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

Uma Técnica para Verificar Não-conformidades em Programas Especificados com Contratos

Catuxe Varjão de Santana Oliveira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I 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

Tiago Massoni e Rohit Gheyi
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

O48u Oliveira, Catuxe Varjão de Santana.

Uma técnica para verificar não-conformidades em programas especificados com contratos / Catuxe Varjão de Santana Oliveira. – Campina Grande, 2013.

68 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2013.

"Orientação: Prof. Dr. Tiago Lima Massoni, Prof. Dr. Rohit Gheyi".
Referências.

1. Engenharia de Software - Contratos. 2. Design by Contract. 3. JML. 4. Testes I. Massoni, Tiago Lima. II. Gheyi, Rohit. III. Título.

CDU 004.41(043)

"UMA TÉCNICA PARA VERIFICAR NÃO-CONFORMIDADES EM PROGRAMAS ESPECIFICADOS COM CONTRATOS"

CATUXE VARJAO DE SANTANA OLIVEIRA

DISSERTAÇÃO APROVADA EM 15/03/2013

Tiago L. Massoni

TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

Rohit Gheyi

ROHIT GHEYI, Dr., UFCG
Orientador(a)

Adalberto Cajueiro de Farias

ADALBERTO CAJUEIRO DE FARIAS, Dr., UFCG
Examinador(a)

MÁRCIO LOPES CORNÉLIO, Dr., UFPE
Examinador(a)

CAMPINA GRANDE - PB

Recife, 26 de abril de 2013

À: **Coordenação de Pós-Graduação em Ciência da Computação - COPIN**
Universidade Federal de Campina Grande – UFCG

De : **Márcio Lopes Cornélio** (mlc2@cin.ufpe.br)

Assunto: **Parecer sobre a defesa de dissertação de Mestrado de Catuxe Varjão de Santana Oliveira**

Prezados senhores,

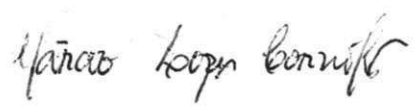
Participei no dia 27 de março de 2013, às 10h, da defesa de dissertação de mestrado de Catuxe Varjão de Santana Oliveira, intitulada "Uma Abordagem para Verificar Não-conformidades em Programas Especificados com Contratos". Minha participação foi remota, utilizando áudio e vídeo via Skype. Sugeri uma mudança no título da dissertação: a palavra "Abordagem" seria substituída por "Técnica".

Considero que a dissertação está bem estruturada no que diz respeito à disposição de capítulos. Um aspecto positivo da dissertação é o ganho na qualidade das especificações a partir do uso da técnica proposta. Sugeri uma discussão mais ampla sobre a avaliação de concretude das especificações, uma vez que as especificações escritas na linguagem JML são, em muitas casos da API da linguagem, extensas em relação ao código especificado. Além desta discussão, sugeri um aprofundamento sobre a questão de encadeamento de chamadas de métodos. Outras sugestões e discussões mais detalhadas foram passadas para a mestranda por meio de anotações no texto da dissertação.

Meu posicionamento foi a favor da aprovação da dissertação de mestrado de Catuxe Varjão de Santana Oliveira.

Sem mais para o momento, coloco-me à disposição para quaisquer esclarecimentos.

Atenciosamente,



Márcio Lopes Cornélio

Professor (CIn-UFPE)

Resumo

A escrita de especificações formais por contratos é uma maneira confiável e prática de construir softwares, em que desenvolvedores e clientes mantêm um acordo contendo direitos e obrigações a serem cumpridos. Essas responsabilidades são expressas basicamente através de pré-condições, pós-condições, e invariantes. Como exemplo de linguagem de especificação por contrato tem-se Java Modeling Language (JML) específica para programas Java. Apesar de a especificação formal melhorar a confiabilidade do software, deve-se haver certificação de que a implementação está em conformidade com a especificação definida. Verificação de conformidade em programas com contratos é geralmente realizada através de análises manuais ou verificação dinâmica, e em fases tardias do processo de desenvolvimento do software, ou seja, quando o produto final encontra-se disponível para o cliente. Nesta situação, o tempo despendido para detectar não-conformidades pode ser muito longo, ocasionando, consequentemente, atrasos no cronograma e aumento nos custos. Neste trabalho, propomos uma abordagem para checar conformidade entre código fonte e especificação formal por contratos através da geração e execução de testes. Testes de unidade são gerados automaticamente, resultando em casos de testes com sequências de chamadas aos métodos e construtores. Os contratos são transformados em assertivas que funcionam como oráculo para os testes. Esta abordagem não garante corretude total do software, mas aumenta a confiança quando uma não-conformidade é encontrada e, além disso, encoraja o uso de especificação por contratos. Nós implementamos JMLOK, uma ferramenta que executa os passos desta abordagem automaticamente no contexto de programas Java especificados com Java Modeling Language (JML). JMLOK foi avaliada em grupos de programas Java/JML, incluindo um módulo do projeto JavaCard. Todas as unidades experimentais totalizam 18 KLOC e 5K de linhas de especificação JML. Todo o processo consumiu menos que 10 minutos de execução e gerou como resultado a detecção de 29 não-conformidades. As causas das ocorrências das não-conformidades foram analisadas manualmente e classificadas em categorias de falhas.

Abstract

Writing formal specifications by contracts is a practical and reliable way to build softwares in which developers and clients keep an agreement with rights and obligations to be fulfilled. These responsibilities are expressed basically by pre-conditions, post-conditions and invariants. As example of specification language by contract there is Java Modeling Language (JML) that is specific to Java programs. Although formal specification improves software reliability, it should exist certification of conformance with defined specification. Verify conformance between programs and contracts is usually performed by manual analysis or dynamic verification, and in late stages of software development process, that is, when the final product is available to client. In this situation, the time required to detect nonconformances could be so long, causing, consequently, schedule delays and increased costs. In this work, we propose an approach to check conformance between source code and contract formal specification through testing generation and execution. Unit tests are generated automatically resulting in test cases with call sequences of methods and constructors. The contracts are translated in assertions that work like test oracle. We have implemented JMLOK, a tool performs the approach steps automatically in the context of Java programs specified with Java Modeling Language (JML). JMLOK was evaluated in Java/JML programs groups, including a module of the JavaCard project. All the experimental units totalize 18 KLOC and 5K lines of JML specification. All process took less than 10 minutes of running and generated as result 29 nonconformances. The causes of nonconformances occurring were analyzed manually and classified in categories of fails.

Agradecimentos

Acredito piamente na teoria que o mundo é habitado por anjos! Eles estão no mundo real por uma invocação divina e não aparecem em sua vida por acaso. Mas sim para fazer diferença, te ensinar algo, te mostrar caminhos, te dar carinho e te chamar atenção sempre que necessário. Eles existem mesmo que seja para realizar coisas pequenas, mas não menos grandiosas, como te desejar um "bom dia", te perguntar "como vai?", te oferecer um café, limpar o recinto que você vai trabalhar. Fico muito surpresa quando sou tocada por gestos enormes e mais grandiosos ainda. E são eles que me fazem agradecer detalhadamente a cada anjo que apareceu e continua fazendo parte da minha vida.

Os maiores anjos, que estão em minha vida desde que eu nasci, meus heróis e que de tão generosos me fazem acreditar que dariam a vida deles pela minha: meus pais, meus eternos amores. Eu os amo demais. Muito obrigada por tudo! Inclusive por me dar duas irmãs maravilhosas que me apóiam, me amam e que são reciprocamente amadas! Meu muito obrigada!

E eis que a vida não me poupa em emoções e me reserva mais uma surpresa: o nascimento de um príncipe com nome de Rei, Artur. Ainda me surpreendo com o fato de uma criaturinha tão pequenininha representar tanto amor. Artuzinho, dindinha te ama demais! Muito obrigada por existir!

Acredito sim! E passei a acreditar ainda mais quando ingressei no mestrado em Campina Grande em meados de 2010. Essa cidade me apresentou pessoas maravilhosas, que conviveram comigo, me respeitaram e me amaram: Minhas colegas de apartamento Cintya, Claudinha, Lidja e Camila. Obrigada por tudo meninas! Amo vocês. Meus orientadores Tiago e Rohit, agradeço-os imensamente, sem vocês eu não conseguiria. Muito obrigada pelos ensinamentos, por terem me acolhido. Aproveito e peço desculpas por todos os transtornos ocorridos. Mesmo nesses momentos vocês não desistiram de mim. Palavras como essas nunca sairão da minha cabeça: "você tem que levantar e fazer. Simplesmente fazer, sem pensar, racionalizar. Entrar em guerra, faca nos dentes, e fazer". Me sinto imensamente honrada por ter feito parte de um grupo tão especial. Grupo esse que me presenteou com a presença cotidiana e amiga de grandes pessoas. Muito obrigada Gustavo, Melina, Laerte, Solon e Alysson por todos os dias de labuta e divertidos ao mesmo tempo! Vocês me

ajudaram inigualavelmente. Meus amigos da UFCG: Andreza (amiguinha linda!), Aninha, Cabral, Elthon, Adriana, Neto, Éverton, Augusto, Elloá, Marco e Magna. Muito obrigada pela força!!! Vocês são responsáveis pela minha vitória. Agradeço a Copin, representada por Aninha, Vera, Rebeka e Nazareno. Sempre disponíveis e dispostos a ajudar. Muito obrigada!

Muito obrigada ao INES, representado por Sérgio Soares, pelo financiamento da minha pesquisa.

Muito obrigada aos estimados professores que me ajudaram nessa caminhada com observações muito pertinentes: Márcio Cornélio, Patrícia Machado, Henrique Rebêlo, Roberta Coelho, Augusto Sampaio, Gary Leavens, Erik Poll e Leila Maciel.

Muito obrigada aos meus grandes e amados amigos: Polly, Jéssica, Marden, Renata, Felipe, Letícia, Dani e João Paulo. Agradeço muitíssimo a compreensão de Givanildo e Fred nessa reta final.

E sem entender porque fui agraciada com tantos anjos em minha volta, eu, então, responsabilizo Deus. Agradeço IMENSAMENTE por ter sido escolhida e rodeada por eles. Com lágrimas nos olhos e muito feliz, somente me resta agradecer. MUITÍSSIMO OBRIGADA!

Conteúdo

Lista de Símbolos

BISL - *Behavioral Interface Specifications Language*

JML - *Java Modeling Language*

KLOC - *Kilo Lines of Code*

KLJML - *Kilo Lines of JML*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Engenharia de software é uma disciplina que engloba a execução sistemática de um conjunto de boas práticas para a produção de software de qualidade, desde as etapas iniciais até a sua manutenção [44]. A produção de software com qualidade é uma busca contínua e que vem ganhando cada vez mais importância dentro do processo de desenvolvimento de software [16]. A melhor definição para qualidade de software é dada pela combinação de vários fatores de qualidade divididos em duas principais categorias: fatores externos e internos [30]. A primeira categoria refere-se às características que podem ser detectadas pelo usuário, como por exemplo performance e usabilidade. Enquanto que a segunda categoria relaciona-se com fatores perceptíveis somente pelos profissionais envolvidos na produção do software. Mas no momento final, somente os fatores externos importam [30].

Dentre os vários fatores externos existentes, corretude e robustez podem ser destacadas como características relevantes para sistemas computacionais. Segundo McCall [26], corretude é o fator de qualidade do quanto um sistema satisfaz a sua especificação e atende às expectativas do cliente, enquanto que robustez é a habilidade que um sistema de software possui ao reagir apropriadamente a condições anormais. Para determinar se um software possui corretude, é necessária a produção de uma descrição precisa dos requisitos, pois um software é correto ou incorreto de acordo com sua especificação [44].

O uso de especificação formal no processo de desenvolvimento de software é uma maneira confiável e precisa de descrever o software corretamente, pois elimina ambiguidades, uma vez que são escritas em linguagens com semântica e sintaxe precisamente definidas [45]. Uma maneira de encorajar a prática de especificação formal é através da utilização de lin-

guagens de especificação de interface comportamental (*Behavioral Interface Specification Language* - BISL) [45]. BISL, linguagens de especificação acessíveis tanto para especificadores quanto para desenvolvedores, são usadas para especificar tanto interfaces quanto comportamentos de módulos.

Neste cenário, *Design by Contract* (DBC) [30] é uma metodologia que tem por conceito chave a visualização de uma relação de acordo formal entre uma classe e seus clientes. Assim, um contrato é expresso por obrigações e direitos de clientes e implementadores, onde o cliente deve garantir algumas condições antes de chamar um método, e a classe, por sua vez, deve garantir algumas propriedades após sua chamada. Especificações no estilo DBC são escritas principalmente em forma de pré-condições, pós-condições e invariantes. Pré-condições definem restrições para a entrada de um método, pós-condições estabelecem qual o resultado deve ser dado pelo método, e invariantes são restrições que devem ser asseguradas antes e depois da execução de todos os métodos, levando em consideração a visibilidade estabelecida por cada método.

Existem algumas BISLs baseadas em *Design by Contract* assim como Eiffel [27], Spec# [2] e Java Modeling Language (JML) [21]. JML é uma linguagem de especificação formal para classes e interfaces Java que especifica comportamento através de asserções como pré-condições, pós-condições e invariantes, e elas podem ser usadas para verificar a corretude de programas. Estas asserções são escritas usando um subconjunto de expressões Java e são anotadas dentro do próprio código fonte.

1.1 Problema

Especificações formais são utilizadas principalmente na descrição de sistemas críticos, para os quais requisitos como segurança e confiabilidade possuem extrema importância [44]. Desta forma, ao usar especificações formais, erros de especificação de requisitos levantados informalmente devem ser descobertos de maneira precisa e a especificação abstrata resultante não pode ser ambígua. Porém, no momento da implementação do software, desenvolvedores podem construir código não conforme com a especificação dada. Em sistemas especificados formalmente através de contratos, os desenvolvedores podem produzir equivocadamente métodos em não-conformidade com as asserções (pré-condições, pós-condições e invarian-

tes). Conformidade neste contexto, refere-se à consistência entre especificação por contratos e implementação.

As abordagens mais utilizadas atualmente para detectar não-conformidades são análise manual ou automática. A primeira técnica é facilmente propensa a continuação da ocorrência de defeitos, uma vez que o ator responsável por tal tarefa pode estar viciado na maneira como inspeciona o código e, consequentemente, pode esquecer de analisar outras possibilidades e, assim, não identificar não-conformidades aparentes. Além disso, ela é muito custosa, pois necessita de um especialista e pode demandar muito tempo. A segunda técnica divide-se em análise estática e análise dinâmica. Análise estática é realizada para verificar o código sem precisar executá-lo. Essa análise pode ser incompleta e inconsistente, uma vez que detecta muitos alarmes falsos, o que torna difícil a tomada de decisão por parte do especialista. Enquanto que a análise dinâmica dá respostas imediatas para os desenvolvedores por meio de verificadores em tempo de execução. Embora não dê certeza que todos os bugs foram encontrados, esse tipo de técnica dá confiança de que os bugs encontrados são verdadeiros positivos.

Dependendo da metodologia de desenvolvimento adotada, muitas não-conformidades podem ser encontradas em momentos tardios do processo de desenvolvimento do software. Assim, a detecção prévia de não-conformidades é altamente desejável, uma vez que quanto mais cedo uma não-conformidade é detectada, maior é a qualidade do software, pois não excede custo e prazo estabelecidos. Verificação dinâmica dá *feedback* imediato para os desenvolvedores, mesmo que as especificações sejam parciais. Assim, detectar não-conformidades, neste caso, depende estritamente da qualidade dos casos de testes que exercitam as asserções produzidas a partir dos contratos.

1.2 Exemplo Motivante

Nesta seção, nós apresentamos uma não-conformidade em um pequeno exemplo codificado e especificado (estilo DBC) em alto nível. Neste exemplo (Código-fonte 1.1), temos a classe `Pessoa`, onde colocamos os contratos entre comentários para destacá-los.

A Classe `Pessoa` contém dois métodos. `atualizaAltura` e `crescimento`. As pré-condições de ambos os métodos estabelecem que eles só podem ser executados se os

argumentos para o parâmetro `a` forem positivos. A pós-condição de `atualizaAltura` garante que o seu retorno deva ser um valor para `altura` maior que o seu valor anterior, indicando, assim, o histórico de crescimento de uma pessoa. Supomos que LIMITE e FATOR_CRESCIMENTO são constantes que estão pré-estabelecidas.

Código Fonte 1.1: Classe `Pessoa`

```
1   class Pessoa {
2
3       int altura;
4
5       /* invariante altura > 0; */
6
7       /* pré-condição a > 0;
8          pós-condição altura.atual  > altura.anterior;
9         */
10      public void atualizaAltura (inteiro a){
11          altura = a;
12      }
13
14      /* pré-condição a > 0; */
15      private int crescimento(inteiro a){
16          SE (a < LIMITE){
17              retorne a + a*FATOR_CRESCIMENTO;
18          SENÃO retorne a;
19          }
20      }
21  }
```

Conformidade entre especificação e implementação é um requisito para desenvolvedores que usam DBC, embora erros sutis e difíceis de serem detectados possam ocorrer. Por exemplo, a classe `Pessoa` contém uma não-conformidade que pode ser detectada através de chamadas para os módulos `atualizaAltura` e `crescimento`, tal como ilustrado no seguinte fragmento de Código Fonte 1.2.

Código Fonte 1.2: Sequência de Chamadas aos Módulos da Classe `Pessoa`

```
1 a = 10;
2 Pessoa p1 = new Pessoa();
3 p1.atualizaAltura(a);
4 a = Pessoa.crescimento(a);
5 p1.atualizaAltura(a); //pós-condição violada
```

Suponha que a constante `LIMITE` seja igual a `4`. Ao atribuir valor `10` à variável `a`, na segunda chamada ao método `atualizaAltura`, a pós-condição é violada. Pois dependendo do valor inicial de `a`, o método `crescimento` pode retornar um valor imutável, se ele ultrapassar a constante `LIMITE`. Consequentemente, ao chamar novamente o método `atualizaAltura`, o valor de `a` pode estar imutável, o que viola a sua pós-condição. Ao violar a pós-condição, o software torna-se incorreto, pois a especificação do usuário não está sendo seguida. Mesmo em programas pequenos, problemas como esse são difíceis de serem detectados manualmente e um erro pode, então, não ser detectado em desenvolvimento e somente ocorrer quando o software já está disponível para o usuário.

Problemas de não-conformidade podem acontecer devido a uma falsa suposição do especificador ou a um equívoco do desenvolvedor ao codificar uma solução incompatível com a especificação definida. No entanto, em muitos casos, os papéis de especificador e desenvolvedor são executados pelo mesmo responsável. Independentemente de quem causou a inconsistência, não-conformidades mais sutis podem ser mais difíceis de serem identificadas em sistemas maiores, pois muitos caminhos úteis para encontrar não-conformidades podem não ser testados. Tendo em vista este cenário, acreditamos que as não-conformidades que possivelmente possam estar presentes em sistemas formalmente anotados devem ser detectadas o quanto antes no processo de desenvolvimento de software para que o nível de qualidade seja assegurado.

1.3 Solução

Neste trabalho, propomos uma técnica baseada em testes para detecção de não-conformidades no contexto de programas anotados com contratos. Para investigar a ocorrência de não-conformidades, nossa técnica gera testes e os executa automaticamente, atestando

a confiança de que uma implementação satisfaz sua especificação. Os testes gerados são basicamente sequências de chamadas aos métodos e construtores da aplicação de entrada. E os oráculos dos testes são os contratos transformados em asserções. Algumas ferramentas como o compilador de JML, *jmlc* e *OpenJML* realizam esse tipo de transformação para programas codificados na linguagem Java. Assim, pós-condições e invariantes (checados após a execução do método) são usados como assertivas para o resultado da execução dos testes, e invariantes (checados antes da execução do método) e pré-condições são delimitadores de dados de entrada. Após o processo de geração e execução dos testes, a técnica dá como resultado um relatório indicando as não-conformidades existentes. Não-conformidade é definida como uma inconsistência entre o oráculo dos testes (resultados esperados) e o resultado da execução dos testes.

JMLOK é a implementação da técnica proposta no contexto de programas escritos em Java e especificados com contratos JML. Em programas JML, os contratos podem ser compilados e transformados em *bytecodes* Java instrumentados com asserções que podem ser exercitadas através de testes. Em JMLOK, testes são gerados automática e aleatoriamente pela ferramenta Randoop [36]. A ferramenta restringe a redundância de testes ao utilizar uma técnica focada no *feedback* de sequências de chamadas já geradas, possibilitando, desse modo, a diversidade dos testes. Após, a execução dos testes, JMLOK indica as não-conformidades quando exceções indicando violações aos contratos são lançadas. Podem ocorrer, basicamente, violações às pré-condições, pós-condições e invariantes.

1.4 Avaliação

Nós avaliamos nossa técnica através da aplicação do JMLOK em programas JML. A avaliação foi realizada em 5 unidades experimentais com o objetivo de detectar não-conformidades e analisar os resultados obtidos pela ferramenta. Todos as unidades experimentais totalizam 18 KLOC e 5 K linhas de especificação JML (KLJML). A partir das exceções lançadas durante a execução do JMLOK nas unidades experimentais, detectamos o total de 29 não-conformidades. A ferramenta despendeu menos que 10 minutos para executar todas as unidades experimentais. Para descobrir a origem do problema, realizamos uma verificação manual e classificamos as não-conformidades em categorias de falhas detectadas. Verifica-

mos, assim, que a maioria dos problemas ocorrem devido à especificação de pré-condições fracas, como foi a do exemplo motivante. A pré-condição mais fraca é `TRUE`, em que há permissão para que todos os parâmetros de entrada sejam executados pelo método. Assim, até valores proibitivos são executados, o que pode causar violações às pós-condições ou invariantes quando vistos como parte da pós-condição. Também realizamos a comparação do JMLOK com a ferramenta JET [8] com o objetivo de comparar os resultados obtidos e analisar o número de não-conformidades detectadas em cada ferramenta. Executamos ainda, o JMLOK na biblioteca de tipos JML (JMLModels), muito utilizada pela comunidade JML, com o intuito de verificar se JMLOK conseguiria detectar não-conformidades, entretanto não conseguimos, pois ocorreram erros durante a compilação.

1.5 Resumo de Contribuições

Em resumo, as principais contribuições deste trabalho são:

- Uma técnica para verificar não-conformidades em programas especificados com contratos (Capítulo 3);
- Uma ferramenta, JMLOK, para a checagem de programas Java especificados com contratos na linguagem JML (Seção 3.2);
- Uma avaliação da ferramenta em pequenos exemplos e aplicações reais que resultou em 29 não-conformidades (Capítulo 4).

1.6 Organização do Trabalho

No próximo capítulo, apresentamos a fundamentação teórica para entender os principais conceitos que permeiam o nosso trabalho (Capítulo 2). O Capítulo 3 apresenta a técnica proposta para verificar não-conformidades em programas especificados com contratos e a ferramenta JMLOK, implementação da técnica proposta em programas Java especificados em JML. No Capítulo 4, descrevemos a avaliação realizada com o intuito de detectar não-conformidades. Por fim, no Capítulo 5, apresentamos as considerações finais, trabalhos relacionados e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este trabalho concentra-se no conceito de conformidade (Seção 2.1), conceitos relacionados ao desenvolvimento baseado em contratos (Seção 2.2), suas linguagens e ferramentas (Seção 2.3) e técnicas para a detecção de não-conformidades (Seção 2.4).

2.1 Definição de Conformidade

Neste trabalho, consideramos o conceito de corretude, entretanto, utilizamos o termo *conformidade* ao invés de *corretude*, pois consideramos *conformidade* uma forma mais leve de *corretude*, uma vez que não realizamos provas formais e levamos em consideração que as especificações dadas para uma implementação estão corretas.

A definição de corretude de programas vem sendo discutida desde a década de 60, principalmente por trabalhos desenvolvidos por Hoare [17; 18] e Dijkstra [13]. Hoare estabelece o conceito de corretude usando axiomas e regras matemáticas para provar que programas estão realmente seguindo os requisitos do usuário em nível de projeto detalhado. A lógica de Hoare para denotar corretude é representada por uma tripla: `{P} C {Q}`, onde `P` e `Q` são asserções de pré-condição e pós-condição, respectivamente, e `C` é um programa. Assim, quando a pré-condição `P` é assegurada, o programa `C`, se terminar, deve estabelecer a pós-condição `Q`. No exemplo apresentado na Seção 1.2 (Código-fonte 1.1), `P` é representada pela pré-condição `a > 0`, `C` é o corpo do método `atualizaAltura` e `Q` é representado pela pós-condição `altura.atual > altura.anterior`.

Meyer [30] define, a princípio, corretude como uma noção relativa, pois um software é

considerado correto quando ele implementa todas as funções a que se propõe realizar, seguindo precisamente uma descrição que especifica seus requisitos. Ou seja, um software não é correto ou incorreto por si só, ele é correto ou incorreto de acordo com uma determinada especificação. Além disso, Meyer segue a mesma linha de Hoare, e transforma a definição informal acima citada na fórmula da corretude representada pela tripla `{P} C {Q}`.

Desta forma, é importante que a corretude do software seja assegurada para que a sua qualidade seja mantida. Um software incorreto (não consistente com sua especificação) ocasiona retrabalho, o que excede o custo e o tempo previstos no projeto do software.

2.2 Desenvolvimento Baseado em Contratos

O uso de conceitos intrínsecos à orientação a objetos como tipos abstratos de dados, objetos, classes, interfaces, são fundamentais para a modularização do software, sendo uma prática que beneficia a legibilidade e organização do código como um todo. Porém tais técnicas não são suficientes para garantir a total qualidade do software, pois além de atingir requisitos de qualidade, o software também deve ser confiável. Segundo a perspectiva de Meyer [30], confiabilidade é um conjugado de corretude e robustez. O uso de asserções no processo de desenvolvimento de sistemas pode garantir a confiabilidade do software ao assegurar que os requisitos propostos para o sistema estão sendo implementados corretamente. É partindo desse pressuposto que as linguagens BISL [45] – *Behavioral Interface Specification Languages* – provêem especificações formais no formato de anotações no nível de código-fonte, permeando, assim, os módulos com pré-condições, pós-condições e invariantes que asseguram se as regras de negócio serão mantidas em tempo de execução. Além de resultar em sistemas confiáveis, tais especificações melhoram a legibilidade e documentação do software ao expressar o seu comportamento dentro dos módulos.

Dentro desta abordagem encontra-se *Design by Contract* (DBC) [30; 29], um conceito de implementação de contratos para sistemas orientados à objetos estudada por Bertrand Meyer com o intuito de ser integrada à linguagem de programação Eiffel [27]. O conceito de contrato é semelhante ao utilizado no mundo real, ou seja, é firmado um acordo entre cliente e um fornecedor de serviço onde são fixados direitos e obrigações para ambos. Assim, a maneira de estabelecer um contrato nos termos de DBC é através da especificação de pré-

condições, pós-condições e invariantes, onde pré-condições são asserções que estabelecem as restrições para execução de uma determinada rotina, pós-condições descrevem as propriedades que a rotina deve garantir após sua execução e invariantes são restrições que devem ser estabelecidas por todas as rotinas de um determinado módulo. Então, o critério de conformidade estabelecido por DBC é o seguinte: uma determinada rotina está correta quando pré-condições e invariantes são verdadeiros antes de sua execução, e então os mesmos invariantes e pós-condições são asseguradas após sua execução. No exemplo da classe `Pessoa` (Código-fonte 1.1), o invariante `altura > 0` e a pré-condição `a > 0` são verificados antes da execução do método `atualizaAltura`. Quando essas restrições são aceitas, então o método é executado e o seu retorno é comparado com a pós-condição `altura.atual > altura.anterior`.

Algumas linguagens de especificação se baseiam em DBC para assegurar confiabilidade ao software, tais como Java Modeling Language (JML) [21] (Seção 2.3), Spec# [2], entre outras. JML foi a linguagem eleita para avaliar a abordagem proposta neste trabalho, pois é específica para a linguagem de programação Java, a qual é bastante difundida e, consequentemente, possui muitos usuários e repositórios de software a serem avaliados. Mais detalhes sobre essa linguagem estão dispostos na próxima seção.

2.3 Java Modeling Language (JML)

JML é uma linguagem de especificação BISL para programas Java que tem por princípio o fácil uso de métodos formais no processo de desenvolvimento de sistemas. Essa linguagem é resultante dos esforços de vários profissionais oriundos de diversos pólos acadêmicos e liderada por Gary Leavens (University of Central Florida[1]). Assim, JML baseia-se no estilo de especificação Hoare, ou seja, pré-condições e pós-condições para especificar o comportamento dos métodos do sistema. Por conseguinte, baseia-se no estilo DBC, estabelecendo um contrato entre o cliente e o sistema, onde o cliente deve entregar entradas que estabeleçam as pré-condições, enquanto o sistema deve entregar para o cliente a execução esperada do sistema estabelecendo as pós-condições.

As especificações JML são anotadas em forma de comentários dentro do código e pos-

[1] http://www.eecs.ucf.edu/~leavens/homepage.html

sui a seguinte sintaxe: `//@ < especificação JML >` ou `/*@ < especificação JML > @*/`. Pré-condições JML são expressas pela anotação `requires`. Enquanto que as pós-condições são representadas pela palavra reservada `ensures`. A cláusula `invariant` denota um predicado que deve ser assegurado antes da execução do construtor e antes e depois de todas chamadas aos métodos.

A principal vantagem do estilo aplicado à linguagem JML é a ausência de efeitos colaterais, uma vez que ela possui em seu escopo somente um subconjunto de expressões Java e também alguns poucos operadores matemáticos, assim expressões que incorporam efeitos colaterais como ++, −, entre outras não fazem parte das expressões matemáticas expressas em JML. Além disso, ela possui uma biblioteca de expressões matemáticas específica para JML, o JMLModels[2], que possui uma vasta implementação de conceitos matemáticos tais como conjuntos e sequências. Essa biblioteca pode ser incorporada aos programas anotados, abstraindo, assim, o uso de expressões matemáticas complexas.

Como exemplo, considere o seguinte trecho de código 2.1 escrito em Java com anotações JML retirado do repositório de exemplos do site oficial da linguagem[3]. Para simplificar, colocamos aqui somente um trecho do código: temos a classe `Rectangular` que implementa números complexos em coordenadas retangulares e que possui, neste trecho de código, o construtor e os métodos `realPart()` e `imaginaryPart()`. Note que estes métodos herdam as especificações da classe `ComplexOps`, que por sua vez herda da interface `Complex` (Código-fonte 2.2). Ou seja, JML suporta herança de especificação, outra vantagem inerente à linguagem.

JML usa herança de especificação para impor especificações de super tipos sobre seus subtipos, suportando o conceito de subtipo comportamental. Em JML, herança de especificação significa que métodos de subtipos devem obedecer as especificações de todos os métodos sobrescritos.

O construtor `Rectangular` possui quatro cláusulas `ensures`. Elas verificam se nos casos em que os valores dos atributos `re` e `img` são números válidos, então os valores retornados pelos métodos `realPart` e `imaginaryPart` devem ser iguais aos valores dos atributos `re` e `img`, respectivamente. E asseguram o caso contrário também, ou seja,

[2]http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/org/jmlspecs/models/package-summary.html

[3]http://www.eecs.ucf.edu/~leavens/JML/examples.shtml

quando os números `re` e `img` não são números válidos, então os métodos `realPart` e `imaginaryPart` também devem retornar números não válidos.

Código Fonte 2.1: Classe `Rectangular`

```
1  public  /*@pure@*/ strictfp class Rectangular extends ComplexOps
       {
2
3      /**Parte real deste número. */
4      private double re;
5      /**Parte imaginária deste número. */
6      private double img;
7
8      /*@
9        @  ensures !Double.isNaN(re) ==> realPart() == re;
10       @  ensures !Double.isNaN(img) ==> imaginaryPart() == img;
11       @  ensures Double.isNaN(re) ==> Double.isNaN(realPart());
12       @  ensures Double.isNaN(img) ==> Double.isNaN(imaginaryPart()
             );
13       @*/
14     public Rectangular(double re, double img) {
15         this.re = re;
16         this.img = img;
17     }
18
19     // especificação herdada
20     public double realPart() {
21         return re;
22     }
23
24     // especificação herdada
25     public double imaginaryPart() {
26         return img;
27     }
```

A interface `Complex`, que possui a especificação dos métodos, importa a biblioteca de tipos `JMLModels`. Mais precisamente o `JMLDouble` através da especificação `@model import org.jmlspecs.models.JMLDouble` (Código-fonte 2.2).

Note que a palavra reservada `model` aparece duas vezes no código: ao realizar a importação da biblioteca `JMLDouble`, e ao declarar a constante `tolerance`. JML declara várias cláusulas com o modificador `model`. Ela pode servir para declarar campos, métodos, tipos e bibliotecas, como vimos. O modificador `model` tem como principal significado declarar que a construção só pode estar presente para propósitos de especificação. Na interface `Complex`, a constante `tolerance`, por exemplo, é somente utilizada durante a especificação, não é uma constante utilizada na implementação Java. A mesma situação acontece com o `model import`, pois a biblioteca `JMLModels`, como um todo, é somente utilizada para propósitos de especificação. Assim, o retorno do método `JMLDouble.approximatelyEqualTo` é somente enxergado do ponto de vista da especificação do método `realPart`.

No Código-fonte 2.2, há também a cláusula `ghost`, que é similar ao `model`. Mas com `ghost`, uma variável pode ser inicializada diretamente na especificação, como acontece com `double tolerance = 0.005`.

Código Fonte 2.2: Interface `Complex`

```
1
2  //@ model import org.jmlspecs.models.JMLDouble;
3
4  public /*@pure@*/ interface Complex {
5
6       //@public ghost static final double tolerance = 0.005;
7
8      /** Return the real part of this complex number. */
9      /*@  ensures JMLDouble.approximatelyEqualTo(
10       @              magnitude()*StrictMath.cos(angle()),
11       @              \result,
12       @              tolerance);
13       @*/
14     double realPart();
15     }
```

2.3.1 Ferramentas de Suporte JML

Por ser uma linguagem BISL que se integra facilmente ao código-fonte, JML suporta uma vasta lista de ferramentas [5] que é mantida por uma comunidade ativa. JML possui ferramentas como compiladores, verificadores estáticos, geradores de testes e geradores de especificação. Dentre elas, existe o compilador JML [10], *jmlc*, que foi largamente utilizada neste trabalho.

O compilador *jmlc* é semelhante aos compiladores Java, pois gera *bytecodes* Java a partir de programas Java que possuem anotações JML. Os bytecodes gerados incluem checadores correspondentes às asserções, de tal modo que haja verificação delas em tempo de execução. Quando os arquivos gerados pelo *jmlc* são executados, os bytecodes correspondentes às asserções podem denunciar violações aos contratos através do lançamento de exceções que indicam o tipo de violação ocorrida. O Código-fonte 2.3 mostra, em alto nível, como os contratos do método `atualizaAltura` da classe `Pessoa` (Código-fonte 1.1) poderiam ser transformados após a compilação.

Código Fonte 2.3: Método `atualizaAltura` após compilação com *jmlc* em alto nível de abstração.

```
1  Classe Pessoa {
2  Assertivas de Inv
3  Assertivas de Pre
4   atualizaAltura(int n){
5      //corpo do método
6   }
7  Assertivas de Pos
8  Assertivas de Inv
9  }
```

As asserções de invariante, pré-condição e pós-condição (Código-fonte 2.3) são representadas concretamente, geralmente, por estruturas de controle `if` e estruturas `try/catch`.

A execução de uma chamada ao método `atualizaAltura` é verificada como mos-

trado, em alto nível, no Código-fonte 2.4. As assertivas `assert (inv e pre)` e `assert (inv e pos)` verificam, respectivamente, invariantes e pré-condições antes da execução do método, e invariantes e pós-condições após a execução do método.

Código Fonte 2.4: Execução em alto nível do método `setAltura`.

```
1 assert(inv e pre);
2         atualizaAltura(10);
3 assert(inv e pos);
4
5 assert(inv e pre);
6         atualizaAltura(1);
7 assert(inv e pos);
```

Existe outro compilador, denominado *OpenJML*, que está atualmente em desenvolvimento. Os esforços atuais para atualizações e incremento de mais cláusulas ao compilador JML estão sendo realizadas nesta ferramenta.

Quando contratos são violados, três tipos principais de exceções podem ocorrer: `JMLPreconditionError`, `JMLPostconditionError` e `JMLInvariantError`. Dentro dessas três classificações de exceções ainda existem outros tipos de exceções que herdam delas. A Figura 2.1 [40] mostra a hierarquia de erros de violação em JML. A classe abstrata `JMLAssertionError` é uma subclasse da classe `java.lang.Error`, e super classe de todos os tipos de erros de asserções. Como mostra a Figura 2.1, além dos três tipos de erros citados inicialmente, existem outros. O tipo de violação à pós-condição `JMLPostconditionError`, por exemplo, se divide em dois tipos: `JMLInternalNormalPostconditionError` e `JMLInternalExceptionalPostconditionError`. O primeiro acontece quando ocorre uma violação à pós-condição após a execução do método. E o segundo acontece quando ocorre uma violação à pós-condição quando o método lança uma exceção (termina de maneira anormal). A classe `JMLHistoryConstraintError` reporta violações referentes à *history constraint*. *History constraint* possui conceito semelhante ao de invariante, mas é usado para obrigar a maneira como os valores de determinadas variáveis mudam com o tempo. Um exemplo de *history constraint* é a especificação `//@ constraint a == \old(a)` que significa que `a` é imutável.

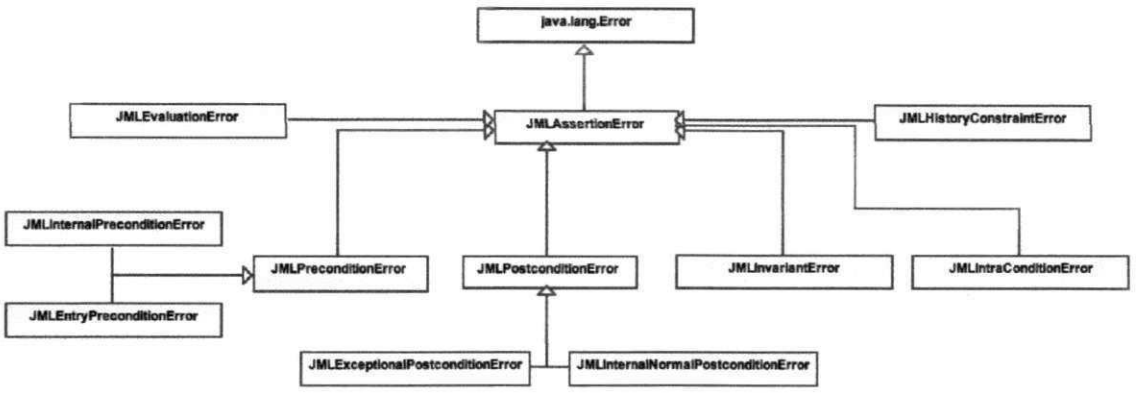


Figura 2.1: Diagrama de Classes dos Tipos de Erros de Asserções em JML.

Note que os nomes das exceções indicam o tipo de erro ocorrido e facilitam a investigação da origem do problema pelo especialista, podendo assim servir como oráculo para a realização de testes. A ferramenta implementada neste trabalho representa a técnica proposta no contexto de programas Java/JML e utiliza as exceções lançadas pelo *jmlc* como seu oráculo para indicar o tipo de não-conformidade ocorrida. Como é uma técnica geral, o compilador OpenJML também pode ser usado como oráculo, da mesma forma que o *jmlc* . Uma versão em desenvolvimento da ferramenta está evoluindo para realizar essa atualização.

Outras ferramentas que suportam JML são: ferramentas de teste de especificação como JMLUnit [7], Jartege [34], JMLUnitNG [46] e JET [9], ferramenta de documentação *jmldoc*, semelhante ao javadoc; compiladores como OpenJML e *ajmlc* [39]; e ferramentas de análise estática como ESC/Java2 [11].

2.4 Testes de Software

O software deve ser previsível e consistente e não deve oferecer surpresas para os usuários [32]. Para garantir isto, existe o processo de teste de software, um tipo de análise dinâmica que executa programas sistematicamente com o intuito de certificar se o software faz o que é designado para ser feito e se ele não realiza tarefas que não pretende fazer, contribuindo, assim, para a melhoria da qualidade de software.

Existem vários tipos de testes, dentre eles: testes de componente ou unidade (validam cada componente do software em relação a sua especificação), testes de integração (validam se grupos de componentes colaboram com o andamento especificado pelo projeto do

sistema), testes de sistema (validam o sistema como um todo de acordo com sua especificação) e testes de aceitação (validam o sistema de acordo com o contrato firmado com o cliente) [44].

Testes de unidade garantem confiança ao afirmar que uma aplicação está correta, uma vez que é um estágio relevante no processo de teste de software que tende a encontrar falhas prematuramente no ciclo de vida do software, aumentando, então, a confiabilidade. Utilizamos testes de unidade largamente neste trabalho. Assim, a primeira etapa (Capítulo 3) da abordagem proposta neste trabalho propõe que testes de unidade sejam utilizados para detectar não-conformidades nos programas com contratos. Nesta etapa, a geração dos testes leva em consideração somente o programa, sendo que os contratos são transformados em asserções e por isso são utilizados como oráculo para os testes.

Além disso, agilizamos o processo de manufatura dos testes na implementação da ferramenta JMLOK, pois realizar testes manualmente é uma tarefa complexa. Considere, por exemplo, um sistema que possua mais de 100.000 instruções, onde exercitar um número razoável de cenários e com uma boa cobertura é manualmente impraticável. Por isso existem algumas ferramentas que facilitam a manufatura de testes ao gerá-los automaticamente. A geração automática de testes de unidade, em contraste com a produção manual, diminui o tempo gasto em sua confecção, a propensão aos erros e os custos, ao mesmo tempo que aumenta a diversidade dos testes. Existem algumas ferramentas acadêmicas com diferentes soluções para a geração automática de testes [1] [36] [37]. Em nossa implementação (Seção 3.2), nós utilizamos a ferramenta Randoop [36] que gera automaticamente testes de unidade aleatoriamente. Obtemos bons resultados, em curto tempo e com boa cobertura, ao aplicá-la em nossa abordagem e relatamos os resultados obtidos no Capítulo 4.

Capítulo 3

Uma Técnica para Verificar Não-conformidades em Programas Especificados com Contratos

Verificar a existência de não-conformidades em programas formalmente especificados é de grande importância, uma vez que a maioria dos sistemas inseridos neste contexto são críticos [44]. Dessa maneira, não-conformidades devem ser detectadas o quanto antes dentro do processo de desenvolvimento do software para que a confiabilidade do sistema seja assegurada. Neste capítulo, descrevemos uma técnica que propõe uma solução para encontrar não-conformidades em sistemas especificados no estilo DBC. A técnica baseia-se em análise dinâmica, a partir da geração e execução automática de testes. Os contratos são transformados em assertivas, as quais são usadas para checar se os resultados dos testes devem passar ou falhar. Neste capítulo, mostramos, primeiramente, uma visão geral da técnica (Seção 3.1). Das Seções 3.1.1 a 3.1.4, descrevemos em detalhes cada etapa da técnica. E na Seção 3.2, descrevemos a implementação da técnica proposta no contexto de programas Java especificados com JML.

3.1 Visão Geral da Técnica

A técnica descrita neste capítulo é proposta com o intuito de descobrir não-conformidades em programas especificados com contratos, que podem ser simples ou sistemas completos,

desde que possuam pelo menos uma classe concreta para que testes possam ser gerados e executados. Quando o sistema sob teste possui um conjunto de classes a serem testadas, é só passar todas as classes como argumento. O oráculo para os testes é gerado a partir dos contratos especificados. A saída gerada é dada a partir do resultado da execução dos testes: caso algum erro seja lançado, então existe não-conformidade no programa; caso contrário, os testes passam.

Resumidamente, a partir da aplicação especificada com contratos dada como entrada, os seguintes passos são realizados, como ilustrado na Figura 3.1:

1. Geração de testes de unidade compostos por sequências de chamadas ao(s) construtor(es) e métodos da(s) classe(s) sob teste (Seção 3.1.1);

2. Geração de oráculo para os testes a partir dos contratos. Linguagens de especificação baseadas em DBC, geralmente possuem compiladores que transformam os contratos em assertivas que podem ser utilizadas em tempo de execução, principalmente para servir como oráculo para testes (Seção 3.1.2);

3. Execução da suíte de testes gerada. Neste passo, o oráculo (conjunto de resultados esperados) é comparado com a saída dos testes (Seção 3.1.3);

4. Após a execução, um filtro distingue os testes que passam e os testes que falham a fim de indicar não-conformidades entre o programa de entrada e os contratos (Seção 3.1.4).

A técnica proposta pode ser aplicada em linguagens de programação com especificação no estilo DBC, assim, cada passo supracitado pode ser adaptado à realidade do contexto escolhido. A aplicação realizada neste trabalho (Seção 3.2) foi implementada no contexto de programas Java/JML.

3.1.1 Etapa 1: Geração dos Testes

Nesta primeira fase da técnica, testes são gerados, juntamente com os dados de testes, com o intuito de que sua execução (Etapa 3, Seção 3.1.3) valide a implementação mediante a especificação dada.

Existem vários tipos de testes que podem ser realizados durante os estágios de desenvolvimento do software, dentre eles tem-se testes de unidade e testes de integração que podem

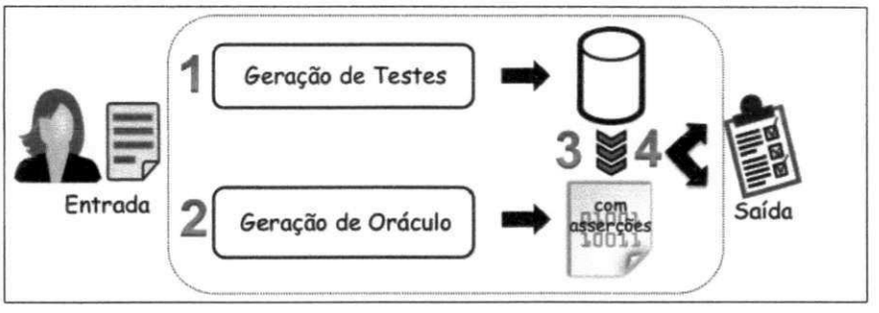


Figura 3.1: Técnica Proposta.

ser realizados sequencialmente para testar módulos individualmente e o comportamento entre módulos, respectivamente [32]. Na aplicação da técnica, geramos testes de unidade. Assim, chamadas a todos os métodos públicos das classes públicas sob teste são geradas. A geração dos testes é baseada somente na implementação e os contratos são utilizados como oráculo (Etapa 2, Seção 3.1.2). O Código-fonte 1.2 exemplifica testes de unidade gerados para testar o método `atualizaAltura` da classe `Pessoa` (Código-fonte 1.1).

Note que chamadas consecutivas aos métodos da classe podem indicar caminhos para o descobrimento de não-conformidades. A produção automática de testes pode criar diversos cenários que realizem chamadas consecutivas, aumentando, assim, a diversidade dos testes. Existem algumas ferramentas acadêmicas com diferentes soluções para a geração automática de testes [1] [36] [37]. Em nossa implementação (Seção 3.2), nós utilizamos a ferramenta Randoop [36] que faz a geração de testes aleatoriamente (maiores detalhes na Seção 3.2.1).

3.1.2 Etapa 2: Geração do Oráculo dos Testes

Nesta etapa da técnica, o oráculo para os testes é gerado a partir dos contratos. Os contratos são transformados em resultados esperados que são confrontados com o resultado da execução dos testes e dizem se a implementação viola ou não a sua especificação. Assim, pré-condições, pós-condições e invariantes se transformam em checadores em tempo de execução que possuem poder de decisão durante a execução dos testes.

Algumas linguagens de especificação por contratos, tais como JML, Eiffel e Spec#, possuem compiladores que geram código objeto com o objetivo de verificação em tempo de execução. Assim, do ponto de vista de testes de software, o código objeto gerado pode ser

usado como oráculo, pois transforma os contratos em checadores em tempo de execução. Na ferramenta implementada, utilizamos o compilador *jmlc* para transformar os contratos em asserções, pois a ferramenta aceita como entrada programas Java/JML.

O Código-fonte 2.3, mostra, em alto nível de abstração, como os contratos são transformadas em asserções após a compilação.

3.1.3 Etapa 3: Execução dos Testes

Nesta etapa, os testes gerados (Etapa 1, Seção 3.1.1) são executados levando em consideração o oráculo gerado na etapa anterior (Seção 3.1.2). Desta forma, após a comparação entre os resultados dos testes e os resultados esperados, então os testes são classificados: em *falha*, quando um contrato é violado, indicando que o programa pode conter uma não-conformidade na implementação com relação a sua especificação; e, em *sucesso*, quando o resultado da comparação é verdadeira.

Quando um contrato é violado, ele pode indicar não-conformidade entre pré-condição e método, método e pós-condição ou método e invariante (tanto na entrada como no retorno do método). Erros de pré-condição ocorrem quando um valor de entrada não permitido é executado no método. Em nossa implementação, alguns valores que são gerados automaticamente não satisfazem às pré-condições, então, neste caso denominamos testes *insignificantes*. Enquanto que erros de pós-condição e invariantes são considerados *relevantes* já que indicam que existe erro na implementação. Violações às pós-condições ocorrem quando a saída retornada pelo método não satisfaz à especificação da pós-condição. E violações aos invariantes podem ocorrer tanto antes da execução do método, quanto após a sua execução, pois eles são verificados nas entradas e saídas de cada método da classe.

O Código-fonte 2.4 mostra uma sequência de chamadas ao método `atualizaAltura` após a instrumentação com o compilador *jmlc*. Sendo que a segunda chamada ocasionará uma violação à pós-condição.

3.1.4 Etapa 4: Resultados da Execução dos Testes

Ao término da execução dos testes (Seção 3.1.3), a técnica entrega como saída para o usuário um relatório contendo os testes que detectaram erros com as exceções que foram lançadas.

Assim, quando um contrato é violado, então são lançadas exceções que indicam o tipo de violação ocorrida. Ao analisar as exceções lançadas, o especialista pode investigar a origem da não-conformidade e planejar a melhor forma de corrigi-la. Na avaliação (Capítulo 4), realizamos uma análise a partir dos relatórios de resultados obtidos e categorizamos manualmente as não-conformidades de acordo com a possível origem do problema.

Os resultados são apresentados através da divisão dos testes em duas classificações: *insignificantes* e *relevantes*. Testes *insignificantes* provocam o lançamento de exceções que ocorrem quando valores de entrada do teste não satisfazem às pré-condições do módulo testado. Uma chamada ao método `atualizaAltura` da classe `Pessoa` (Código-fonte 1.1) como mostrado no Código-fonte 3.1 é um exemplo de tipo de teste insignificante, pois ao passar como parâmetro de entrada um valor menor que `0`, o lançamento de uma exceção indicando violação à pré-condição é lançada.

Código Fonte 3.1: Exemplo de teste insignificante na classe `Pessoa`

```
1 a = -10;
2 Pessoa p1 = new Pessoa();
3 p1.atualizaAltura(a); //teste insignificante, "a = -10" não
    satisfaz a pré-condição
4 a = Pessoa.crescimento(a);
5 p1.atualizaAltura(a);
```

Nesta situação tais exceções são descartadas, pois não há não-conformidade entre o código e os contratos, uma vez que o acontecimento da exceção foi causado por um dado gerado pelo teste. Porém, exceções de pré-condição que são lançadas em situações onde um módulo do programa invoca, com algum parâmetro não satisfeito pela pré-condição de outro módulo, caracteriza a ocorrência de uma não-conformidade, já que houve um erro de implementação do módulo invocador, não sendo, então, descartada. O Código-fonte 3.2 mostra uma outra implementação e especificação para a classe `Pessoa`, onde uma chamada (Código-fonte 3.3) para o método `atualizaAltura` gera uma violação interna à pré-condição.

Código Fonte 3.2: Exemplo de código que pode gerar violação interna à pré-condição.

```
1 class Pessoa {
```

```
2
3       int altura;
4
5       /* invariante altura > 0; */
6
7       /* pré-condição a > =0;
8          pós-condição altura.atual  >= altura.anterior;
9         */
10      public void atualizaAltura (inteiro a){
11         altura = altura + crescimento(a); // chamada ao método
              crescimento que pode gerar violação interna à pré-
              condição.
12      }
13
14      /* pré-condição a >  0; */
15      private int crescimento(inteiro a){
16         SE (a < LIMITE){
17           retorne a + a*FATOR_CRESCIMENTO;
18         SENÃO retorne a;
19         }
20      }
21   }
```

Note que uma chamada com valor de entrada igual `0` (Código-fonte 3.3) para o método `atualizaAltura` satisfaz sua pré-condição, mas não satisfaz a pré-condição do método `crescimento`, que é invocado dentro de `atualizaAltura`. Desta forma, consideramos este tipo de violação uma não-conformidade.

Código Fonte 3.3: Teste que gera uma violação interna a pré-condição do método `crescimento`.

```
1 a = 0;
2 Pessoa p1 = new Pessoa();
3 p1.atualizaAltura(a); //"a = 0" é normalmente executado no método
     atualizaAltura, pois satisfaz a sua pré-condição. Entretanto,
```

```
não satisfaz a pré-condição do método crescimento.
```

Enquanto que *relevantes* são os testes que provocam o lançamento de exceções que indicam não-conformidades entre a implementação do programa e os contratos. Assim, exceções referentes à violação de pré-condições internas, pós-condições e invariantes se enquadram nessa classificação. A execução do Código-fonte 3.4 lançará uma exceção indicando uma violação a pós-condição, pois a segunda chamada ao método `atualizaAltura` quebra a restrição de pós-condição que estabelece que uma atualização de `altura` seja realizada com valor maior que o valor anterior imediato.

Código Fonte 3.4: Teste que gera uma violação a pós-condição do método atualizaAltura.

```
1 Pessoa p1 = new Pessoa();
2 p1.atualizaAltura(2);
3 p1.atualizaAltura(1); //chamada que viola a pós-condição.
```

3.2 Implementação da Técnica Proposta: JMLOK

Como prova de conceito da técnica, e para avaliar seus resultados, implementamos uma ferramenta, denominada JMLOK, que executa automaticamente os passos da técnica em programas JML. Assim, primeiro, classes de testes são geradas automaticamente e os casos de teste são compostos por sequências de chamadas a métodos e construtores. Nesta fase somente o código-fonte é levado em consideração (Seção 3.2.1). Em seguida, o oráculo para os testes é gerado a partir das especificações JML, onde um compilador específico para JML transforma os contratos em checadores de asserções em tempo de execução (Seção 3.2.2). Na terceira fase, os testes gerados são executados e o oráculo é utilizado para checar se o programa apresenta não-conformidades (Seção 3.2.3). Por fim, um relatório expõe os testes que identificaram não-conformidades e as exceções lançadas. Desta forma, o desenvolvedor pode analisar a origem dos problemas (Seção 3.2.4). As Seções 3.2.5 e 3.2.6 apresentam, respectivamente, a arquitetura e as limitações do JMLOK.

Assim, de maneira geral, JMLOK é composta por uma entrada, quatro passos e uma saída, como segue.

Entrada: uma aplicação Java/JML.

Passo1: Conjuntos de testes de unidade são gerados automática e aleatoriamente através do Randoop levando em consideração somente o código-fonte.

Passo 2: A aplicação é compilada pelo *jmlc*, ou seja, é neste passo que os verificadores de asserções são adicionados ao *bytecode*.

Passo 3: JMLOK executa os testes gerados no primeiro passo sobre o *bytecode* gerado pelo *jmlc*.

Passo 4: Um filtro separa os testes que passam e os que falham, e dentre os que falham, os testes são classificados em *meaningless* ou *relevants*.

Saída: Um relatório que é gerado como saída contendo os resultados da execução dos testes. A Figura 3.2 mostra todos estes passos.

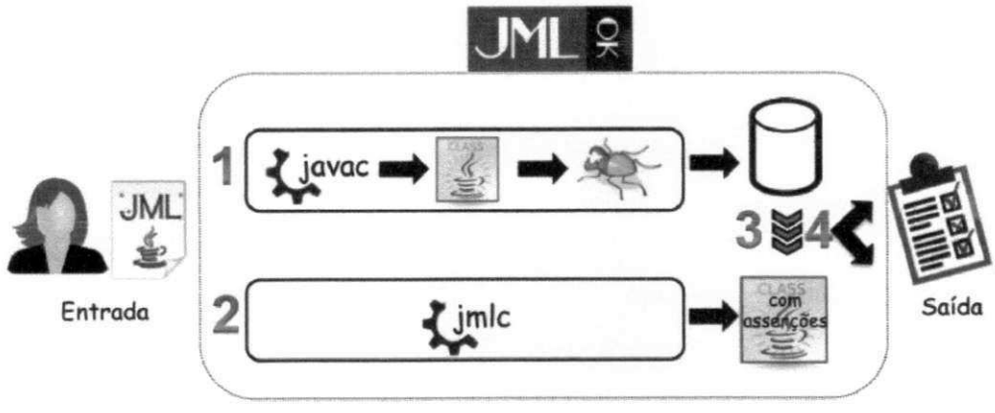


Figura 3.2: Ferramenta JMLOK.

3.2.1 Etapa 1: Geração Automática dos Testes pelo Randoop

A primeira etapa de funcionamento do JMLOK gera classes de testes de unidade através da ferramenta Randoop. Esta ferramenta gera aleatória e automaticamente os testes de unidade onde não são necessários os dados de testes fornecidos pelo usuário. A execução dessa ferramenta teve bons resultados ao descobrir erros em aplicações muito utilizadas, como a Sun JDK 1.5 [36] e tem sido acoplada também em outras soluções [43] [42]. A seguir, mostramos com mais detalhes o funcionamento desta ferramenta.

Randoop

O Randoop gera testes para programas Java de maneira aleatória e automática. Essa ferramenta possui um mecanismo que descarta testes redundantes através de um algoritmo que é guiado pelo feedback do comportamento do programa. Então, o Randoop gera testes de unidade incrementalmente para as classes sob teste e *assertions* que capturam o comportamento do sistema. Como configuração prévia para que a geração seja realizada, são passados como parâmetros: a lista de classes a serem testadas e um tempo limite para a geração dos testes (o tempo *default* do Randoop é 120 segundos). Como resultado da geração, temos um conjunto de classes de testes que são compostas, cada uma, por vários casos de testes no formato JUnit [24], os quais são compostos por cenários que consistem em sequências de chamadas aos métodos e construtores seguidos por assertions. A Figura 3.3 mostra um exemplo de configuração do Randoop com os parâmetros necessários, e as classes e casos de testes gerados.

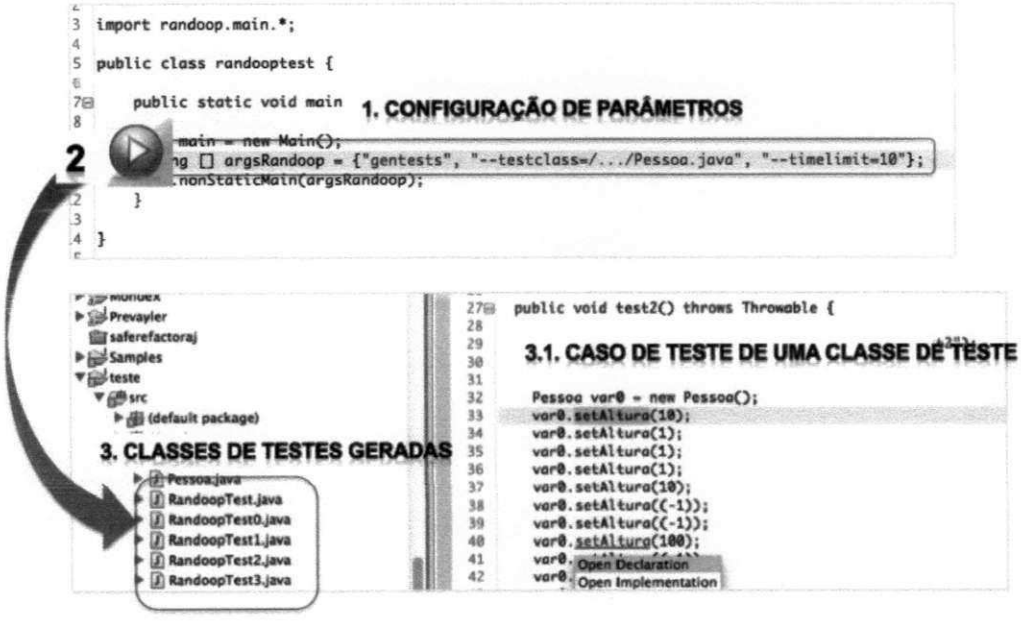


Figura 3.3: Exemplo de Configuração do Randoop. 1) Estabelecendo parâmetros como: classe sob teste e tempo limite (timelimit) de geração de testes. 2) Executando a classe de configuração. 3) Várias classes de testes são geradas. 3.1) Dentro de cada classe existe um conjunto de casos de testes com sequências de chamadas aos métodos da classe sob teste.

Para compor os dados dos testes, o Randoop utiliza variáveis primitivas como `int`, `char`

e `boolean`, além de `String`, e também instâncias geradas anteriormente. Para fazer chamadas a métodos que possuem como entrada tipos abstratos de dados (ADT), basta adicionar o ADT, que deve ser uma classe concreta, como parâmetro à lista de classes a serem testadas, assim o Randoop pode criar instâncias que serão usadas nas chamadas.

Randoop executa e recebe o *feedback* do comportamento do programa para checar o resultado com contratos e filtros. Os filtros identificam expressões ilegais e redundantes e geram mais entradas. Enquanto que os contratos refletem o comportamento do programa e são responsáveis pela criação das *assertions*. A próxima seção explica o funcionamento do Randoop dentro do JMLOK.

Funcionamento do Randoop no JMLOK

No JMLOK, optamos por utilizar a ferramenta Randoop como gerador automático de testes, pois esta ferramenta tem oferecido resultados satisfatórios como base de verificação em programas Java em algumas abordagens [43] [42] e por ela gerar sequências de chamadas aos métodos sob teste que podem detectar não-conformidades.

Como exemplo de funcionamento do JMLOK, considere o Código-fonte 3.5, escrito em Java e com especificações JML, como entrada para a ferramenta. Note que a palavra reservada *spec_public* na declaração do atributo `altura` denota que para fins de especificação ele será tratado como atributo público.

Código Fonte 3.5: Classe Pessoa anotada com JML.

```
1  public class Pessoa {
2    private /*@ spec_public @*/ int altura;
3    /*@ requires n > 0;
4      @ ensures this.altura >= \old(this.altura);
5      @*/
6    public void setAltura (int n) {
7      this.altura = n;
8    }
9  }
```

Na primeira etapa da ferramenta, o Randoop gera uma coleção de casos de testes contidos em cinco classes de testes (Figura 3.3).

Um exemplo de caso de teste gerado a partir da classe Pessoa é exibido no Código-fonte 3.6. Existem duas chamadas consecutivas ao método `setAltura`, sendo que essa sequência detecta uma não-conformidade, pois há uma atualização do atributo `altura` com valor menor (1) que o valor anterior (10).

Código Fonte 3.6: Teste de unidade gerado pelo Randoop

```
1  public void test5() throws Throwable {
2    Pessoa var0 = new Pessoa();
3    var0.setAltura(10);
4    var0.setAltura(1);
5    }
```

Como visto na Seção 3.2, a execução do JMLOK é dividida em três principais fases. Na fase da geração dos testes, o JMLOK recebe basicamente como entrada:

1. Uma lista de classes a serem testadas; e
2. Um tempo limite de geração de testes.

3.2.2 Etapa 2: Geração do Oráculo dos Testes com o Compilador *jmlc*

Nesta etapa do JMLOK, o oráculo para os testes é gerado pelo compilador *jmlc*, que consiste basicamente em um compilador Java que adiciona asserções advindas de pré-condições, pós-condições e invariantes JML. Assim, o procedimento de decisão para o oráculo dos testes é, portanto, verificar as asserções adicionadas pelo *jmlc*, interpretando as verificações como sucesso ou falha após a execução dos testes. Se uma asserção é violada, então uma exceção específica é lançada. Vale salientar que algumas estruturas JML não são compiladas pelo *jmlc*, como quantificadores generalizados (`\min`, `\max`, `\sum` e `\product`) e especificações JML de corpo de métodos (invariantes de laço), por limitação da implementação atual do *jmlc*. Nós utilizamos a versão *jmlc* 5.6_rc4.

A Código-fonte 3.7 mostra como é um arquivo gerado pelo *jmlc*. Ignoramos detalhes do arquivo para simplificar a visualização. Note a transformação das asserções em estruturas *try-catch* e estruturas de controle *if*, ou seja, checadores das asserções em tempo de execução.

Código Fonte 3.7: Trecho de código gerado pelo *jmlc* para o construtor da classe `Pessoa`.

```
1     public Pessoa() {
2     {...}
3     // checar pré-condição
4     if (JMLChecker.isActive(JMLChecker.PRECONDITION)) {...}
5     boolean rac$ok = true;
6     boolean rac$inv = true;
7     try {
8       internal$$init$();
9       // checar pós-condição normal
10      if (JMLChecker.isActive(JMLChecker.POSTCONDITION) &&
          rac$dented()) {...}
11    }
12    catch (JMLEntryPreconditionError rac$e) {...}
13    catch (JMLAssertionError rac$e) {...}
14    catch (java.lang.Throwable rac$e) {
15      rac$inv = false;
16      try {
17        // checar pós-condição excepcional
18        if (JMLChecker.isActive(JMLChecker.POSTCONDITION) &&
            rac$dented()) {...}
19      }
20      catch (JMLAssertionError rac$e1) {...}
21    }
22    finally {
23      if (rac$ok && rac$inv) {
24        // checar invariante
25        if (JMLChecker.isActive(JMLChecker.INVARIANT) && rac$dented
            ()) {...}
26      }
27  {...}
28    }
29  }
```

Pode ocorrer basicamente o lançamento de quatro tipos principais de exceções: `JMLPreconditionError`, `JMLPostconditionError`, `JMLInvariantError`, que indicam violação à pré-condição, pós-condição e invariante, respectivamente, e `JMLEvaluationError`, lançada quando uma exceção ocorre em meio à execução da avaliação das asserções. A Figura 2.1 ilustra mais tipos de exceções que podem ocorrer quando o `bytecode` instrumentado é executado. Violações às pré-condições do tipo `JMLEntryPreconditionError` (subtipo de `JMLPreconditionError`) não indicam não-conformidade, pois alguns testes podem passar valores de entrada que não satisfazem algumas pré-condições. Este tipo de violação é denominado de *meaningless* [8]. Desta forma, nós consideramos não-conformidades as faltas existentes no código que são evidenciadas através do lançamento de exceções referentes às pós-condições, aos invariantes e às pré-condições, sendo que tais exceções referentes às pré-condições – `JMLInternalPreconditionError` – são lançadas através das chamadas internas entre métodos sob teste. O teste da seção anterior (Código-fonte 3.6) detecta uma não-conformidade referente à violação da pós-condição, lançando, assim, uma exceção do tipo `JMLPostconditionError`.

É importante ressaltar que a execução do Randoop diretamente no *bytecode* gerado pelo *jmlc* não detectaria não-conformidades, pois para gerar oráculo para uma sequência de chamadas, o Randoop executa a sequência e utiliza o resultado dela para criar asserções. Para o código presente no Código-fonte 3.5 compilado com *jmlc*, o Randoop geraria o teste abaixo (Código-fonte 3.8) ao invés do teste exibido no Código-fonte 3.6:

Código Fonte 3.8: Teste gerado pelo Randoop com o código compilado pelo *jmlc*

```
1  public void test5() throws Throwable {
2    Pessoa var0 = new Pessoa();
3    var0.setAltura(10);
4    try {
5      var0.setAltura(1);
6      fail("Exceção esperada");
7    } catch (JMLInternalNormalPostconditionError e) {}
8  }
```

Note que o Randoop considera o lançamento da exceção como correto. E apenas quando

ela não for lançada, o teste falhará. Por isso, ele não detectaria a não-conformidade.

3.2.3 Etapa 3: Execução dos Testes pelo JUnit

Nesta etapa, o JMLOK executa a coleção de testes gerados sobre o bytecode instrumentado com as asserções. A ferramenta utilizada para executar os testes é o JUnit, uma vez que os testes gerados pelo Randoop estão no formato específico para essa ferramenta. Assim, após a execução os resultados são divididos em *meaningless* e *relevants*, denotando, respectivamente, violações às pré-condições e violações às pós-condições e aos invariantes. A Figura 3.4 mostra, em alto nível de abstração, como ocorre todo o processo no JMLOK para verificar conformidade na classe Pessoa (Código-fonte 3.5).

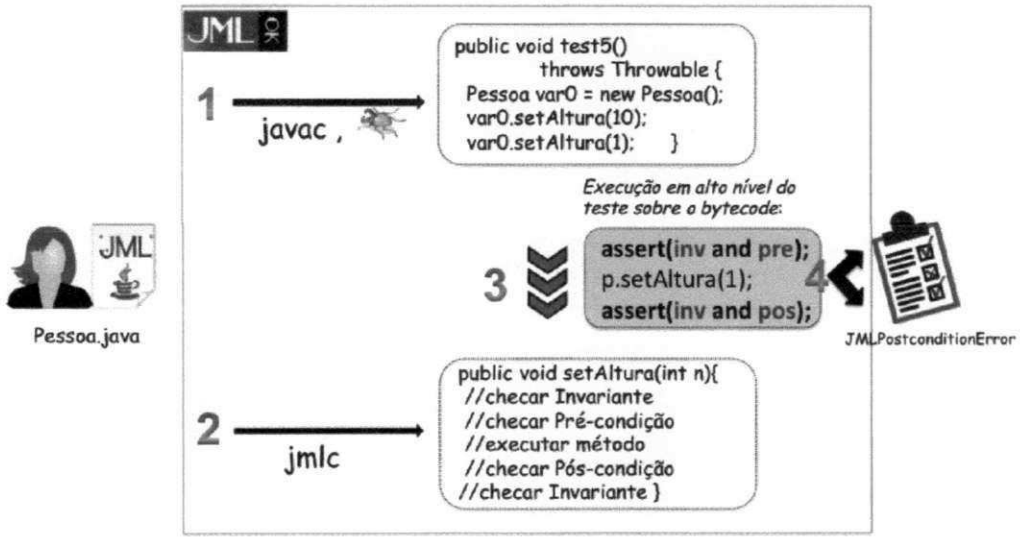


Figura 3.4: Exemplo de Execução do JMLOK na classe `Pessoa`. A execução da classe no JMLOK identificou uma não-conformidade de pós-condição.

A execução do teste (Código-fonte 3.6) na classe `Pessoa` detecta uma não-conformidade – `JMLInternalNormalPostconditionError` – referente à violação à pós-condição do método `setAltura`.

3.2.4 Etapa 4: Relatório com o Resultado da Execução dos Testes

Por fim, após a geração dos testes de unidade e a produção dos oráculos dos testes, tais testes são executados e, então, uma tela contendo os resultados obtidos é exibida (Figura

3.5). Na tela inicial da ferramenta, o usuário informa os diretórios do projeto Java/JML e clica no botão JMLOK. Logo após, a ferramenta informa os testes que podem indicar não-conformidades e as exceções lançadas. Na tela de resultados, a ferramenta exibe a lista de testes que detectaram possíveis não-conformidades, incluindo os resultados *meaningless*. O usuário pode clicar no teste para ver qual foi a exceção JML lançada e a sua mensagem, bem como o código do teste que revela o problema.

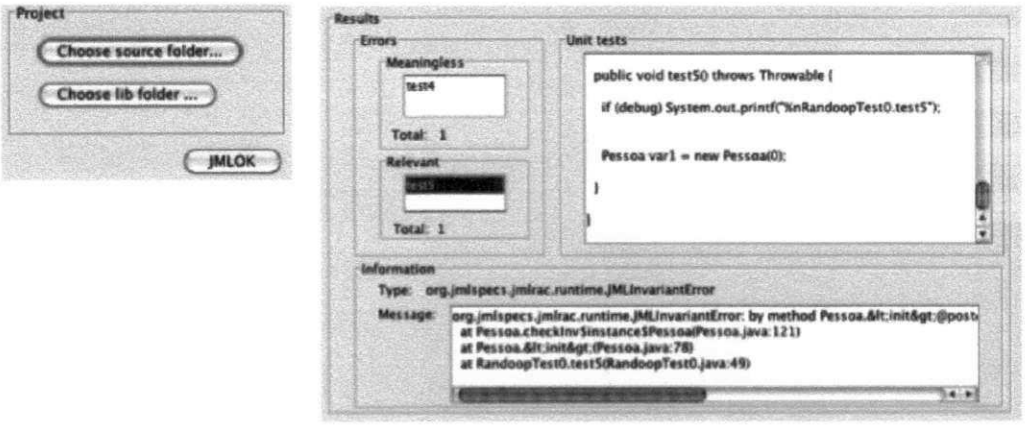


Figura 3.5: Telas do JMLOK.

3.2.5 Arquitetura

A arquitetura do JMLOK é baseada no padrão *Pipes-and-Filters* [6], onde *Filters* são as ferramentas utilizadas no processamento do JMLOK para uma entrada Java/JML: Randoop, *jmlc* e JUnit (execução dos testes). E os *Pipes* são as conexões entre estas ferramentas. A Figura 3.6 mostra esta arquitetura.

3.2.6 Limitações

Como a ferramenta de geração de testes acoplada ao JMLOK é de caráter aleatório, nós assumimos o risco de que uma dada coleção de testes não seja eficaz para encontrar não-conformidades, pois é impraticável testar todas as possibilidades de entradas de um sistema. Além disso, o limite de valores gerados para as entradas usadas em chamadas a métodos pode ter desvantagem, pois, dependendo do sistema testado, muitas exceções de erro de pré-condição podem ser lançadas, indicando muitos falsos positivos. Ademais, temos, por

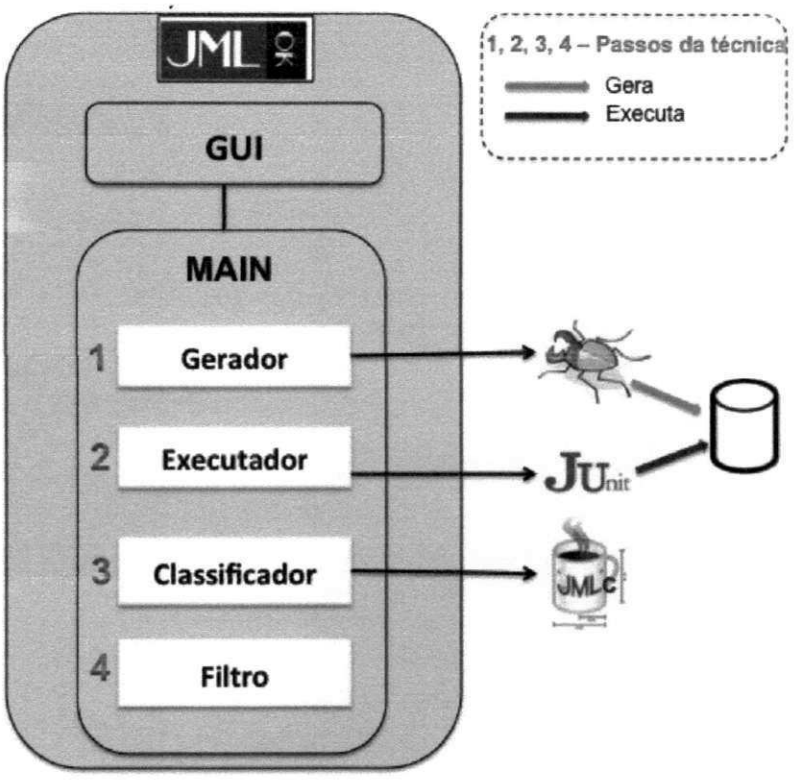


Figura 3.6: Arquitetura do JMLOK.

enquanto, uma ferramenta prototipal que pode ser atualizada com melhorias, tais como: indicação da origem das não-conformidades, apresentação na tela de medidas como porcentagem de cobertura dos testes, número total de linhas de especificação JML e KLOC, entre outras informações que possam melhorar a análise das não-conformidades pelo especialista.

Capítulo 4

Avaliação

Neste capítulo, descrevemos o estudo de caso que foi realizado com o intuito de avaliar a técnica proposta através da execução da ferramenta JMLOK. A avaliação foi realizada sobre grupos de programas com o intuito de encontrar não-conformidades e analisar os resultados extraídos pela ferramenta. JMLOK foi executado em 5 unidades experimentais, que são repositórios de software contendo programas Java/JML (Seção 4.1). Todos os grupos de programas totalizam 18 KLOC e 5 KJML. Baseado nas exceções lançadas pelo compilador *jmlc*, foram encontradas 29 não-conformidades em todos os programas (Seção 4.3). A discussão sobre os resultados obtidos nos experimentos é mostrada na Seção 4.4. Realizamos também uma comparação entre o JMLOK e a ferramenta JET [8] (Seção 4.5.1) com o intuito de comparar os resultados obtidos e verificar o número de não-conformidades encontradas por cada ferramenta. Tentamos ainda realizar avaliação na biblioteca de tipos JML, JMLModels, com o objetivo de verificar se essa biblioteca, muito utilizada pela comunidade JML, está implementada em conformidade com a sua especificação. Relatamos nossa experiência na Seção 4.5.2. Apresentamos algumas ameaças à validade do JMLOK e da análise realizada nos resultados obtidos na Seção 4.6. Antes da descrição de tais experimentos, a Seção 4.2 apresenta a configuração do ambiente utilizado nas avaliações. Todos os dados da avaliação podem ser encontrados no site da ferramenta[1].

[1] http://www.dsc.ufcg.edu.br/~spg/jmlok

4.1 Caracterização das Unidades Experimentais

Utilizamos cinco unidades experimentais para avaliar o JMLOK com o intuito de detectar não-conformidades. São repositórios de software contendo programas Java/JML que são largamente especificados e alguns usados em aplicações reais. Os repositórios analisados foram *Samples* (Seção 4.1.1), *JAccounting* (Seção 4.1.2), *Bomber* (Seção 4.1.3), *TransactedMemory* (Seção 4.1.4) e *Healthcard* (Seção 4.1.5).

A Tabela 4.1 mostra a caracterização de todos os repositórios avaliados. A coluna *Repositório* descreve o nome do repositório avaliado. A coluna *Classes* indica o número de classes presentes no repositório. A coluna *Métodos* mostra o número total de métodos contidos no repositório. A coluna *LOC* indica o número de linhas de código do repositório. E a coluna *LJML* mostra o número de linhas de especificação JML presentes no repositório.

Tabela 4.1: Caracterização das Unidades Experimentais.

Repositório	Classes	Métodos	LOC	LJML
Samples	41	231	1833	2022
JAccounting	82	683	6586	311
Bomber	44	557	6458	255
TransactedMemory	28	83	1806	335
HealthCard	15	191	1752	2410
Total	**210**	**1745**	**18435**	**5333**

4.1.1 Repositório Samples

O repositório *Samples*, hospedado no site JML [2] tem como principal propósito demonstrar como especificações JML podem ser escritas. A maioria dos exemplos foi escrita por especialistas na linguagem. Este repositório inclui vários exemplos que são separados de acordo com a regra de negócio implementada. Dentre os exemplos, avaliamos 9 deles, incluindo: *dbc*, *digraph* (grafos direcionados), *dirobserver* (*callbacks*), *jmlkluwer* [19], *jmltutorial* [22], *misc* (amostras diversas de especificações JML), *reader* (abstrações de I/O), *sets* and *stacks*. Estes exemplos totalizam 1833 LOC e 2022 LJML. Observa-se que existem mais linhas de

[2] http://www.eecs.ucf.edu/~leavens/JML/examples.shtml

especificação do que de código, o que se justifica justamente pelo intuito a que se destinam esses exemplos: explicar a linguagem JML.

4.1.2 Repositório JAccounting

O *JAccounting*, que possui 6 KLOC e 331 LJML, foi um repositório avaliado no projeto do compilador *ajmlc*. É um sistema Java/JML de contabilidade. A classe concreta `Account` é a classe principal que gerencia a execução de toda aplicação, por isso é a classe configurada sob teste.

4.1.3 Repositório Bomber

O repositório *Bomber*, que possui 6 KLOC e 255 LJML, também foi avaliado no compilador *ajmlc*. Ele é um sistema para dispositivos móveis. Neste caso, todas as classes concretas foram testadas, pois não havia uma classe principal gerenciando a sua execução.

4.1.4 Repositório TransactedMemory

TransactedMemory [38] (~2 KLOC e 335 LJML) é uma funcionalidade específica para a API Javacard. Ela foi originalmente especificada na linguagem Z e implementada na linguagem C [35]. Este programa é a tradução do projeto original para o contexto Java/JML. Este repositório possui três subpacotes que são três versões para esta tradução. A primeira versão é uma tradução abstrata, a segunda é uma implementação concreta e a terceira é uma versão evoluída da segunda versão. Executamos as duas últimas versões no JMLOK por possuírem implementação de classes concretas.

4.1.5 Repositório HealthCard

O repositório *Healthcard* (2 KLOC e 2 KLJML) foi um estudo de caso de dissertação de mestrado [41]. Esse repositório representa uma aplicação para *smartcard* que gerencia e armazena históricos médicos. *Healthcard* possui 8 funcionalidades: *appointments*, *allergies*, *cardServices*, *diagnostics*, *medicines*, *personal*, *treatments* e *vaccines*. Executamos JMLOK em todas as classes para simular um real histórico de um usuário.

4.2 Configuração do Ambiente

Os experimentos foram realizados em máquina com processador Intel Core 2 Duo com 2.13 GHz e 4 GB SDRAM, MAC OS X Versão 10.5.8 e Java 6. Coletamos a cobertura alcançada pelos testes nas avaliações através do plugin *eclEmma* [3], versão 2.2.0. Para coletar o número de linhas de código (LOC) dos programas avaliados, utilizamos o plugin *metrics* [4], versão 1.3.6. Implementamos uma aplicação para coletar o número de linhas de especificação JML (LJML). No JMLOK, usamos a versão 1.3.2 do Randoop, versão 5.6_rc4 do *jmlc* e versão 3.8 do JUnit. Está em andamento a evolução do JMLOK para novas versões das ferramentas utilizadas em sua arquitetura.

Para a geração de testes, o valor *default* no JMLOK para o tempo limite é fixado em 10 segundos. Para configurar esse valor no JMLOK, realizamos um experimento piloto simples para correlacionar o tempo limite de geração dos testes e LOC em pequenos exemplos e variamos o tempo limite em 10, 30 e 60 segundos e verificamos duas variáveis: a cobertura de instrução dos testes gerados e a quantidade de não-conformidades encontradas para cada tempo. Após essa análise, verificamos que a cobertura não se modificou em nenhum dos tempos limites, mantendo-se em 100% de cobertura e que o número de não-conformidades encontradas manteve-se o mesmo. Desta forma, decidimos deixar como padrão o tempo de 10 segundos para o tempo limite de geração de testes para quaisquer classes na ferramenta. Mas o valor do tempo limite de geração de testes pode ser alterado pelo usuário nas configurações da ferramenta. Algumas ferramentas que também utilizam Randoop dentro de sua estrutura [43] [42] também obtiveram resultados satisfatórios com pequenos valores de tempo limite de geração.

Após a execução de cada repositório, coletamos o tempo total de execução do JMLOK. A Tabela 4.2 indica o tempo de processamento do JMLOK (em segundos) em cada programa avaliado. Utilizamos o tempo padrão para cada programa, por ser um tempo satisfatório para gerar testes com cobertura razoável. Repositórios como *Samples* (Seção 4.1.1) e *Transacted Memory* (Seção 4.1.4) utilizam mais que o valor padrão de tempo limite, pois são subdivididos em outros pacotes, os quais foram testados separadamente. No total, entre geração dos testes, compilação e execução, o JMLOK despendeu aproximadamente 7 minutos.

[3] http://www.eclemma.org/

[4] http://metrics.sourceforge.net/

Tabela 4.2: Tempo de Execução no JMLOK. Esta tabela indica o tempo (em segundos) demandado para processar cada etapa do JMLOK em cada experimento. Alguns repositórios como o *Samples* e *TransactedMemory* foram subdivididos em subpacotes, onde cada subpacote foi configurado com o tempo *default* de geração de testes.

Repositório	Tempo Limite(s)	*jmlc* (s)	Execução Testes(s)	Total (s)
Samples	90	90	2	**182**
JAccounting	10	30	0.5	**40.5**
Bomber	10	20	0.2	**30.2**
Transacted Memory	30	45	0.5	**75.5**
Healthcard	10	45	0.5	**55.5**
Total	**150**	**230**	**3.7**	**383.7**

4.3 Resultados obtidos

Após executar o JMLOK em cada repositório, coletamos os resultados obtidos e verificamos as não-conformidades encontradas. Descrevemos nas próximas seções os resultados de cada repositório avaliado. A Tabela 4.3 sumariza os resultados obtidos.

4.3.1 Repositório Samples

Após executar o JMLOK, foram encontradas sete não-conformidades em quatro exemplos. Cinco das não-conformidades foram identificadas como violações às pós-condições pelo lançamento da exceção `JMLNormalPostconditionError`. Os exemplos onde encontramos estas não-conformidades foram *dbc*, *jmlkluwer* e *misc*. Uma das não-conformidade foi identificada através do lançamento da exceção do tipo `JMLExceptionalPostconditionError`, no exemplo *stacks*. E a última não-conformidade é do tipo `JMLInvariantError` também no pacote *stacks*.

Por exemplo, o pacote *dbc* tem duas classes concretas, `Rectangular` e `Polar`, que implementam números complexos de duas maneiras diferentes. A interface `Complex`, superclasse de todas as classes, possui em algumas especificações dos seus métodos, referências à biblioteca JMLModels, mais precisamente, ao tipo JMLDouble. O método referenciado é `JMLDouble.approximatelyEqualTo` que realiza a comparação entre dois valores. No Código-fonte 4.1 temos a especificação do método `imaginaryPart()` como

Tabela 4.3: Resultados Obtidos através da Execução do JMLOK em Todos os Repositórios. A coluna *Repositório* descreve o nome do repositório avaliado. A coluna *N. Testes* mostra o número de testes gerados na ferramenta. A Coluna *Cobertura* indica a cobertura de instruções alcançada pelos testes gerados. As colunas *Classe* e *Método* mostram, respectivamente, a classe e o método onde a não-conformidade foi encontrada. Método *<init>* indica construtor. E a coluna *Não-conformidade* indica o tipo de exceção lançada que indica a violação à especificação.

Repositório	N. Testes	Cobertura	Classe	Método	Não-conformidade
Samples	1566	92%	Rectangular	realPart	JMLPostconditionError
			Rectangular	imaginaryPart	JMLPostconditionError
			Polar	imaginaryPart	JMLPostconditionError
			EqualsN	<init>	JMLPostconditionError
			PriorityQueue	getLevelList	JMLPostconditionError
			BoundedStack	<init>	JMLPostconditionError
			BoundedStack	<init>	JMLInvariantError
JAccounting	1705	91%	Account	getCurrency	JMLEvaluationError
			Account	getName	JMLPostconditionError
Bomber	946	7.8%	Explosion	<init>	JMLInvariantError
			Building	<init>	JMLInvariantError
			FlakSmoke	getBoundingBox	JMLPostconditionError
TransactedMemory	717	32%	AbstractTransactedMemory	ANewTag	JMLPostconditionError
			TransactedMemory	<init>	JMLInvariantError
			DTagData	<init>	JMLInvariantError
			GenGenbyte	<init>	JMLInvariantError
HealthCard	552	90%	Medicine_Impl	<init>	JMLInvariantError
			Personal_Impl	<init>	JMLInvariantError
			Treatment_Impl	setMedicalRecommendation	JMLPostconditionError
			Treatment_Impl	<init>	JMLInvariantError
			Treatment_Impl	setDiagnosticID	JMLInvariantError
			Diagnostic_Impl	<init>	JMLInvariantError
			Vaccines_Impl	removeVaccine	JMLPostconditionError
			Vaccines_Impl	getVaccineDesignation	JMLPostconditionError
			Vaccines_Impl	validateVaccinePosition	JMLPostconditionError
			Vaccine_Impl	setDesignation	JMLInternalPreconditionError
			Common	getVaccinationDate	JMLHistoryConstraintError
			Common	toString	JMLHistoryConstraintError
			Allergies_Impl	removeAllergy	JMLPostconditionError

exemplo.

Código Fonte 4.1: Especificação JML do método `imaginaryPart()` da interface `Complex`.

```
1   /*@ ensures JMLDouble.approximatelyEqualTo(
2     @ \result,
3     @ magnitude()*StrictMath.sin(angle()),
4     @ tolerance);
5     @*/
6     double imaginaryPart();
```

Note que o método `JMLDouble.approximatelyEqualTo` recebe três parâmetros e retorna um booleano indicando se os dois primeiros argumentos são iguais dada uma taxa de tolerância (terceiro argumento). A chamada para o método `imaginaryPart()` no teste Código-fonte 4.2 deveria retornar valor `zero` como resultado, pois o construtor da classe `Rectangular` assume que o valor `-1.0d` dado como entrada, é o valor da parte real do número complexo, e por *default*, a parte imaginária receberia valor `zero`.

Código Fonte 4.2: Teste gerado pelo JMLOK para a classe `Rectangular`

```
1 public void test25() throws Throwable {
2     Rectangular var1 = new Rectangular((-1.0d));
3     double var2 = var1.imaginaryPart();
4 }
```

No entanto, durante a execução do teste, ocorre um erro de pós-condição devido a aproximações imprecisas retornadas pelo `JMLDouble.aproximatelyEqualTo`.

4.3.2 Repositório JAccounting

Após executar o JMLOK sobre o *JAccounting*, encontramos duas não-conformidades. As exceções lançadas foram `JMLEvaluationError` e `JMLInternalNormalPostconditionError`. Após investigação manual, descobrimos que as duas não-conformidades estão relacionadas a falhas similares. O construtor `Account` não inicializa nenhum atributo da classe. Assim, quando o construtor é invocado, os atributos não inicializados recebem valores nulos por *default* na semântica Java. É importante ressaltar que a classe `Account` estava especificada com `/*@nullable_by_default@*/`, que permite valores nulos.

Código Fonte 4.3: Teste gerado para testar o repositório `JAccounting`.

```
1 public void test3() throws Throwable {
2
3     Account var0 = new Account();
4     var0.setCompanyKey((java.lang.Integer)10);
5     java.math.BigDecimal var3 = var0.getBalance();
6     java.lang.Integer var4 = var0.getParentKey();
```

```
7       java.lang.String var5 = var0.getCurrency();
8
9       assertNull(var3);
10      assertNull(var4);
11      assertNull(var5);
12  }
```

Assim, quando um teste como o do Código-fonte 4.3 instancia `Account` e logo depois faz uma chamada ao método `getCurrency` (Código-fonte 4.4), a não-conformidade do tipo `JMLEvaluationError` é lançada, pois quando esse método retorna o valor *null* e durante a checagem da pós-condição é realizada a instrução `null.equals(null)`, então é lançada a exceção `NullPointerException` (`JMLEvaluationError` indica que uma exceção é lançada durante a execução da avaliação da especificação).

Código Fonte 4.4: Método `getCurrency` do repositório JAccounting.

```
1   //@ requires true;
2   //@ ensures \result.equals(currency);
3   public String getCurrency() {
4       return currency;
5   }
```

No caso da exceção lançada durante a chamada do método `getName` (Código-fonte 4.5), a primeira pós-condição `ensures \result instanceof java.lang.String` retorna `false`, pois o resultado da instrução `null instanceof <object>` sempre retorna falso, o que viola a pós-condição do método.

Código Fonte 4.5: Método `getName` do repositório JAccounting.

```
1   //@ ensures \result instanceof java.lang.String;
2   //@ ensures \result.equals(this.name);
3   public String getName() {
4       return name;
5   }
```

4.3.3 Repositório Bomber

A execução do JMLOK no repositório *Bomber* encontrou três não-conformidades. Duas das não-conformidades são violações ao invariante e a outra é uma violação à pós-condição. As duas violações ao invariante referem-se à atribuição de valores nulos a alguns atributos após a invocação de construtores de duas classes(`Explosion` e `Building`), semelhantemente ao que acontece no *JAccounting*. Mas nestes casos, as não-conformidades ocorreram pelo fato de JML não permitir, por *default*, valores nulos para os atributos da classe. No caso do repositório *JAccounting*, a classe `Account` estava especificada com `/*@nullable_by_default@*/` que permite valores nulos.

A outra não-conformidade acontece quando há chamadas ao método `getBoundingBox` na classe `FlakSmoke`. Este método retorna valor `null` o que provoca uma violação à pós-condição, que proíbe essa situação.

4.3.4 Repositório TransactedMemory

No repositório *TransactedMemory* encontramos quatro não-conformidades. Uma delas relacionada ao lançamento de uma exceção inesperada durante a pós-condição, e as últimas três relacionadas ao lançamento de exceções que indicam violações aos invariantes. Uma das violações ao invariante ocorre quando o construtor (Código-fonte 4.6) da classe `TransactedMemory` é invocado. Note que ele realiza uma instanciação da classe `DTagData` (Código-fonte 4.7) passando como parâmetro o valor `0` para o atributo `size`. Entretanto, existe um invariante que estabelece que `size` pode receber somente valores maiores que zero. As outras duas não-conformidades aconteceram na terceira versão do *TransactedMemory*. As duas tiveram causas semelhantes, pois os construtores das classes (`DTagData` e `GenGenbyte`) não possuem especificação de pré-condição, permitiam a entrada de muitos valores o que ocasionava violações aos invariantes em algumas situações.

Código Fonte 4.6: Construtor da classe *TransactedMemory*.

```
1 public TransactedMemory()
2  { short i;
3    for (i = 0; i < TSIZE; i++) ddata[i] = new DTagData (false,0,false)
       ;
```

```
4     for (i = 0; i < MSIZE; i++) dmem[i] = new DPage (false, 0, 0, 0,
          0, 0);
5   }
```

Código Fonte 4.7: Classe `DTagData` do repositório *TransactedMemory*.

```
1  public class DTagData{
2
3   public boolean tagInUse;
4
5   public int size;
6  //@ invariant 0 < size;
7
8   public boolean committed;
9
10  public DTagData(boolean tagInUse, int size, boolean committed)
11  { this.tagInUse = tagInUse;
12    this.size = size;
13    this.committed = committed;
14  }
15 }
```

4.3.5 Repositório HealthCard

No *Healthcard*, JMLOK encontrou treze não-conformidades (Tabela 4.3). Cinco delas relacionadas ao lançamento da exceção `JMLInvariantError`, mais cinco relacionadas ao lançamento da exceção `JMLPostconditionError`, uma não-conformidade relacionada ao lançamento da exceção `JMLInternalPreconditionError` e mais duas relacionadas ao lançamento da exceção `JMLHistoryConstraintError`. Essas últimas não-conformidades, ocorridas por causa do lançamento de exceção indicando erro na *history constraint*, aconteceram somente neste repositório. A classe `Common` possui algumas cláusulas *constraints* que especificam o formato do atributo `date`, como exemplo, uma especificação no Código-fonte 4.8. Quando o método `getVaccinationDate` (Código-fonte 4.9)

é chamado, então os valores instanciados para data são verificados com relação ao modelo especificado nas *constraints*. Assim, alguns cenários de teste gerados identificaram essa situação, a qual é causada pela entrada de valores proibitivos nas classes que herdam essa especificação.

Código Fonte 4.8: *Constraint* especificada no repositório *HealthCard*.

```
1 //Constraint para meses(i.e., 1 até 12)
2 /*@ constraint (byte) 0x00 <= getByte(date_model,0)
3   @ && getByte(date_model,0) <= (byte) 0x0C;
4   @*/
```

Código Fonte 4.9: Especificação do método `getVaccinationDate`.

```
1 /*@ assignable commons.CardUtil;
2   @ ensures toJMLValueSequence(\result).equals(date_model);
3   @*/
4     public byte[] getVaccinationDate ();
```

4.4 Discussão dos Resultados Obtidos

Para analisar os resultados obtidos pelo JMLOK, realizamos uma análise manual. A análise começou a partir do momento do lançamento da exceção. Com o nome da exceção em mãos, então averiguamos o teste que conseguiu gerar o cenário. Rastreamos os caminhos passados a partir do teste até chegar à origem do problema. Discutimos estes problemas nesta seção. Para entender melhor as causas dos problemas, classificamos as falhas em quatro categorias e depois as ranqueamos. As próximas seções detalham estas categorias.

4.4.1 Categorização das Falhas

Após a obtenção dos resultados, analisamos manualmente as faltas ocorridas no código. Para rastrear as origens dos problemas seguimos alguns passos.

Passo 1. A ferramenta JMLOK divide os testes que causam lançamentos de exceções JML em dois tipos, *meaningless* e *relevants*. A partir desse ponto, o escopo das exceções

lançadas é diminuído, visto que *meaningless* são descartadas. Então, a partir dos *relevants*, verificamos quais os tipos de exceções foram lançadas. É comum que muitos testes desencadeiem a ocorrência da mesma falha. Assim, dividimos os testes por falhas encontradas.

Passo 2. Analisamos os testes e depuramos os caminhos dentro do programa até chegar à origem do problema.

Passo 3. Categorizamos as faltas ocorridas e mostramos quais as principais causas das não-conformidades ocorridas. Para classificar verificamos o seguinte *checklist*:

(a) O método possui pré-condição explicitamente especificada? Em caso falso, então indicamos o tipo de falta como "A Pré-condição é Fraca", pois, por *default*, JML atribui `@requires true` quando não há especificação explícita, deixando, então, que todos os valores de entrada de um método sejam aceitos e o método executado. Em caso verdadeiro, ainda verificamos se todos os parâmetros de entrada são especificados. Em caso falso, então indicamos também que o tipo de falta é "A Pré-condição é Fraca". Vale ressaltar que avaliamos quando um método herda uma especificação de outra classe.

(b) O tipo de erro é de pós-condição? Em caso verdadeiro, indicamos o tipo de falta como "O Resultado do Método [M] Viola a Pós-condição".

(c) O tipo de erro é de invariante? Em caso verdadeiro, indicamos o tipo de falta como "O Resultado do Método [M] Viola Invariante".

A partir desse *checklist*, uma análise criteriosa foi realizada para cada resultado obtido na avaliação realizada. A partir dessa análise, conseguimos classificar as categorias em quatro tipos (Seção 4.4.3). A Tabela 4.4 mostra cada classificação dada.

Tabela 4.4: Categorização das Falhas.

Categoria	Descrição
(1)	Não-conformidades advindas de cláusulas *non_null* implícitas, por *default*, pelo JML.
(2)	Não-conformidades advindas de pré-condições fracas ou pela falta de pré-condições.
(3)	Não-conformidades advindas de erros de implementação.
(4)	Não-conformidades advindas de erros de especificação.

4.4.2 Categorização das Falhas nos Resultados Obtidos

A partir do *checklist*, a análise realizada com os resultados obtidos na avaliação nos permitiu classificá-los em quatro categorias que seguem:

(1) Não-conformidades advindas de cláusulas *non_null* implícitas, por default, pelo JML

Nessa categoria se encaixam as duas não-conformidade encontradas no repositório *Bomber*. Como explicamos na Seção 4.3.3, o construtor inicializa somente alguns atributos da classe. Assim, Java inicializa automaticamente os demais atributos com *null*. Entretanto, a cláusula *non_null* está implícita na semântica JML, causando, assim, o lançamento de exceções que violam invariantes, pois valores nulos são atribuídos a alguns atributos das classes `Explosion` e `Building`, por não instanciação em seus construtores. Além dessas não-conformidades, também se encaixam duas das não-conformidades encontradas no repositório *HealthCard*.

Para resolver problemas semelhantes, algumas medidas podem ser tomadas, de acordo com o contexto do programa. O especificador pode resolver esse problema colocando simplesmente a cláusula *nullable* na classe afetada. Enquanto que o desenvolvedor pode resolver o problema instanciando todos os atributos da classe no construtor. É difícil indicar a culpa pelo surgimento do problema. Cada projeto possui uma regra de negócio e uma metodologia de desenvolvimento a ser seguida. A melhor decisão a ser tomada dentro da conformidade com os requisitos, no menor tempo e com o menor custo para o projeto depende de cada realidade.

(2) Não-conformidades advindas de pré-condições fracas ou pela falta de pré-condições

A maioria das não-conformidades encontradas no repositório *HealthCard* (dez delas) foram classificadas nesta categoria. Nos repositórios *Samples* e *TransectedMemory* sete não-conformidades também se encaixaram nessa categoria. A partir da análise realizada e nossa experiência, consideramos esta categoria uma das mais fáceis de ser identificada, uma vez que basta analisar, à princípio, somente a pré-condição estabelecida no método. Quando uma pré-condição não é especificada em um método, então JML, por *default* a especifica

como `true`, assim, todas as entradas são aceitas pelo método, mesmo havendo requisitos de restrições para isso.

Uma não-conformidade detectada no *HealthCard* ocorreu devido ao lançamento da exceção `JMLInternalPreconditionError`. Diferentemente da exceção de violação à pré-condição `JMLEntryPreconditionError` (*Meaningless*), este tipo de exceção indica uma não-conformidade. Veja o Código-fonte 4.10 que mostra um teste realizado. Note que o método `setVaccineDesignation` recebe duas entradas (um `short` e um array do tipo `byte`).

Código Fonte 4.10: Teste gerado pelo JMLOK para o pacote `HealthCard`

```
1 public void test94() throws Throwable {
2     Vaccines_Impl var0 = new Vaccines3_Impl();
3     var0.setVaccineDesignation((short)1, [0,0,0,0,0,0]);
4 }
```

As duas entradas do método denotam, respectivamente, o índice do array e um array de designações de vacina que devem ser atualizadas. Dentro da implementação do método `setVaccineDesignation` (Código-fonte 4.11) existe uma chamada para o método `setDesignation` para realizar esta atualização.

Código Fonte 4.11: Método `setVaccineDesignation` do pacote `HealthCard`

```
1 public void setVaccineDesignation (short position,
2                                    byte[] designation)
3                     throws RemoteException, UserException {
4         vaccines[position].setDesignation(designation);
5 }
```

Assim, a pré-condição do método `setDesignation` estabelece que o tamanho do array seja igual a quatro, mas o teste faz uma chamada com um array de tamanho igual a seis. Essa exceção foi lançada, pois foi uma invocação interna ao método `setDesignation` que gerou a não-conformidade.

Desta forma, esse erro foi classificado nesta categoria, pois o método `setVaccineDesignation` não possui nenhuma pré-condição que restrinja este tipo de situação.

Esses problemas relatados podem ser resolvidos tanto na especificação quanto na implementação. O especificador pode estabelecer todas as pré-condições necessárias, e assim, quando alguma entrada não é permitida, as exceções lançadas nesse caso serão descartadas, pois não configuram uma não-conformidade. Enquanto que o desenvolvedor pode implementar algumas estruturas de controle e fazer tratamentos de exceções.

(3) Não-conformidades advindas de erros de implementação

Uma das não-conformidades do repositório *Bomber* se encaixa nessa categoria. Pois o método `getBoundingBox` sempre retorna `null`, diferente da especificação dada para a pós-condição. A não-conformidade do repositório *TransactedMemory* relacionada ao lançamento da exceção indicando violação ao invariante, também se encaixa nessa categoria, pois o invariante estabelece um determinado conjunto de valores para um atributo, e na implementação do construtor, o atributo recebe um valor proibitivo. Por fim, uma das não-conformidades detectadas no repositório *HealthCard* também se encaixa nessa categoria, pois após a instanciação da classe `Personal_Impl`, uma exceção do tipo `JMLInvariantError` é lançada pois um atributo possui restrições de valores, ao tempo que esse mesmo atributo não é instanciado no construtor.

Os três problemas relatados acima são mais difíceis de serem detectados em comparação às categorias anteriores, pois não existe nenhuma ferramenta JML que realize *debug* na especificação, e dependendo da complexidade da especificação e da experiência do desenvolvedor, realizar uma análise manual para achar a origem do problema pode ser uma tarefa dispendiosa.

Nessa categoria se encaixam as duas não-conformidade encontradas no repositório *JAccounting*.

(4) Não-conformidades advindas de erros de especificação

Apesar de afirmar que em nossa técnica, a especificação realizada é dada como correta, após análise criteriosa encontramos um equívoco de especificação em uma não-conformidade encontrada no repositório *Samples*, no pacote *dbc*. A não-conformidade foi encontrada na interface `Complex`, ou seja, que não possui implementação, somente as especificações e as assinaturas dos métodos. Verificamos que a chamada ao método

`JMLDouble.approximatelyEqualTo` da API JMLModels dentro da especificação de um método, passa como argumento um valor para o qual a saída esperada não é obtida. O problema pode estar na pós-condição estabelecida, que pode ser muito forte. Ou até mesmo na implementação do método do `JMLDouble`. Independente da resposta para a solução, este tipo de erro é o mais difícil de ser encontrado, pois a especificação pode ser muito complexa para ser analisada manualmente, uma vez que JML não possui debugador. Ao mesmo tempo que este tipo de situação atesta a experiência do especificador pela complexidade da especificação realizada, ela também indica que, de qualquer forma, erros podem acontecer.

A Tabela 4.5 sumariza as não-conformidades encontradas e as respectivas categorias onde as origens dos problemas delas se encaixaram.

Tabela 4.5: Classificação dos Resultados Obtidos nas Categorias Propostas.

Samples	Rectangular	realPart	JMLPostconditionError	(4)
	Rectangular	imaginaryPart	JMLPostconditionError	(4)
	Polar	imaginaryPart	JMLPostconditionError	(4)
	EqualsN	<init>	JMLPostconditionError	(2)
	PriorityQueue	getLevelList	JMLPostconditionError	(2)
	BoundedStack	<init>	JMLPostconditionError	(2)
	BoundedStack	<init>	JMLInvariantError	(2)
JAccounting	Account	getCurrency	JMLEvaluationError	(3)
	Account	getName	JMLPostconditionError	(3)
Bomber	Explosion	<init>	JMLInvariantError	(1)
	Building	<init>	JMLInvariantError	(1)
	FlakSmoke	getBoundingBox	JMLPostconditionError	(3)
TransactedMemory	AbstractTransactedMemory	ANewTag	JMLPostconditionError	(2)
	TransactedMemory	<init>	JMLInvariantError	(3)
	DTagData	<init>	JMLInvariantError	(2)
	GenGenbyte	<init>	JMLInvariantError	(2)
Healthcard	Medicine_Impl	<init>	JMLInvariantError	(2)
	Personal_Impl	<init>	JMLInvariantError	(3)
	Treatment_Impl	setMedicalRecommendation	JMLPostconditionError	(1)
	Treatment_Impl	<init>	JMLInvariantError	(2)
	Treatment_Impl	setDiagnosticID	JMLInvariantError	(2)
	Diagnostic_Impl	<init>	JMLInvariantError	(2)
	Vaccines_Impl	removeVaccine	JMLPostconditionError	(2)
	Vaccines_Impl	getVaccineDesignation	JMLPostconditionError	(1)
	Vaccines_Impl	validateVaccinePosition	JMLPostconditionError	(2)
	Vaccine_Impl	setDesignation	JMLInternalPreconditionError	(2)
	Common	getVaccinationDate	JMLHistoryConstraintError	(2)
	Common	toString	JMLHistoryConstraintError	(2)
	Allergies_Impl	removeAllergy	JMLPostconditionError	(2)

4.4.3 Conclusões da Categorização

Ao realizar a análise, verificamos, informalmente, a natureza de cada repositório. Assim, vimos que mesmo em programas com especificações complexas (alguns exemplos do *Samples*, *HealthCard* e *TransactedMemory*) e outros não tão complexos como o *Bomber* e *JAccounting*, nós encontramos não-conformidades. Por isso, não podemos relacionar a ocorrência de não-conformidades com a complexidade da especificação. Enquanto que fatores como experiência do especificador e desenvolvedor possam desencadear problemas.

O repositório *Samples* foi desenvolvido com o intuito de demonstrar construções JML para a academia e alguns exemplos foram atualizados a última vez no ano de 2006, então podemos afirmar que a maturidade desses exemplos é incipiente, ao mesmo tempo que as não-conformidades encontradas não tem impacto forte e não requerem tanto rigor como a API JMLModels requer, por exemplo, pois é uma biblioteca amplamente utilizada pela comunidade que utiliza JML.

O repositório *TransactedMemory* ainda é uma aplicação difundida somente na academia e, portanto, não é utilizada comercialmente. Embora o JMLOK tenha indicado que este projeto necessita de melhorias para que seja implantada dentro da API Javacard, pois é uma tecnologia que requer segurança para ser executado em cartões inteligentes e dispositivos semelhantes, de pequenas dimensões de memória.

O repositório *HealthCard* é uma das aplicações que necessita de maiores ajustes e possui um forte impacto, por ser um sistema crítico, pois lida diretamente com o usuário final. As não-conformidades encontradas neste pacote indicam que muitas melhorias devem ser realizadas e a utilização do JMLOK ajudaria esta tarefa de maneira mais prática e rápida.

A Tabela 4.6 mostra o número de não-conformidades classificadas entre as quatro categorias de falhas. Note que a maioria das falhas advém de pré-condições fracas que permitem que valores proibitivos entrem na execução dos métodos, causando, assim, não-conformidades. Podemos concluir que a maioria das especificações estejam permitindo que não-conformidades ocorram, talvez pela experiência do especificador na linguagem de especificação, bem como não conhecimento total das regras de negócio do projeto. A utilização do JMLOK pode ajudar a escrita de especificações, uma vez que problemas podem ser resolvidos mais rapidamente ao longo do processo de especificação e implementação.

Tabela 4.6: Número de Não-conformidades em cada Categoria de Falha.

Categoria	Não-conformidades
(1)	4
(2)	17
(3)	5
(4)	3

4.5 Outras Avaliações

Como avaliações adicionais, realizamos um comparativo entre JMLOK e a ferramenta JET [8] (Seção 4.5.1) com o intuito de verificar a quantidade de não-conformidades que cada ferramenta conseguiria encontrar nas mesmas unidades experimentais. Realizamos esta comparação nos exemplos contidos no repositório *Samples* 4.1.1. Além disso, relatamos nossa experiência ao executar o JMLOK na biblioteca de tipos JML, JMLModels (Seção 4.5.2). Essa API é largamente utilizada pela comunidade que utiliza JML, então realizamos esta avaliação com o intuito de verificar se ela continha não-conformidades.

4.5.1 Comparativo entre JMLOK e JET

A ferramenta JET possui técnica similar a do JMLOK para verificar conformidade em programas Java/JML. A ferramenta também gera testes de unidades, executa-os e depois decide os resultados usando as especificações JML como oráculo para os testes. A geração dos testes utiliza algoritmos genéticos para construir automaticamente todos os dados de testes e, então, executa-os exercitando as asserções em tempo de execução. Algoritmos genéticos são eficazes no que se propõem a realizar, porém seu não-determinismo em gerar casos de testes e dados de testes não garantem, por exemplo, que uma mesma não-conformidade seja detectada em duas execuções consecutivas. A ferramenta JET segue duas metodologias: *specified-based*, que deriva dados de testes de pós-condições empregando um *constraint solver*; e, *program-based*, que deriva dados de testes somente dos programas, similar ao JMLOK. Esta avaliação foi realizada nos exemplos contidos no repositório *Samples*. Não conseguimos realizar essa avaliação em repositórios que precisavam de bibliotecas específicas (arquivos *.jar*) associadas para conseguir compilar, como o repositório *Bomber*, por exemplo. Por isso nos detivemos nos exemplos contidos no *Samples*.

A Tabela 4.7 ilustra a diferença de resultados entre as ferramentas.

Tabela 4.7: Comparação entre JMLOK e JET.

Pacote	JMLOK	JET
dbc	3	1 ou 2
jmlkluwer	1	0
misc	1	0
stacks	2	1
Total	**7**	**2 ou 3**

A comparação mostrou diferentes resultados em quatro subpacotes: *dbc*, *jmlkluwer*, *misc* e *stacks*. Sendo que no subpacote *dbc*, encontramos resultados diferentes no JET em execuções diferentes, em algumas execuções encontramos 1 não-conformidade, em outras encontramos 2 não-conformidades. Este fato pode ser explicado pelo não-determinismo dos algoritmos genéticos no momento da geração dos testes. Assim, a principal diferença entre as duas abordagens é a técnica usada para gerar os testes. A principal vantagem em JMLOK sobre o JET é que JMLOK possui uma técnica de geração de testes que cria conjuntos de sequências de chamadas que conseguem criar cenários de testes satisfatórios para encontrar não-conformidades. Ambas técnicas possuem características similares como a criação de testes *meaningless*.

4.5.2 Execução do JMLOK no JMLModels

A biblioteca de tipos JMLModels facilita a escrita de especificações, pois como muitas expressões matemáticas, tais como *sets* e *sequences*, são implementadas, então especificadores não necessitam especificar expressões matemáticas complexas. Como essa API é totalmente especificada com JML, tentamos executá-la com JMLOK. Entretanto, muitas expressões matemáticas não foram compiladas pelo *jmlc*. Mensagens de erro como o que segue,

```
File ../org/jmlspecs/models/JMLValueSequence.java, line
212, character 34 error:  Variable owner is not defined in
current context
```

ocorre com frequência em todas as classes da biblioteca. Neste caso, variáveis *model*,

ou seja, variáveis somente utilizadas para fins de especificação, não tinham correspondência alguma com variáveis concretas.

É importante salientar que JML é um projeto acadêmico e que por essa razão o aparato ferramental ainda não é totalmente estável. Por isso, dois fatores principais corroboram para este fato: o compilador *jmlc* não compila algumas expressões JML, ao tempo que o JML-Models é carregado de especificações complexas; e, a última atualização de algumas classes do JMLModels é do ano de 2006, enquanto que algumas expressões JML foram atualizadas até o ano de 2011 (data de atualização do manual JML), como por exemplo, em versões antigas do JML, especificações de métodos que ficavam localizadas em arquivos diferentes do código-fonte (*.jml*) tinham que começar com a cláusula *also*. Entretanto, na versão atual do JML, especificações de métodos em arquivos separados devem somente começar com *also* se o método for herdado de outro método que possua especificações. Havendo uma versão estável do JMLOK com o compilador *OpenJML*, voltaremos a testar JMLModels com o JMLOK.

4.6 Ameaças à Validade

Existem algumas ameaças à validade desta avaliação. As próximas Seções detalham cada tipo.

4.6.1 Validade Interna

Alguns fatores podem ameaçar a validade interna destes experimentos. A ferramenta utilizada pelo JMLOK para a geração dos testes é um deles. O processo para a geração dos testes é randômico e, desta forma, assumimos que alguns cenários de teste podem não ser criados. Na tabela 4.3 estão ilustradas as coberturas de instrução conseguidas durante a execução dos testes. Note que os projetos *Bomber* e *transactedMemory* não obtiveram cobertura maior que 50%. Embora, mesmo colocando um tempo maior para geração dos testes e obtendo maior cobertura, o número de não-conformidades continuaram os mesmos. Além disso, o tempo limite padrão de geração dos testes é outro fator ameaçador. Realizamos um estudo piloto simples para adequar a ferramenta a um tempo *default*. Esta variável fica ao critério do usuário, ele pode deixar o tempo padrão da ferramenta, ou configurar outro tempo.

Dependendo da natureza do programa a ser testado, o número de testes *meaningless* pode ser um fator ameaçador à validade interna. Uma vez que um número muito grande da ocorrência deles pode afetar o descobrimento de não-conformidades relevantes.

O compilador *jmlc* também representa uma ameaça à validade interna. Pois, apesar de ser uma ferramenta estável, atualmente ela está descontinuada. Então qualquer problema que venha a ocorrer no compilador não será corrigido na versão utilizada por nós. Existe um projeto atual, em desenvolvimento, que está migrando a versão atual do JMLOK para uma nova versão utilizando o compilador OpenJML.

A experiência do analista dos resultados obtidos na avaliação também representa uma ameaça à validade interna. Os repositórios advêm de outros projetos, então tivemos o primeiro contato durante a avaliação. Assim, o poder de entendimento das regras de negócio destes projetos pode afetar a validade das interpretações e conclusões obtidas.

4.6.2 Validade Externa

A categorização das falhas pode não ser totalmente utilizadas em outras realidades. Como foi realizada sob a ótica de poucos especialistas e manualmente, ela pode ser melhor estruturada e automatizada. Assim, estas categorizações podem não ser generalizadas para outras avaliações. Outras interpretações podem detalhar as categorias definidas, bem como inserir novas outras. Além disso, para a realidade de outras linguagens de especificação e programação, essas categorias podem ser reformuladas.

Capítulo 5

Conclusões

Apresentamos neste trabalho, uma técnica para detectar não-conformidades entre implementação e especificações formais por contratos através da geração e execução de testes de unidade. Geralmente, especificações formais são definidas em sistemas críticos que requerem que fatores como corretude, segurança e confiabilidade sejam asseguradas. Entretanto, no momento da implementação, *bugs* podem ser inseridos e sistemas inconsistentes com as especificações definidas podem ser construídos. Da mesma maneira, em sistemas especificados formalmente através de contratos, os desenvolvedores podem implementar código em não-conformidade com as suas pré-condições, pós-condições e invariantes. Como consequência, a maioria das não-conformidades são encontradas em momentos tardios do processo de desenvolvimento do software, o que implica em aumento do prazo e do custo do projeto. Assim, é importante que a detecção de não-conformidades seja realizada previamente, uma vez que quanto mais cedo uma não-conformidade seja detectada, maior é a qualidade do software.

Nossa técnica tem por objetivo encontrar não-conformidades de maneira rápida e em fases iniciais do desenvolvimento para que problemas sejam resolvidos mais rapidamente e a qualidade do software seja mantida. Utilizamos verificação dinâmica ao realizar a execução de testes de unidade. A técnica proposta é composta por uma entrada, três passos e uma saída. A entrada é uma aplicação especificada com contratos. No primeiro passo, conjuntos de testes de unidades são gerados. No segundo passo, os contratos são transformados em oráculo, alguns compiladores realizam essa transformação. No terceiro passo, os testes são executados e os resultados obtidos são comparados com os resultados esperados contidos no oráculo. Por fim, um relatório contendo as não-conformidades encontradas é gerado.

Implementamos a ferramenta JMLOK que realiza os passos do processo da técnica proposta automaticamente no contexto de programas Java/JML. A entrada para a ferramenta é, então, uma aplicação codificada em Java e especificada com JML. No primeiro passo, os testes são gerados pela ferramenta Randoop, que gera testes de unidade aleatória e automaticamente. Em seguida, o oráculo para os testes é gerado pelo compilador *jmlc* que transforma os contratos em asserções. Este compilador trabalha de forma semelhante ao compilador Java, gerando um *bytecode* a partir do código-fonte, a diferença é que o *bytecode* criado pelo *jmlc* é instrumentado com asserções. No terceiro passo, os testes gerados são executados no *bytecode* instrumentado. Quando ocorre uma violação à especificação, então uma exceção indicando o tipo de violação ocorrida é lançada. Após a execução, as não-conformidades encontradas são apresentadas.

Como uma maneira de garantir confiança de que nossa solução realmente detectaria não-conformidades, realizamos avaliação do JMLOK em diversos programas anotados com JML, incluindo pequenos exemplos e repositórios de aplicações reais, totalizando 18 KLOC e 5 KLJML. Nesta avaliação, JMLOK demandou menos que 10 minutos de execução e encontrou 29 não-conformidades. A técnica aleatória de geração de testes do JMLOK gera sequências de chamadas aos construtores e métodos que criam cenários que conseguem encontrar não-conformidades entre especificação e implementação. Após a obtenção dos resultados da avaliação, realizamos uma análise manual para investigar as origens das faltas. Assim, categorizamos as principais causas das não-conformidades ocorridas em 4 grupos: (1) não-conformidades advindas de cláusulas `non_null` implícitas, por *default*, pelo JML; (2) não-conformidades advindas de pré-condições fracas ou pela falta delas; (3) não-conformidades advindas de erros de implementação; (4) não-conformidades advindas de erros de especificação. Verificamos que a maioria das não-conformidades se encaixaram na categoria (2), pois a maioria das especificações possuem pré-condições fracas. Quando não há pré-condição explicitamente especificada, JML atribui, por *default*, especificação `@requires true`. Assim, é permitido que valores proibitivos entrem na execução dos métodos e causem não-conformidades. Esta situação pode ter como justificativa a experiência do especificador na linguagem de especificação, bem como não conhecimento total das regras de negócio do projeto. A utilização do JMLOK pode ajudar a escrita de especificações, uma vez que problemas podem ser resolvidos mais rapidamente ao longo do processo

de especificação e implementação. Além disso, verificamos que as categorias (3) e (4) são difíceis de serem detectadas, uma vez que, não existe ferramenta atual JML que realize *debug*, e realizar inspeção manual demanda tempo e experiência do especificador.

Ademais, realizamos comparação entre o JMLOK e JET nos exemplos presentes no repositório *Samples*. O JMLOK apresentou melhores resultados em 4 subpacotes, detectando mais não-conformidades do que o JET. Atestamos, assim, que a técnica de geração aleatória de testes realizada pelo JMLOK é mais eficaz que a técnica de geração com algoritmos genéticos do JET. Ainda relatamos nossa experiência em analisar a biblioteca de tipos JML (JMLModels) e constatamos que o compilador *jmlc* juntamente com as construções complexas de JML presentes nas classes não permitiram a sua compilação. Para solucionar isto, estamos migrando o JMLOK para utilizar o compilador OpenJML, que está em constante atualização.

Nossa técnica não garante que todas as não-conformidades de uma dada aplicação serão encontradas. Uma vez que, a geração de todos os cenários de testes possíveis é uma tarefa impossível de ser realizada, ou seja, testes podem detectar não-conformidades, mas não garantem a ausência de outras. No entanto, garantimos que as não-conformidades encontradas são realmente falhas existentes na implementação do software. Além disso, não-conformidades podem ser detectadas em fases iniciais do processo do software. A execução do JMLOK pode ser incremental, assim, à medida que erros são consertados, a ferramenta pode ser executada novamente para verificar novas não-conformidades, ou não-conformidades veladas pelas anteriores.

5.1 Trabalhos Relacionados

Nosso trabalho é relacionado com trabalhos que verificam corretude do software, com técnicas automáticas para inspecionar o código com o intuito de detectar erros (Seção 5.1.1), com trabalhos que apresentam ferramentas que suportam JML (Seção 5.1.2) e com trabalhos sobre geração automática de testes (Seção 5.1.3).

5.1.1 Provas, Verificação Dinâmica e Verificação Estática para Inspecionar Software

A corretude de um software pode ser alcançada através de técnicas informais, provas matemáticas e técnicas automáticas como verificação dinâmica e verificação estática. Verificar corretude informalmente, ou seja, através de uma investigação manual, está propensa a erros e não é, então, aceitável como garantia de qualidade. Provas matemáticas de corretude vem sendo discutidas desde a década de 60, principalmente por Hoare [17] [18] e Dijkstra [13]. Provas são realizadas manualmente ou através do auxílio de provadores de teoremas. Elas requerem mais esforço e demandam mais tempo, embora forneçam mais confiança, pelo menos em teoria. Hoare estabelece o conceito de corretude usando axiomas e regras matemáticas para provar que as propriedades de um programa estão realizando realmente os requisitos do usuário. Além disso, Hoare [18] determina corretude entre um programa abstrato e concreto com uma prova de que a representação concreta apresenta todas as propriedades esperadas pelo programa abstrato. As duas partes da prova correspondem aos estágios sucessivos do desenvolvimento do programa, portanto, é uma técnica construtiva para provar corretude em programas, assim como Dijkstra demonstra em [13]. Em nosso contexto, a parte abstrata seria a especificação formal e o programa concreto seria a implementação correspondente a especificação. Nós consideramos, então, que corretude entre especificação e implementação é garantida quando a implementação segue a especificação, assim, a corretude é verificada através da checagem das asserções geradas pela especificação durante a execução do programa. Neste trabalho, nós utilizamos o termo *conformidade* ao invés de *corretude*, pois consideramos *conformidade* uma forma mais leve de *corretude*, uma vez que nós não levamos em consideração e não provamos que a especificação está realmente consistente com os requisitos do cliente, ou seja, consideramos que a especificação dada está correta.

Como um dos trabalhos para provar corretude, Darvas e Müller [12] propõem uma técnica para mapear classes *model* para estruturas matemáticas provando a *consistência* (tudo que pode ser provado usando os contratos de uma classe *model* também pode ser provado usando a estrutura matemática correspondente do provador de teorema) e a *completude* (tudo que é provado usando uma estrutura definida pelo provador de teorema também pode ser provado usando contratos). Eles avaliaram essa técnica na classe `JMLObjectSet`, uma classe

presente na biblioteca de tipos modelos JML (JML Models) utilizando o provedor de teoremas Isabelle [33]. Nessa avaliação eles encontraram uma equação *unsound* dentro da `equational_theory` e a ausência de algumas especificações de métodos. Essa técnica é diferente da técnica proposta nesse trabalho, pois realiza verificação estática e lida somente com provas de especificação, ou seja, ela não lida com código e não necessita da compilação das classes. No contexto da técnica apresentada nesse trabalho (verificação de conformidade entre código e especificação através de testes), nós tentamos avaliar esta mesma classe *model*, mas não conseguimos compilar a biblioteca de tipos modelo JML com o compilador *jmlc*. Futuramente, podemos combinar os dois tipos de abordagens de maneira a obter uma especificação precisa em conformidade com seu mapeamento de código e as estruturas matemáticas do código, e então provando a implicação entre ambas. Embora essa técnica seja ideal e precisa, nós temos que levar em consideração o custo benefício em aplicá-la, pois o mapeamento e a prova não são atividades triviais e consomem tempo.

5.1.2 Técnicas que Verificam Conformidade em Programas Especificados com Contratos

Vários trabalhos de pesquisa vem sendo dedicados à verificação de software [31] [34] [9] [7] [46] no contexto de especificação de código fonte com linguagens por contratos [20; 2; 27] e a metodologia (DBC) [28]. Autotest [31] é um *framework* de teste automatizado que produz e executa testes utilizando contratos de programas orientados à objetos como oráculo. Ele unifica geração manual e automática de testes integrando casos de testes dos desenvolvedores (produção manual) dentro de um processo automatizado de teste dirigido por contratos. Atualmente, Autotest funciona em código Eiffel. A diferença básica entre nossa técnica e o Autotest é a unificação com testes manuais, nós consideramos somente a geração automática de testes. Autotest justifica esta unificação ao afirmar que a integração entre as duas estratégias de testes extrai o de melhor de cada uma. Em contrapartida, o tempo padrão estipulado para realizar todo o processo é de 10 minutos, enquanto que JMLOK apresentou bons resultados e com cobertura satisfatória em alguns programas utilizando o mínimo de 10 e o máximo de 90 segundos. TestEra [23] é uma outra ferramenta de geração de casos de testes para programas Java. Ela utiliza a

especificação de pré-condições dos métodos para gerar entradas de testes e as pós-condições para verificar as saídas dos métodos. Ela suporta especificações escritas em Alloy e utiliza o conjunto de ferramentas Alloy que gera como saída um modelo, que satisfaz determinadas restrições, para gerar os conjuntos de testes. Jartege [34] é uma ferramenta semiautomática para geração de casos de testes no formato JUnit. Ela também utiliza técnica aleatória para a geração dos testes. A Aleatoriedade é estabelecida através da atribuição de pesos às classes e métodos sob teste (peso igual a 1 é o valor padrão). No entanto, nenhum critério é descrito para o estabelecimento do peso, além disso, o tempo gasto para gerar e executar os testes não é informado. JMLUnit [7] é uma ferramenta semiautomática que gera casos de testes combinando chamadas aos métodos sob teste com o intuito de checar conformidade. Similar ao JMLOK, JMLUnit constrói oráculos para os testes através das asserções geradas pelo compilador JML (*jmlc*). Embora essa ferramenta seja efetiva na geração de esqueletos de testes, ela não gera dados de testes para os casos de testes; o usuário realiza essa tarefa. Com o intuito de transpor essa limitação, a ferramenta JMLUnitNG [46] foi criada como uma versão melhorada do JMLUnit, pois gera automaticamente dados de testes para tipos não-primitivos – como entradas para construtores e métodos. Uma das diferenças entre as duas ferramentas é que o JMLUnitNG substituiu a utilização do framework JUnit pelo TestNG [4] o qual constrói listas de parâmetros sob demanda, não armazenando casos de testes na memória. As melhorias, entretanto, não eximem o usuário de prover os dados de testes. Além disso, o tempo necessário para executar os testes é consideravelmente longo, uma vez que um número excessivo de testes é gerado. Outra ferramenta, JET [8], utiliza algoritmos genéticos para construir automaticamente todos os dados de testes e, então, executá-os exercitando as asserções em tempo de execução. Algoritmos genéticos são eficazes no que se propõem a realizar, porém seu não-determinismo em gerar casos de testes e dados de testes não garantem, por exemplo, que uma mesma não-conformidade seja detectada em duas execuções consecutivas. Nós realizamos uma comparação entre JMLOK e JET (Seção 4.5.1), onde mostramos que o JMLOK apresentou melhores resultados ao encontrar mais não-conformidades nas mesmas unidades experimentais. A principal diferença entre as duas abordagens é a técnica usada para gerar os testes. A principal vantagem do JMLOK sobre o JET é que JMLOK possui uma técnica de geração de testes que cria conjuntos de sequências de chamadas que conseguem criar cenários de testes que

descobrem não-conformidades.

No âmbito da análise estática, existe a ferramenta ESC/Java2 [11] que realiza verificação estática em programas JML. Essa ferramenta aplica uma técnica baseada em lógica que verifica estaticamente se alguma violação à especificação JML vai acontecer durante a execução. Em suma, ESC/Java2 tenta provar programas corretos com relação a sua especificação usando o provador de teorema *Simplify*. ESC/Java2 fornece garantias mais fortes, mas por outro lado exige que as especificações estejam completas, não só para os módulos a serem verificados, mas também para os módulos e bibliotecas de que eles dependem, transformando a tarefa de especificar mais exigente. ESC/Java2 pode ser utilizada pra ampliar a nossa técnica, através da sua utilização antes do JMLOK. Este passo a mais pode demandar mais tempo, mas em contrapartida haverá maior garantia de que a implementação está em conformidade com a especificação dada.

Sireum/Kiasan [3] é outra ferramenta que realiza análise estática em programas Java/JML. Kiasan não necessita de parâmetros de entrada para gerar conjuntos de testes de unidade. Ele utiliza os contratos JML para filtrar parâmetros de entrada que não satisfaçam pré-condições e assegura a conformidade entre a saída do método e a pós-condição. Sem asserções ou contratos, Kiasan, por padrão, detecta possíveis exceções em tempo de execução como *NullPointerException*, *ArrayIndexOutOfBoundsException* e etc. Assim como ESC/Java2, também podemos utilizar esta ferramenta para ampliar a nossa técnica e obter maior confiabilidade nos programas verificados.

5.1.3 Ferramentas de Geração Automática de Testes

É desejável que toda possível permutação de um programa seja averiguada por testes. Entretanto, na maioria dos casos, isso não é possível. Mesmo que o programa pareça simples, ele pode ter milhares de possíveis combinações de entradas e saídas. Assim, criar casos de testes para todas as possibilidades possíveis é impraticável. Testar completamente uma aplicação complexa levaria muito tempo e esforço, o que é economicamente inviável. Porém os testes são práticos e conseguem descobrir *bugs* em fases iniciais do processo de software, trazendo feedback rápido e garantindo a qualidade, mesmo quando as especificações são parciais. Existem algumas ferramentas que facilitam a geração de testes e automatiza esse processo partindo de diferentes estratégias [37;

1; 15].

A ferramenta Eclat [37] é um trabalho anterior ao Randoop. As entradas para essa ferramenta são (1) código-fonte, (2) um conjunto de execuções corretas do programa, como por exemplo, uma suíte de testes em que nenhum caso de teste revele falha e (3) um conjunto de entradas candidatas que podem ser uma entrada inválida, uma entrada normal ou uma entrada que cause falha. Depois das entradas, o processo possui três passos: (1) Geração de um modelo: gera um modelo operacional (a partir da suíte de teste *pass* de entrada) com o auxílio do Daikon [14] para extrair propriedades do código. (2) Classificação: classifica as entradas candidatas em ilegal, operação normal ou *fault-revealing* através da execução de cada candidata e comparando o comportamento do programa com o modelo operacional. (3) Redução: particiona o conjunto de *fault-revealing* em um conjunto de propriedades violadas e reporta somente um candidato de cada partição. Em nossa técnica, nós consideramos que as propriedades já estão definidas pelas especificações e nossos resultados (Capítulo 4) sugerem ser melhores, uma vez que nós consideramos que as especificações estão corretas com os requisitos dos usuários. EvoSuite [15] é uma ferramenta de geração automática de casos de testes com asserções para classes escritas em código Java. Essa ferramenta usa uma técnica de busca evolutiva que evolui suítes de testes em relação a um critério de cobertura completa. EvoSuite utiliza testes de mutação para produzir um conjunto reduzido de asserções que maximiza o número de defeitos semeados em uma determinada classe. Testful [1] é uma ferramenta semi automática de geração de testes que gera uma população inicial de dados de testes a partir de uma pré-análise das classes sob teste e depois utiliza algoritmos genéticos para melhorar a aptidão de sua população. Para realizar a pré-análise, o TestFul requer que as classes sob teste sejam instrumentadas pelo usuário. A ferramenta não gera bons resultados com tempo de geração curto, 10 segundos, por exemplo. A ferramenta apresenta resultados melhores perante outras ferramentas com tempo de geração a partir de 5 minutos, e teoricamente, o melhor tempo para ter uma boa cobertura deve ser de 10 minutos. Realizamos uma avaliação piloto comparando o Randoop e o TestFul. Testamos a classe `Rectangular` do repositório *Samples* para averiguar a quantidade de não-conformidades que cada técnica conseguiria detectar. Com o Randoop, conseguimos detectar 3 não-conformidades, enquanto que com o TestFul encontramos somente 1 não-conformidade, utilizando o tempo de 10 segundos. Além disso, despendemos tempo considerável para instrumentar a Classe

`Rectangular` e todas as classes que ela herda.

5.2 Trabalhos Futuros

Em trabalhos futuros, pretendemos testar nossa técnica em mais avaliações de repositórios, para com isso, explorar novas situações e poder melhorar o JMLOK. Realizaremos comparações do JMLOK com outras ferramentas semelhantes, como JMLUnitNG. Novas funcionalidades serão adicionadas ao JMLOK, tais como extrair métricas a partir dos resultados obtidos e automatizar a categorização das não-conformidades, para ajudar os desenvolvedores a despender menos tempo para consertar as não-conformidades encontradas. Além disso, pretendemos melhorar a interface da ferramenta, pois atualmente ela funciona como um protótipo. Também expandiremos a técnica proposta para outros contextos, como a linguagem de programação C# e especificação Spec# [2]. O compilador OpenJML será incorporado em nova versão da ferramenta, por ser o compilador atualmente desenvolvido e evoluído pela comunidade JML. Além disso, a técnica proposta pode ser útil quando inserida no contexto de evolução do software, ao passo que mudanças ocorridas no código através de refatoramentos [25] realizados em implementações Java com especificações JML podem requerer adaptações em pré ou pós-condições, por exemplo. Para isso, implementaremos uma ferramenta que garanta segurança durante a evolução do software no contexto de refatoramentos.

Bibliografia

[1] Luciano Baresi and Matteo Miraz. Testful: automatic unit-test generation for java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 281–284, New York, NY, USA, 2010. ACM.

[2] Mike Barnett, Manuel Fähndrich, Rustan Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The spec# experience. *ACM*, 54:81–91, June 2011.

[3] Jason Belt, Robby, and Xianghua Deng. Sireum/topi ldp: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 355–364, New York, NY, USA, 2009. ACM.

[4] Cédric Beust and Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[5] L. Burdy, Y. Cheon, D. Cok, M. Ernest, J. Kiniry, G. Leavens, R. Leino, and E. Poll. An overview of jml tools and applications. 2005.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[7] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP*, pages 231–255. 2002.

[8] Y. Cheon and C. Medrano. Random test data generation for java classes annotated with jml specifications. pages 385–392, 2007.

[9] Yoonsik Cheon, Antonio Cortes, Gary T. Leavens, and Ceberio Martine. Integrating random testing with constraints for improved efficiency and diversity. In *SEKE*, pages 861–866, 2008.

[10] Yoonsik Cheon and Gary Leavens. A runtime assertion checker for the java modeling language (jml). In *Proceedings of SERP*, 2002.

[11] David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting esc/java and jml – progress and issues in building and using esc/java2. In *CASSIS*, 2004.

[12] Ádám Darvas and Peter Müller. Faithful mapping of model classes to mathematical structures. volume 2, pages 477–499, 2008.

[13] Edsger W. Dijkstra. *A constructive approach to the problem of program correctness.* 1967. circulated privately.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. volume 69, pages 35–45, December 2007.

[15] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[16] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[17] C. Hoare. *An Axiomatic Basis for Computer Programming*, volume 12. 1969.

[18] C. Hoare. *Proof of Correctness of Data Representation.* 1975.

[19] Gary Leavens, A. Baker, and C. Ruby. Jml: A notation for detailed design. chapter 12, pages 175–188. In Behavioral Specifications for Businesses and Systems, 1999.

[20] Gary Leavens, Albert Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, 2006.

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

[22] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml.

[23] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE '01: Proceedings. 16th Annual International Conference on Automated Software Engineering*, page 22. IEEE Computer Society, 2001.

[24] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[25] Tiago Lima Massoni. *A Model-driven Approach to Formal Refactoring*. PhD thesis, UFPE, Recife, 2008.

[26] Jim A.l McCal, Paul K. Richards, and Gene F. Walters. *Factors in Software Quality*, volume 1. Pressman, 1977.

[27] B Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Not.*, 22:85–94, 1987.

[28] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

[29] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[30] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[31] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[32] Glenford J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004.

[33] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[34] C. Oriat. Jartege: a tool for random generation of unit tests for java classes. Rapport de recherche LSR-IMAG, RR 1069, 2004.

[35] E. Jong P. Hartel, M.Butler and M. Longley. Transacted memory for smart cards. pages 478–499. 10th Formal Methods for Increasing Software Productivity, 2001.

[36] C. Pacheco, S. Lahiri, M. Ernest, and T. Ball. Feedback-directed random test generation. pages 75–84, 2007.

[37] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 504–527. Springer, 2005.

[38] E. Poll, P. Hartel, and E. Jong. A java reference model of transacted for smart cards. In *CARDIS*, pages 75–86. 2002.

[39] Henrique Rebêlo, Sérgio Soares, Ricardo Lima, Leopoldo Ferreira, and Márcio Cornélio. Implementing java modeling language contracts with aspectj. In *Proceedings of SAC '08*, SAC '08, pages 228–233, New York, NY, USA, 2008. ACM.

[40] Henrique Emanuel Mostaert Rebêlo. Implementing jml contracts with aspectj. Master's thesis, Departament of Computing and Systems, State University of Pernambuco, May 2008.

[41] R. Rodrigues. *JML-Based Formal Development of a Java Card Application for Managing Medical Appointments*. Universidade da Madeira, 2009.

[42] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[43] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 27:52–57, 2010.

[44] Ian Sommerville. *Software Engineering*. Pearson Studium, 2001.

[45] Jeannette M. Wing. Writing larch interface language specifications. *ACM Trans. Program. Lang. Syst.*, 9(1):1–24, January 1987.

[46] Daniel M. Zimmerman and Rinkesh Nagmoti. Jmlunit: The next generation. In *FoVeOOS*, pages 183–197, 2010.