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

Proposal for Master’s Thesis of

Stephan Seifermann

At the Department of Informatics
Institute for Program Structures
and Data Organisation (IPD)

Reviewer: Prof. Dr. Ralf H. Reussner
Second reviewer: Jun.-Prof. Dr.-Ing. Anne Koziolke
Advisor: Max E. Kramer
Second advisor: Michael Langhammer

# Contents

1 Introduction
   1.1 Motivation ................................................. 1
   1.2 Objectives ................................................... 1

2 Foundations
   2.1 Mode-Driven Software Development ......................... 2
   2.2 Co-Evolution ................................................. 2
   2.3 Refactorings .................................................. 2
   2.4 Contracts ..................................................... 3
   2.5 Unit Tests ..................................................... 3

3 Concept
   3.1 Research Question ........................................... 4
   3.2 Procedure ..................................................... 4
   3.3 Approach ....................................................... 4
   3.4 Validation ..................................................... 5

4 Related Work
   4.1 Working with Contracts and Code ............................ 7
   4.2 Working with Contracts and Tests ........................... 8
   4.3 Infrastructure for Implementation .......................... 9

5 Organisational
   5.1 Thesis Context ............................................... 10
   5.2 Planning ...................................................... 10

Bibliography ...................................................... 12
1. Introduction

1.1 Motivation

Software failures are costly for the producers and the users of the software. The producers have to fix the bug, which might be challenging. The users of the software got wrong results in the best case or experienced harmful situations in the worst case. Therefore failures shall be avoided. In order to be attractive for the producer this shall be done with as less as possible effort.

Many techniques and tools exist to prevent software failures. One of the most popular techniques are automatically executable tests. They aim for increasing software quality and fast location of failures. Another emerging technique are executable specifications called contracts. They aim for improved documentation and avoiding mistakes caused by wrong assumptions about the functionality of a software part. Both techniques lower the costs for and during maintenance since failures are detected earlier and can be located much faster.

Unfortunately the advantages do not come without additional costs. Keeping tests and contracts consistent to code is tedious, since they can not be seen as distinct parts but have a semantic overlap. For example the change of a method name in the code requires the method name to be renamed in the tests too. As such changes occur very often an approach for maintaining the consistency is necessary, which does not leave all the work to the developer.

1.2 Objectives

Supporting the developer in keeping code, tests and contracts consistent during evolution of them is the main objective. This is achieved by a semi automated approach based on model transformations, which apply the changes to the other artefacts. For this thesis only changes, which can be mapped unambiguously and are considered useful, are handled.

The objective is divided into several sub-objectives, which consist of  a) specifying the overlap, b) defining which parts can be edited in which way in order to be useful for developers, c) developing an approach and d) implementing it.

Several techniques and approaches for transferring changes from one artefact to the other exist. Therefore the development of the approach also consists of finding and integrating existing ones. For an overview please refer to chapter 4.
2. Foundations

2.1 Mode-Driven Software Development

Model-Driven Software Development (MDSD) elevates the meaning of models from documentation to an inherent part of the development process. They can be considered as much important as code at least. In order to use models effectively an abstraction of the domain of the system under development has to be found.\cite{SV06, chapter 1.1}

The abstraction is embodied in meta-models and domain specific languages (DSLs). A meta-model defines the set of all possible models by describing possible elements and relations between them. A model always corresponds to one meta-model. A DSL is a special language tailored to best describe the domain for which it has been designed.

The advantages of using MDSD are better maintainable software and increased quality because of the abstraction, which protects the developer from many technical details. Anyhow developers have to implement a stack of methods to transform the models into code one time.\cite{SV06, chapter 1.1}

Such a stack consists of the DSL, the model to code transformations and several technologies like transformation languages, modelling languages, and so on.\cite{SV06, chapter 1.1}

2.2 Co-Evolution

Software is subject of regular changes since requirements have changed, failures have been found, and so on. In order to stay useful it has to be adjusted. As stated in \cite{Som12, p. 276f.} a system can not evolve in isolation since almost always there are other systems, which depend on it or it depends on. The same holds true for evolution of software, which consists of multiple artefacts. Developing only one of these artefacts further may cause incompatibility with the others making the software useless. So co-evolution is a principle to enhance the software without introducing incompatibilities.

2.3 Refactorings

A refactoring according to \cite{Fow99, p. 53} is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.
Also it is a regular task during development to improve the design of the software, which degrades over time. Not changing the observable behaviour holds true for the software, not for individual modules, which are subject of the refactorings and form the software. A comprehensive collection of refactorings can be found in [Pow99].

2.4 Contracts

A contract mainly consists of assertions. Assertions are statements, which must be true. A false assertion is an indicator for an incorrect system state or failure respectively. There are three types of assertions in a contract: a) preconditions describe what must be true when calling a method, b) postconditions describe what must be true after a method call and c) invariants describe what must be true during the complete lifetime of an object. [MM02, p. XI] Some programming languages like Eiffel already include support for contracts whereas for others extensions exist. For Java one of the most popular extension is Java Markup Language (JML).

The compliance with the specified contracts is checked during the runtime of the application. Whenever an assertion of a contract is violated, it is detected by the Runtime Assertion Checker (RAC). [MM02, chapter 1.6]

Some advantages of using contracts are improved reliability, better documentation and easier debugging because of a more accurate location of the failure. [MM02, chapter 8]

2.5 Unit Tests

Testing is an important part during software development because according to [HGS05] it a) checks that the software meets its requirements, b) points out defects in the design or structure of the software and c) finds bugs.

Unit tests are tests for individual units of the application. This removes the overhead of setting up and testing multiple units together. These tests are from a developers point of view and check the API of the unit. In almost every case a unit testing framework is used. A prominent example of this is JUnit, which supports Java code. Unit tests are easy to run and fast, which leads to frequent executions. Additionally they lead to an improved design, which enables testing units individually. [HGS05, p. 28-31]
3. Concept

3.1 Research Question

How can code, tests and contracts be mapped to each other in order to keep them consistent in a semi-automated approach?

- Are there any guidelines for writing contracts or tests such that consistency can be maintained easier?
- What changes of the artefacts can be considered useful?
- How can useful changes be mapped to the other artefacts?

Since there are lots of possible changes especially in code, the changes will be limited to refactorings. Furthermore it might not be possible to process all changes as indicated by [Fel03] in his research about refactorings and contracts.

3.2 Procedure

Specifying the overlap of code, contracts and unit tests is the foundation of this thesis. In general not all relations between these artefacts are useful. Therefore they have to be separated from the less useful ones.

Based on the useful relations possible changes have to be collected and their effects on the unchanged artefacts have to be found. In fact this is the specification of the mapping between the artefacts.

The relation between two of our three artefacts (code, contracts and unit tests) is subject of various related work (c.f. chapter 4). Approaches, which support the specified mapping have to be found and combined to a consistency preserving framework. This includes the implementation.

The functioning of the developed approach is shown by a case study using several open source projects.

3.3 Approach

Keeping code, contracts and unit tests consistent is the main objective of this thesis as stated in chapter 1. In order to achieve this, model transformations will be used, which
apply changes of one artefact to the others. Since model transformations only work with models, the artefacts have to be transformed to models. These models are synchronised with the artefacts permanently. The foundation of the model transformations will be the specification of the overlap and a combination of existing technologies. The whole setup is shown in Figure 3.1.

This thesis focuses on specific representations of the artefacts. Namely Java, JUnit and JML are used. The choice is based on the popularity, which implies the amount of related approaches and projects for case studies. The specific representations do not restrict the applicability because they are only used indirectly via models. Adapting the transformations to the new models should be possible with less effort.

For the creation of models from artefacts and backwards model printers and parsers will be used. JaMoPP\(^1\) is such a model printer and parser for working with Java code. As unit tests are just java code with some annotations from the syntactical point of view JaMoPP can be used for these either. For working with JML no such tool has been found yet. It might be necessary to build it from scratch using an existing parser from an open source JML library. JMLeclipse\(^2\) or OpenJML\(^3\) look promising.

Since the models are based on Eclipse Modeling Framework (EMF) the model transformations will be done with Xtend\(^4\). An approach for detecting changes and representing them as models is done in [Mes14]. The meta model for the changes is part of the implementation of the Vitruvius approach [KBL13]. Both work can be reused.

![Figure 3.1: Approach for keeping code, contracts and unit tests consistent using model transformations.](image)

### 3.4 Validation

The specified changes, which have an implemented mapping, will be tested in multiple open source projects. The open source projects will already contain JML annotations.

\(^1\)http://jamopp.org
\(^2\)http://jmleclipse.projects.cis.kasu.edu
\(^3\)http://jmlspecs.sourceforge.net
\(^4\)http://www.eclipse.org/xtend
During a short research two usable projects have been identified: The first project is a reference implementation for the JAVA CARD API. JAVA CARD is used to run Java based programs on smart cards. Further details on the project can be found in \[\text{Mos07}\]. The second project is The Kiezen op Afstand (KOA), which is a remote voting system for the Netherlands. A report on the project can be found in \[\text{KMC+07}\].
4. Related Work

Integrating contracts in the development process and combining them with existing techniques is a current subject of research. This chapter contains a short description of the state of the art.

Approaches for combining contracts with existing development processes like Extreme Programming [HN01] or for creating new processes like Agile Specification-Driven Development [OMP04] show the relevance of the subject. Both approaches state that contracts and tests have much in common but are also complementary. So using both is desirable.

The following sections give a short overview on approaches, which bring contracts and either code or tests together.

4.1 Working with Contracts and Code

The existence of a relation between changes in code and changes of contracts is stated in [Fel03]. The author inspected 68 refactorings described in [Fow99]. As a result he distinguishes refactorings, which a) have only syntactical effects on contracts, b) require additional contracts because of new code elements, c) might be not allowed because of violation of existing contracts. Contracts can be adjusted automatically only for a part of these refactorings even when using theorem provers.

Based on these findings [GFT06] introduce a tool called Crepe. It is an Eclipse plugin, which hooks itself into the refactoring engine and adjusts the contracts in the right way. It uses a Java parser to treat the contracts as code (e.g. to perform renaming). Mathematica compares contracts (e.g. checks if the precondition of subclass is weaker than the precondition of its superclass) and simplifies them. For creating new contracts Discern [FG06] is used (e.g. for a new method). Although the approach seems promising there is no implementation available.

As already mentioned [FG06] is an approach to infer pre- and postconditions from code. Contracts for the Java standard library are an additional input. Simple contracts can be inferred, which are propagated through the method and thereby are adjusted. E.g. an unchecked access to an object reference leads to the contract, that this reference has to be not null. Whereas the postconditions are still under development, the preconditions

\[^{1}\text{http://www.wolfram.com/mathematica}\]
for the classes Vector and StringBuffer could be calculated completely with some help of a developer. Unfortunately there is also no implementation available.

[CCLB12] target the Extract Method refactoring, which is hard to realise according to [Fel03]. The approach is rather complex because in contrast to Discern [FG06] the refactoring shall be proof-preserving. Although the results and the performance are good, this seems to be too complex to implement for Java and JML in the given time.

Refactoring of contracts instead of code is discussed in [Hul10]. He introduces the Pull Up Specification and Push Down Specification refactoring on contracts, which moves parts of the contract to a superclass and subclass respectively. He argues that moving specifications can simplify the code refactoring. Adjusting the callers of the changed method is still an open point. The implementation is available as open source.

### 4.2 Working with Contracts and Tests

The major advantage of using contracts is the simplification or at least improving of tests as stated in a case study on JML-based validation [dBLM+04]. Therefore several approaches exist to create tests from contracts. They all have in common that the contracts are used as test oracles. Instead of generating assert statements for the tests, runtime assertion checkers are used, which indicate an error if a pre- or postcondition is violated. Already existing runtime assertion checkers seem to be the reason for this, but even in cases where no such checker exists, the generation of it is favoured cf. [CA10]. One advantage of doing so is that one can use it while the application is running to.

[CL02] is one of the first approaches, which uses contracts as test oracles by incorporating a runtime assertion checker. A test case can have a result meaningless in addition to the usual states successful and failed. A test is meaningless iff the precondition of the method under test is violated, which means that the input data for the test is invalid. This excludes violations of preconditions of embedded method calls. A test failed iff it is not meaningless and any contract is violated. The input data for the tests is given per type. Each method is tested with all possible combinations of type compatible values. The following approaches adopt these principles and mainly differ in the generation of the test data.

[LCO+07] generate regression tests based on debug traces. The basic idea is that developers run the application many times during development with parameters, which force the new parts to be executed in order to test them. A modified IDE processes the debug trace and reduces it to the necessary system state and input parameters. Test cases can have the same three states as in described [CL02].

[EGL07] propose model driven unit testing based on contracts. Test cases are generated from a structural model and contracts are used as test oracles. The test data is generated randomly. Production rules, which are based on states indicated by the contracts, create the object under test. Anyhow the last part seems to be missing.

[Gup10] generates the test data and object under test via a state machine. He constructs it by using abstract states and evaluating the pre-, postconditions and invariants. The wanted coverage criterion for the state based tests determines the amount of test cases. In addition to the contracts the abstract state is a second test oracle. The approach is evaluated using three simple Java applications. How the approach performs in more complex systems with more possible states is unclear.

[ZN11] extends the existing tool JMLUnit, which is based on [CL02]. The developed tool is called JMLUnitNG. Both tools automatically generate complete tests including the object under test and the input data. JMLUnit uses default values for primitive data
types and prompts the developer to implement a method, which generates the necessary objects if they do not contain a default constructor. JMLUnitNG performs constructor tests based on contracts to create objects. Technical aspects changed as well: JMLUnit uses JUnit whereas JMLUnitNG uses TestNG, which reduces the memory consumption. Anyhow the amount of generated tests is overwhelming and takes much time to complete.

4.3 Infrastructure for Implementation

Since this is not the first implementation of models and model transformations to preserve consistency several methods and techniques exist, which can be reused.

VITRUVIUS [KBL13] is an approach for keeping several artefacts consistent using model transformations and a DSL to describe the mapping between the artefacts. It is developed at the chair for Software Design and Quality, where this thesis takes place. Therefore especially meta models for the representation of changes and for mapping elements on instance level can be reused. Further parts might be reused as well.

[Mes14] presents an approach to be informed about changes in the source code and how they can be mapped to refactorings. The code is represented as model using JaMoPP and changes are represented as models. Parts of [KBL13] might be reused.
5. Organisational

5.1 Thesis Context

The first advisor of the thesis is Max E. Kramer, the second advisor is Michael Langhammer. Both are researchers at the Chair for Software Design and Quality, which belongs to the Institute for Program Structures and Data Organisation at the Karlsruhe Institute of Technology.

5.2 Planning

The duration of the thesis is set to 6 months plus the time for the proposal. The proposal will end on May 12th, 2014. The remaining 6 months of the thesis will be used as shown in Figure 5.1. The schedule consists of the following parts:

Preparation This part consists of preparation tasks for the following parts. The major part will be specifying the overlap and evaluating existing technologies, which can be used for the infrastructure of the implementation. Further developing a model parser and printer for some basic statements of JML and implementing the infrastructure will be done.

Code $\rightarrow$ Unit Tests The overlap of code and tests will be kept consistent in the direction from code to unit tests. Existing approaches will be used to create and adjust unit tests after code changes.

Code $\rightarrow$ Contracts The overlap of code and contracts will be kept consistent in the direction from code to contracts. Specifying the mapping of changes will be a major part as well as combining existing approaches for synchronisation.

Code Review The implementation up to this point will be reviewed as required by the SDQ standards for thesis with implementation part.

Tests $\rightarrow$ Contracts The overlap of tests and contracts will be kept consistent in the direction from tests to contracts. Since so far no matching approach for this topic has been found, it might be necessary to develop a new approach. Since the overlap is assumed to be small, this should be affordable.

Contracts $\rightarrow$ Code The overlap of contracts and code will be kept consistent in the direction from contracts to code. Most probably this part will mainly consist of the approach of [Hul10].
5.2. Planning

Figure 5.1: Simplified schedule for thesis

Case Study  The case study will take place as described in section 3.4. The order of these tasks is based on the estimated frequency of changes in the artefacts. While changes caused by code occur quite often, changes caused by tests or contracts are seldom. So the tasks which provide the most help for developers are prioritised.

The thesis plan contains some risks. These are mainly caused by uncertainty of necessary effort and reusability of related work. This has the most significant influence on the preparation and the relation between code and contracts as it takes the longest time. Additionally working with the most third party approaches is necessary. If the time will exceed the planned amount either later parts could be skipped (e.g. the relation between contracts and code) or only the concept of some parts could be described.
Bibliography


