Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code

Master's Thesis Presentation – Advisors: Max Kramer, Michael Langhammer, Erik Burger
Supervisors: Prof. Dr. Ralf Reussner, Jun.-Prof. Dr.-Ing. Anne Koziolek

Stephan Seifermann | November 21, 2014
public void fund(int amount) {
    ...  
    balance += amount - FUND_FEE;  
    ...  
}
Introduction

Motivation

/*@
   requires amount >= FUND_FEE;
   ensures balance == \old(balance) +
       amount - FUND_FEE;
*/

public void fund(int amount) {
    ...
    balance += amount - FUND_FEE;
    ...
}
Introduction
Motivation

/*@
    requires amount >= FUND_FEE;
    ensures balance == \old(balance) + amount - FUND_FEE;
*/

public void fund(int amount) {
    ... balance += amount - FUND_FEE;
    ...
}

@Test
public void testFund() {
    testInstance.fund(10);
    assertEquals(10 - BankAccount.FUND_FEE,
                 testInstance.getBalance());
}
**Introduction**

**Motivation**

```java
/*@ 
   requires amount >= FUND_FEE;
   ensures balance == \old(balance) + amount - FUND_FEE;
@*/

public void fund(int amount) {
    ...
    balance += amount - FUND_FEE + FUND_BONUS;
    ...
}

@Test
public void testFund() {
    testInstance.fund(10);
    assertEquals(10 - BankAccount.FUND_FEE,
                 testInstance.getBalance());
}
```
Introduction

Motivation

/*@  
    requires amount >= FUND_FEE;
    ensures balance == \old(balance) +  
        amount - FUND_FEE + FUND_BONUS;
    @*/

public void fund(int amount) {
    ...
    balance += amount - FUND_FEE + FUND_BONUS;
    ...
}

@Test
public void testFund() {
    testInstance.fund(10);
    assertEquals(10 - BankAccount.FUND_FEE,
        testInstance.getBalance());
}

---

Stephan Seifermann – Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code

November 21, 2014
Introduction

Motivation

/*@ 
  requires amount >= FUND_FEE;
  ensures balance == \old(balance) + 
    amount - FUND_FEE + FUND_BONUS;
*/

public void fund(int amount) {
    ...
    balance += amount - FUND_FEE + FUND_BONUS;
    ...
}

@Test
public void testFund() {
    testInstance.fund(10);
    assertEquals(10 - BankAccount.FUND_FEE +
        BankAccount.FUND_BONUS, testInstance.getBalance());
}

@Test
public void testFund() {
    testInstance.fund(10);
    assertEquals(10 - BankAccount.FUND_FEE +
        BankAccount.FUND_BONUS, testInstance.getBalance());
}
Definition of overlap between code, contracts and unit tests
  - Relations
  - Constraints

Consistency concept for artifacts (change reactions)

Simplified model printer and parser for JML

Implementation of prototype
  - Change detection, (Messinger 2014)
  - Vitruvius, (Kramer et al. 2013)
  - Enforcement of consistency concept
Introduction
Foundations

Changes

Modifications of an artifact, which are classified by the *abstraction*, *representation* and *range of influence*
Changes

Modifications of an artifact, which are classified by the abstraction, representation and range of influence

Co-Evolution

Changes of implementation of code, tests and contracts together
# Introduction

**Foundations**

## Changes

Modifications of an artifact, which are classified by the *abstraction*, *representation* and *range of influence*.

## Co-Evolution

Changes of implementation of code, tests and contracts together.

## Contracts

Obligations of user and provider of functionality (preconditions, postconditions, invariants)

### Table of Contents

<table>
<thead>
<tr>
<th>Introduction</th>
<th>Related Work</th>
<th>Concept</th>
<th>Prototype</th>
<th>Evaluation</th>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stephan Seifermann – Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code</td>
<td></td>
<td></td>
<td></td>
<td>November 21, 2014</td>
<td>4/17</td>
</tr>
</tbody>
</table>
Related Work
Relation between Code and Contracts

- Hull 2010
- Feldman 2003
- Feldman, et al. 2006
- Goldstein, et al. 2006

Problems
- No detailed information on overlap
- No public tools except for Hull 2010
- Findings not traceable
Related Work
Relation between Code and Contracts

- Contracts to Code
  - Hull 2010
- Code to Contracts
  - Feldman 2003
  - Goldstein, et al. 2006

Problems
- No detailed information on overlap
- No public tools except for Hull 2010
- Findings not traceable
Related Work
Relation between Tests and Contracts

Recommendation of Combining Tests and Contracts

Feldman 2003
du Bousquet, et al. 2004
Goldstein, et al. 2006

Processes
Heinecke, et al. 2001
Ostroff, et al. 2004

Test Data Generation
## Related Work

### Relation between Tests and Contracts

<table>
<thead>
<tr>
<th>Recommendation of Combining Tests and Contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feldman 2003</td>
</tr>
<tr>
<td>du Bousquet, et al. 2004</td>
</tr>
<tr>
<td>Goldstein, et al. 2006</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heinecke, et al. 2001</td>
</tr>
<tr>
<td>Ostroff, et al. 2004</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Data Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defaults and Manual</td>
</tr>
<tr>
<td>Cheon, et al. 2002</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model-Based</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Debug Traces</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Contracts</th>
</tr>
</thead>
</table>

---

**Introduction**

**Related Work**

**Concept**

**Prototype**

**Evaluation**

**Conclusions**
Related Work

Relation between Tests and Contracts

**Recommendation of Combining Tests and Contracts**

- Feldman 2003
- du Bousquet, et al. 2004
- Goldstein, et al. 2006

**Processes**
- Heinecke, et al. 2001
- Ostroff, et al. 2004

**Test Data Generation**

- Defaults and Manual
  - Cheon, et al. 2002
- Model-Based
- Debug Traces
- Contracts

**Problems**

- One way approaches
- Unfeasible amount of test cases
Concept Overview

Find Technologies

- Specification Languages
- Programming Languages
- Test Frameworks
Concept

Overview

Find Technologies

- Specification Languages
- Programming Languages
- Test Frameworks

Select for Analysis

- Java
- JML
- JUnit
Concept Overview

Find Technologies
- Specification Languages
- Programming Languages
- Test Frameworks

Select for Analysis
- Java
- JML
- JUnit

Define Overlap and Constraints
- JML
- Java
- JUnit

Introduction Related Work Concept Prototype Evaluation Conclusions
Stephan Seifermann – Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code November 21, 2014 7/17
Concept Overview

Find Technologies
- Specification Languages
- Programming Languages
- Test Frameworks

Select for Analysis
- Java
- JML
- JUnit

Define Overlap and Constraints
- JML
- Java
- JUnit

Define Change Reactions
- JML
- Java
- JUnit
**Concept**

**Overview**

Find Technologies

- Specification Languages
- Programming Languages
- Test Frameworks

Select for Analysis

- Java
- JML
- JUnit

Transfer

- Contracts
- Code
- Tests

Define Overlap and Constraints

- JML
- Java
- JUnit

Define Change Reactions

- JML
- Java
- JUnit
Concept

Assumptions
Concept

Assumptions

- JML language level 0 and 1 only

Level 0
requires, ensures, invariant

Level 1
pure, model method

Level 2
duration
JML language level 0 and 1 only

JML shortcuts are used

```java
//@ requires item != null;
public void add(Object item) {
    ...
}

public void add(/*@ non_null */
    Object item) {
    ...
}
```
Assumptions

- JML language level 0 and 1 only
- JML shortcuts are used
- Changes covered by IDEs omitted

```java
void setReason(short reason) {
_reason[0] = reason;
}
```
Concept

Assumptions

- JML language level 0 and 1 only
- JML shortcuts are used
- Changes covered by IDEs omitted
- No dead code

```java
void test(int number) {
    throws Exception {
    for (int i = 0; i < number; ++i) {
        if (i == number) {
            throw new Exception();
        }
    }
}
```
Assumptions

- JML language level 0 and 1 only
- JML shortcuts are used
- Changes covered by IDEs omitted
- No dead code
- Java elements only changed in Java artifact
Assumptions

- JML language level 0 and 1 only
- JML shortcuts are used
- Changes covered by IDEs omitted
- No dead code
- Java elements only changed in Java artifact
- Generated tests with manual test data used
Concept
Covered JML Constructs

Basic Syntax

- Identifier 🌟
- Visibility
- Types / Return Types
Concept
Covered JML Constructs

Basic Syntax
- Identifier 🌼
- Visibility
- Types / Return Types

Specification-Only Elements
- Ghost Field
- Model Field
- Model Method
- Model Import
Concept
Covered JML Constructs

Basic Syntax
- Identifier
- Visibility
- Types / Return Types

Specification-Only Elements
- Ghost Field
- Model Field
- Model Method
- Model Import

Method or Type Specs
- Pure
- Helper
- Nullable
- Non Null
- Nullable Defaults
- Generic Behavior Specs
  - requires
  - ensures
  - ...
- Exception Specs
- Assignable

Introduction Related Work Concept Prototype Evaluation Conclusions
Stephan Seifermann – Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code
November 21, 2014 9/17
One page of the document contains a diagram labeled "Concept" with subtopics "Overlap and Constraints." The diagram illustrates the relationships between various technologies and frameworks relevant to software development, particularly focusing on JML, Java, and JUnit. The text suggests a process involving finding technologies, selecting for analysis, defining overlaps and constraints, and transferring changes.

The diagram includes nodes for "Find Technologies," "Select for Analysis," "Define Overlap and Constraints," and "Define Change Reactions." Branches connect these nodes to specific technologies such as JML, Java, and JUnit. The diagram is part of a flowchart that outlines a methodology or framework for co-evolving contracts, unit-tests, and source-code.
The JML-Modifier *pure* marks a method as side-effect free

Relations

Constraints
The JML-Modifier *pure* marks a method as side-effect free

Relations
- Relation between method body and modifier

Constraints
The JML-Modifier *pure* marks a method as side-effect free

Relations
- Relation between method body and modifier

Constraints
- *pure* has to be removed if method is not side-effect free anymore
The JML-Modifier *pure* marks a method as side-effect free

**Relations**
- Relation between method body and modifier

**Constraints**
- *pure* has to be removed if method is not side-effect free anymore
- *pure* can be added if method becomes side-effect free
The JML-Modifier *pure* marks a method as side-effect free

Relations
- Relation between method body and modifier

Constraints
- *pure* has to be removed if method is not side-effect free anymore
- *pure* can be added if method becomes side-effect free
- Methods used in specification clauses *have to be* *pure*
Concept
Change Reactions

Specifications → Code

detect changes

[pure added]

look for non-pure statements

[found]

block change
**Concept**
Change Reactions

**Code → Specifications**

- detect changes
  - [method body changed]
  - look for non-pure statements
    - [found]
      - remove pure
      - perform S2S handling
Concept
Change Reactions

Specifications → Specifications

- detect changes
- look for references to method
  - [pure removed]
  - [referenced in spec]
- block change
  - [any failed]
- collect calling methods
- perform C2S handling for each
Concept
Transfer

Find Technologies
- Specification Languages
- Programming Languages
- Test Frameworks

Select for Analysis
- Java
- JML
- JUnit

Define Overlap and Constraints
- JML
- Java
- JUnit

Transfer
- Contracts
- Code
- Tests

Define Change Reactions
- JML
- Java
- JUnit
Not all specification languages contain a *pure* modifier or a similar mechanism.

Evaluating specifications without side-effects is mandatory.
Specifications → Code

- detect changes
- [pure added]
- look for non-pure statements
  - [found]
- block change
Concept Transfer

Specifications → Code
Concept
Transfer

Code → Specifications

- Detect changes
- [Method body changed]
- Look for non-pure statements
  - [Found]
  - Remove pure
  - Perform S2S handling
Stephan Seifermann – Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code

November 21, 2014

12/17
Stephan Seifermann – Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code

Introduction
Related Work
Concept
Prototype
Evaluation
Conclusions
November 21, 2014
12/17
 Specifications → Specifications

- detect changes
  - [query status lost]
  - [referenced in spec]
  - [any failed]
- look for references to method
- collect calling methods
- perform C2S handling for each
Prototype Architecture

- Initializer
- JML MonitoredEditor
- Java MonitoredEditor
- SourceDirProvider
- CST Synchronization
- Vitruvius Factory
- InitializationDataProvider
- JML
- Vitruvius
- JaMoPP

Monitoring
Synchronization
Foundations
JaMoPP
Vitruvius
JML
VitruviuspFactory
CSTpSynchronization
JavapMonitoredEditor
JMLpMonitoredEditor
Initializer

«import»
«import»
«import»
InitializationDataProvider
SourceDirProvider
«import»
«import»
«import»
«import»
Prototype
Exemplary Execution

Initial artifacts:

Java

```java
public class Test {
    private int i;
    public void pureMethod() {
        int q = 1;
    }
    public void pureMethod2() {
    }
    public void nonPureMethod() {
        i = 0;
    }
}
```

JML

```java
public class Test {
    private int i;
    public /*@pure*/ void pureMethod();
    public /*@pure*/ void pureMethod2();
    public void nonPureMethod();
}
```

Introduction Related Work Concept Prototype Evaluation Conclusions

Stephan Seifermann – Model-Driven Co-Evolution of Contracts, Unit-Tests and Source-Code
November 21, 2014
Prototype
Exemplary Execution

Prepared artifacts:

**Java**

```java
public class Test {
    private int i;
    public void pureMethod() {
        int q = 1;
    }
}

public void pureMethod2() {
}

public void nonPureMethod() {
    i = 0;
}
```

**JML**

```java
public class Test {
    private int i;
    public /*@pure*/ void pureMethod();
    public /*@pure*/ void pureMethod2();
    public void nonPureMethod();
}
```
Prototype
Exemplary Execution

Changed artifacts:

Java

```java
public class Test {
  private int i;
  public void pureMethod() {
    int q = 1;
    nonPureMethod();
  }
  public void pureMethod2() {
  }
  public void nonPureMethod() {
    i = 0;
  }
}
```

JML

```java
public class Test {
  private int i;
  public /*@pure*/ void pureMethod();
  public /*@pure*/ void pureMethod2();
  public void nonPureMethod();
}
```
### Prototype

Exemplary Execution

Consistent artifacts:

<table>
<thead>
<tr>
<th>Java</th>
<th>JML</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Test {</td>
<td>public class Test {</td>
</tr>
<tr>
<td>private int i;</td>
<td>private int i;</td>
</tr>
<tr>
<td>public void pureMethod() {</td>
<td>public void pureMethod();</td>
</tr>
<tr>
<td>int q = 1;</td>
<td></td>
</tr>
<tr>
<td>nonPureMethod();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public void pureMethod2() {</td>
<td>public /<em>@pure</em>/ void pureMethod2();</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>public void nonPureMethod() {</td>
<td>public void nonPureMethod();</td>
</tr>
<tr>
<td>i = 0;</td>
<td>}</td>
</tr>
</tbody>
</table>
Evaluation
Procedure

Overall objective: Show that the concept is correct.
- Do the implemented parts work in general?
- Do the implemented parts work in all contexts?

Approach: Validation by automated tests in JavaCard API project
Evaluation

Procedure

Overall objective: Show that the concept is correct.

- Do the implemented parts work in general?
- Do the implemented parts work in all contexts?

Approach: Validation by automated tests in JavaCard API project

<table>
<thead>
<tr>
<th>Property</th>
<th>Test Suite 1</th>
<th>Test Suite 2</th>
<th>Test Suite 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage</td>
<td>path</td>
<td>context</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>system</td>
<td>system</td>
<td>unit</td>
</tr>
<tr>
<td>Selection</td>
<td>manual</td>
<td>automatic</td>
<td>manual</td>
</tr>
<tr>
<td>Syntax Check</td>
<td>yes</td>
<td>yes</td>
<td>(yes)</td>
</tr>
<tr>
<td>Semantics Check</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td># Tests</td>
<td>32</td>
<td>1085</td>
<td>134</td>
</tr>
</tbody>
</table>
Evaluation

Results

Test Suite 1 (path coverage)

- 100% succeeded
- correct syntax and semantics after transformation
- in general approach works
Test Suite 2 (context coverage)

- 95% succeeded
- correct syntax after transformation
- 0.3% revealed real errors
- implementation works in all contexts

Classification of failed tests
## Conclusions

- Approach covers complete overlap systematically
- Many constructs can be processed in a semi-automated way
- Implementation of synchronization between code and contracts
- 95% of evaluation tests succeeded

## Future Work

- Integrate approaches for contract inferring and merging
- Further evaluation – especially transfer
- Improve model printer and parser for JML
- Production-ready tools


