JMLOK: Uma Ferramenta para Verificar Conformidade em Programas Java/JML

Catuxe Varjão[1], **Tiago Massoni**[1], **Rohit Gheyi**[1], **Gustavo Soares**[1]

[1]Department of Systems and Computing – UFCG

catuxe@copin.ufcg.edu.br,{massoni,rohit,gsoares}@dsc.ufcg.edu.br

Abstract. *It is non-trivial to verify whether a Java code is in conformance with its JML specification. Usually, developers perform this task by manual reasoning, which is error-prone and time-consuming, or by including assertions in the compiled Java code. However, these assertions may not be sufficiently exercised, then problems arise when the system is in production. We implemented JMLOK to identify non-conformances between Java code and its JML specification by randomly generating unit tests. We evaluate JMLOK in toy examples and real case studies, finding 13 non-conformances. Even in toy examples presented in the literature, we found non-conformances between programs and specifications.*

1. Introdução

Design by Contract (DBC) tem por conceito chave o estabelecimento de uma relação formal onde um contrato é expresso por obrigações e direitos de clientes e implementadores. Assim, o cliente deve garantir algumas condições antes de chamar um método, e a classe, por sua vez, deve garantir algumas propriedades após sua chamada. *Java Modeling Language* (JML) [Leavens et al. 1999] é uma linguagem de especificação formal para classes e interfaces Java que contém como subconjunto a notação essencial utilizada por DBC. Em JML, o comportamento de interfaces e classes Java é especificado em 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.

As não-conformidades que possivelmente possam estar presentes em sistemas formalmente anotados devem ser detectadas o quanto antes dentro do processo de software para que a qualidade do sistema seja assegurada. Existem duas maneiras comumente utilizadas para verificar consistência em códigos anotados, uma delas é manualmente, que está propensa a erros, consome muito tempo e depende da experiência do especialista, e a outra forma é realizada através da tradução de cada anotação em asserções incluídas no código Java. Este processo é realizado por um compilador, como o *jmlc* (Seção 2). Embora existam tais métodos de detecção de não-conformidades, a maioria delas é encontrada quando o sistema já está em produção, o que dificulta a correção, pois quanto maior é o tempo para encontrar erros, maior é o tempo demandado para consertá-los.

Neste artigo nós descrevemos JMLOK, uma ferramenta que verifica não-conformidades entre o código Java e a especificação JML (Seção 3). Ela utiliza geração automática massiva de testes de unidade para exercitar o programa, e verificadores de asserções como oráculo para os testes. Nós avaliamos JMLOK em pequenos exemplos e em um estudo de caso real, onde encontramos 13 não-conformidades (Seção 4).

2. Exemplo Motivante

Nesta seção, apresentamos um exemplo de código Java anotado com JML que expõe a ocorrência de um problema de não-conformidade. Considere a classe `Pessoa`, presente no Código-fonte 1, que contém o atributo `altura` e o método `setAltura`. O método `setAltura` possui uma pré-condição – cláusula **requires** — que restringe o parâmetro `n` com valor maior que `zero`, e a pós-condição — cláusula **ensures** – que verifica se o valor atual do atributo `altura` está maior que o valor que ele possuía antes da chamada.

Código-fonte 1. Parte da classe Pessoa anotada com JML.

```
public class Pessoa {
  private /*@ spec_public @*/ int altura;
  /*@ requires n > 0;
    @ ensures this.altura >= \old(this.altura);
    @*/
  public void setAltura (int n) {
    this.altura = n;
  }
}
```

Entretanto, o código anterior Java não está em conformidade com a sua especifição JML. Considere a sequência de chamadas a seguir:

```
Pessoa p1 = new Pessoa();
p1.setAltura(2);
p1.setAltura(1);
```

Na primeira chamada ao método `setAltura`, a pré-condição e a pós-condição não foram violadas, pois suas respectivas restrições foram satisfeitas. Enquanto que a segunda chamada viola a pós-condição. O parâmetro `1` satisfaz a pré-condição, mas a pós-condição não é satisfeita já que a altura final (`1`) é menor que a anterior (`2`). O exemplo é pequeno, e este erro pode ser facilmente detectado por um desenvolvedor. Porém, em sistemas reais, não-conformidades mais sutis podem ser mais difíceis de serem identificadas.

3. Ferramenta JMLOK

Nesta seção, nós apresentamos a ferramenta JMLOK que identifica não-conformidades entre código Java e especificações JML. Nossa técnica propõe a deteção de não-conformidades em programas Java anotados com especificações JML através da geração de testes de unidade. De maneira geral, ela funciona a partir de um programa Java/JML dado como entrada, através dos seguintes passos: 1) Testes de unidade compostos por sequências de chamadas de métodos e construtores são gerados automaticamente, levando em consideração o código-fonte Java; 2) Em seguida, oráculos de testes são gerados a partir das especificações JML; 3) Os testes gerados no Passo 1 são executados, e os oráculos gerados no Passo 2 são utilizados para checar não-conformidades entre o código e a especificação; 4) Após a execução, nós geramos um relatório contendo um conjunto de testes expondo as possíveis não-conformidades existentes no programa Java/JML. O processo da técnica é descrito na Figura 1.

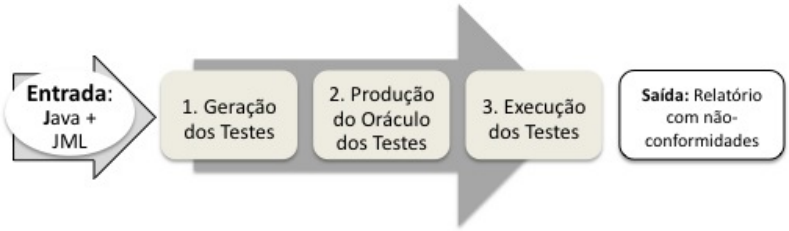

Figura 1. Abordagem proposta.

O JMLOK é uma ferramenta que implementa a técnica detalhada na Figura 1. Para que o processo se inicie, o usuário seleciona um diretório contendo classes Java/JML como entrada e clica no botão JMLOK (Figura 2). Logo após, como primeiro passo, a ferramenta JMLOK gera testes de unidade, utilizando o Randoop [Pacheco et al. 2007] (versão 1.3.2). Randoop gera, aleatoriamente, sequências de métodos e invocações aos construtores para as classes sob testes, no formato JUnit. Randoop ainda gera asserções juntamente com os testes, mas nós não as consideramos, pois elas são produzidas de acordo com o código-fonte Java, ou seja, não leva em consideração os contratos em JML.

No segundo passo, oráculos para os testes são produzidos através do verificador de asserções que é gerado pelo compilador JML (*jmlc*). O *jmlc* produz *bytecode* do programa compilado, assim como os compiladores Java, mas com a adição, a este *bytecode* produzido, de verificadores de asserções para as pré-, pós-condições e invariantes. Nós utilizamos a versão *jmlc* 5.6_rc4. Na nossa solução, o procedimento de decisão para o oráculo dos testes é, portanto, o verificador de asserções que é utilizado para detectar violações às asserções, 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 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 2). Nesta tela são disponibilizados os testes que detectaram violações e os tipos de exceções lançadas.

Como exemplo de funcionamento do JMLOK, considere o Código-fonte 1 como entrada para a ferramenta. Primeiro, uma coleção de testes contento cinco testes de unidade é gerada pelo Randoop. Um exemplo de teste gerado para o Código-fonte 1 é exibido no Código-fonte 2. Em seguida, a ferramenta gera oráculos para os testes utilizando o compilador *jmlc*. Por fim, os testes são executados no código Java/JML. A execução do teste (Código-fonte 2) na classe `Pessoa` detecta uma não-conformidade – `JMLInternalNormalPostconditionError` – referente à violação à pós-condição do método `setAltura`.

Código-fonte 2. Teste de unidade gerado pelo Randoop

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

Violações às pré-condições do tipo `JMLEntryPreconditionError` 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* [Cheon and Medrano 2007]. Desta forma, nós consideramos não-conformidades as faltas existentes no código que são evidenciadas através do lançamento de exceções referentes às pós-condição, 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.

É importante ressaltar que só a utilização do Randoop diretamente no código contendo as asserções geradas pelo *jmlc* não detectaria não-conformidades. Para gerar oráculos 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 1 compilado com *jmlc*, o Randoop geraria o teste abaixo ao invés do teste exibido no Código-fonte 2:

Código-fonte 3. Teste gerado pelo Randoop com o código compilado pelo *jmlc*

```
public void test5() throws Throwable {
  Pessoa var0 = new Pessoa();
  var0.setAltura(10);
  try {
    var0.setAltura(1);
    fail(''Exceção esperada'');
  } catch (JMLInternalNormalPostconditionError e) {}
}
```

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.

A arquitetura do JMLOK é baseada no padrão *Pipes-and-Filters*, 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 2 mostra esta arquitetura, bem como a interface gráfica do JMLOK. Na janela 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 sua a mensagem, bem como o código do teste que revela o problema.

4. Avaliação

Nesta seção, avaliamos o JMLOK em diferentes grupos de programas Java/JML. Primeiro, nós executamos JMLOK em um conjunto de pequenos exemplos, denominado *Samples*, disponível no repositório de exemplos do site oficial JML[1]. *Samples* possui treze subconjuntos de exemplos principais divididos pela sua natureza, dos quais avaliamos nove deles – total de 3343 LOC e 5073 linhas de especificação JML (a partir daqui indicaremos esta métrica por LJML) – e encontramos nove não-conformidades. Depois, nós avaliamos JMLOK em um estudo de caso real, *Transacted Memory* [Poll et al. 2002], uma funcionalidade específica para API Javacard[2]. Nós avaliamos este estudo de caso

[1] http://www.eecs.ucf.edu/~leavens/JML/examples.shtml
[2] http://www.oracle.com/technetwork/java/javacard/overview/index.html

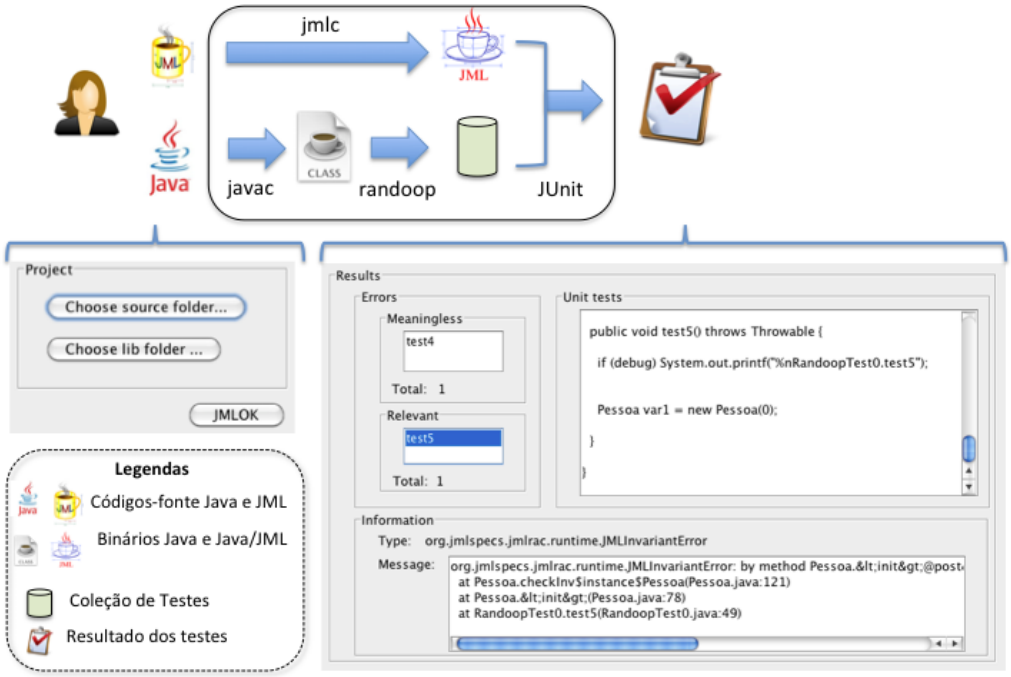

Figura 2. Arquitetura e Interface gráfica do JMLOK. Na tela inicial da ferramenta, o usuário informa os diretórios do projeto Java/JML e clica no botão JMLOK. Logo após, a ferramenta informa os testes que podem indicar não-conformidades e as exceções lançadas.

em três versões disponíveis, o que totaliza 1806 LOC e 335 LJML, e encontramos quatro não-conformidades.

Como mencionado na Seção 3, a execução do JMLOK é dividida em três principais fases. Na fase da geração dos testes, o Randoop recebe como entrada uma lista de classes a serem testadas e um tempo limite de geração de testes, o *timelimit*. Através da realização de um experimento piloto simples para correlacionar *timelimit* e LOC em alguns programas onde as não-conformidades eram conhecidas, verificamos que, aplicando entre dez e sessenta segundos no *timelimit*, as não-conformidades esperadas eram encontradas independente do *timelimit* e da cobertura dos testes. Nos programas aqui avaliados, nós utilizamos dez segundos de *timelimit* como valor padrão. No total, entre geração e execução dos testes, a ferramenta despendeu aproximadamente nove minutos em todos os programas avaliados. Realizamos este experimento em um Intel Core 2 Duo com 2.13 GHz e 4 GB SDRAM, MAC OS X Versão 10.5.8 e Java 6. Todos os dados da avaliação podem ser encontrados no site da ferramenta[3].

Para os exemplos avaliados no *Samples*, foram gerados o total de 4497 testes com cobertura média de 86,5% para os nove subconjuntos de exemplos avaliados — utilizamos o plugin EclEmma[4] para coletar este dado, que informa o percentual de instruções que os testes cobrem. Após a execução dos testes, encontramos nove não-conformidades (NC) em quatro dos exemplos testados (a Tabela

[3] http://www.dsc.ufcg.edu.br/~spg/jmlok
[4] http://www.eclemma.org/

1 mostra as não-conformidades encontradas). Dentre os quais tivemos: seis não-conformidades do tipo `JMLInternalNormalPostconditionError` nos exemplos *dbc* (três NC), *jmlkluwer* (uma NC) e *list* (duas NC); uma não-conformidade do tipo `JMLEvaluationError` no exemplo *list*; e, uma não-conformidade `JMLExitExceptionalPostconditionError` e uma `JMLInvariantError` no exemplo *stack*.

No estudo de caso *Transacted Memory* foram gerados o total de 717 testes com cobertura média de 32%. Nós encontramos quatro não-conformidades (Tabela 1) entre as três versões avaliadas: um `JMLExitExceptionalPostconditionError` e três `JMLInvariantError`. A Coluna Programa na Tabela 1 refere-se a versão onde a não-conformidade foi encontrada.

Tabela 1. Resultados da Execução do JMLOK; Programa: nome do programa onde existe a não-conformidade; Estudo de Caso: nome do Estudo de Caso; LOC: LOC do programa; LJML: número de linhas JML; Testes: número de testes gerados; Cobertura: percentual de cobertura de instruções alcançada pelos testes; NC: número de não-conformidades; Total: valores totais de cada coluna.

Programa	Estudo de Caso	LOC	LJML	Testes	Cobertura	NC
dbc	Samples	313	150	199	91,2%	3
jmlkluwer	Samples	164	238	259	94,5%	1
list	Samples	1541	3089	317	64,3%	3
stacks	Samples	260	403	392	88,3%	2
TransactedMemory (versão 1)	Transacted Memory	129	144	59	11,1%	1
TransactedMemory (versão 2)	Transacted Memory	470	45	395	37,3%	1
TransactedMemory (versão 3)	Transacted Memory	1207	146	263	47,1%	2
	Total	**4084**	**4215**	**1884**	**58,2%**	**13**

5. Trabalhos Relacionados

Algumas ferramentas relacionadas ao JMLOK como JMLUnit [Cheon and Leavens 2002], JMLUnitNG [Zimmerman and Nagmoti 2010], JET [Cheon and Medrano 2007] e Jartege [Oriat 2004] também pretendem detectar não-conformidades através da geração de testes. A principal diferença entre cada ferramenta está justamente nas técnicas utilizadas para essa geração.

JMLUnit é um framework que gera dados de testes realizando chamadas aos métodos sob teste através da combinação de valores para seus parâmetros de entrada – para tipos primitivos, esta ferramenta disponibiliza alguns valores default. O JMLUnitNG é uma versão atualizada do JMLUnit, assim, o JMLUnitNG gera dados de testes para entradas com tipos não-primitivos de construtores e métodos e, principalmente, substituiu a ferramenta JUnit por TestNG a qual não necessita de suites de testes armazenadas em memória, mas que gera listas de parâmetros de acordo com a necessidade. Apesar das melhorias, esta ferramenta ainda necessita da intervenção do usuário para gerar alguns dados de testes. Além disso, o tempo necessário para executar os testes é demasiado, mediante o exorbitante número de testes gerados.

Jartege é uma ferramenta semiautomática inspirada no JMLUnit. Jartege propõe a geração aleatória de testes de unidade, em que o aspecto randômico é parametrizado

através da atribuição de pesos a cada classe e método sob teste - por padrão, são atribuídos pesos com valor `1`. Todavia, este trabalho não apresenta critérios de escolha dos valores dos pesos quando um valor diferente do padrão é necessário. Ademais, o tempo despendido para executar tal ferramenta não foi mencionado.

A ferramenta JET utiliza algoritmos genéticos para gerar os dados de testes. Esta ferramenta utiliza duas abordagens: *specified-based*, que deriva os dados de teste a partir das pós-condições empregando um *constraint solver*; e, *program-based* que deriva os testes a partir do código-fonte, semelhantemente ao JMLOK. Em uma avaliação preliminar, entre JET e JMLOK, nossa ferramenta identificou mais não-conformidades.

6. Conclusões

Nosso trabalho apresentou JMLOK, uma ferramenta para detectar não-conformidades em códigos Java anotados com especificações JML através da geração aleatória e automática de testes de unidade. Nós avaliamos a ferramenta em programas catalogados na literatura, totalizando aproximadamente 5 KLOC e 5K LJML. JMLOK identificou 13 não-conformidades mostrando que inconsistências podem acontecer mesmo em sistemas especificados e implementados por atores experientes.

Futuramente, pretendemos avaliar mais programas, avaliar algumas das ferramentas relacionadas (JMLUnitNG e JET) em experimentos mais elaborados, categorizar os tipos de erros encontrados nas avaliações realizadas, e extrair métricas que possam facilitar a identificação de não-conformidades através da análise dos resultados.

Agradecimentos

Agradecemos aos revisores anônimos pelas sugestões úteis. Este trabalho foi parcialmente financiado pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES), financiado pelo CNPq subsídios 573964/2008-4, 477336/2009-4, e 304470/2010-4.

Referências

[Cheon and Leavens 2002] Cheon, Y. and Leavens, G. (2002). A simple and practical approach to unit testing: The jml and junit way. pages 231–255. In ECOOP'02.

[Cheon and Medrano 2007] Cheon, Y. and Medrano, C. (2007). Random test data generation for java classes annotated with jml specifications. pages 385–392. In ICSE'07.

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

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

[Pacheco et al. 2007] Pacheco, C., Lahiri, S., Ernest, M., and Ball, T. (2007). Feedback-directed random test generation. pages 75–84. In ICSE'07.

[Poll et al. 2002] Poll, E., Hartel, P., and Jong, E. (2002). A java reference model of transacted for smart cards. pages 75–86. In CARDIS'02.

[Zimmerman and Nagmoti 2010] Zimmerman, D. and Nagmoti, R. (2010). Jmlunit: The next generation. pages 183–197. In FoVeOOS'10.