais, do projeto de software.

A análise das propriedades estruturais pode ser realizada através da ferramenta *DesignWizard* (Apêndice A). A especificação das propriedades estruturais baseia-se em teste de unidade. Para isso, utilizamos o *framework* JUnit e a API *DesignWizard* para especificação das propriedades estruturais. Por fim, estes testes estruturais são executados da mesma maneira que os testes de unidades convencionais. No entanto, o seu uso não é viável para a detecção de violações das propriedades comportamentais, o que resultou no definição de uma nova técnica, denominada *DesignMonitor*.

Visando oferecer suporte ferramental a essa técnica foi desenvolvida o protótipo da ferramenta *DesignMonitor*. Na Figura 7.1 é ilustrada uma visão geral do funcionamento da ferramenta *DesignMonitor*. Essa ferramenta tem como entrada um conjunto de especificações do comportamento desejado para determinados pontos no código aonde esse comportamento é esperado. A partir dessas especificações são gerados os artefatos necessários para observar e analisar o comportamento do sistema de software alvo. O comportamento desejado é expresso através de uma fórmula LTL, que constituem os pilares para a geração do código para a realização da análise do comportamento observado. Estes comportamento observado é capturado pelo código aspecto (código de monitoração) nos pontos de interesse especificados. O código aspecto é adicionado junto ao código-fonte do sistema de software alvo.

Já o código analisador está presente no monitor. No entanto, é preciso definir o cenário de monitoração ao qual a ferramenta *DesignMonitor* deve realizar a detecção de violações da conformidade do comportamento do sistema. Definido este cenário, o sistema de software alvo é executado, durante a execução do sistema o código de monitoração captura o comportamento nos pontos de interesse que são encaminhados para o monitor, que paralelamente é realizada a verificação deste comportamento através do código analisador. Caso algum dos comportamentos observados violar alguma das propriedades comportamentais especificadas o usuário da ferramenta *DesignMonitor* é notificado.

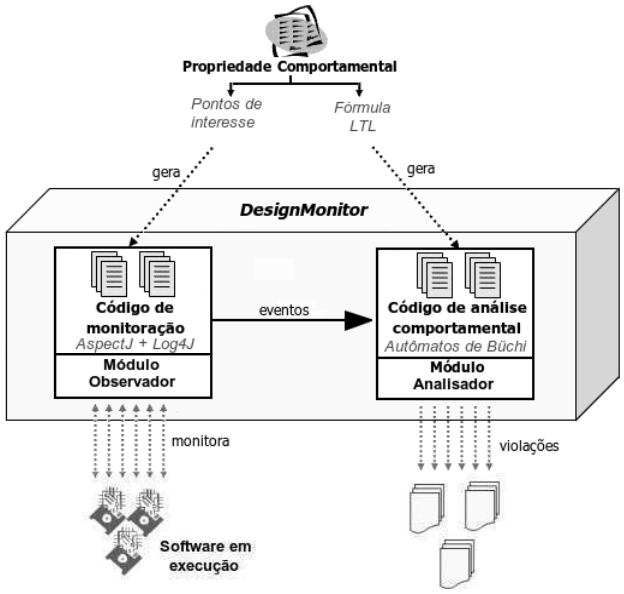

Figura 7.1: Visão geral da ferramenta *DesignMonitor*.

O desenvolvimento desse trabalho foi orientado a um estudo de caso junto ao projeto *OurGrid*. A idéia do uso dessa metodologia foi permitir a avaliação contínua da aplicabilidade da técnica proposta, bem como da funcionalidade da ferramenta *DesignMonitor* em um sistema real. O estudo de caso forneceu requisitos reais para a realização deste trabalho e os resultados do uso da técnica a partir da ferramenta *DesignMonitor* foram apresentados neste documento. Os resultados obtidos dão indícios reais da facilidade de aplicação da técnica e do uso da ferramenta.

7.1 Contribuições

Com a conclusão deste trabalho, podemos destacar algumas contribuições trazidas pelo mesmo. Primeiramente, apesar de muitos trabalhos abordarem diretamente o problema da divergência entre o código e os modelos abstratos [MNS95; SSC96], poucos oferecem um mecanismo automático para verificação da relação de conformidade entre o que é desenvolvido e o modelo idealizado. Este cenário torna-se ainda mais complexo para sistemas desenvolvidos que requerem o uso de paralelismo e concorrência. Sendo assim, este trabalho tem como contribuição a formulação de uma técnica, com suporte ferramental, que permite detectar violações automaticamente de propriedades comportamentais com enfoque em sistemas paralelos e concorrentes.

Uma outra contribuição relevante é no sentido de amenizar os problemas decorrentes da deterioração do software. Este é um problema que afeta a maioria, se não todos, os sistemas de software diante do surgimento de novos requisitos e restrições. *DesignMonitor* possibilita acompanhar a evolução de sistemas de software complexos, já que ao longo do desenvolvimento sua implementação tende a divergir do projeto de software esperado, tornando estes sistemas difíceis de entender, modificar e manter. O reflexo do processo de deterioração do software é a baixa qualidade do código, alta taxa de erros e o aumento dos custos do projeto [vGB02]. Assim, essa técnica prover meios automáticos que auxiliam na minimização da distância entre o que está sendo desenvolvido e o projeto de software previsto.

Por último, um dos nossos sentimentos é a diminuição da distância existente entre a definição da técnica e a sua real utilização. *DesignMonitor* visa permitir que os desenvolvedores a apliquem dentro do mesmo ambiente de desenvolvimento, fazendo uso de práticas comuns na academia e indústria para utilização de nossa técnica.

7.2 Trabalhos futuros

Com a finalização deste trabalho, algumas propostas para a sua continuidade podem ser destacadas:

- **Aumentar a abrangência da técnica** - a técnica permite verificar propriedades comportamentais referentes ao projeto de software de um determinado sistema. Contudo,

essa limita-se a considerar propriedades comportamentais passíveis de verificação de execuções locais, ou seja, em uma única máquina. Como trabalho futuro, propomos a incorporação de métodos que permitam verificar propriedades comportamentais durante a execução de sistemas distribuídos;

- **Facilitar a especificação das propriedades comportamentais** - definição e criação de uma API para especificação das propriedades comportamentais, para que os usuários dessa técnica não necessitem aprender um novo formalismo para especificar o comportamento desejado ;

- **Evolução da ferramenta *DesignMonitor*** - apesar de existir uma primeira versão da ferramenta *DesignMonitor*, algumas funcionalidades ainda necessitam ser aperfeiçoadas, a exemplo da geração dos código de monitoração e de verificação. Além disso, apesar da técnica não ser intrusiva no sentido de alterar o código-fonte do sistema de software alvo, a captura de informações durante a execução deste pelo código aspecto inclui um acréscimo de tempo total de execução do sistema;

- **Realização de outros estudos de caso** - mesmo com aplicação da técnica e o uso da ferramenta *DesignMonitor* a um estudo de caso real que nos forneceu uma visão prática, é necessário aplicarmos a um número maior de estudos de casos para adquirirmos uma maior confiança a cerca da viabilidade do uso desta técnica.

Bibliografia

[ABDM01] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Computer Society Press, Piscataway, NJ, USA, 2001.

[BFMW01] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[BKL] Eric Bruneton, Eugene Kuleshov, and Andrei Loskutov. Asm project. Web site. Disponível em: <http://asm.objectweb.org/>. Acessado em: 3 maio 2007.

[Büc62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Logic, Methodology and Philosophy of Science*, pages 1–12, Stanford, CA, USA, 1962. Stanford University Press.

[CBC+04] Walfredo Cirne, Francisco Brasileiro, Lauro Costa, Daniel Paranhos, Elizeu Santos-Neto, Nazareno Andrade, Cesar De Rose, Tiago Ferreto, Miranda Mowbray, Roque Scheer, and Joao Jornada. Scheduling in bag-of-task grids: The pauÁ case. In *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 124–131, Washington, DC, USA, 2004. IEEE Computer Society.

[CDR04] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. A formal monitoring-based framework for software development and analysis. In *ICFEM '04: Proceedings of the 6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 357–373, Seattle, WA, USA, 2004. Springer.

[CdR06] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. Checking and correcting behaviors of java programs at runtime with java-mop. *Electronic Notes in Theoretical Computer Science*, 144(4):3–20, 2006.

[Col87] James S. Collofello. Introduction to software verification and validation. Technical Report Curriculum Module SEI-CM-13-1.0, Software Engineering Institute, Carnegie Mellon University, October 1987.

[CR03] Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[CR05] Feng Chen and Grigore Roşu. Java-mop: A monitoring oriented programming environment for java. In *TACAS '05: Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems*, volume 3440 of *Lecture Notes in Computer Science*, Edinburgh, UK, 2005. Springer-Verlag.

[CR06] Feng Chen and Grigore Roşu. Mop: Reliable software development using abstract aspects. Technical Report UIUCDCS-R-2006-2776, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA, october 2006.

[CVW06] Rafael Cepêda, Aline Vasconcelos, and Claudia Werner. Tracer e phoenix: Ferramentas integradas para a engenharia reversa de modelos dinâmicos de java para uml. *XIII Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software*, pages 25–30, Outubro 2006.

[DGR04] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, 2004.

[EAK+01] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, 2001.

[FK99] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[FMP04] Jean-Claude. Fernandez, Laurent Mounier, and Cyril Pachon. Property oriented test case generation. In *FATES '03: Third International Workshop on Formal Approaches to Testing of Software*, volume 2931 of *Lecture Notes in Computer Science*, pages 147–163, Montreal, Quebec, CAN, 2004. Springer.

[Fra] JUnit Framework. Junit. Web site. Disponível em: <http://junit.sourceforge.net/>. Acessado em: 3 maio 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GLG03] Joseph D. Gradecki, Nicholas Lesiecki, and Joe Gradecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[GO01] P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *CAV '01: Proceedings of the 13th Conference on Computer Aided Verification*, number 2102 in Lecture Notes in Computer Science, pages 53–65. Springer-Verlag, 2001.

[GRG+05] Ann Q. Gates, Steve Roach, Irbis Gallegos, Omar Ochoa, and Oleg Sokolsky. Javamac and runtime monitoring for geoinformatics grid services. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 105–112, Washington, DC, USA, 2005. IEEE Computer Society.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[Kat99] J. P. Katoen. *Concepts, Algorithms, and Tools for Model Checking*. Lectures Notes of the Course Mechanised Validation of Parallel Systems. Friedrich-Alexander Universitat Erlangen-Nurnberg, 1999.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[Kin] Joe Kiniry. Esc/java2. Web site. Disponível em: <http://secure.ucd.ie/products/opensource/ESCJava2/>. Acessado em: 3 maio 2007.

[Kis02] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams, Indianapolis, IN, USA, 2002.

[KK03] M. D. Kaastra and C. J. Kapser. Toward a semantically complete java fact extractor. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, CAN, 2003.

[KKL+01] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-mac: a run-time assurance tool for java programs. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[KKL+02] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring - fundamentals of java-mac. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming*, volume 1241. Springer Berlin / Heidelberg, Jyväskylä, Finland, FIN, 1997.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[LBR06a] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LBR06b] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[Mey85] Bertrand Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, 1985.

[Mey86] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[MG06] João Arthur Brunet Monteiro and Dalton Dario Serey Guerrero. Extração e verificação automática de modelos de software. In *III Congresso de Iniciação Científica da Universidade Federal de Campina Grande*, Campina Grande, PB, BRA, Outubro 2006. UFCG.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20(4):18–28, 1995.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, USA, 1977. IEEE Computer Society Press.

[Proa] Apache Ant Project. Apache ant. Web site. Disponível em: <http://ant.apache.org/>. Acessado em: 3 maio 2007.

[Prob] AspectJ Project. The aspectj project at eclipse.org. Web site. Disponível em: <http://www.eclipse.org/aspectj/>. Acessado em: 3 maio 2007.

[Proc] Eclipse Project. Eclipse. Web site. Disponível em: <http://www.eclipse.org/>. Acessado em: 3 maio 2007.

[Prod] Log4j Project. Log4j. Web site. Disponível em: <http://logging.apache.org/log4j/>. Acessado em: 3 maio 2007.

[Proe] OurGrid Project. Ourgrid. Web site. Disponível em: <http://www.ourgrid.org/>. Acessado em: 3 maio 2007.

[RKW+06] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. pages 43–56, May 2006.

[SB02] Sérgio Soares and Paulo Borba. Aspectj - programação orientada a aspectos em java. In *VI Simpósio Brasileiro de Linguagens de Programação*, Rio de Janeiro, RJ, BRA, 2002. SBC.

[Som04] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[SSC96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. pages 387–396, 1996.

[Tan92] Andrew S. Tanenbaum. *Modern operating systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Too] LTL2BA4J Tool. Ltl2ba4j. Web site. Disponível em: <http://www-i2.informatik.rwth-aachen.de/Research/RV/ltl2ba4j/>. Acessado em: 3 maio 2007.

[vGB02] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems & Software*, 61(2):105–119, 2002.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM Press.

Apêndice A

Verificação Estrutural do Código

A.1 Especificação estrutural

As propriedades estruturais compõem os modelos estáticos do sistema de software. Para expressar as propriedades estrututurais de um sistema de software é necessário definir como as informações a respeito da estrutura do código desenvolvido estão organizadas. As informações sobre a estrutura do código é denominada de **fatos**. Os fatos se referem a todos os relacionamentos entre as entidades que constituem o código, onde um fato possui o seguinte formato:

```
Fato = <Entidade Chamador> <Relação> <Entidade Chamada>
```

Considerando sistemas de software desenvolvidos na linguagem Java, temos que os parâmetros que compõem um fato são os seguintes:

- **entidade** - classes, métodos e atributos;
- **relação** - todas as possíveis conexões entre duas entidades, por exemplo, uma classe `A` implementa a interface `B` (`A implements B`) ou o método a_1 da classe `A` é chamado pelo método b_3 da classe `B` (a_1 `invokeVirtual` b_3).

Contudo, existem restrições entre quais os tipos de entidades podem ocorrer determinadas relações. Na Tabela A.1 é apresentada para cada tipo de relação quais as possíveis combinações entre os tipos de entidade e o significado de cada relações.

Tabela A.1: Exemplos de tipos de relações entre entidades.

Tipo de Relação	**Significado**
`A instanceOf B`	relação entre atributo e classe, onde um atributo `A` é intância da classe `B`
`A contains B`	relação entre classe e atributo ou classe e método, em que a classe `A` contém um atributo `B` ou um método `B`
`A extends B`	relação entre duas classes, a classe `A` herda caracteríticas da classe `B`
`A implements B`	relação entre classe e interface (classe), a classe `A` implementa todos os métodos da interface `B`
`A is_accessedby B`	relação entre atributo e método, ocorre quando o atributo `A` é acessado ou modificado pelo método `B`, podendo esse tipo de relação classificada como: `getStatic`, `putStatic`, `getField` e `putField`, descrita a seguir
`A getStatic B`	relação entre método e atributo estático, o método `A` acessa o atributo estático `B`
`A putStatic B`	relação entre método e atributo estático, o método `A` modifica o atributo estático `B`
`A getField B`	relação entre método e atributo, o método `A` acessa o atributo `B`
`A putField B`	relação entre método e atributo, o método `A` modifica o atributo `B`
`A is_invokedby B`	relação entre método e método, ocorre quando o método `A` é invocado pelo método `B`, esse tipo de relação pode ser classificada como: `invokeVirtual`, `invokeSpecial`, `invokeStatic` e `invokeInterface`, que são descritas a seguir
`A invokeVirtual B`	relação entre método e método, ocorre quando o método `A` de uma classe faz chamada ao método `B` de outra classe
`A invokeSpecial B`	relação entre método e método, quando o método `A` chama o método de inicialização (construtor) ou uma super-classe ou um método privado
`A invokeStatic B`	relação entre método e método, ocorre quando um método `A` chama o método estático `B`
`A invokeInterface B`	relação entre método e método, ocorre quando o método `A` chama outro método da interface `B`
`A catch B`	relação entre método e classe, ocorre quando o método `A` possui em seu corpo um *catch* da execeção representada pela classe `B`

A estrutura de um sistema de software pode ser vista como um conjunto de fatos. Estes fatos são organizados em sub-conjuntos, conforme mostra a Figura A.1. Para cada entidade que compõem o sistema existe um conjunto de todos os seus fatos. Este fatos são organizados em outros sub-conjuntos a partir do tipo de relação de cada fato. A idéia é que facilmente seja possível expressar as propriedades estruturais a partir de operações sob estes conjuntos de fatos. As operações básicas sob conjuntos são as de união, diferença e intersecção.

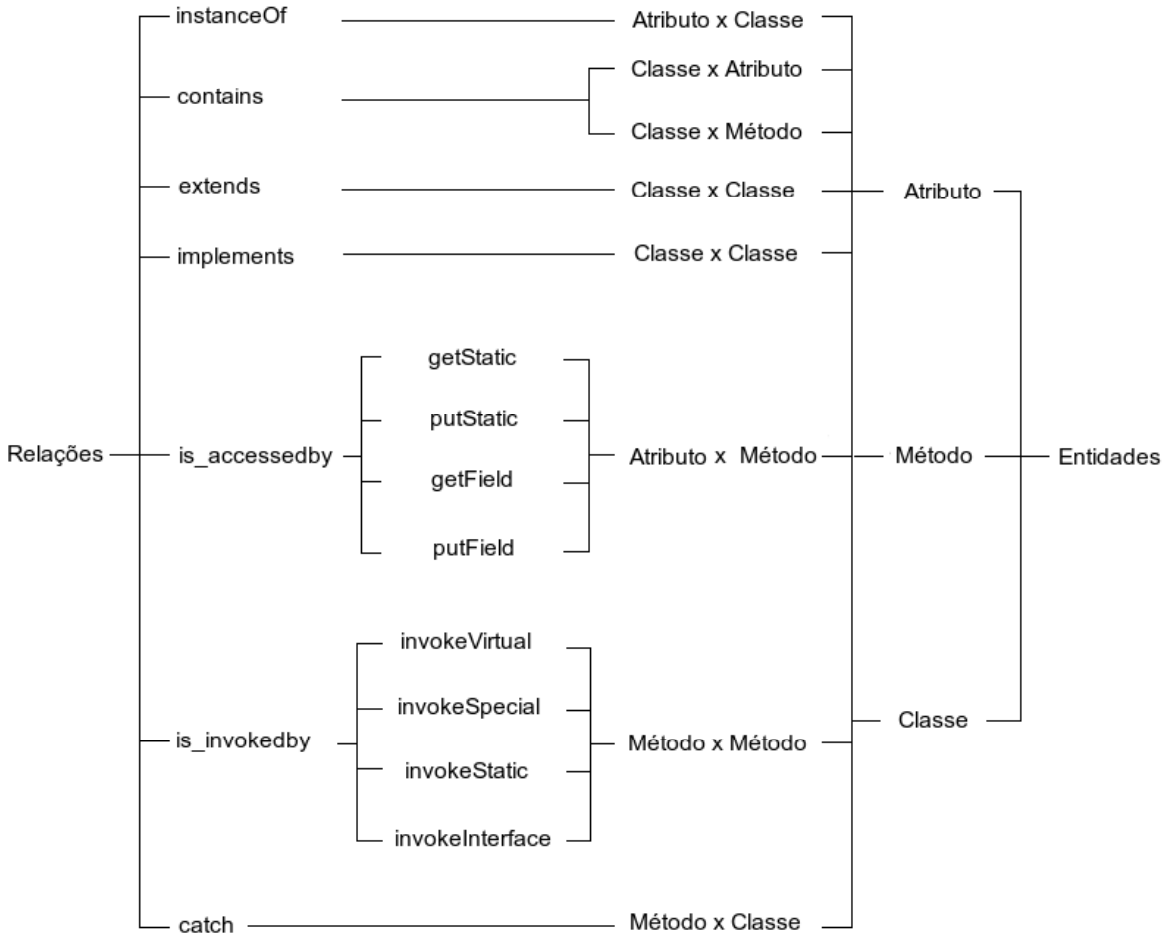

Figura A.1: Representação da organização do conjunto de fatos de um sistema de software.

Exemplo A.1 *Seja o sistema de software S, temos que este deve obedecer a seguinte propriedade estrutural:*

a classe `A` *dever ser acessada apenas pela classe* `B`*.*

Para expressar essa propriedade estrutural é necessária algumas operações sob o conjunto de fatos coletados do sistema de software `S`*. Assim temos a seguinte sequência de*

operações:

1. $methods_A$ *- inicialmente obtemos o conjunto de todos os métodos que pertencem a classe* `A`*, ou seja, o conjunto resultante da união de todos o fatos da relação do tipo* `A contains` $*$ *(onde* $*$ *é qualquer entidade do tipo método);*

2. $methods_B$ *- semelhante a operação anterior obtemos o conjunto de todos os métodos que pertencem a classe* `B`*, ou seja, o conjunto resultante da união de todos o fatos da relação do tipo* `B contains` $*$ *(onde* $*$ *é qualquer entidade do tipo método);*

3. $invoke_{a_i}$ *- para cada* $a_i \in methods_A$*, onde* a_i *é uma entidade do tipo método, temos que* $invoke_{a_i}$ *seja o conjunto resultante da união de todos os métodos que fazem chamadas ao método* a_i *(*`*` `is_invokedby` a_i*, onde* $*$ *é qualquer entidade do tipo método);*

4. $invoke_{methods_A}$ *- a união de todos os conjuntos de chamadas externas aos métodos da classe* `A` *resulta no conjunto* $invoke_{methods_A}$ *(*$invoke_{a_1} \cup invoke_{a_2} \cup ... \cup invoke_{a_i} \cup ... \cup invoke_{a_n}$*);*

5. *Logo, temos que a propriedade estrutural não está sendo violada no código se* $invoke_{methods_A} - methods_B = \oslash$ *for verdadeira.*

Com base nas possíveis operações entre os conjuntos de fatos do sistema, organizados conforme a Figura A.1, definimos algumas das principais operações apresentadas a seguir:

- ***methodIsCalledBy(called:String, caller:String) : Boolean*** - informa se um método (*called*) é invocado por outro método (*caller*);

- ***getCallers(method:String) : Set<String>*** - retorna todas as entidades que invocam um determinado método;

- ***getCalled(method:String) : Set<String>*** - retorna todas as entidades que são invocadas por um método;

- ***getMethods(className:String) : Set<String>*** - retorna o conjunto de métodos de uma determinada classe (*className*);

- ***getStaticMethods(className:String) : Set<String>*** - retorna o conjunto de métodos estáticos de uma determinada classe (*className*);
- ***getAllClasses() : Set<String>*** - retorna o conjunto das classes do sistema;
- ***getSubClasses(className:String) : Set<String>*** - retorna o conjunto de subclasses de uma determinada classe (*className*);
- ***getSuperClass(className:String) : String*** - retorna a super-classe de uma determinada classe (*className*) do sistema;

Tabela A.2: Conjunto de asserções para especificação de propriedades estruturais.

Asserção	**Descrição**
`assertTrue(condição)`	afirma que a condição é verdadeira.
`assertFalse(condição)`	afirma que a condição é falsa.
`assertEquals(condição, condição)`	afirma se dois conjuntos são iguais.
`assertNotNull(condição)`	afirma que a condição não é nula.
`assertNull(condição)`	afirma que a condição é nula.

A partir dessas operações, temos que as propriedades estruturais podem ser descritas como asserções. A sintaxe e a semântica do conjunto de asserções, apresentadas na Tabela A.2, são semelhantes a utilizada pelo *framework JUnit* [Fra]. Assim, a propriedade descrita no Exemplo A.1 pode ser especificada da seguinte maneira:

```
classA = getClass("A");
Set callers = classA.getClassesThatUses();
assertEquals(1,callers.size());
assertTrue(callers.contains("B"));
```

A.2 Análise estática do código

A análise das propriedades estruturais é realizada através da análise estática da conformidade de código por meio da extração de um modelo de projeto a partir do código fonte, que posteriormente é confrontado com o modelo idealizado.

As informações necessárias para a análise das propriedades estruturais podem ser determinadas estaticamente, ou seja, em tempo de compilação. Para verificar se a implementação do sistema de software condiz com as propriedades estruturais especificadas utilizamos a ferramenta *DesignWizard* [MG06]. *DesignWizard* é uma ferramenta de extração direta de modelos de software a partir do código fonte e que faz uso dessas informações extraídas para verificar propriedades estruturais. Essa ferramenta é oriunda de um projeto de iniciação científica desenvolvido no Grupo de Métodos Formais (GMF) da Universidade Federal de Campina Grande (UFCG).

A arquitetura da ferramenta *DesignWizard* é composta, basicamente, pelos seguintes módulos:

- **módulo extrator** - módulo responsável por extrair as informações estruturais a partir do código e/ou *bytecode*[1] de sistemas de software escritos em Java;
- **módulo tradutor** - o objetivo desse módulo é manter um padrão das informações extraídas do sistema de software;
- **módulo verificador** - esse módulo realiza a análise das propriedades estruturais, especificadas a partir de uma API[2], com as informações obtidas pelo extrator;

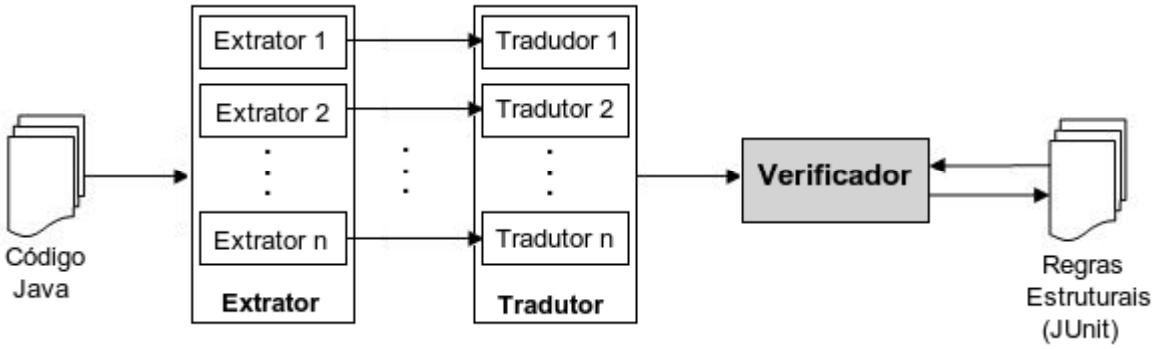

Figura A.2: Arquitetura da ferramenta *DesignWizard*.

[1] É um formato de código intermediário entre o código fonte - o texto que o programador consegue manipular - e o código de máquina - que o computador consegue executar.

[2] API (*Application Programming Interface*) é um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades por programas aplicativos, isto é, programas que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços

A Figura A.2 ilustra a arquitetura da ferramenta *DesignWizard*. A partir do código fonte e/ou o arquivo .jar o módulo extrator obtêm e armazena os fatos que constituem o sistema de software alvo, de modo que possamos interpretar e manipular essas informações. O modo de extração dos fatos utilizado é a partir do *bytecode* gerado ao compilar o código fonte. Sua escolha deve-se pela grande quantidade de informações relevantes obtidas e por haver ferramentas que fazem a extração de maneira bastante eficiente. O *DesignWizard* é, atualmente, formado por dois extratores:

- **ASM** [BKL] - *framework*[3] para manipulação de *bytecode*. Ele permite extrair fatos do *bytecode* gerado em tempo de compilação de um código Java e fornecer esse conjunto de fatos de forma textual. É extremamente eficiente com relação a outros extratores, como por exemplo o *Java Fact Extractor* [KK03], e compatível com o *bytecode* gerado a partir de código Java na versão 1.5;

- **Reflexão** - extrator próprio desenvolvido a partir da API de reflexão de Java. Reflexão permite que um programa Java examine ou faça introspecção em suas propriedades e na sua estrutura. Com isso podemos, por exemplo, obter o nome de todos os membros de uma classe (como atributos e métodos) ou executar um método utilizando introspecção (*Introspection*).

As informações extraídas são então armazenadas seguindo um padrão através do módulo tradutor. Com isso, podemos inserir diversos extratores, cada qual com suas particularidades, fazendo a comunicação entre o módulo responsável pela extração dos fatos e o módulo de verificação. Desta maneira, as informações extraídas podem ser interpretadas e manipuladas pelos demais módulos da ferramenta *DesignWizard*.

As propriedades estruturais são especificadas através de testes estruturais usando o *framework* JUnit. Assim, podemos escrever testes estruturais de maneira muito semelhante a testes de unidade (Exemplo A.2). O objetivo é permitir uma fácil adaptação ao uso da ferramenta *DesignWizard*, uma vez que o usuário não necessita aprender uma nova técnica para especificar testes estruturais, visto que, compor um teste de unidade é uma prática bastante difundida e utilizada pela comunidade científica.

[3] É uma estrutura de suporte em que um outro projeto de software pode ser organizado e desenvolvido.

Exemplo A.2 *Considere o Exemplo A.1, utilizando a ferramenta DesignWizard a especificação desta propriedade estrutural fica conforme o Código A.1.*

```
1  package example.test;
2  import java.io.IOException;
3  import java.util.*;
4  import junit.framework.TestCase;
5  import designWizard.exception.InexistentEntityException;
6  import designWizard.main.DesignWizard;
7  public class AssertionsTest extends TestCase {
8       private DesignWizard dw;
9       public void testRule01() throws IOException,
10                          InexistentEntityException {
11           dw = new DesignWizard("myproject.jar");
12           Set<String> methods = dw.getMethods("A");
13           Iterator<String> it = methods.iterator();
14           while(it.hasNext()) {
15                assertTrue(dw.methodIsOnlyCalledFacade(it.next(), "B"));
16           }
17      }
18 }
```

Código A.1: Regra estrutural do Exemplo A.1 utilizando a ferramenta *DesignWizard*.

A extração dos fatos ocorre quando é criada uma nova instância da classe `DesignWizard` *(linha 12). Com a extração dos fatos, a ferramenta* `DesignWizard` *possui todas as informações referentes a estrutura do código que podem ser manipuladas fazendo uso dos métodos fornecidos pela API DesignWizard (linha 16). O resultado da execução dos testes estruturais são expressos da mesma maneira que o JUnit, ou seja, barra verde indica que os testes foram bem sucedidos e barra vermelha informa que algum teste falhou.*

Apêndice B

Verificação Estrutural do *OurGrid*

B.1 Propriedades estruturais

Os módulos *Scheduler* e *ReplicaExecutor* fazem uso do padrão de projeto *Facade* [GHJV95]. As classes `EBSchedulerFacade` e `EBReplicaExecutorFacade` são as fachadas dos módulos *Scheduler* e *ReplicaExecutor*, respectivamente. A comunicação entre os módulos só deve ocorrer entre a fachada de ambos. Estes módulos possuem objetos que não podem sair de um módulo para outro, sendo caracterizados pelas seguintes propriedades:

1. `EBGridManager` só pode ser acessadas pela fachada `EBSchedulerFacade`;
2. `EBJobManager` só pode ser acessadas pela fachada `EBSchedulerFacade`;
3. `EBReplicaManager` só pode ser acessada pela fachada `EBReplicaExecutorFacade`.

Para verificar se essas propriedades estruturais estão sendo obedecidas pelo código desenvolvido utilizamos a ferramenta *DesignWizard*. Adicionamos ao projeto *OurGrid* o arquivo `designwizard.jar`. Como as propriedades estruturais são especificadas através de testes de unidade a partir do *framework* JUnit é necessário adicionar também o arquivo `junit.jar`. Em seguida, com base na decrições das propriedades escreva os testes estruturais e os execute de maneira similar aos testes de unidade.

No Código B.1, podemos ver a especificação dessas propriedades como testes estruturais. Nas linhas 12, 23 e 34 vemos as especificações das propriedades estruturais 1, 2 e 3,

```
1  package org.ourgrid.test.structuraltests;
2
2  import java.io.IOException;
3  import java.util.Set;
4
4  import junit.framework.TestCase;
5  import designwizard.exception.InexistentEntityException;
6  import designwizard.main.DesignWizard;
7
7  public class MygridModuleRulesTest extends TestCase {
8    private DesignWizard dw;
9
9    protected void setUp() throws IOException {
10     this.dw = new DesignWizard("lib/ourgrid.jar");
11   }
12
12   public void testMyGridRules1() throws InexistentEntityException {
13     designwizard.design.entity.ui.Class
14       clazz = dw.getClass(
15               "org/ourgrid/mygrid/scheduler/gridmanager/EBGridManager");
16     Set<String> uses = clazz.getClassesThatUse();
17     assertTrue(uses.contains(
18       "org/ourgrid/mygrid/scheduler/gridmanager/EBGridManager"));
19     assertTrue(uses.contains(
20       "org/ourgrid/mygrid/scheduler/EBSchedulerFacade"));
21     assertTrue(uses.size() == 2);
22   }
23
23   public void testMyGridRules2() throws InexistentEntityException {
24     designwizard.design.entity.ui.Class
25       clazz = dw.getClass(
26         "org/ourgrid/mygrid/scheduler/jobmanager/EBJobManager");
27     Set<String> uses = clazz.getClassesThatUse();
28     assertTrue(uses.contains(
29       "org/ourgrid/mygrid/scheduler/jobmanager/EBJobManager"));
30     assertTrue(uses.contains(
31       "org/ourgrid/mygrid/scheduler/EBSchedulerFacade"));
32     assertTrue(uses.size() == 2);
33   }
34
34   public void testMyGridRules3() throws InexistentEntityException {
35     designwizard.design.entity.ui.Class
36       clazz = dw.getClass(
37         "org/ourgrid/mygrid/replicaexecutor/EBReplicaManager");
38     Set<String> uses = clazz.getClassesThatUse();
39     assertTrue(uses.contains(
40       "org/ourgrid/mygrid/replicaexecutor/EBReplicaManager"));
41     assertTrue(uses.contains(
42       "org/ourgrid/mygrid/replicaexecutor/EBReplicaExecutorFacade"));
43     assertTrue(uses.size() == 2);
44   }
45
45 }
```

Código B.1: Propriedades estruturais do componente *MyGrid*.

respectivamente.

O ambiente para o uso da ferramenta *DesignWizard* é o mesmo utilizado pelos desenvolvedores, no nosso caso o Eclipse. Na Figura B.1, vemos a tela de execução dos testes estruturais gerados para as propriedades especificadas no Código B.1.

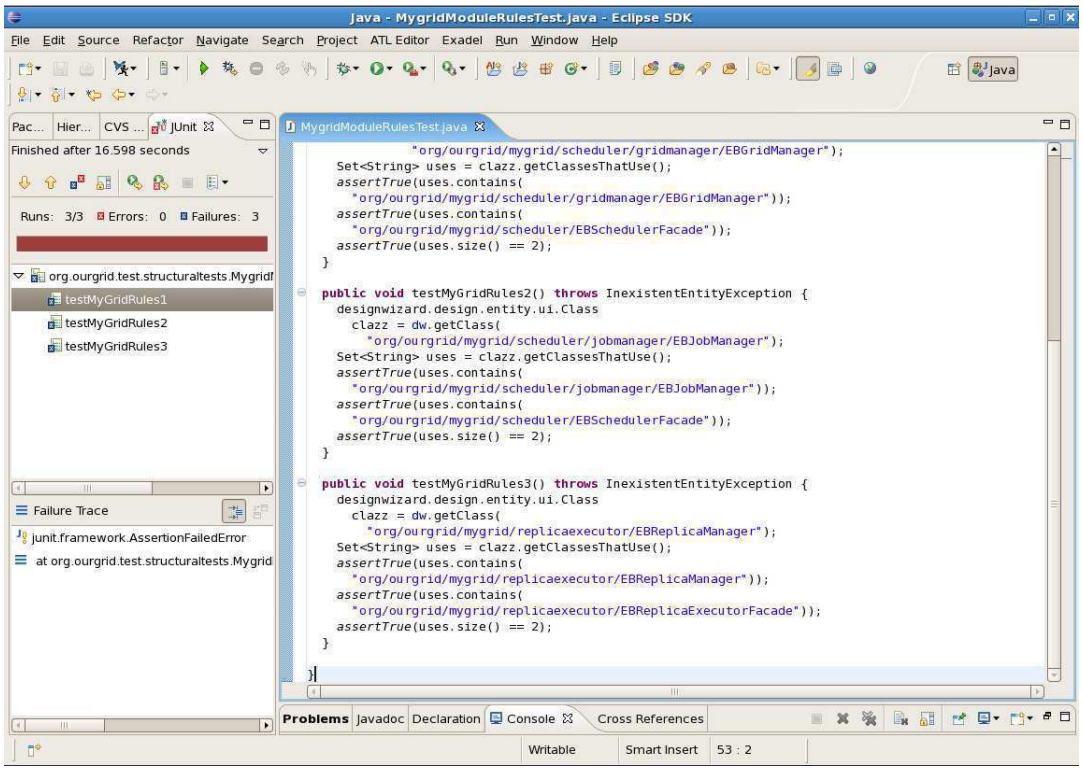

Figura B.1: Execução dos testes estruturais do *MyGrid*: violação de propriedades.

Nesta execução, testamos uma implementação na qual uma outra classe diferente da `EBSchedulerFacade` faz chamada a classe `EBJobManager`. Como houve pelo menos uma violação de propriedades estruturas na execução dos testes, a barra de progresso assumiu a cor vermelha ao final da execução. Neste caso devido as seguintes situações:

1. `EBGridManager` só pode ser acessadas pela fachada `EBSchedulerFacade`, através da classe `SchedulerEventEngine`;

2. `EBJobManager` só pode ser acessadas pela fachada `EBSchedulerFacade`, através da classe `SchedulerEventEngine`;

3. `EBReplicaManager` só pode ser acessada pela fachada `EBReplicaExecutorFacade`.

Além disso, existem as classes de testes que também fazem uso dessas classes, o que altera o número de classes que as usam. Após a alteração das chamadas inválidas temos como resultado o Código B.2. Os testes estruturais foram executados mais uma vez e, como pode ser visto na Figura B.2, a barra de progresso assumiu a cor verde em sinal da ausência de violação de propriedades.

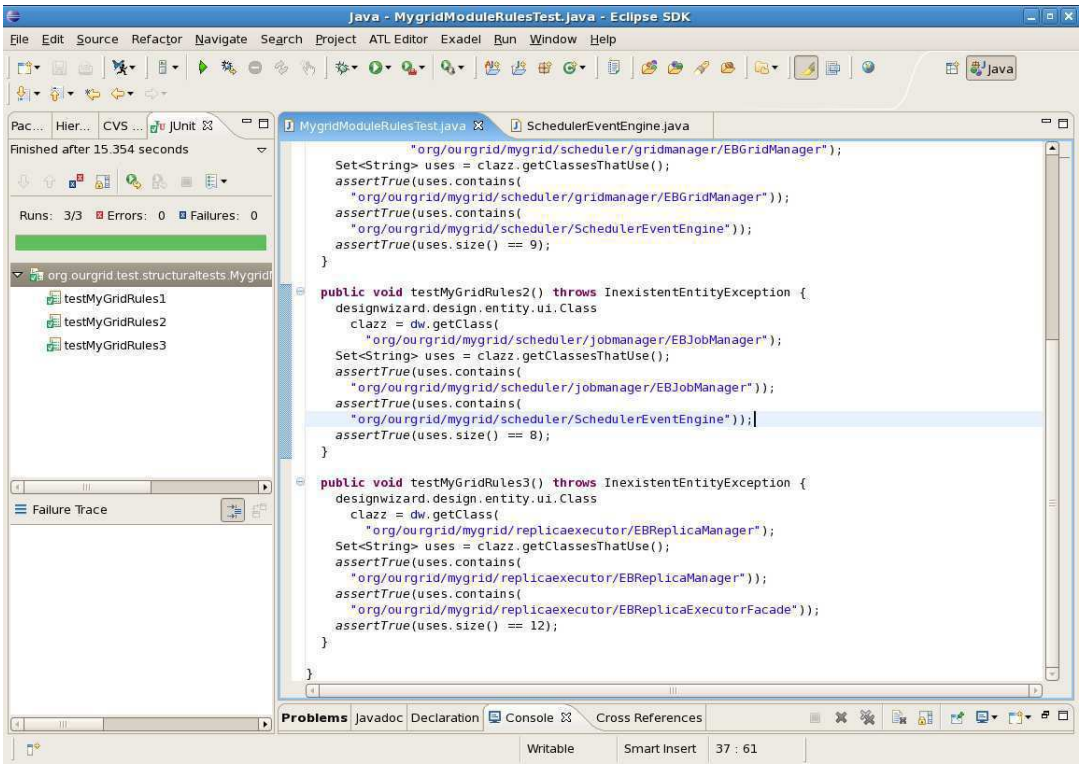

Figura B.2: Execução dos testes estruturais do *MyGrid*: propriedades ok.

```
1  package org.ourgrid.test.structuraltests;
2
2  import java.io.IOException;
3  import java.util.Set;
4
4  import junit.framework.TestCase;
5  import designwizard.exception.InexistentEntityException;
6  import designwizard.main.DesignWizard;
7
7  public class MygridModuleRulesTest extends TestCase {
8    private DesignWizard dw;
9
9    protected void setUp() throws IOException {
10     this.dw = new DesignWizard("lib/ourgrid.jar");
11   }
12
12   public void testMyGridRules1() throws InexistentEntityException {
13     designwizard.design.entity.ui.Class
14       clazz = dw.getClass(
15               "org/ourgrid/mygrid/scheduler/gridmanager/EBGridManager");
16     Set<String> uses = clazz.getClassesThatUse();
17     assertTrue(uses.contains(
18       "org/ourgrid/mygrid/scheduler/gridmanager/EBGridManager"));
19     assertTrue(uses.contains(
20       "org/ourgrid/mygrid/scheduler/SchedulerEventEngine"));
21     assertTrue(uses.size() == 9);
22   }
23
23   public void testMyGridRules2() throws InexistentEntityException {
24     designwizard.design.entity.ui.Class
25       clazz = dw.getClass(
26         "org/ourgrid/mygrid/scheduler/jobmanager/EBJobManager");
27     Set<String> uses = clazz.getClassesThatUse();
28     assertTrue(uses.contains(
29       "org/ourgrid/mygrid/scheduler/jobmanager/EBJobManager"));
30     assertTrue(uses.contains(
31       "org/ourgrid/mygrid/scheduler/SchedulerEventEngine"));
32     assertTrue(uses.size() == 8);
33   }
34
34   public void testMyGridRules3() throws InexistentEntityException {
35     designwizard.design.entity.ui.Class
36       clazz = dw.getClass(
37         "org/ourgrid/mygrid/replicaexecutor/EBReplicaManager");
38     Set<String> uses = clazz.getClassesThatUse();
39     assertTrue(uses.contains(
40       "org/ourgrid/mygrid/replicaexecutor/EBReplicaManager"));
41     assertTrue(uses.contains(
42       "org/ourgrid/mygrid/replicaexecutor/EBReplicaExecutorFacade"));
43     assertTrue(uses.size() == 12);
44   }
45
45 }
```

Código B.2: Alteração das propriedades estruturais do componente *MyGrid*.