Karlsruhe Institute of Technology

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

Master's Thesis of

Stephan Seifermann

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

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

Duration: 4. June 2014 – 3. December 2014

Declaration of Authorship

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 14. November 2014

.....

(**Stephan Seifermann**)

Zusammenfassung

Software-Systeme bestehen heutzutage aus verschiedenen Artefakten, die jeweils einen dezidierten Zweck haben und nur einen Teil des vollständigen Systems aus einem bestimmten Blickwinkel beschreiben. Während der Implementierungsphase können Quelltext, Unit-Tests und Verträge solche Artefakte sein. Letztere werden genutzt, um Quelltext formal zu spezifizieren und diese Spezifikation gegebenenfalls auch durchzusetzen. Die genannten Artefakte besitzen inhaltliche Überlappungen, wie beispielsweise die Signatur der Methoden. Diese Überlappung muss konsistent gehalten werden, da ansonsten das Software-System nicht aus ihnen zusammengesetzt werden kann. Um die Artefakte nach einer Änderung konsistent zu halten, ist ein Ansatz zur Co-Evolution notwendig.

Es existieren verschiedene Ansätze zur Konsistenzerhaltung für Quelltext, Verträge und Tests. Diese Ansätze behandeln jedoch nur eine sehr begrenzte Menge an möglichen Änderungen und liefern häufig kein öffentlich zugängliches Konzept für eine automatische Anwendung des Ansatzes. Nach unserem Kenntnisstand existiert kein vollständiger Ansatz zur automatisierten Co-Evolution aller Artefakte.

Der Beitrag dieser Arbeit ist ein Ansatz zur Co-Evolution von Quelltext, Unit-Tests und Verträgen. Wir haben die Überlappung für Java-Quelltext, JUnit-Tests und Java Markup Language (JML)-Verträge analysiert und Reaktionen auf mögliche Änderungen definiert. Anschließend haben wir die Ergebnisse für objekt-orientierte Sprachen und beliebige Vertragssprachen verallgemeinert. Implementiert haben wir unseren Ansatz für Java-Quelltext und JML-Verträge mit Hilfe von modellgetriebenen Techniken basierend auf einem Synchronisations-Framework. Wir transformieren dabei die Artefakte von Quelltext in Modelle, wenden passenden Änderungsreaktionen an und serialisieren die Modelle wieder zurück zu Quelltext. Solche Reaktionen können recht einfach oder aber komplex sein, wie beispielsweise die Reaktion auf eine geänderte Methode: Wir prüfen, ob die Methode seiteneffektfrei ist und ergänzen oder entfernen die dazu passende Vertrags-Annotation bei der Methode und all ihren Aufrufern, falls notwendig. Zusätzlich nutzen wir einen existierenden Ansatz zur Detektion von Änderungen in Java-Quelltext und wenden detektierte Änderungen automatisch auf die anderen Artefakte an. Dieser Prozess ist vollständig transparent für den Nutzer. Dieser erhält nach einer Änderung lediglich konsistente Artefakte. Er benötigt kein Wissen über Modelle oder Transformationen.

Wir haben unseren Ansatz und unsere Implementierung mit Hilfe einer Fallstudie mit Schwerpunkt auf korrekter Behandlung von Änderungen in einem Projekt evaluiert. Dabei hat unser Ansatz für 1036 von 1085 Umbenennungen über Refactorings funktioniert (ca. 95 %). Wir konnten damit zeigen, dass er die Co-Evolution in der Implementierungsphase unterstützen kann.

Abstract

Today software systems are composed from various artifacts, which have a specific purpose and describe only a part of the whole system from a particular perspective. Source code, unit tests and contracts can be such artifacts during the implementation phase. The latter formally specifies the source code and can enforce these specifications. These artifacts have some overlap such as the signatures of methods. This overlap must not be contradictory. Otherwise, the software system cannot be composed. Therefore, an approach for the co-evolution of these artifacts is necessary in order to keep them consistent after changes.

Several approaches exist for keeping code, contracts and unit tests consistent after changes. These approaches only focus, however, on a very limited amount of possible changes and often the concept for applying them in an automated way is not publicly available. To our knowledge, no comprehensive approach exists for co-evolving all artifacts in an automated way.

The contribution of this thesis is an approach for co-evolution of source code, unit tests and contracts. For Java code, JUnit tests and Java Markup Language (JML) contracts we analyze the overlapping information and define reactions on possible changes. We generalize the results for object-oriented languages and arbitrary specification languages afterwards. We implemented our approach for Java code and JML contracts with model-driven techniques based on a synchronization framework. We transform the artifacts from code into models, apply corresponding reactions for the change and serialize the models into code again. These reactions can be quite simple or complex such as the reaction on a changed method: We determine whether the method body is side-effect free and add or remove the corresponding annotation of the contract for the method and all callers if required. Additionally we use an existing approach to monitor changes in Java code and apply the changes to the other artifacts automatically. This is fully transparent to users because they only receive consistent code artifacts after performing a change and do not need to know about models or the transformations.

We evaluated our approach and our implementation with a case study focusing on the correct handling of changes in a real-world project. We showed that our approach is able to keep code and contracts consistent for 1036 out of 1085 rename refactorings (ca. 95 %) and that it can support co-evolution in the implementation phase.

Table of Contents

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 require 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 – so-called contracts. They aim for improved documentation and avoiding mistakes caused by wrong assumptions about the functionality of a software module. Both techniques lower the costs for and during maintenance because they can detect failures early and can locate them fast.

Unfortunately, the advantages do not come without additional costs. Keeping tests and contracts consistent with code is tedious because they are no distinct parts but have a semantic overlap. For example, the change of a method name in the code requires the method name in the test to be changed too. Because such changes occur often, an approach for maintaining the consistency is necessary, which does not leave all the work to the developer.

1.2. Contributions

Our overall objective is supporting users in keeping code, contracts and unit tests consistent after changes. We do not target creating a production ready tool, but we want to define the foundations for doing this by providing a concept. Additionally, we want to evaluate this concept by implementing a prototype and using it in a case study. More precisely, we make the following contributions.

1. We define the overlap between code, contracts and unit tests as well as the constraints, which must hold true in order to preserve the consistency.
2. We develop reactions that have to be applied after a change of the overlap. Changes can be single changes or refactorings. This leads to a concept for preservation of consistency, which can be used as draft for implementing a semi-automated synchronization tool.

3. We implement a part of the consistency concept by using model-driven synchronization techniques. In doing so, we
   a) enable semi-automated consistency enforcement for Java code and Java Markup Language (JML) contracts.
   b) implement a simplified model printer and parser for the JML, which can be reused for other tools.
4. We evaluate a part of our concept with model-based techniques and show that our concept can support the co-evolution of code and contracts.

Especially our consistency concept represented by the first two contributions is highly reusable for other research. Based on the definition of the overlap additional change reactions can be developed, for instance. The whole concept can be used during the development of tools for specification languages. Precisely, our concept can support the implementation of refactorings in contracts.

2. Foundations

First, we introduce Model-Driven Software Development and how Eclipse and Eclipse-based plug-ins support it. A short explanation of co-evolution of system artifacts follows, which introduces Vitruvius and a change monitoring approach. Afterwards, a definition of change, a classification scheme used in this thesis and refactorings are described. Subsequently, we introduce contracts, which includes a general definition of contracts, a description of the Java Markup Language, OpenJML and verification techniques based on contracts. The last part subsumes unit testing with focus on JUnit and test oracles.

2.1. Model-Driven Software Development

A model is an element with three properties: First, it is a *representation* of another element, which in turn can be a model itself. Second, not all attributes of the represented element are contained in the model, so a *reduction* took place. Third, the model has to be created in a *pragmatic* way to achieve a specific goal. It does not have to be appropriate for any other use. [Sta73, p. 131-133] For a long time, models were only used as documentation of design decisions or as drafts during software development.

Model-Driven Software Development (MDSD) elevates the meaning of models from documentation to an inherent part of the development process. Models are developed and treated as any other artifacts used in the software system. Often, the models are used to generate code artifacts that are deployed. Hence, models can be considered as much important as code at least. In order to use them effectively, the already mentioned abstraction is necessary. For instance, a model for freight shipping contains the dimensions and weight and omits the other details of the objects. This abstraction is useful for freight shipping software but not for online shops. So, the abstraction is tightly coupled to the business domain. Meta-models and domain specific languages (DSLs) embody this abstraction. [SV06, chap. 1.1]

A DSL is a special language tailored to describe the domain for which it has been designed in the best way. It defines a so-called concrete syntax of a meta-model. The Java syntax is an example of a concrete syntax. A meta-model defines a set of possible models by describing legal elements and relations between them as well as constraints. Therefore, it defines the abstract syntax (elements and relations) and the static semantics (constraints). The Java abstract syntax tree (AST) is an example for an abstract syntax with respect to Java. There can be multiple concrete syntaxes but only one abstract syntax. The latter

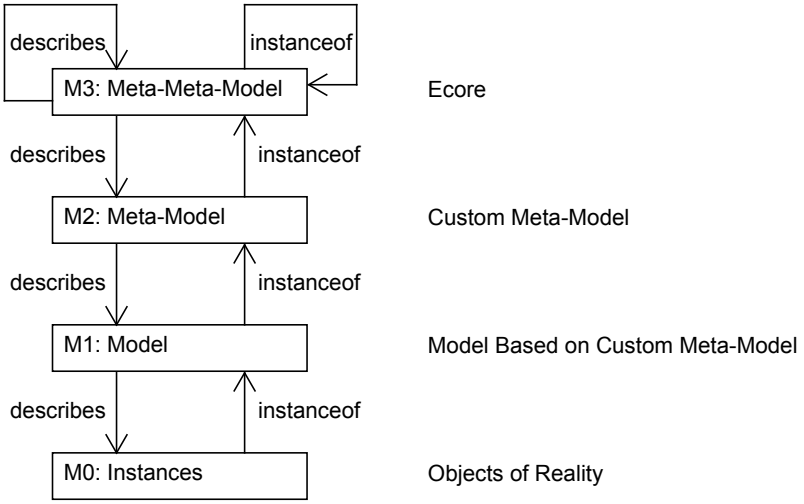


Figure 2.1.: The four meta-levels of the OMG and their typical representations in the EMF.

is important for automation processes such as model transformations while the former are important for the developer because modeling is done by using a concrete syntax. [SV06, chap. 6.1]

A model always corresponds to one meta-model. This means that a model is an instance of its meta-model. Because a meta-model is itself a model, there is a need for a meta-meta-model that defines its structure and constraints. In theory, this leads to an infinity number of meta-levels. In practice, standards exist which limit the number of these levels. The Meta-Object Facility (MOF) is the most important standard because the Unified Modeling Language (UML) is defined using this reference architecture. Figure 2.1 illustrates the architecture, which consists of only four meta-levels. M3 is the meta-meta-model, which describes all meta-models on M2. Ecore is the modeling language for this level when using the Eclipse Modeling Framework (EMF). It is almost fully compatible to Essential MOF (EMOF), which is a simplified version of the meta-meta-model specified by MOF. A combination of natural language and MOF itself defines MOF. M2 contains Meta-models created by developers. Ecore – or any other modeling language located on M3 – can be used to create them. M1 contains all models that are instances of these meta-models. The real-world objects reside on M0. [SV06, chap. 6.1]

The advantages of using MDSD are better maintainable software and increased quality because of the abstraction and automatic code generation. The former protects the developer from many technical details. The latter supersedes writing repetitive code by using code generation techniques like model transformations. The meta-models and transformations have to be created only once, but can be reused during any change or for new software systems with the same business domain. [SV06, chap. 1.1]

The concrete syntax for models often is graphical, e.g. in class diagrams. Another evolving approach is using textual syntax. One approach can be more intuitive than another one, depending on the business domain. In the context of this thesis, the artifacts are represented as code or textual respectively. In the following subsections, we introduce two well-established approaches for modeling with textual syntax.

2.1.1. EMFText

EMFText [HJK09] is a framework that allows creating a textual syntax for an existing meta-model. The framework generates the initial syntax, which is based on the Human

Usable Textual Notation[1] and some best practices. The definition of this syntax is stored in a specification model, which has to be modified to change the syntax. The modifications affect the language grammar, internal handling and pretty printer. After a change in the meta-model, the new grammar rules are added and obsolete rules are removed automatically. A warning is issued in case of customized rules. Hence, generating an initial textual syntax is possible with very less effort, but the refinement can take some time. Certainly, the refinement is an integral part of the workflow.

A model printer and parser can be generated from the textual syntax and the meta-model. It can parse existing artifacts into models and serialize models back into artifacts. Under the hood the ANTLR[2] parser is used. Therefore, there are some restrictions on the syntax such as missing support for left recursion. Non-trivial errors are not repaired automatically, but have to be fixed manually after receiving a warning.

JaMoPP [HJSW10] is one of the most prominent applications of EMFText. It provides a full Ecore meta-model for Java and a concrete syntax conform to the language specification. It contains a printer and parser for Java source and class files and resolves cross-references. Thereby, Java code can be processed like any other models and all standard mechanisms of EMF can be used.

2.1.2. Xtext

Xtext [EKZC14] is a framework for DSL developers. It targets a tight integration into the Eclipse integrated development environment (IDE) by DSL editors, which support code formatters, quick assists, cross-references and some usual Eclipse views such as the outline. The developer has to create the grammar for the textual syntax first. Various grammar fragments and extensions can be reused to save time. For instance, the Xbase expressions are a popular extension, which provide expressions in a Java like syntax.

The framework generated the meta-model, parser, serializer and all Eclipse integration mechanisms from the grammar. Under the hood, the ANTLR parser is used. Therefore, the same restrictions as for EMFText hold true with respect to the grammar. The structure of the meta-model depends on the structure of the grammar. Adjustments can only be made by expressing the same concrete syntax in different ways. A file written with the DSL can be parsed into a model which corresponds to the generated meta-model. While editing the file, the Eclipse editor updates the model after every change. It can be serialized again or can be used in code generators. Because the code generators are template based, any new representation can be produced.

2.2. Changes

A change is a modification of an artifact, which leads to a different state compared to the state before the change. This section introduces the change classification scheme that we use in this thesis. Afterwards, we define refactorings and classify them by the previously described scheme.

2.2.1. Change Classification Scheme

Changes can be classified in various ways. Instead of using an existing classification scheme, we create a new one. Thereby, we can include only necessary attributes and keep the descriptions as concise as possible. We use the properties shown in Figure 2.2 to classify changes in this thesis.

[1] `http://www.omg.org/spec/HUTN/1.0`
[2] `http://www.antlr.org`

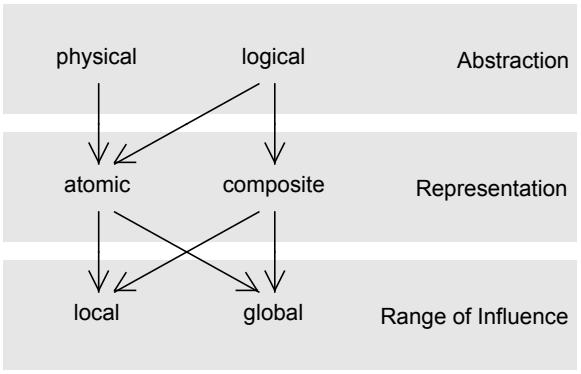


Figure 2.2.: Properties used for the classification of a change.

First, the abstraction is important. We distinguish between a physical and a logical change, where a logical change has a higher abstraction than a physical one. A physical change is a modification of an artifact that only affects one property or relation. For example, changing the name of a single method is a physical change. In contrast, a logical change is the intended change, which might be carried out by multiple physical changes. Renaming a method and updating all references to that method is considered a logical change. A change can be considered physical and logical if the intended change is equal to the physical change. Adding a comment is an example of such a change.

Second, the representation of a change can vary from a single atomic to a composite change. An atomic change consists of only one change, while a composite change consists of at least two changes. The meta-model for changes dictates when to use an atomic or composite change. It is more likely to have more composite changes when using a generic meta-model than when using a specific one because the change might not be describable in a concise way: For instance, a rename refactoring could be represented as one rename-refactoring-change in a specific meta-model whereas it could be represented by a composite change consisting of many update changes of the name attributes in a generic meta-model.

The range of influence is the third classification property. We distinguish local and global changes. A local change can affect directly corresponding elements only. In contrast, a global change might affect other elements. Therefore, as soon as we have to consider any non-corresponding elements, we classify the change as global. Obviously, this depends on the structure of the correspondence. If we maintain correspondences very detailed, there might be less global changes.

The context of the change has to be considered when classifying with the mentioned properties. For instance, comparing the representation is only possible if one change meta-model is as generic as the other one. So comparing properties without comparing contexts is not useful. In this thesis, we have a fixed context consisting of detection mechanisms, used meta-models and correspondence structures. Therefore, we can omit the context comparison.

2.2.2. Refactorings

According to [Fow99, p. 53], a refactoring is

> a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

It is a regular task during development to improve the design of the software, which degrades over time. Not changing the observable behavior holds true for the software

but not for individual modules that are subject of the refactorings and form the software. [Fow99] gives a comprehensive collection of common refactorings.

Classifying a refactoring in general is hard because there are many different refactorings. Instead, we classify the average refactoring. Reading the refactoring description of Fowler reveals that they are logical change because the focus relies on the intention. The refactorings will most likely be represented as composite changes in a generic change model because they affect many different locations and features. The range of influence depends on the concrete refactoring and context, but most likely many not directly corresponding elements have to be adjusted. Therefore, we consider refactorings global.

2.3. Co-Evolution of System Artifacts

Software is a subject of regular changes because of changed requirements, found failures and so on. In order to stay usable, it has to be adjusted. As stated in [Som12, p. 276f.], a system cannot evolve in isolation because usually other systems depend on it or it depends on other systems. The same holds true for evolution of software that consists of multiple artifacts. Elaborating only one of these artifacts can make it incompatible to other artifacts, which makes the software useless. Co-evolution is a principle to enhance the software without introducing such incompatibilities.

There are multiple approaches for co-evolving multiple corresponding artifacts. Manually adapting artifacts after changing another one is one simple approach. This is error-prone, however, because a change can be forgotten or the adaption can be made in a wrong way. This approach is not desirable. Nevertheless, this can work well for a limited amount of small artifacts. Synchronization techniques are a safer approach for co-evolution because they use rules to adapt artifacts. This leads to correct results as long as all changes on the artifacts are detected and the rules for the adaption are correct. There are two requirements for co-evolution by using a synchronization technique: We need a synchronization framework, which handles the most basic task and is extensible by adaption rules. Additionally, we need a mechanism to detect changes in the artifacts. We introduce the approaches for these requirements in the following subsection. Both will be used in the context of this thesis.

2.3.1. Vitruvius

In MDSD, multiple models and meta-models describe various aspects of the software system. Multiple models are useful because specific models can describe specific aspects more concise than general models. A universal meta-model cannot fulfill this need because it has to be general enough to describe all aspects. These models do not have to describe disjunctive parts of the software but might overlap. We need to solve two issues in order to develop a system with the MDSD approach: a) How shall the models be separated in order to support all roles during the development in the most effective way? b) How can shared (overlapping) information be kept consistent?

Vitruvius [KBL13] is an approach that aims to solve both of these issues using some well-known techniques. Views prohibit the fragmentation of the models, which is caused by providing all roles during development with the most concise data (item a). The concept of views is well known in the context of databases. They can combine elements from various tables and display them in an appropriate way. Changes in the views are propagated back to the original tables. This concept works in MDSD as well by replacing tables with models. Thereby, all roles can be supported without the need for high model fragmentation. This leads to flexibility because new views can be added without changing the persisted data in the models or the corresponding meta-models.

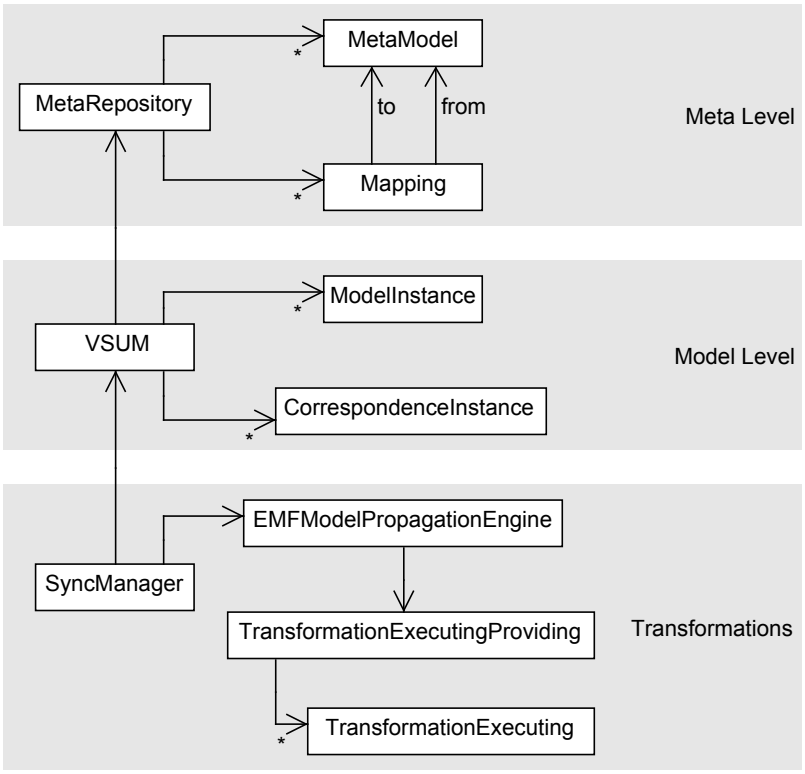


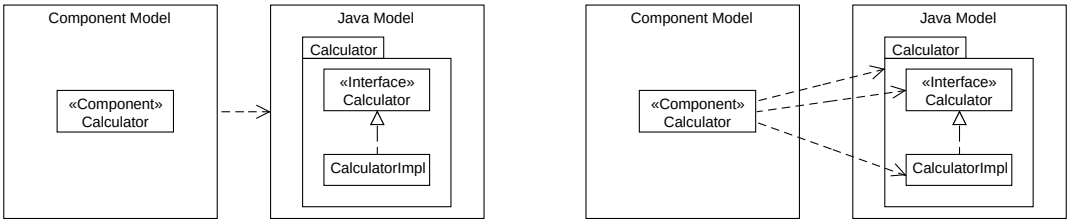
Figure 2.3.: Overview of the core classes of the Vitruvius framework.

Model transformations keep the remaining models and the views consistent (item b). Writing such transformations requires a considerable effort and can be error-prone if many meta-model elements exist. Therefore, Vitruvius suggests a DSL for formulating the dependencies between the models. The transformations are generated based on that information. The user has to specify mappings between the models, invariants for and between models and response actions, which may inform the developer or ask him to solve a complex situation. The generated model transformations are executed after a corresponding change occurred.

While Vitruvius is still a subject of research, a prototype of the framework already exists. At the moment it is focused on the synchronization part. The DSL and the view generation is not included yet. Figure 2.3 gives an overview of the most important classes of the framework. The concepts behind these classes are necessary for other synchronization approaches too. Therefore, they are described in the following paragraph. They are separated in the meta-level, model-level and transformations.

First, we need to know the meta-models of the synchronized models because transformation tools use the abstract syntax to transform models rather than any concrete syntax. Additionally, we need to know the relation between these meta-models (so-called mappings). A mapping is a unidirectional relation that defines that a synchronization from instances of one meta-model to instances of another one is necessary and which constraints have to hold true for the relation. A meta-repository encapsulates this information.

On the model level, a virtual single underlying model (VSUM) encapsulates all information. It contains all models as well as correspondences between model elements. There is an important difference between the mappings described above and correspondences: Mappings are defined on the meta-level. Hence, they define relations between types of elements. In contrast, the correspondences reside on the model level. Hence, they define relations between elements of the models. For example, you could have a component and a



(a) A mapping stating that changes in the component model have to be propagated to the Java model.

(b) Correspondences stating that some Java elements are related to a component.

Figure 2.4.: Illustration of difference between mappings and correspondences as used in Vitruvius.

Java meta-model as well as instances of these meta-models. A mapping says that changes in component models have to be propagated to the Java models as seen in Subfigure 2.4(a). A correspondence says that changes on a specific component have to be propagated to specific elements in the Java model as seen in Subfigure 2.4(b). Correspondences are necessary to find the elements that need to be adjusted. For example, after changing a component we need to know which Java package has to be adjusted. Therefore, it is crucial to keep the correspondences up-to-date after changes in the models. Otherwise, we cannot adjust the correct model elements. Correspondences are represented as a model internally. A single entry holds the two corresponding elements and dependent correspondences. The latter is useful when an element (and thereby all of its contained elements) has been deleted and the correspondences shall be deleted as well.

The third part of Vitruvius are transformations. A synchronization manager holds a propagation engine for EMF model changes. It propagates an arriving change to a provider for transformations, which chooses the matching transformation. Inside the transformations, the VSUM is used to update the models and correspondences.

The synchronization engine of Vitruvius distinguishes between three types of changes: File changes, EMF model changes and composite changes. File changes represent create and delete operations on files. This intentionally excludes the contents of the file. EMF model changes represent any change operation on models. A composite change is an ordered list of changes, which can include further composite changes. This is useful if multiple changes occurred and the change monitor was unable to separate the changes into logical ones. The synchronization framework tries to process the changes as a group.

The changes are models too. There are 25 different change types. They are separated in changes of features and objects. Feature changes are further separated into create, update, delete and permute operations on attributes and references. Reference are divided into containment and non-containment references. For object changes there are create, update and delete operations for lists and single valued references. Some of the change types overlap.

Every change type has specific attributes, which are necessary to perform the transformation. The example in Figure 2.5 illustrates the change model after replacing a non-root object in a single valued reference. It has references to the object affected by the change before and after the change occurred. Additionally, it references the old value, the new value and the feature, where the change took place. The old elements are necessary to find the corresponding elements in the other models, which have not been changed yet. The new value is necessary to apply the change. The new affected object is necessary to update the correspondences.

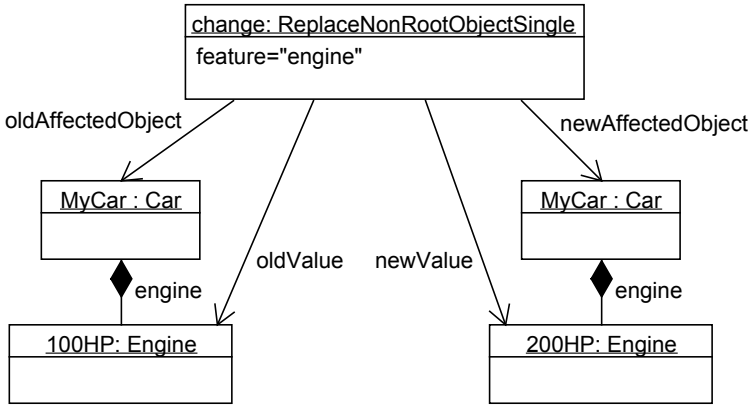


Figure 2.5.: Example for a change model after replacing a non-root object in a single valued feature.

2.3.2. Change Monitoring

Changes have to be detected in order to co-evolve artifacts. A detection approach often focuses on a specific type of artifact and environment. A common situation is editing Java code in the Eclipse IDE. [Mes14] presents an approach for detecting and classifying such changes. Especially, the latter is interesting because a logical change can consist of multiple physical changes. For instance, a rename operation can consist of a delete operation for the old and an add operation for the new method. Determining the logical change is not trivial in every case.

In order to detect the changes, listeners for multiple extension points of Eclipse are registered. Thereby, manual changes, quick assists and refactorings are covered. Classifiers create logical changes by combining multiple physical changes or creating new ones. Even corresponding changes that are separated by time can be combined by withholding the changes and classifying them later. The last step consists of sending the changes to the Vitruvius framework. Because the framework can only handle model changes, the internal change representation is converted to model changes. For that purpose, the Java code is parsed into a model before and after the change and the changed model elements are used to create the model change. This process is extensible via Eclipse extension points. Thereby, new classifiers and change converters can be added.

A combination of the change detection with Vitruvius has already been tested in a case study for that approach [Mes14, pp. 68 sqq.]. The setup consisted of Java code and an architectural description located in a Palladio Component Model (PCM) model, which is used for performance analysis of components and whole systems [BKR09]. The renaming of a Java class and the removal of the public modifier from a Java class have been tested. The changes could be detected and the framework was able to synchronize the change successfully.

2.4. Contracts

A contract specifies the desired observable behavior of a part of a software. It mainly consists of assertions. Assertions are statements that must be true at the time of evaluation. 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 the Java Markup Language (JML) (see Subsection 2.4.1).

The advantage of using the integrated contract features or equivalent extensions instead of exception mechanisms provided by the programming language is that the contract is executable. This is often called *executable specification* [PFP+13]. This means that the compliance of the implementation with the contracts is checked during the runtime of the application. Whenever an assertion of a contract is violated, the Runtime Assertion Checker (RAC) detects it.[MM02, chap. 1.6] Some tools distinguish between precondition violations of the first called method and violations of internally called methods. The former is called *violation of an entry precondition*. It indicates an error of the caller because the provided parameters or the current state of the object did not fulfill the precondition. This means the caller did not meet its obligations. Other precondition violations indicate an error of the implementation because the caller met its obligations.

Some advantages of using contracts are improved reliability, better documentation and easier debugging because of a more accurate location of the failure.[MM02, chap. 8] Anyway, the effort to write useful contracts has to be considered and compared to the advantages. [PFP+13]

The following subsections deal with the JML and OpenJML. We will just cover the most basic features required for this thesis in short.

2.4.1. JML

The Java Markup Language (JML), according to [LBR99], is

> a behavioral interface specification language designed to specify Java modules. Java modules are classes and interfaces.

The behavior covers the interface of the modules, which is visible to the client, and the internal behavior, which is invisible to the client. The specifications are noted inside comments. Therefore, the Java application can be compiled and run without the need for a JML compiler. Obviously, the contracts do not have any influence anymore when doing this. [LBR99] Noting contracts in comments can be considered as disadvantage too because the Java compiler has to be replaced in order to translate such specifications. Section 3.3 gives a good overview of various other approaches. Anyway, none of these is as popular as JML [CJLK10] because it is used in teaching formal methods at universities and in research.

The JML features are grouped into language levels. The higher the level of a feature is the less implementations support it. Therefore, we focus on level 0 and 1, which can be considered available in all and most JML tools respectively. Additionally, we do not consider features, which are not supported by OpenJML. OpenJML is a tool suite for JML, which is described later (see Subsection 2.4.2). For a comprehensive overview of all JML features, please refer to [LPC+13] instead.

The most common language features are demonstrated in Listing 2.1. The class implements a very simple bank account with no credit line. JML specifications are noted in the comments which start with `//@` for single line or `/*@` for multi-line specifications. The specification and implementation of the example have weaknesses and are only used to support the understanding.

In JML, specification elements can only be used if they are visible according to Java rules and their visibility is higher than the visibility of the specification. Hence, we cannot use a private field in a public specification. To circumvent this restriction we can declare the field spec_public, which makes the field public for the specification. The visibility for Java is not changed. The keyword spec_protected makes the element protected respectively.

```
1  public class BankAccount {
2    private /*@ spec_public */ int balance;
3    private /*@ spec_public */ int creditLine;

5    //@ public model int creditLine_;
6    //@ represents creditLine_ = creditLine;

8    /*@ model public pure helper int getCreditLine_() {
9          return creditLine_;
10       }*/

12   //@ invariant getBalance() >= 0;

14   //@ initially getCreditLine_() == 0;

16   //@ requires initialBalance >= 0;
17   //@ ensures getBalance() == initialBalance;
18   public BankAccount(int initialBalance) {
19     balance = initialBalance;
20     creditLine = 0;
21   }

23   //@ ensures \result == balance;
24   public /*@ pure helper */ int getBalance() {
25     return balance;
26   }

28   //@ requires amount >= 0;
29   //@ ensures getBalance() == \old(getBalance()) + amount;
30   //@ ensures receiver.getBalance() == \old(receiver.getBalance()) - amount;
31   //@ signals (IllegalArgumentException e) getBalance() == \old(getBalance())
       && receiver.getBalance() == \old(receiver.getBalance());
32   //@ signals_only IllegalArgumentException;
33   //@ assignable balance, receiver.balance;
34   public void transfer(/*@ non_null */ BankAccount receiver, int amount) {
35     if (receiver.getBalance() < amount) {
36       throw new IllegalArgumentException();
37     }
38     balance += amount;
39     receiver.balance -= amount;
40   }
41 }
```

Listing 2.1: Example for contracts written in JML for an unfinished bank account.

There are even more statements, which affect the specifications only: In the sixth line, we create a new field creditLine_. This is called a model field because it is marked with the model keyword. The value of this field is calculated by the expression after represents every time it is accessed. In contrast, the value of ghost fields is not calculated but set explicitly via set statements. Model methods can be created too. They have the same syntax as regular Java methods except for the model keyword and the access to specification-only elements such as the model field creditLine_. Such methods can be used to outsource some calculations used in specifications. If elements have to be imported just because they are used inside a model method, then a model import can be used. So, the Java compiler will not warn about unused imports.

The three basic elements of a contract are also covered by JML. Invariants can be defined with the invariant keyword following the assertion. All visible Java and JML elements such as methods or fields can be used. Using methods in invariants can lead to infinite loops depending on the implementation: Assume that invariants are checked as soon as

a method is called. If the invariant contains a method, then this method is called during the evaluation of the invariant. This leads to another check of all invariants, in which the method is called again. The helper modifier can be used to work around this issue. It excludes the methods from the invariant checking, so no infinite loop can occur. This should only be used with pure private methods.

Preconditions and postconditions are added directly before the method. The requires keyword introduces preconditions, which are evaluated when the method is called. ensures introduces postconditions, which are evaluated when the method is left. If there are multiple preconditions or postconditions, the conjunction of them has to be fulfilled. In postconditions, the returned value of the method can be referred by using \result. The result of an expression before execution of the method can be referred by surrounding it with \old(expr). The initially clause can be used to specify postconditions which have to be true for all constructors.

There is support for multiple specification cases by using heavyweight specification cases. They are introduced by normal_behavior, exceptional_behavior and behavior (not covered by the example). Each specification case has its own precondition to determine whether it can be applied and the corresponding postconditions have to hold true. Some simpler handling of exceptional behavior can be achieved by using so-called lightweight specification cases. They have no introducing keyword like all of the specifications in the example. Exceptions are covered by signals and signals_only. The former specifies exceptions that might be thrown. The corresponding predicate has to hold true after throwing it. The latter specifies which exceptions can be thrown and by that, which exceptions cannot be thrown.

Frame rules specify the parts of an object's state that can be changed by the method. Any part not mentioned in this clause cannot be changed. In the example, the transfer method can only change the balance values because they are mentioned in the assignable clause. The credit line cannot be changed.

To ease writing specification for common cases JML provides some additional keywords that are shortcuts for preconditions and postconditions. Query methods are marked with the pure modifier. A query method is a method, which must not have any side-effects. Parameters, fields or methods can be annotated with non_null to indicate, that their value or return type respectively can never be null. In fact, object references are considered not to be null by default. This can be changed with nullable_by_default.

Separate specification files allow specifying existing libraries or source code that cannot be changed. In contrast to the example, the JML statements are not added to the implementation but to a separate specification file. This file contains all elements of the regular source file except for method bodies and initializers. The specifications are placed in the already mentioned way. This mechanism can be used to specify regular Java files too. Thereby, the implementation can be separated from the specification.

This introduction did not cover all level 0 and 1 features. Especially, self-explanatory expressions were not mentioned. The remaining features are data groups (see [LPC+13, chap. 10]), loop specifications (see [LPC+13, chap. 13.2]) and the following statements: assert, assume, initializer, constraint, axiom.

2.4.2. OpenJML

OpenJML is a tool suite for working with JML. In the context of this thesis, the compiler is the most important tool. It can process Java up to version 7 and almost all JML features of level 0 and 1. After compilation, the specifications can be checked during runtime. Under the hood, the OpenJDK compiler is used. It has been extended to support easy evolution of OpenJML. [Cok14]

The integration in other tools is supported by an application programming interface (API), which gives access to the parsed AST, pretty printing, compiling, verification results and so on. Because it is open source, it can even be extended to fit some unmet needs.

There are many other tools besides OpenJML, which are described in [CJLK10] for instance. The main differences are the used technologies, the usability and the language support for Java and JML. Because OpenJML supports a recent Java version, is open source and provides an API, it is a good choice to build other tools on. The current implementation status is given in [Cok14], but a better support has already been announced.

2.4.3. Static Checking and Verification

Using contracts offers new possibilities to detect errors in the specified code. Sure, this only holds true as long as the contracts are correct with respect to the wanted behavior. As soon as this changes any bug finding approach described in the following paragraphs will result in false positives. Additionally, all advantages of contracts are eliminated by wrong specifications. This subsection describes the most used and well-known approaches for finding bugs in code using contracts.

Program verification is a powerful approach to find bugs. Special verification frameworks are available, which vary in handling, requirements for the system to verify and abilities. For instance, the KeY system [BGH+07] is such a verification framework for Java and JML. It is able to solve non-linear arithmetic and combines automatic and interactive proving. Many bugs can be discovered by such verification techniques as long as the specifications are strong enough and cover the whole behavior. This is one of the biggest disadvantages because often contracts are not as detailed as required. Therefore, the specifications have to be strengthened, which leads to a considerable effort.

Extended static checking [FLL+02] is a trade-off between the effort to write specifications and the percentage of discovered errors. The checking is performed in a separate step after the compilation. In short, predicates and conditions are inferred and proven by a theorem prover. At any time, only the implementation of a single method is checked at once. As soon as other elements are used in specifications or the method implementation, only its specification is considered. This reduces the runtime and makes the checks modular. This approach is neither sound nor complete.

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 developer's point of view and check the API of the unit. In almost every case, a unit testing framework is used. JUnit is a prominent example of a framework supporting 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 of units individually.[HGS05, p. 28-31]

The following subsections describe JUnit in short and explain two possible types of test oracles, which are used in this thesis.

2.5.1. JUnit

JUnit is the most popular unit testing framework for Java. It is well integrated in IDEs and there is many documentation for it. The framework contains a library with base and

utility classes and test runners. A test runner executes test suits. According to [HT04, chap. 3], a test suite is a set of tests, which shall be executed. A test is located in a test case, which is a set of tests for the same subject and implemented as a class. An example is given in Listing 2.2.

```
1  public class ClassUnderTestTest {
2    private static Resource res;
3    private ClassUnderTest objectUnderTest;
4    @BerforeClass
5    public static void init() {
6      res = new ReadOnlyFile("/path/to/file");
7    }
8    @Before
9    public void setup() {
10     objectUnderTest = new ClassUnderTest(res);
11   }
12   @Test
13   public void testGet() {
14     assertEquals(42, objectUnderTest.get());
15   }
16   @After
17   public void tearDown() {
18     objectUnderTest.freeResources();
19   }
20   @AfterClass
21   public static void finalize() {
22     res.close();
23   }
24 }
```

Listing 2.2: Simplified structure of a JUnit test case.

The test case ClassUnderTestTest consists of various methods, which support the typical steps during a test mentioned in [HT04, chap. 3]: a) setup structures and resources, b) execute the method under test, c) check the result and d) clean up. Setup and cleanup are necessary because a test shall be independent [HT04, chap. 7.4], which includes independence of previous tests. JUnit distinguishes between setup and cleanup for the whole test case and for an individual test. In Listing 2.2 the methods init and finalize are the setup and cleanup methods for the whole test case. So, they are executed before and after all tests respectively. The methods setup and tearDown are executed before and after every test. The test method testGet is quite simple because many tasks are performed by the other methods. It consists of input data generation, calling of the method under test and checking the results. The results are checked by so-called test oracles (see next subsection). A unit test has three states. If a check failed, then the corresponding test fails too. A test can be ignored by telling the test runner to ignore it. A test succeeds if no checks failed and the test is not ignored.

The runner of a test suite can be individually chosen by annotating a test case. The parameterized runner is quite interesting in our context. It takes the test case and a set of test data. The test case is initialized with every tuple of test data and all tests are run. For instance the same calculation method can be tested with various input data without the coding overhead of multiple test cases. [JUn14, Parameterized tests]

2.5.2. Traditional and Contract-Based Test Oracles

Test oracles decide whether a test fails or succeeds. The necessary input data for this decision varies depending on the type of test oracle. In the context of this thesis, two types of oracles are important: The traditional assert statements and runtime assertion checks.

Assertions are boolean conditions, which must hold true in the context and at the time of evaluation. Unit testing frameworks contain special assert statements, which are used to check the results and context after running the unit test. Such statements usually are used to compare concrete values for the current test case. Therefore, they have to be adjusted in order to use them in other test cases. For example, an assert statement for a calculation method to sum up two numbers would check the concrete result for concrete input values. Such assertions are easy to understand because they are specific for the current context.

Runtime assertion checks are verifications of the contracts during runtime. They are inserted in the code during compilation. In contrast to the assert statements, the runtime checks use a generic description of the result and context for the verification. Therefore, it is not necessary to adjust them for any test case. For example, a runtime assertion check for the mentioned sum up method would ensure that the result is the sum of the operands without storing a set of input data and matching result values. The most important advantage of such checks is the automation: No explicit check statements have to be inserted in the code or the test. The results are guaranteed to be correct or invalid data is detected as long as the contracts are strong enough and correct.

Various oracle types can be combined and mixed. Additionally, some runtime assertion checks can be simulated using traditional assert statements. For instance, the assert statement could use the input values, perform an addition and compare the result.

3. Related Work

Integrating contracts in development processes and existing tools is a subject of research. There are approaches, which try to integrate contracts in existing processes like Extreme Programming [HN01] or which introduce completely new specification-driven processes [OMP04]. Because contracts are heavily used and evolved as part of such processes, there is a requirement for systematic evolution approaches.

The remainder of this chapter is structured as follows: The first section deals with the relation between contracts and code. The relation between contracts and tests is subject of the second section. The third section covers techniques to integrate contracts into Java code.

3.1. Relation between Contracts and Code

This section deals with related work about the relation between contracts and code. Figure 3.1 gives a first overview of the approaches that are described in the following paragraphs.

[Fel03] gives a high-level overview of code changes and their effects on contracts. The author inspected 68 refactorings of Fowler [Fow99] and grouped them into three categories: Refactorings that a) have only syntactical effects on contracts, b) require additional contracts because of new code elements, and c) might be forbidden because of a violation of existing contracts. The effects of the *Extract Class*, *Self-Encapsulate Field*, *Move Field* and *Move Method* refactorings on the contracts are described. Additionally three refactorings that maintain contracts rather than code are introduced: *Pull Up Assertion*, *Pull Down*

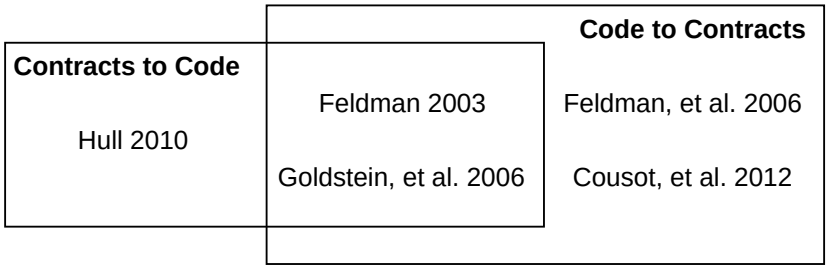


Figure 3.1.: Overview of the related work covering the relation between contracts and code.

Assertion and *Create Abstract Precondition*. The latter introduces a precondition in an abstract superclass, which refers to an abstract method. During the implementation of this method, the subclasses can specify it. This leads to concrete specifications in the subclasses. Regarding the author, not all refactorings can be processed automatically even if theorem provers are used. The *Extract Method* refactoring is presented as an example. Although it can be automated to some degree, the developer has to check the results afterwards. Unfortunately, the full analysis and the corresponding change handling approaches are not made public to our knowledge.

Based on these findings [GFT06] introduce a tool called Crepe. It is an Eclipse plug-in, which hooks itself into the refactoring engine for Java code and adjusts the contracts in the right way. It uses a Java parser to treat the contracts as code, which is required for syntactical refactorings like renaming. This only works as long as the specification language for the contracts uses Javadoc comments to inject the contracts into Java. For instance, JML contracts cannot be processed with this tool because they are placed in regular comments and the Java parser does not process the contents of regular comments. Mathematica[1] is used to compare contracts and to simplify them. For instance, this is necessary for checking whether the precondition of a subclass is weaker than the precondition of its superclass. For creating new contracts, Discern [FG06] is used. It is used after adding a new method, for instance. Although the approach seems promising, there is no implementation available.

[FG06] is an approach to infer preconditions and postconditions. It uses the source code and predefined specifications of the Java standard library. Postconditions are not covered by the prototype yet, however. The tool aims for inferring simple contracts to prevent the the developers from writing them. They can concentrate on contracts that are more complex instead. The approach uses the propagation of weakest preconditions to determine the contracts. *Weakest* means that the least restrictive contract is inferred, which still guarantees that the method returns without errors. For example, an unchecked access to an object reference leads to the specification that this reference must not be null. When a method is called, the specification for it is used to determine the weakest precondition. The whole process is iterative: After inferring contracts, the developer can provide new invariants and restart the whole process, for instance. This leads to specifications that are more restrictive. The tool has been evaluated by inferring the specifications for the Java classes Vector and StringBuffer. Unfortunately, there is no implementation available.

[CCLB12] target the *Extract Method* refactoring, which is hard to realize according to [Fel03]. The approach is quite complex because, in contrast to Discern [FG06], the refactoring shall be proof preserving. This means that a verifying tool can proof the correctness of the old and the newly created method. 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]. The author 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 as a preparation of a code-refactoring can simplify the actual refactoring. The approach is implemented as Eclipse plug-in for Java code and uses a try and error algorithm. For instance, when moving a specification from a subclass to a superclass, the source is compiled and verified to find missing fields and methods and to find conflicting specifications. Adjusting the callers of the changed method is still an open point, so developers have to adjust them manually at the moment. The implementation is available as open source.

[1] `http://www.wolfram.com/mathematica`

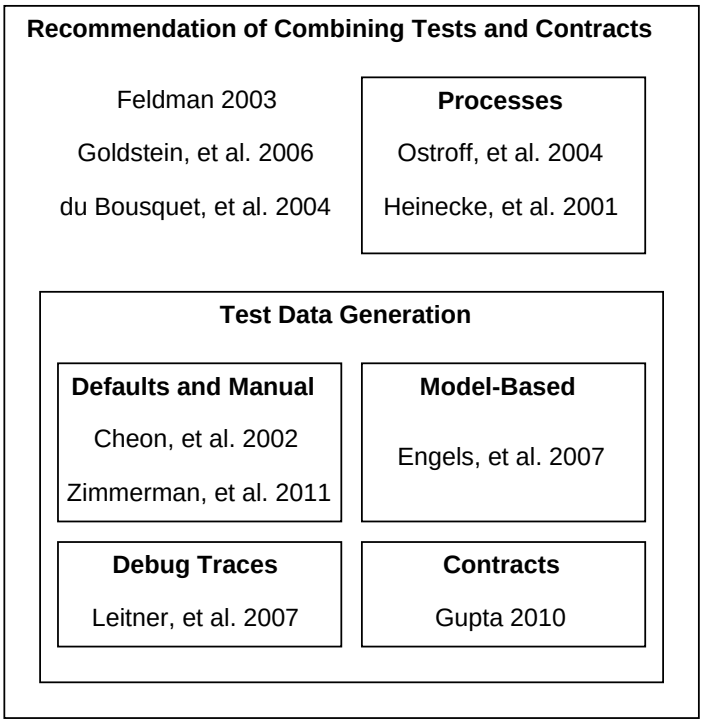


Figure 3.2.: Overview of the related work covering the relation between contracts and tests.

3.2. Relation between Contracts and Tests

This section describes related work covering the relation between contracts and tests. Figure 3.2 gives a first overview of the approaches that are described in the following paragraphs.

During the development of a specification-driven process [OMP04] the authors discovered many similarities between contracts and tests: Both specify the wanted behavior, are lightweight verification methods and aim for higher quality. The authors recommend a combination of both techniques: Unit tests are used to exercise methods and fill the gaps of weak specifications. Contracts are used to strengthen the tests and make writing them easier. The previously mentioned approaches for the relation between code and contracts [Fel03; GFT06] and a case study on JML-based validation [dBLM+04] propose the combined usage of contracts and tests too.

Several approaches exist to create tests, which incorporate 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 a violation of the specification or the wanted behavior respectively. Existing runtime assertion checkers enable these approaches, but even if no such checker exists, the creation of it is favored by approaches such as [CA10], which transforms specifications to executable code.

[CL02] is one of the earliest approaches that 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 if 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 fails if it is not meaningless and any contract is violated. A set of test data is given by the user or randomly generated for every possible parameter type. The set of test data for a single method is defined in Definition 3.1. The approach has been published as JMLUnit. The following approaches adopt these principles and mainly differ in the generation of the test data.

Definition 3.1 (Test data set for a single method using the [CL02] approach)**.** *The test data for a given method signature m is defined as $\mathcal{X}(m)$ with*

- $\mathcal{T}$:= *set of all data types*
- $\mathcal{V}(t)$:= *set of data values for given data type* $t \in \mathcal{T}$
- $\mathcal{T}^n$:= *set of all parameter type tuples representing signatures with* $n \in \mathbb{N}$ *parameters*
- $\mathcal{X}(m) := \mathcal{V}(m_1) \times \cdots \times \mathcal{V}(m_n)$ *is the set of test data tuples with* $m \in \mathcal{T}^n$

For example, assuming that $\mathcal{T} = \{int, char\}$, $\mathcal{V}(int) = \{0, 1\}$, $\mathcal{V}(char) = \{\text{'a'}, \text{'b'}\}$
the set of test data for a method with the parameters $m = (char, int)$ *is*
$\mathcal{X}(m) = \{(\text{'a'}, 0), (\text{'b'}, 0), (\text{'a'}, 1), (\text{'b'}, 1)\}$.

[LCO+07] generate regression tests based on debug traces. The basic idea is that during development developers run the application many times with parameters that 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 described in [CL02].

[EGL07] propose model-driven unit testing based on contracts. Test cases are generated from a structural model of the software system and contracts are used as test oracles. The test data is generated randomly. The object under test is created by production rules, which are based on the states implied by the contracts. The latter is not described in the paper, so we consider it as future work.

[Gup10] generates the test data and object under test via a state machine. The states represent the possible system states. A state is defined by the values of the available variables. Abstractions are used instead of concrete values to prevent a state explosion. The transitions from one state into another are defined by the contracts. Preconditions indicate the start for the transition and postconditions the possible targets. The wanted coverage criterion for the state based tests determines the amount of test cases. The approach is evaluated using three simple Java applications. It is unclear how the approach performs in more complex systems that have more possible states.

[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 are changed as well: JMLUnit uses JUnit whereas JMLUnitNG uses TestNG, which reduces the memory consumption. Anyhow, the amount of generated tests is overwhelming, so they take much time to complete.

3.3. Integration of Contracts in Java

There are multiple approaches for introducing contracts in Java. In [CCH08] a subset of them is classified into six categories and compared as part of the development of a new approach. We pick one approach from each category and give a short overview of the features as well as the advantages and disadvantages based on [CCH08] and our findings. Approaches that use JML as specification language are ignored. We do not aim for a comprehensive comparison of all features, implementation details and other differences. Instead, we want to introduce other ideas, which can be compared with JML. We will use the results later to generalize the mapping concepts in Chapter 4, which are developed with focus on JML.

In [LKP02] the open-source Java compiler from the Kopi tool suite is extended by new keywords for contract specification. During the compilation, the new statements are translated

to byte code, which enforces the contracts. Preconditions, postconditions and invariants are supported. If such a specification is violated, a specific exception is thrown. The advantage of such an approach is the direct relation between the contracts and the code from the compiler's point of view. Indeed this comes with some disadvantages like a special compiler and a considerable effort to disable existing contracts.

The jContractor approach [AK02] uses naming conventions to introduce contracts. Specifications are written into methods following these naming conventions, which define the meaning of the method. There are method names for preconditions, postconditions and invariants. During a post-processing, checks are added which use these methods to determine whether the specification is met during runtime. Because the contracts are plain Java code, existing refactorings of the IDE and standard compilers can be used. Disabling the contract checking is possible by simply omitting the post-processing step. On the other side, introducing contract specific statements is only possible by workarounds. For example, the old state has to be accessed via a special field in jContractor.

A similar approach is used by ezContract [CCH08]: The specifications are located in Java code too and can be treated as regular code. As opposed to jContractor, the specifications are located inside the body of the method that shall be specified. The specification statements are grouped by enclosing method calls of marker methods. The marker methods determine the meaning of the group. There are methods for preconditions, postconditions and invariants. Additionally, there are special methods for accessing old or result values. The checking of the specification is activated during a byte code instrumentation after the regular compilation step. The advantages and disadvantages are the same as for the naming conventions approach mentioned above.

Jass [BF01] uses comments to introduce contracts. A special compiler is used to parse the comments and to translate the specifications to byte code. It supports preconditions, postconditions, invariants and loop specifications. Additionally, special blocks exist which can be used to perform exception handling. The advantages of this approach are the easy introduction of new keywords, compatibility with existing Java compilers (as long as contracts shall be ignored) and easy disabling of contract checking. The most important disadvantage is the usage of comments because IDE features such as refactorings or code completion cannot be used anymore.

Java annotations are used by the Conaj approach [SL04]. There are predefined annotations, which take a string argument that contains the specification. A set of customized preprocessors, parsers and compilers is used before the checks are woven in. Preconditions, postconditions and invariants can be specified using this approach. The advantages and disadvantages are the same as for the contracts specified in comments.

Handbreak [DH98] uses aspects too, but the source of the contracts are no annotations but separate specification files. These files are compiled to binary code with a special compiler. This code is used by a library that intercepts the class loading of the Java virtual machine and modifies the loaded files on the fly. Preconditions, postconditions and invariants are supported. This approach has the same advantages and disadvantages as the comment and annotation approaches, but it is simpler to remove contracts completely from the project.

Altogether, there are various approaches to introduce contracts to Java. The most important difference with respect to co-evolution of code, contracts and tests is the support for IDE operations such as refactorings. This support can be preserved by using homogeneous artifacts. This means that the contracts have to be written in Java in order to remove the so-called *symbolic barrier* [CCH08]. After the compilation, the code still has to be modified to activate the contract checking. Nevertheless, no approach includes a concept for semantic changes.

4. An Approach for the Co-Evolution of Code, Contracts and Tests

Keeping code, contracts and tests consistent is the main objective of this thesis. In order to do this, the meaning of consistency with respect to these artifacts has to be known. Often this is expressed in terms of relations. Parts without a relation can never be inconsistent and related parts can be inconsistent if they violate the constraints of the relation.

This chapter defines the relations between code, contracts and test artifacts in terms of reactions on changes, which ensure the constraints of the relation. These reactions are described pairwise between two artifacts. It might be necessary to apply a reaction multiple times or alternating with other rules. Atomic changes as well as refactorings are considered. To reduce the complexity some limitations on the context, scope and artifact representation are formulated in the next section. Afterwards, the restrictions on the test artifact are defined in more detail. The relations between all artifacts are described in the third section with focus on JML contracts, Java code and the introduced test structure. Thereafter, we generalize the findings in a separate section.

4.1. Assumptions and Limitations

There are many possibilities to express the solution for a certain task in Java and JML. In our context the tests are based on JUnit, so they are written in Java too. While the amount of possibilities is an advantage for its users, it makes it hard to define relations between the artifacts because they have to be either very generic or very detailed to cover any corner case. Neither is optimal for implementation, so we have to find a trade-off by restricting the possible contexts for such relations in this thesis. We formulate assumptions and limitations in order to be able to define them concise. Our goal is to create a useful solution for average developers by covering the most common use cases while excluding special cases.

Additionally we try to increase the outcome of this thesis by excluding relations that are already well known. This is the case for refactorings performed via IDEs. Because the prototypical implementation is an Eclipse plug-in, we exclude all relations that are already covered by refactorings and source code operations of Eclipse. Because they can process Java artifacts, we do not cover effects of refactorings between code and tests or within these artifacts. Instead, we focus on the relations between code and contracts. As we

will see later, the amount of relations between tests and contracts is quite small with our approach.

Choosing the correct tuple of change and reaction is considered hard if the change does not affect the syntax only but the semantics too. In that case, the selection of the reaction can be ambiguous. We need to gather and understand information about the behavior and the purpose of the involved artifacts to make a decision. Often there are only few precise hints about this. In order to make use of them, we have to be sure of the correctness of those hints. For instance, the throws declaration in Java tells us which checked exceptions can be thrown by this method. Any checked exception that is not compatible to an exception mentioned in the throws clause cannot be thrown. This tells us that the behavior includes throwing a checked exception. Unfortunately, there are situations in which a declared exception might never be thrown as can be seen in Listing 4.1. The declared exception can never be thrown because of the conditional statement of the loop. This situation can also occur because of restrictive specifications (e.g. iterations $\leq$ 0). It cannot always be detected by static code analysis or checking techniques. Therefore, we restrict our approach to contexts where such hints are always correct. For the exception example, this is true if there is no dead code – no matter whether this is caused by statements in code or specifications.

```
1 public void loop(int iterations) throws LoopException {
2   for (int i = 0; i < iterations; ++i)
3     if (i == iterations)
4       throw new LoopException();
5 }
```

Listing 4.1: Example of a declared exception that can never be thrown.

For the contracts written in JML, we make assumptions and restrictions too: We only consider the language features from level 0 and 1 mentioned in Subsection 2.4.1 because almost all JML tools support them. In contrast to the example given in the foundations chapter, we assume that specifications are noted in separate specification files because handling artifacts in separate files is easier from a technical point of view. The concept remains the same because one can perform a preprocessing that separates specifications from code or combines them. This does not hold true for specification files without corresponding source file. This can occur when specifying a library. Therefore, we do not cover such specification files. Additionally, we forbid changing Java elements (e.g. renaming a Java method) in the specifications and tests, because the code artifact is the origin of these elements. Thereby, we cut down on the effort for supporting rarely used changes. Further, we assume that the shortcuts provided by JML are favored over using basic statements. For example, we assume that the not_null annotation is used instead of writing a full precondition by using the requires keyword. Thereby, we can focus on the logical relation and are independent of the concrete representation. A classification mechanism can be added, however, which translates the basic statements to shortcuts. Thus, the existing mappings can be reused.

Tests and contracts share a redundant part of information. While contracts specify the correct result in a universal way, tests specify correct results by example. As the former is more powerful and redundancy should be avoided even if it is kept consistent, we propose tests, which use the contracts as test oracles as described in Subsection 2.5.2. Thereby, the redundancy and the maintenance effort for the developer and our approach is reduced. Our approach assumes that the test structure is completely generated and the user provides the test data. Therefore, we can apply changes to the test structure by regenerating it. We decided to build our approach on manual maintained test data because we want it to be as concise as possible. This keeps the runtime for all tests short while having a high

coverage. Nevertheless, any approach from Section 3.2 can be used to generate the test data instead. Changes that affect the test data are applied by transformations, which are described later. In the next section, we introduce our proposed test case structure and a migration path from legacy to generated tests.

4.2. Assumed Test Case Structure

In the context of this thesis, we only consider generated tests that use contracts as test oracles. The concept of such test cases heavily relies on the idea of [CA10] that proposes to use specifications as oracles and generates complete test suites. Test data generation is the problematic part of this approach, for which many ideas exist (cf. Section 3.2). We want to keep the set of test data as concise as possible, so we let the user maintain it manually. Anyway, the user could use any of the referenced test data generation approaches instead. In that case, no complex synchronization rules have to be applied to keep the tests in sync because we can simply regenerate them to achieve this. Although we did not implement the handling for tests in this prototype, we introduce our concept for the test cases and describe how it can be implemented. Additionally, we describe a migration path from existing tests to our proposed structure.

The structure of the test cases is shown in Figure 4.1. Because we want to test each method with a set of test data, we make use of the parameterized test runner of JUnit mentioned in Subsection 2.5.1. We define a base class for the test and its corresponding test data object. The latter is passed to the test during its initialization. These two base classes are not generated but part of the synchronization implementation. For every method a concrete test case and the corresponding test data container is generated. They follow a naming convention, so they can be identified easily. Test cases contain a static method, which is required by the parameterized test runner of JUnit and returns the set of test data. This call is delegated to a test data provider, which is the same for all tests of the same class. The class itself and the methods are generated. The user has to construct the test data manually and therefore has to fill the generated method stubs. The test data includes all parameters for the method under test as well as the test subject or the object where the method is called, respectively. The subject cannot be constructed in a generic way because various initial states might be required for the test.

Usually, existing tests do not comply with the introduced test structure. A migration path for converting legacy tests to the new test structure is necessary. Because we generate all classes and methods that are necessary for the test, we only have to determine the implementation of the test data provider. Therefore, we focus on the test data provider and do not cover the generated elements. A simple process exemplified in Listing 4.2 can be used for migration assuming that a) the structure for the new unit tests is already generated and only the implementation of the test data provider is missing, b) the existing unit tests follow the JUnit conventions and c) the existing unit tests do not use any form of injection. First, the signature of the test methods is changed to return a test data object. Everything after the call of the method under test is removed because those parts contain the superfluous test oracles. The call to the method under test is replaced by the construction of the corresponding test data object. The first parameter is the object under test. The following parameters are the parameters of the method call. The newly created test data object is returned. In the test data provider, we collect these data objects and add them to the test data set. In order to receive correct test data objects from the old test case, we have to initialize it correctly. This is done by calling the initializer (`@BeforeClass`) and setup (`@Before`) methods before the old test method. Obviously, the annotations of JUnit should be removed from the old test case because we do not use it as test anymore but only as delegation target for the test data provider.

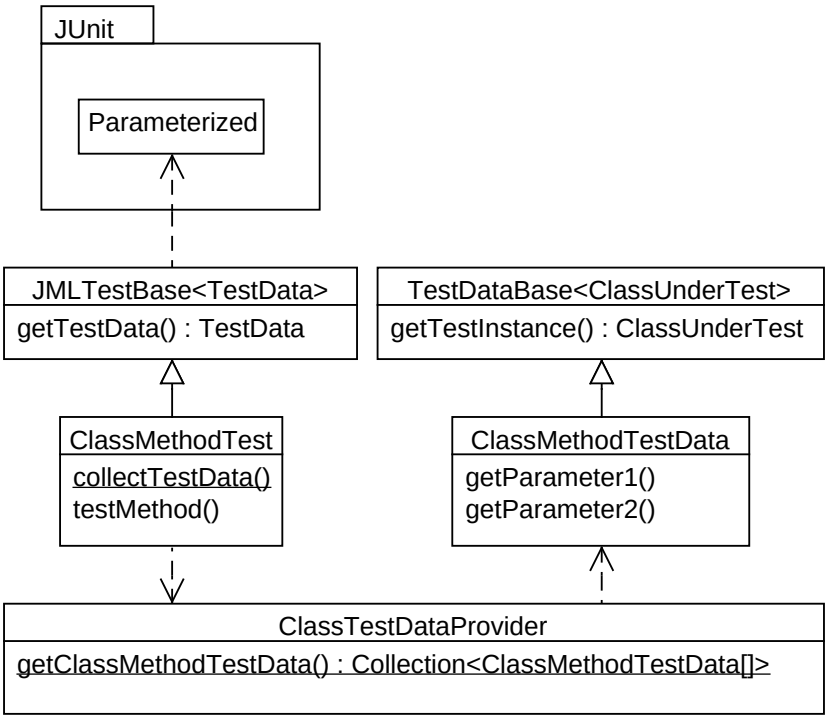


Figure 4.1.: Proposed static structure of the generated test cases and the hand written test data provider.

```
1  public class BankAccountTest {
2    private BankAccount subject;
3    //@Before
4    public void setUp(){
5      subject = new BankAccount();
6    }
7    //@Test
8    public BankAccountTransferTestData testTransfer() {
9      BankAccount receiver = new BankAccount();
10     subject.setBalance(100);
11     return new BankAccountTransferTestData(subject, receiver, 100);
12     //subject.transfer(receiver, 100);
13     //assertEquals(100, receiver.getBalance());
14     //assertEquals(0, subject.getBalance());
15   }
16 }
17 public class BankAccountTransferTestDataProvider {
18   static BankAccountTransferTestData[][] getBankAccountTransferTestData() {
19     return new BankAccountTransferTestData[][] {
20         {createTest().testTransfer()}
21     };
22   }
23   private static BankAccountTest createTest() throws Exception {
24     BankAccountTest test = new BankAccountTest();
25     test.setUp();
26     return test;
27   }
28 }
```

Listing 4.2: Example for migration from old test case to new required structure.

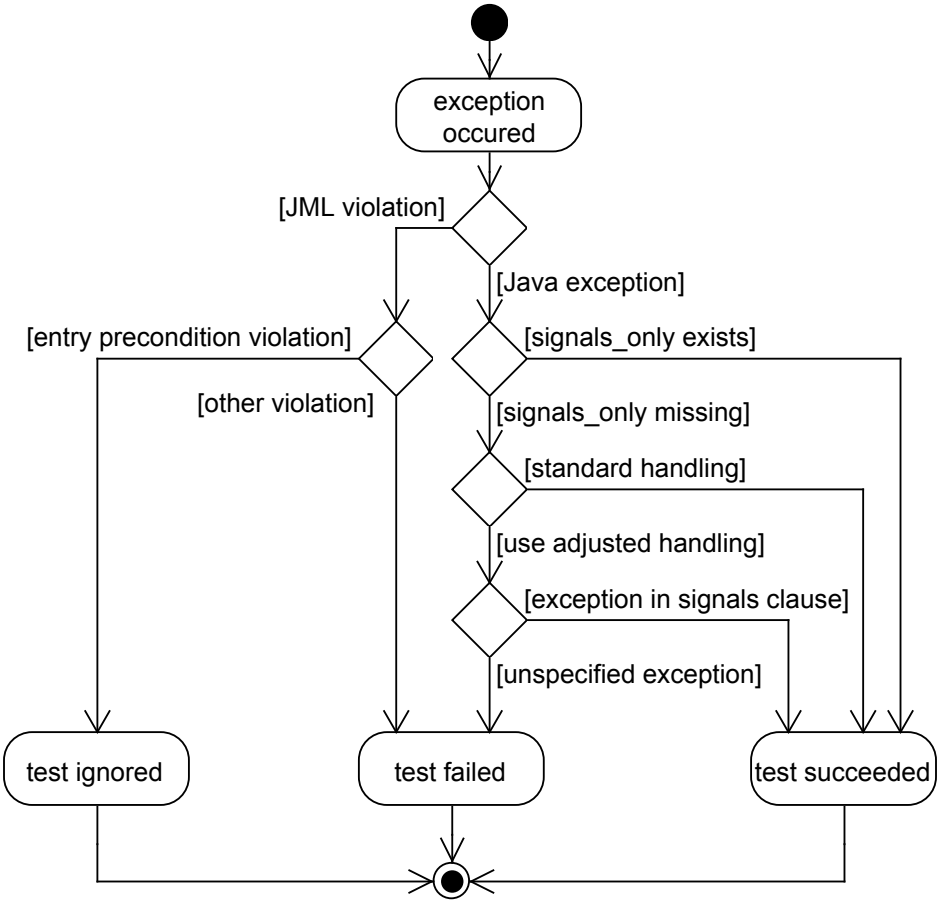


Figure 4.2.: Modified classification of exceptions for generated unit tests.

It is crucial that the test framework supports ignored tests when using contracts as test oracles. A test shall be ignored if the test data is invalid and a precondition of the method under test is violated, therefore. OpenJML differs between an entry precondition violation and a regular precondition violation. This is important because the former implies an error in the test data while the latter implies an error in the specification or the implementation of the method under test. JUnit supports ignoring tests statically by annotating the method only, so the framework has to be adjusted to ignore tests on entry precondition violations. Additionally, it has to mark tests as succeeded which throw specification conform exceptions. A custom test runner for our generated tests can inject the wanted behavior. So other tests are not affected by any modification on the exception classification. Figure 4.2 illustrates the new classification.

First, the source of the exception has to be checked. A JML violation can only lead to an ignored test in case of an entry precondition violation or to a failed test in any other case. Regular Java exceptions might be specification conform. If a signals_only clause exists, the test does not fail after an exception because any non-specified exception would have raised a JML violation. If it is missing, there are two options. One could use the standard handling and accept the exception because no JML specification has forbidden the exception – otherwise a violation would have occurred. We assume that the RAC can detect such violations and reports them as JML violation to the unit testing framework. As denoted in [Cok14] this might not hold true for default cases, which are used if specifications are omitted. Checking if the exception has been specified in the signals clause is a more intuitive but nonstandard approach. An unspecified exception is treated as error in that case.

Element	$C \to S$	$S \to C$	$S \to S$	$S \to T$	$T \to S$	$C \to T$
Identifier	■◒	■◒	■◒	□○	□○	■◒
Visibility	■●	□○	■◒	□○	□○	■●
Types / Return Types	◪◒	□○	◪◒	□○	□○	◪◒
Pure	■◒	■◒	■◒	□○	□○	□○
Helper	□○	□○	■◒	□○	□○	□○
Nullable / Non_Null	◪◒	◪◒	◪◒	■●	□○	□○
Nullable Defaults	◪◒	◪◒	◪◒	■●	□○	□○
Generic Behavior Specs	◪◒	◪◒	◪◒	■●	□○	□○
Exception Specs	◪◒	■◒	■◒	□○	□○	□○
Assignable	◪●	■◒	◪●	□○	□○	□○
Ghost/Model Field	◪◒	■◒	■◒	□○	□○	□○
Model Method	◪◒	□○	◪◒	□○	□○	□○
Model Import	■●	□○	■◒	□○	□○	□○

Table 4.1.: Overview of mappings between code C , contracts S and tests T covered by our concept. The first symbol defines whether the mapping is possible (□ not relevant, ◪ partial mapping, ■ full mapping). The second symbol defines the degree of automation (○ not automated, ◒ semi-automated, ● fully automated).

4.3. Mapping Code, Contracts and Tests

Mappings are unidirectional relations between elements on the meta-level. They consist of the relation itself and constraints associated with that relation. Each mapping is characterized by the related elements, the artifacts that contain these elements and the direction of the relation. Directions considering contracts are most important for us, because tests are assumed to be completely generated – except for the test data – and a major part of the mappings inside the code artifact is covered by refactorings of IDEs already. We investigate mappings for JML contracts, Java code and the previously introduced tests. In Section 4.5 the results of this section are generalized for object-oriented languages and arbitrary specification languages.

We focus on the mappings of the level 0 and 1 language features of JML, which have already been introduced in Subsection 2.4.1. An overview of the relevant language elements and their mappings is shown in Table 4.1. C stands for code, S for specification and T for tests. $\alpha \to \beta$ denotes the propagation of a change from α to β. We consider $S \to S$ because at the moment there is no refactoring engine for JML available to our knowledge. Therefore, we also have to deal with the effects inside the specification artifact.

Because we process generated tests only, we regenerate them after every relevant change. We only care about the test data providing class (TDPC), which is written by the developer and called by the generated code. For the sake of brevity, we do not mention the regeneration of tests for every element.

The following parts describe the mappings between the three artifacts grouped by the affected JML language feature. To provide a better overview we grouped the JML language features into four groups: a) basic syntax elements, b) modifiers, c) method or type specifications and d) specification only elements. While the latter groups are intuitive and distinct, the former contains cross cutting elements used almost everywhere. For every group we start with a description. For every element inside such a group, we give an informal recap of the JML element's meaning and the general relation constraints. Afterwards, the possible changes and the corresponding reactions are described in terms of a manual.

4.3.1. Basic Syntax Elements

Basic Syntax Elements is a term used in this thesis to describe cross cutting syntax elements. They are cross cutting because they are not bound to a small number of syntactic legal locations. In contrast, this would be the case for preconditions, which are only allowed to occur immediately before the method to be specified. Because it is not useful to describe cross cutting elements everywhere they occur, we define their mapping in a separate group. The defined mappings hold true wherever the elements are used. For example, an identifier inside a precondition is treated the same way as an identifier inside a model method.

Identifier

In this context, an identifier is an element that can be referenced by name. Examples of identifiers are classifiers, fields, methods, parameters or variables. They occur in every artifact and there might be cross-references. Because referencing is done by name, they have to be consistent in and between all artifacts.

$\boldsymbol{C \rightarrow S}$ Relevant change operations are create, rename and delete. Renaming has to be propagated by finding the corresponding JML element and replacing the identifier in every reference revealed by the static code analysis. After adding an identifier, an action is only required if the created element has to be represented in the specification file as well. For instance, this is not true for local variables but for fields. Creating and renaming can lead to name clashes in Java or JML. Name clashes in Java are detected and blocked as long as refactorings are used. In JML, no refactoring support exists, so we have to detect and block name clashes – either manually or by using a JML compiler. The removal of elements has to be propagated too, but this might lead to syntax errors in Java as well as JML. The change has to be blocked if there is at least one reference to this element. Changes of method parameters belong to this category too. When changing the order of the parameters, the order of the arguments for all method calls for the corresponding method in JML has to be changed. Again, static code analysis is required for this. Deleting parameters can be processed in a similar way. Adding parameters leads to syntax errors at all callers, so the user has to be informed about compilation errors.

$\boldsymbol{S \rightarrow C}$ There are no identifiers that can be created, renamed or deleted in Java because of a change in the specification because this is not allowed by our restrictions. Nevertheless, we have to make sure that no name clashes with existing Java elements are introduced by adding or renaming specification only methods or fields. The change has to be blocked in that case.

$\boldsymbol{S \rightarrow S}$ The only relevant elements for this direction are model methods and fields. Creating such an element does not have to be propagated, but name clashes with existing JML elements have to be avoided. For rename operations, the change has to be propagated to all callers in addition to the name checking. Deleting such an element can lead to syntax errors in JML so the change has to be blocked if there are references to it. The steps for changing the method signature of a model method are the same as for the $C \rightarrow S$ direction.

$\boldsymbol{C \rightarrow T}$ Test classes can be regenerated after the change of an identifier, so we focus on the test data providing class (TDPC). After every addition or removal of a class or method, the corresponding test classes and TDPCs have to be created or deleted. Changes on the method under test lead to changes of the corresponding TDPC: After a rename or move operation, the name or location of the TDPC has to be adjusted accordingly. After the parameter order changed, the parameter order in the constructor of the test data container has to be swapped too. This can be done by using a refactoring engine. After the removal

of a parameter, the corresponding parameter has to be removed from the constructor of the test data container and all calls of it. After adding a parameter, the user has to be asked for new test data for this parameter.

Altogether, changes of identifiers are relevant for the syntax only. Therefore, common static code analysis and refactorings can be used to perform changes in a fully automated way. Adding a parameter is the only change, which affects the semantics, so it can only be processed in a semi-automated way.

Visibility

Elements can be decorated with visibility modifiers in Java and JML. There are public, protected, private and additional specification related visibility modifiers that can override the regular ones for specification purposes. The visibility of elements determines in which places they can be used. Increasing the visibility means making them accessible from more places. For Java elements, the visibility modifiers have to be the same in Java and JML. Additionally, callers must not violate the restrictions implied by the visibility (e.g. accessing a private method of a superclass from a subclass).

$\boldsymbol{C \rightarrow S}$ Every change of the visibility has to be applied to the JML specification file by replacing the old with the new modifier. After increasing the visibility, no action has to be performed because it can still be used in all places. If the visibility is reduced and the element is used in a specification with a lower visibility, then there is a syntax error. We have to enforce the old visibility by adding spec_public or spec_protected to the changed element. The user should be informed about the change because this is only a temporary fix. The specification visibility should match the Java visibility because a caller should always be able to check the specification by itself too. The $S \rightarrow S$ handling has to be executed after the visibility in the specification has been changed.

$\boldsymbol{S \rightarrow S}$ After the visibility of a specified element has been decreased and it is used by any element with a lower visibility, the change has to be blocked and reverted. Otherwise, there would be a syntax error because the element is not usable anymore in that place. The called elements do not have to be checked because they still must have a matching (higher) visibility. After the visibility of a specified element has been increased, the visibility of all used elements has to be increased to the same or a higher visibility in order to make them callable. This can be done by changing the modifier of specification only elements directly. For Java elements, we have to add spec_public or spec_protected instead, because we restricted the source of changes for Java elements to the Java artifact. Afterwards, the $S \rightarrow S$ rule has to be applied to all changed elements too. If the visibility of any element cannot be changed, the change has to be blocked and all modifications have to be reverted.

$\boldsymbol{C \rightarrow T}$ If the visibility of a method is changed to anything lower than public, the corresponding method has to be considered as removed for the $C \rightarrow T$ direction of identifiers because it is not testable anymore. If the visibility is set to public, the method has to be considered as added. The test data is lost if it is not stored anywhere.

Overall, visibility changes do not involve semantics. Adjusting the visibility of Java elements for specification purposes is easy by overriding existing visibilities. Adjusting the visibility of JML elements involves a recursive change of related elements, which can experience unsolvable problems. Those issues have to be fixed manually before performing the change.

Types / Return Types

Java and JML elements are typed. During compilation a type check is performed, which ensures that two types support the wanted operation. Such an operation could be an

assignment or a method call. The type of Java elements has to be the same in code and specifications. Additionally, the type check has to succeed in all artifacts, which require correctly typed expressions.

C → S Type changes of structural Java elements have to be applied to JML. Such elements are fields, methods and parameters. The types of other elements are not relevant because they do not occur in JML. This holds true for local variables or statements in the method body. A type change of an element does not influence the element itself only but all users of it. Therefore, a simple replace operation on the type is not possible in general. For example, a type change of a constant, which is used as an initializer of a field, has an influence on the type of the field too or vice versa. There is one exception, however: A replace operations is sufficient if the new type is a subtype of the old type and as long as the elements adhere to the Liskov substitution principle. In general, we have to wait until the code is fixed after a type change because it usually breaks after changing a type to a non-subtype. We have to perform a type check by using the compiler afterwards. If the changed element is public or spec_public, we have to check the specification of the whole project. For a protected element, the module and all modules of the subclasses have to be checked. In the other case, we only have to check the specification of the module. If errors have been found in the specifications, we have to infer new contracts for these elements and merge them with the old ones. [PFP+13] mentioned a combining approach as future work but so far, no approach exists. In fact, the merging step has to fix syntax errors caused by incompatible types and consider the new semantics caused by the changed types. This is considered hard, so it might be necessary to ask the user for a solution. We have to repeat this step in the $S \rightarrow S$ direction until there are no errors anymore.

S → S In general after a type change, we have to type check the whole specification in case of a change to a public or spec_public element or single modules in other cases. We have to infer new correctly typed contracts for the erroneous elements and merge them as described for the $C \rightarrow S$ direction. We have to repeat this until there are no more errors. We cannot handle changes of the return type of a model method because we would have to infer a new implementation with the same semantics, which is not trivial. It is not clear, however, whether the semantics shall even remain the same. Type changes that affect set statements (used to set the value of ghost fields) are not supported for the same reason.

C → T After changing a parameter type, we regenerate the corresponding tests and test data containers. If a type check reveals errors, the user has to provide new data for the test. Fixing the existing data is not possible because we do not know the new semantics implied by the changed type or how to keep the old semantics. Other type changes like changing the return type are not relevant for tests because specifications are used as test oracles.

A type change influences the syntax and semantics, so it is hard to process. The only fully automated change is changing a type to a subtype. This change conserves the semantics as long as the Liskov Substitution Principle has not been violated before the change. Only the user can solve any other change because contract inferring and merging techniques cannot accomplish such a complex task at the moment.

4.3.2. Method or Type Specifications

This group contains elements used to specify methods or types. This includes modifiers and regular specification statements. Modifiers specify syntactical elements and consist of a keyword. In this context, such syntactical elements are fields, methods and parameters. Regular specification statements are similar but usually consist of expressions connected by keywords. Type specifications are invariants, constraints and axioms. The remaining statements are method specifications.

The mappings for method specifications and modifiers are problematic because they have to be consistent with the method implementation. In general, verifying a Java program against a given specification is undecidable [DD07, p. 493]. Hence, we are unable to check whether the specification has to be adjusted or such an adjustment has been carried out correctly for any change of the implementation or the specification. For some situations, we can formulate assumptions that allow us to process some changes, but this is only desirable if the assumptions do not exclude common use cases. We tried to adhere to this guideline while formulating the mappings for the following elements in this group.

Pure

Pure is a JML modifier for methods that marks them as side-effect free. Such a method is called a query method. Because preconditions and postconditions have to be evaluable without affecting the object state, only query methods can be used inside specifications.

C* → *S Loosing or gaining a query property is possible by changing the method body. Every statement that can change the state of any object has to be considered. For Java, only assignments to fields or calls to non-pure methods have to be considered. Those two statements already cover changes on parameters. If one of those statements is added, the pure modifier has to be removed. If no such statement exists in the whole body, the pure modifier can (optionally) be added. Afterwards, the $S \rightarrow S$ handling has to be considered too.

S* → *C After removing the pure modifier, no action is required in the code. After adding the pure modifier, we have to make sure that the method body does not contain assignments to fields or calls to non-pure methods. If there are such statements, we have to block the change. The $S \rightarrow S$ handling has to be considered too, which can block the change as well.

S* → *S After adding the pure modifier to a method represented in Java, no action is required in the specification. Nevertheless, the $S \rightarrow C$ handling has to be considered. After adding the modifier to a model method, the $S \rightarrow C$ handling has to be applied to that method. After removing the pure modifier, we have to make sure that the method is not used in any specification. If it is used, we have to block the change because we cannot replace the method call in the specification with a semantically equivalent statement. If the change is not blocked, we have to perform the $C \rightarrow S$ handling for all methods, which call the changed method, because their query status could be lost.

In all, the query property of a method can be determined by static code analysis. Therefore, changes that lead to the addition or removal of the pure modifier can be processed in a fully automated way. The same holds true for adjusting all transitively calling methods, which can have lost their query status because of a previous change. If the method is used inside a specification, the method call had to be replaced with a semantic equivalent method call to a query method. Such a change is blocked because this is not decidable.

Helper

Helper is a JML modifier for methods that excludes them from the invariant and constraint checking. This can be necessary to prevent infinite loops during the invariant evaluation. Invariants are checked in every visible state, which are entering and leaving a method, for instance. Depending on the implementation, the method call during the evaluation of the invariant leads to another evaluation of the invariant and so on. Adding pure to any method used in an invariant is the safest way to circumvent possible problems during the evaluation. Indeed, this should only be done for query methods, which do not require invariant to hold true for a correct result.

$\boldsymbol{S \rightarrow S}$ Adding the helper modifier requires no further action because there are no special requirements for helper methods. Before removing the modifier, we have to make sure that the method is not used in any invariant or constraint. Otherwise, the change has to be blocked in order to avoid an infinite loop. Additionally, changing the expression of an invariant or constraint might make the helper modifier unnecessary. This is the case if the method is not used within any other invariant or constraint. We can (optionally) remove the modifier.

Altogether, addition and removal of the helper modifier can be processed by analyzing the syntax with static code analysis techniques. $S \rightarrow S$ is the only relevant direction.

Nullable / Non_Null

Nullable is a modifier for parameters, fields or methods that specifies that null can be assigned or returned for the latter. Non_null means the opposite and is the default if no such modifier exists. Indeed, the default handling can be influenced by compiler settings and default specifications. In general, the following must hold true: A method must not return a nullable value if it is specified non_null. Additionally, a nullable value must not be assigned to a non_null element.

$\boldsymbol{C \rightarrow S}$ Any change of the method body can make a parameter, return type or field null. This can be compatible with the specifications or not. Compatibility can be checked by using static checking or verification techniques on the changed method. After finding possible issues, the user has to be asked for his intention. Performing changes on the specifications automatically is likely to introduce errors because existing implementations and specifications often rely on a valid reference and perform no check for a null value. If the user did not intend to make the element nullable, we block and revert the change. Otherwise, we can mark the element nullable and perform the $S \rightarrow C \ \cup \ S \rightarrow S$ handling, which can block the change as well. Removing a nullable modifier is never necessary because this would restrict the set of possible values even more. Adding a non_null modifier is not necessary because it is the default.

$\boldsymbol{S \rightarrow C \ \cup \ S \rightarrow S}$ Removing or adding the nullable modifier can lead to inconsistencies between code and specifications as well as between specifications and specifications. The handling for $S \rightarrow C$ and $S \rightarrow S$ cannot be separated because the techniques used to check the compatibility between the specifications and the implementations do not differentiate between them. For example, during the static checking of a method its implementation and the specifications of the called methods are used instead of their implementations. So, specifications and implementations are used. Because the handling of the change is the same for both directions, this is no handicap. After adding or removing the nullable modifier, we have make sure that all artifacts are consistent with the specification. Therefore, we check the changed element as well as all users of this element. For example, after removing the modifier from a method parameter, we have to find all callers of the corresponding method and check them. After adding the modifier to such a parameter, we have to check the method itself because the parameter could violate the preconditions of used elements (e.g. assignment to a non_null field) or the postcondition of the method (e.g. returning null value from a non_null method). If the check reveals errors, we enter a check-and-fix-loop: We determine the conflicting statements in the implementation and specification and ask the user for the correct modification. Possible modifications are adding the nullable modifier to elements or adding a check for a null value to the implementation or to the specification. After the modifications, the compatibility check is performed again. If there are still errors, we ask the user for the correct modifications again. Additionally, at any time the user can cancel the operation, which reverts all changes.

$\boldsymbol{S \rightarrow T}$ After a modifier change of a parameter, the test cases have to be run in order to find incompatible test data. If an entry precondition violation occurred, the corresponding test data has to be removed. Changes in fields and methods do not have to be considered because errors caused by them are related to code and should be treated during regular testing.

In summary, adding nullable automatically should be avoided because it weakens the specification, can introduce errors and is likely to be introduced by mistake. Therefore, adjustments on the specification are done manually, but suggestions are given to the user based on checking and verification techniques.

Nullable Defaults

The keywords nullable_by_default and non_null_by_default can be used to change the default modifier regarding null values for parameters, fields and methods (see above). The scope of the keywords is limited to the class where it is located.

Overall, the intention of the user is not clear after changing the default. His intention could be a) reducing the effort for adding a modifier or b) change the meaning of the existing specifications. The former would require him to replace all existing or non-existing modifiers. This can be done automatically while suppressing further handling. The latter is quite critical because this usually breaks the existing code. We can perform a check or verification of the system and suggest changes, which can be used by the user during manual fixing. Asking for the user's intention is necessary to perform a correct handling.

Generic Behavior Specifications

Generic behavior specification is a category introduced in this section. It subsumes statements, which can take arbitrary specifications. This includes invariant, constraint, requires, ensures, initially, behavior, normal_behavior, exceptional_behavior, assert, assume, axiom, maintaining and decreasing. There are shortcut statements that have the same or a subset of the meaning of a specification represented by these statements. For example, the non_null annotation has the same meaning as a matching precondition shown in Listing 4.3. Such shortcuts and their generic representation are excluded from this category.

```
1 //@ requires o != null;
2 public void someMethod(Object /*@ non_null */ o) {
3   // implementation
4 }
```

Listing 4.3: Example for equivalent specification statements in JML.

Because these statements extend Java expressions by JML keywords, the same behavior can be expressed in various ways. So, the constraints for the relation can only be described in a generic way. From a syntactical point of view, the constraints for identifiers, visibility and types must hold true. Additionally, the syntax of JML must not be violated. The semantics of the statements must not be contrary to the behavior of the specified element, but shall describe it instead.

$\boldsymbol{C \rightarrow S}$ Syntactical changes inside the specifications are handled by the identifier, visibility and type rules. After a semantic change, we have to ensure that the specification still matches the implementation. Tools for checking this exist but have limitations: Verification tools work with very detailed specifications only, which are often not available because they require too much effort to write. Static checking does not find all errors in general (see Subsection 2.4.3 for an explanation of verification and static checking). Executing unit tests most probably uncovers even less errors than static checking because tests often

have a low code coverage. If the specification does not match the implementation, we have to infer a new contract and merge it with the existing one. At the moment, no such approach exists but is planned by [PFP+13] for example. In general, inferred contracts are less restrictive than hand written ones. So replacing the existing contracts completely is not desirable. Suggestions for problematic parts are still possible by using the results of the specification check.

$\boldsymbol{S \rightarrow C}$ After a change, we have to ensure that the implementation matches the new specification. We can use the same approaches like the ones in the $C \rightarrow S$ handling. If the implementation does not match, we have to apply code-fixing techniques. Unfortunately, approaches such as [PFN+14] take too much time for fixing the implementation and are limited with respect to the complexity of the code. Instead, suggestions based on checking approaches can be made for manual fixing.

$\boldsymbol{S \rightarrow S}$ Specifications must not be contradictory after a change. We have to check this via the approaches already mentioned in $C \rightarrow S$. If the specifications are inconsistent, we cannot determine the correct one. We can infer the contracts and eliminate the wrong one by using a merging approach – assuming the code is correct. Unfortunately, at the moment no such approach exists (see $C \rightarrow S$).

$\boldsymbol{S \rightarrow T}$ After a change of precondition-like statements (invariant, constraint, requires, initially), the tests are executed. If an entry precondition violation occurs, we have to remove the corresponding test data. After a change of any other specification statements, no actions are required because they are tested automatically by the RAC.

In summary, generic behavior specifications describe the semantics. Propagating changes from the code to these specifications is hard because the new semantics of the code have to be determined and merged with the old ones. Ideas for doing this exist but are not production ready. At the moment, $S \rightarrow T$ is the only direction, which is fully supported.

Exception Specifications

Exception specifications are statements that specify throwable exceptions as well as the preconditions and postconditions for throwing it. This includes the JML keywords exceptional_behavior, signals and signals_only. signals_only specifies all exceptions, which can be thrown and thereby which exceptions must not be thrown too. signals specifies postconditions after throwing an exception. exceptional_behavior is a so-called heavyweight specification case, which uses those clauses. Multiple heavyweight specification cases can be added to the method specification. Each one has a precondition to decide which specification case has to be applied. The exceptional_behavior keyword can only be handled in some restricted cases. In general, the handling for generic behavior specifications applies.

$\boldsymbol{C \rightarrow S}$ The handling can be separated in processing changes on checked and unchecked exceptions. Both changes arise by adding or removing a (transitive) throw statement, but the former takes place in the throws clause of a method too.

After a checked exception has been removed from code and there are no other occurrences, it can be removed from the signals and signals_only clauses. If an exceptional_behavior specification now contains an empty signals clause, we can remove it. After a checked exception has been added to code, it has to be added to the signals and to the signals_only clause if it exists. This only applies for lightweight specification cases. We cannot process the changes in case of heavyweight specification cases because we do not know for which precondition the exception might be thrown. Therefore, we do not know in which specification case we have to put the new exception. For the same reason, adding a new exceptional_behavior is not possible. We have to ask the user for the correct specification case or for the precondition, which should be added to a new exceptional_behavior block. Inferring and merging contracts could be used too as soon as approaches exist.

The same procedure has to be performed after adding an unchecked exception. Removing an unchecked exception cannot be propagated as easily as the removal of a checked exception, because it does not have to be declared but can be thrown anywhere in the code. In general, we can solve this issue by using static code analysis, which searches for throw statements in all transitively called methods. If the removed unchecked exception has not been found, we can remove it from the specification. Unfortunately, this approach is limited to methods, which are available as source code. This does not hold true for libraries. Therefore, we have to ask the user whether the exception shall be removed from the specification or not as soon as we find a library method.

Besides the obvious changes of exceptions, there are more subtle ones, which influence the preconditions and postconditions for exceptions. A simple example is illustrated in Listing 4.4. The precondition for throwing the exception depends on the guard clause. If the guard clause is changed to $z > 0$, then the precondition for the behavior clause has to be adjusted, for instance. Because such checks affect the semantics, we cannot know whether and how the preconditions and postconditions are influenced in general. Inferring and merging contracts is one possible solution, but to our knowledge no tool exists, which can infer contracts with focus on exception handling. Therefore, the user has to maintain those specifications manually at the moment.

```
1  /*@
2  public exceptional_behavior
3    requires z < 0;
4    signals (IllegalArgumentException) true;
5  */
6  public void someMethod(int z) {
7    if (z < 0) {
8      throw new IllegalArgumentException();
9    }
10 }
```

Listing 4.4: Example for relation between implementation and precondition with focus on exceptions.

S* → *C If an exception shall be removed from the specification, neither the throws clause nor the bodies of the transitively called method shall contain this exception anymore. The latter has to be ensured for unchecked exceptions only. This does not guarantee that the exception is not thrown but at this point, we should trust the developer, who tells us to remove this exception. The other option would be to display a confirmation dialog. If the exception is still used based on the performed check, the change has to be blocked. If an exception shall be added to the specification and it is a checked exception, we have to make sure that it occurs in the throws clause. If this is not true, the change has to be blocked. For unchecked exceptions, we can perform check mentioned for the $C \rightarrow S$ direction. As soon as a library method is called, we have to assume that the exception can be thrown. Therefore, we do not block the change.

S* → *S After adding a specification element for an unchecked exception, we can suggest to add it to all calling methods too. Removing such an element cannot be handled by removing it from all callers of the method. For instance, a caller can call other methods, which still throw this exception. We can suggest checking this on request.

Altogether, the basic idea for handling exception changes is ensuring that the implementation and the specification match by using static code analysis and performing the necessary adjustments. This approach is limited to syntactic changes. After semantic changes, we cannot make the necessary adjustments without specification inferring and merging techniques. Adding and removing exceptions is easy but determining the preconditions for heavyweight specification cases or postconditions for the signals clauses is not.

Assignable

Assignable specifies frame rules that define the changeable data for a method. Thereby, it also defines what must not be changed. In context of JML, the changeable data is expressed as locations, which includes fields, expressions referring multiple fields or array ranges, groups of model fields and so on. We focus on fields because they are most heavily used and some of the other elements are above level 0 and 1 JML language features. The following mappings are described for lightweight specification cases only. For heavyweight specification cases, the approach is the same, but we would have to calculate the preconditions for every assignment and filter the fields.

C → S Assignments to fields and method calls affect the assignable clause. Added or removed assignments to fields can be added to or removed from the clause directly. If a call to a non-query method is added, all elements from the called method's assignable clause have to be added to the caller's clause. Before adding the elements, the called object has to be prepended because the elements have to be callable from the method's point of view. For instance, a field name of the parameter student would result in adding the expression student.name to the clause. If the called method does not have any assignable clause, the default everything is assumed. Hence, everything on the called object can be changed, which is denoted by calledObject.*. Additionally, all parameters passed to called method can be changed, so we have to add parameter.* to the clause. If a call to a non-query method is removed, the elements of its assignable clause cannot simply be removed because a field can be assigned by any other called method too. Instead, all elements from the assignable clause of all called non-query methods have to be collected. The whole assignable clause is replaced afterwards. In general, it might be necessary to raise the visibility of the field in the clause to match the specifications visibility. The $S \rightarrow S$ direction has to be considered as well after the changes have been applied.

S → C If an element from the assignable clause is about to be removed and there are still assignments to it or method calls to a method, which assigns to it, the change has to be blocked. If an element is added to the assignable clause and there are no assignments or method calls, which require this addition, the user is warned. The $S \rightarrow S$ direction has to be considered as well after the changes have been applied.

S → S After adding an element to the assignable clause, it has to be added to all transitively calling methods, which have an assignable clause. After removing an element from the assignable clause, the assignable clause of all calling methods has to be recalculated by the mechanism described for the $C \rightarrow S$ direction. This procedure has to be applied recursively to all transitively calling methods.

Altogether, all (transitively) assigned fields can be determined by looking at the assignable clauses of called methods and the assignments in the method body. After a change, the specified set of assignments has to be compared with the real one and differences have to be eliminated. For heavyweight specification cases, this procedure is more complex because preconditions for all assignments and method calls have to be calculated. Afterwards, the assignments have to be distributes over existing specification cases by checking the compatibility of the preconditions.

4.3.3. Specification-Only Elements

Specification only elements are only definable and usable in specifications. This includes specification only fields and methods as well as model imports, which can be used to import modules only used in specifications.

Ghost/Model Field

Model and ghost fields are specification only fields. Model fields are always represented by a typed expression, which often involves regular Java fields (at least for JML language levels 0 and 1). In contrast, the value of a ghost fields is not calculated automatically but set explicitly with a set statement inside a method body. The usual syntactic constraints have to hold for them. Additionally, the semantics especially of the set statement have to be considered.

$\boldsymbol{C \rightarrow S}$ Changes on elements used in the expression for model fields have to be propagated back to the model field itself. For instance, if the expression is changed from integerField to stringField, then the type of the model field has to be a string now. As described in the type handlings this can only be processed in a fully automated way if the new type is a subtype of the old one. Otherwise, the user has to fix the issue manually. Ghost fields and related set statements can be influenced by code changes of the method body. The identifier, visibility and type rules can handle many changes. Anyway, if the semantics of the method body changed, we cannot automatically adjust the set statements because we do not know the relation between the method's semantics and the ghost field's semantics. The user has to adjust this manually.

$\boldsymbol{S \rightarrow C}$ There are no changes on model or ghost fields, which have to be propagated to code because changing structural Java elements from specifications is restricted. The check for name clashes described in the identifier handling has to be performed anyway.

$\boldsymbol{S \rightarrow S}$ The usual changes like renaming are handled by the identifier, type and visibility rules. As described in $C \rightarrow S$ a type change of the expression defining the model field is often not applicable automatically.

Overall, the rules for identifiers, visibility and types can be used to process changes on ghost or model fields as well as on set statements. As soon as the semantics are affected, we cannot automatically process changes. This holds true for type changes of model fields and semantic changes of method bodies, which affect set statements.

Model Method

A model method is like a Java method, but it can make use of specification only elements such as model fields and can only be called from specifications. Therefore, the syntactic constraints for identifiers, types and visibility have to hold true.

$\boldsymbol{C \rightarrow S}$ The usual rules for identifiers, types, visibility have to be applied.

$\boldsymbol{S \rightarrow S}$ The usual rules for identifiers, types, visibility have to be applied. The $S \rightarrow C$ direction of pure, exception handling, assignable and nullable has to be considered too as well as the $S \rightarrow S$ direction for helper.

In all, syntactic changes on model methods can be applied using the identifier, type and visibility rules. Model methods have to be treated as Java code for some rules like the one for pure.

Model Import

A model import is necessary for importing elements that are only used in the specifications.

$\boldsymbol{C \rightarrow S}$ After removing a regular import, we have to make sure that it is not used in the specification anymore. This can be verified by compiling the module with the JML compiler. If there is an error, a model import has to be added instead of the removed import. After adding a regular import, we have to remove a possible existing model import for the same module.

$\boldsymbol{S \rightarrow S}$ Every change in the specifications can make it necessary to import a new element. Hence, after every change, the specification has to be compiled. If there are any type resolution errors, we have to determine the module name and find possible imports. If it is ambiguous, the user has to select the correct one. Approaches for this already exist in the Eclipse Java Development Tools. We can try to clean up the model imports as well by removing one import and checking for compile errors afterwards. After removing a model import by hand, we have to compile the module and block the change if there are compilation errors. After adding a model import, not action is required.

Altogether, an import is replaced by a model import if it is used in the specification but not in the code. On compilation problems, the missing type is imported or suggestions are made if the type is ambiguous at this location.

4.4. Handling of Code Refactorings

Besides the atomic changes mentioned in Section 4.3 there are code refactorings, which lead to a set of changes. This set has to be executed together. Because refactorings are defined as changes to the internal structure that do not change the observable behavior one could mistakenly believe that they are easy to apply. Whether a changed behavior is observable depends, however, on the location of the observer. Therefore, a refactoring does not just create syntactical changes but semantic changes too. Usually, the refactorings can be processed by using the mappings described in Section 4.3.

The following parts describe how the Java refactorings supported by Eclipse can be mapped to JML. The selection of refactorings is based on [Ecl14a, Java developer user guide - Refactoring]. All of them produce syntactically correct code, so before execution only options are offered, which lead to a correct result. For our analysis, we assume that the code is in a valid state after executing the refactoring. For a sake of brevity, the refactorings without impact on the specifications are omitted.

The **Rename Refactoring** renames an element and all references to it across multiple files. The following Java elements are relevant and can be mapped by using the identifier rules: Methods, method parameters, fields, types, type parameters, enum constants and packages.

The **Move Refactoring** moves an element to another destination and updates all references. For JML the movement has to be reproduced and references have to be updated. The move can be done by copying the element. It is crucial to copy the specifications together with the structural element. Therefore, implementing this by using create and delete is not possible. The update of references is described in the identifier rules with respect to rename operations and can be used. The following Java elements are relevant: Instance methods, static methods, static fields, types and packages.

The **Change Method Signature Refactoring** can adjust all parts of the method signature including name, visibility, return type, parameter names, parameter types and parameter order and updates all references. All these changes are already described by the identifier, type and visibility rules. Additionally, the refactoring can add and remove parameters and exceptions. The latter is already described by the exception rule. The former is described too, but because of the refactoring, we can use some more context knowledge to solve arising problems. Removing a parameter is easy because we just have to remove it in every call of the method. Adding a parameter is easy too because the refactoring requires the user to enter a default value for the new parameter. This default value can be inserted in any call of the method. This is an advantage over the identifier rule, in which this scenario cannot be solved automatically.

The **Extract Method Refactoring** creates a new method form a selected part of the method body. The old method calls the new method. The specification of the old method does not change because the processed statements remain the same. Inlining the called method would lead to the same method as before. The specification for the new method is unknown, however. [CCLB12] describes an approach for inferring it. One of its advantages over general specification inferring techniques is that the specification of the original method influences the new specification. Thereby, the approach achieves better results.

The **Extract Constant Refactoring** creates a static final field for a selected expression and replaces the selection with the constant. It can replace other occurrences of this expression too. If the latter is used, the occurrences have to be replaced in the specifications. Not all equivalent expressions might be replaced because the order of the operands is relevant. For instance, the extracted constant $2 * 21$ would not replace expression $21 * 2$. This is conform to the handling of Eclipse, but can introduce inconsistencies. On the other side, replacing an equivalent or even the same expression can also lead to inconsistencies as shown in Listing 4.5. As can be seen from the implementation, the precondition and postcondition have no relation. Replacing the mentioned expression with a constant would introduce a wrong relation in this case. Hence, we can replace occurrences but suggest checking all replacements manually. Additionally, the visibility rule has to be applied in order to make the constant usable in the specification.

```
1 //@ requires number == 2 * 21 * 2;
2 //@ ensures \result == 2 * 21;
3 int calculate(int number) {
4   otherMethod(number);
5   return 2 * 21;
6 }
```

Listing 4.5: Initial situation in which an illegal relation might be introduced by replacing $2 * 21$ with a constant for all occurrences.

The **Inline Refactoring** takes the content of a method or constant and replaces a reference with the content. Optionally, all references can be replaced as well. If the latter is chosen, we can inline the elements in the specification too. Otherwise, we do not know whether the inlining is correct. Listing 4.6 shows an example for such a situation. Inlining the constant in line 4 does not have any influence on the specification, which makes replacing the constant in the specification and the return statement wrong. After such a type of change, we should ask the user to check the specification statements.

```
1 public static final int INT = 42;
2 //@ ensures \result == INT;
3 public static int extractConstant(int q) {
4   q += INT;
5   return INT;
6 }
```

Listing 4.6: Initial situation in which inlining the first occurrence of the constant in the method body has no influence on the specification

The **Convert Anonymous Class to Nested Refactoring** takes an anonymous class and converts it to a nested one. The influence on the specifications are similar to the extract method refactoring. The specification in the original location of the anonymous class is not changed because the semantics remain the same. In many situations, specifications for the new inner class should be already available because they are often an implementation of an interface or abstract class, which should already be specified. Only specifications for elements that are not included in the base class or interface are required. We can use specification-inferring approaches to achieve this or ask the user to enter them manually.

The **Move Type to New File Refactoring** creates a new file containing the selected type and updates all references. Additionally, a field to access the old outer class is added and the visibilities are adjusted. The specifications of the referencing elements remain the same because the semantics are the same. The specifications for the moved class do not have to be adjusted but only copied to the new location. The only additional specification is the requirement that the reference to the outer class and the corresponding parameter in the constructor of the moved type must not be null if the field is used in any method. Visibility changes are handled by the general rules.

The **Extract Superclass Refactoring** creates a superclass from two or more classes by extracting the common parts. The specifications of the classes do not have to be adjusted. The critical part is inferring a specification for the new superclass. [GFT06] describes a process to find the strongest precondition and the weakest postcondition for every method. Stronger preconditions and weaker postconditions are required because otherwise the subclasses would violate the Liskov substitution principle. For every method or field that is used in the superclass specification and is not part of the superclass, a model field or method is created, which is specified in the subclasses. All occurrences of that elements are replaced with the corresponding model elements in the superclass specification.

The **Extract Interface Refactoring** creates a new interface containing method declarations for the selected methods. It is quite similar to the extract superclass refactoring. The only difference is that specifications have to be copied from the source class instead of generalizing it because there is no second class for comparison. The user has to adjust the specifications afterwards in order to allow implementing the interface without violating the Liskov substitution principle.

The **Use Supertype Where Possible Refactoring** replaces all occurrences of a type with a supertype where syntactically possible. This affects the semantics because preconditions can be stronger and postconditions can be weaker than before. We have to apply the type change rules, which tell us to ensure that specification and implementation match and ask the user to fix it manually otherwise.

The **Push Down Refactoring** takes fields or methods and moves them to all subclasses. [Hul10] already examined the necessary specification handling in his *Push Down Specification* refactoring. Our situation is, however, even simpler because we do not have to consider possibly missing method declarations in subclasses because we already moved them during the code refactoring. We have to push down any specification for the moved code elements. It can be necessary to adjust the visibility to make elements of the base class accessible to the subclass specifications.

The **Pull Up Refactoring** takes fields and moves them to the superclass or takes methods and adds abstract method declarations to the superclass. In [Hul10] the required actions are described for the *Pull Up Specification* refactoring. Again, our situation is simpler because we already created the code elements. The specifications can simply be copied. For unresolved references (e.g. a reference to a non-existing field) model fields and methods have to be added, which are implemented in the subclasses. This implementation has to be useful for the original subclass only. Other subclasses have to be adjusted manually.

The **Extract Class Refactoring** creates a new class C containing the selected fields F, removes those original fields F, adds a new field of type C and replaces all old occurrences of F with calls to getters and setters. Usages of those fields have to be replaced with getters in specifications too. The visibility of the data accessor might have to be adjusted. It can be possible to move some invariants into the newly created class, but this is not necessary.

The **Introduce Parameter Object Refactoring** creates a new class containing fields that represent a set of parameters, replaces the original parameters and updates all callers of the changed method. Because the semantics remain the same, only the syntactical changes have to be applied to specifications. All occurrences of the parameters in the method specification have to be adjusted to use the getters on the new parameter object. All calls to the method in specifications have to be adjusted to construct the parameter object first and pass it as parameter.

The **Introduce Indirection Refactoring** adds a static method that delegates the call to another method of a given object. Optionally, calls to the method can be changed to call the newly created method. We can just copy the specification from the delegation target to the new method after adjusting them to prepend the object, which is passed as parameter. Additionally, we have to add the non_null modifier to the first parameter, which is the object to which the call is delegated. If the references shall be updated, the same has to be done in the specifications.

The **Introduce Factory Refactoring** adds a factory method to the class that delegates to the selected constructor. References to this constructor are replaced with a call to the factory method. We can just copy the specification from the constructor to the factory method after adjusting them to prepend \result in the postconditions. If the initially clause is used, those statements have to be copied to the factory method as well, because it specifies conditions which must hold for every constructor.

The **Introduce Parameter Refactoring** adds a new parameter to a method and adjusts every call to pass a selected expression as value for it. This is a subset of the *Change Method Signature* refactoring and can be treated like that one.

The **Encapsulate Field Refactoring** adds getters and setters for a field and replaces all usages with calls to those. The usages of this field have to be adjusted in the specifications too. Additionally, we can add specifications for the getters (result has to be equal to field) and setters (field has to be equal to given value afterwards).

The **Generalize Declared Type Refactoring** replaces the current type with a supertype if this is possible with respect to the syntax. This has to be handled with the general type change rules.

Altogether, we have shown that refactorings can be mapped with a combination of existing mappings for atomic changes. Because we have more information about the intention of the user during a refactoring, we can find mappings that cover more aspects of the change. For instance, an extract method refactoring would produce a change which only consists of a create operation for a method. Without knowing the context we have to use generic specification inferring techniques to find a new specification. Because we know that the method has been extracted rather than created, we can use a more precise approach for specification inferring tailored to this refactoring.

4.5. Generalization of Mappings for Contracts

The previous two sections covered mappings between Java code, JML contracts and unit tests. We chose JML because it is one of the most mature description language for contracts and it contains many concepts that are not explicitly covered by the simple approaches mentioned in Section 3.3. Nevertheless, we show that the developed concepts can be applied to any other specification approach for object-oriented languages:

The basic syntax elements include identifiers, visibility and types. These elements occur in object-oriented languages and therefore in its specification too. As long as elements are referenced by name, the identifier mappings hold true. The visibility mappings can

be reused as well. Depending on the new specification language, some adjustments are necessary: If the new specification language does ignore visibilities completely, the visibility mappings for contracts become obsolete. If there are no separate visibility modifiers in the specification language, we have to adjust the visibility of the code element anytime we would have used the specification-only modifiers. The type mappings can be applied for every strongly typed language. Obviously, not all parts of the mappings have to be applied depending on the concrete specification technique. For instance, if contracts are represented as regular code inside the specified method, then renaming is handled automatically by the IDE.

The method or type specifications also contain elements and mappings that can be generalized: Many specification techniques require the concept of query methods (pure modifier) even if they do not provide a special keyword for it. Without query methods, evaluating a contract produces side-effects, which can be harmful. Therefore, the mapping for query methods can be applied after a little modification: The implementations of the methods have to be changed to detect calls to non-query methods instead of using the pure modifier. The nullable modifier can be generalized in a similar way: Instead of looking for the modifier, we have to find specification statements that represent this modifier. For instance, the parameter p could be specified by requires p != null, which would be equivalent to non_null. We can remember these findings or always check the implementations after changes. The mapping for the default value regarding null cannot be generalized because it is implementation specific. The same holds true for the helper modifier. If an infinite loop during invariant evaluation cannot arise because of a different implementation, the modifier is not needed and no mapping has to be enforced.

The generic behavior specifications include preconditions, postconditions and invariants, for instance. These elements are an essential part of contract approaches described in Section 3.3. Hence, their mapping can be applied as is. The same holds true for other statements of this group, which have a representation in the used contract approach. Exceptions are not covered in much detail by most basic approaches. Nevertheless, some approaches like Jass [BF01] provide mechanisms to specify exceptions too. The analysis described in the mapping of the exceptions is still valid for those approaches but it has to be extended by a pattern matching and creation mechanism, which can detect and produce exception specifications. Obviously, this requires a uniform representation of such specifications, so restrictions have to be formulated as part of the new mappings. Frame rules are not part of any investigated contract approach. They are, however, mentioned in [MM02], which is a general introduction to design by contract. Therefore, we consider the assignable clause as an important part of contracts. If such a clause exists in the used contract approach, the mapping can be applied without modifications.

Specification-only elements are quite specific for JML. We cannot generalize mappings for these elements because we have no counterpart of other approaches to compare them.

Altogether, most mappings can be generalized with small effort as long as there is a representation of the JML keyword and the specified language is object-oriented.

5. Implementing the Synchronization between Code and Contracts

After describing concepts for keeping artifacts consistent in Chapter 4, we present the implementation of a part of these concepts in this chapter. We use a synchronization framework to preserve consistency by enforcing the described mappings and their constraints. It a) detects changes, b) handles the input and output of models, c) keeps track of mappings and correspondences and d) transforms the models. The Vitruvius framework in combination with the change monitoring approach described in Section 2.3 can perform most of these tasks. Because both are not production ready, they had to be modified in order to make them usable. Additionally, we had to write some glue code, initialize the framework and write the transformations, which ensure the consistency of the artifacts. The combination of these elements is called *synchronization system* in the following sections.

This chapter gives an overview of the implementation of the synchronization between Java code and JML contracts. Although tests are described in the mappings, no handling has been implemented for them because of a limited amount of time. Instead, we focused on code and contracts, which are described in detail. First, the overall architecture of the framework is introduced, which incorporates Vitruvius and the change monitoring techniques. A description of the supported (implemented) changes is given in the next section. Finally, the processing changes is illustrated by a small example for a quite complex change.

5.1. Architecture

The synchronization system is organized in a relaxed multilayered architecture, which is shown in Figure 5.1. This lead to multiple layers, which can only call the layers below them. In the figure, we assume that an imported package is always re-exported. Every layer has its own task: The monitoring layer is responsible for detecting changes in the supported artifacts and forwarding them to the synchronization layer. This layer performs the synchronization by using the foundations layer, which consists of the Vitruvius framework, the model printers and parsers for the languages and their meta-models. The layers are not reusable because they contain very specific artifacts for the supported meta-models. Nevertheless, single artifacts such as the monitored editors, the factory for the Vitruvius framework, the meta-models or the initialization can be reused.

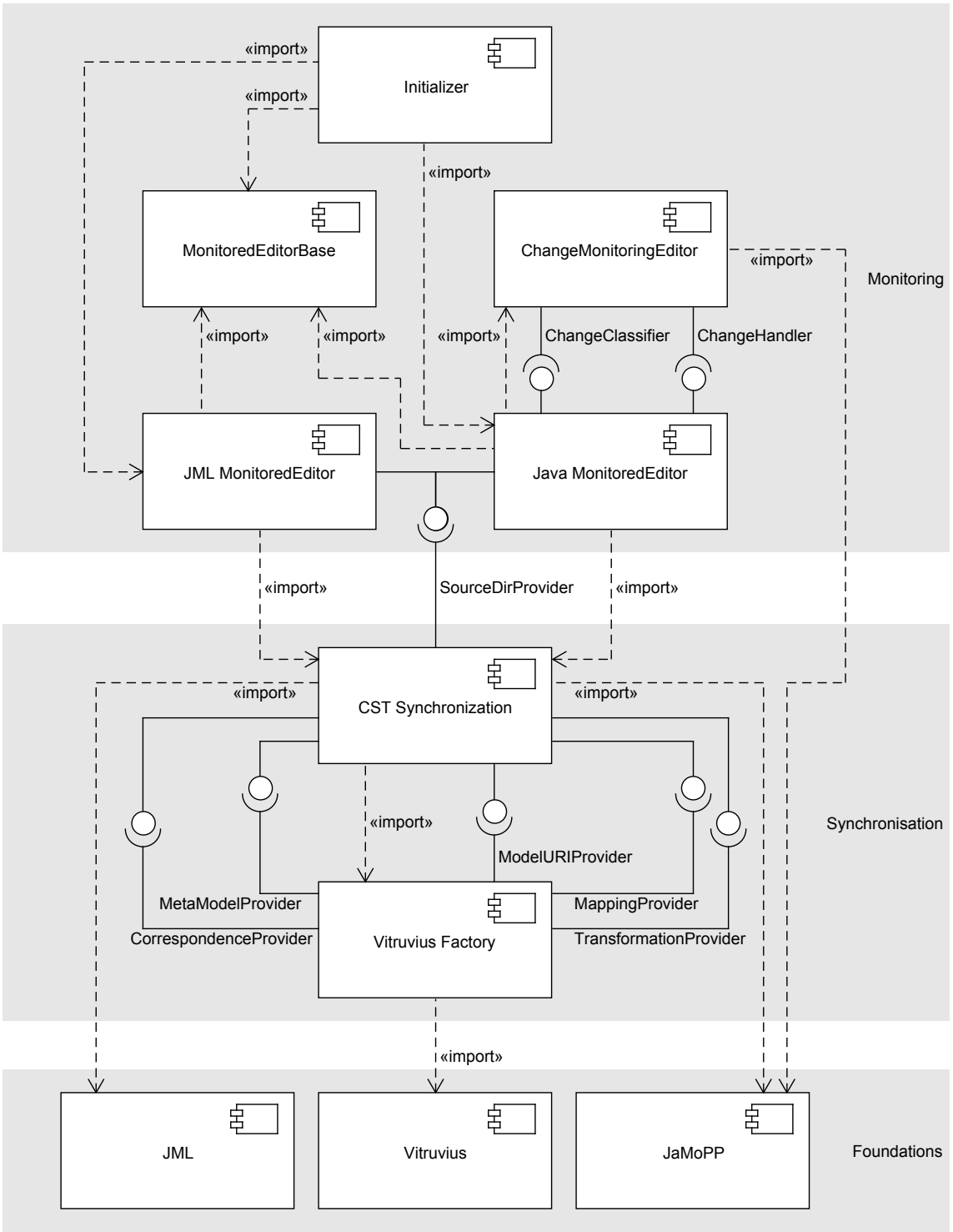


Figure 5.1.: The artifacts of the synchronization system for code, specifications and tests and their relationships.

All artifacts are implemented as Eclipse plug-ins. We will not call them components because they do not fulfill the definition of [SGM02, chap. 4.1.1]: a) they cannot be used for independent deployment or b) third party composition and c) have an observable state. They are assembled by package imports and Eclipse extension points. For the package import to work, the plug-in makes the relevant packages accessible for other plug-ins in its plug-in definition. Other plug-ins declare dependencies to that plug-in and import the package. In Figure 5.1, we assume that the imported packages are re-exported. The extension point mechanism works in a similar way. The plug-in, which shall be extended, defines extension points with obligations like the implementation of a specific interface. The extending plug-in registers extending classes for the given extension point in its plug-in definition. During the runtime the extended plug-in asks for the values from the extension points and makes use of them. So the control flow is inverted with respect to the package import.

The following sections cover the assembling mechanisms used on and between the layers and the layers itself in brevity.

5.1.1. Monitoring Layer

The monitoring layer detects changes in the related artifacts and propagates them to the synchronization layer. It is the starting point of the control flow. The initializer plug-in starts and stops the monitored editors and thereby the whole synchronization system. There is a monitoring editor for each supported language. Each of them has two responsibilities: First, it has to detect changes, classify and forward them to the synchronization layer. Second, a list of all monitored directories has to be passed to the synchronization layer so it can find source files and build its model database, which we will call virtual single underlying model (VSUM) in the following sections (as introduced in Subsection 2.3.1).

While the JML editor has no support for change detection but only for injection of change models in this prototype, the Java editor has. It is based on the existing change monitoring solution for Java code described in Subsection 2.3.2 and therefore imports it. The original implementation is extended by a few more classifiers, change handlers, some little adjustments for existing change handlers and bug fixes. The extensions are necessary because changes that do not affect the AST are not always properly classified in the change monitor for some of our use cases. Renaming parameters is a popular example for this. It is an important change in our context because specifications usually specify parameters, which are referenced by name. Therefore, the monitored editor for Java makes use of the extension mechanisms of the change monitor, which allow defining custom change classifiers and handlers. The change handler takes the previously created change and forwards it to the synchronization layer by using the imported synchronizer.

5.1.2. Synchronization Layer

The synchronization layer consists of two plug-ins. The first plug-in creates and initializes the Vitruvius synchronization framework. In order to do this, it needs a) the meta-models, b) the mapping between the meta-models, c) all models, which shall be synchronized, d) the correspondences between the elements of the models and e) the transformations, which shall be executed after a change. The creation is done by importing the Vitruvius framework and initializing the imported classes.

The initialization information is provided by the second plug-in, which is meta-model specific. It uses the already mentioned Eclipse extension point mechanism to pass the information by registering various data providers. Additionally, it imports the factory

plug-in to access the synchronization object, to which changes received from the monitoring layer are forwarded. The meta-model specific plug-in also imports the JML and JaMoPP meta-models because they are necessary to construct the initialization data. The source directories received from the monitoring layer are used to find models for the VSUM.

5.1.3. Foundations Layer

All upper layers (transitively) use the foundations layer. It provides the meta-models and model utilities for the synchronized models. The Vitruvius framework is located in the foundations too because it does not only contain the synchronization logic but data types and meta-models too. For example, the whole change meta-model is located in the framework. Because changes are produced in the monitoring and consumed in the synchronization layer, it is used in both places.

5.2. Prerequisites for Synchronization

Before the transformation can be carried out, the following preparations have to be executed: The most basic preparation is the conversion of artifacts to appropriate models. Without properly formalized models, the transformation would become extensive and the advantages of using models would be lost. This includes compatibility to generic transformation tools and abstraction from technical details, for instance. The first section deals with the approach to create models from existing artifacts. Afterwards, the technique used for referencing elements across models is described. The third section deals with the application of this technique to track correspondences between model elements.

5.2.1. Constructing Models from Artifacts

Model transformations can be carried out on models. Therefore, all artifacts that shall be synchronized have to be represented as models. According to [Sta73, p. 131-133] a model is an abstraction and reduction of a (real-world) element, which is created pragmatically to ease a specific task. Therefore, any form of code can be considered a model. The serialization and parsing is done via language specific parsers and serializers.

Unfortunately, we cannot imply that artifacts are pragmatic with respect to model transformations just because they are models. In general, it is easier to handle models with a common meta-modeling language instead of different ones, which is the case for code artifacts for different languages. In the Eclipse ecosystem, this common meta-modeling language is Ecore. The Vitruvius framework has a good support for Ecore-based models while it lacks support for regular objects, which would result from using a regular parser. For those objects one has to write the parsing and serialization code by hand and adjust various internal processes to fit the needs. Hence, using Ecore-based models is favored where possible.

For Java, there are already techniques that can parse code into models and serialize it to code again. JaMoPP is one of the most mature techniques and it is already used in the change monitoring approach. For JML there are no existing techniques to create other models than code and ASTs. Building a technique like JaMoPP for JML is favored over reusing the existing parser because a) the modifications in the Vitruvius framework to support models not based on Ecore are not trivial and b) the transformations are more complex when processing models with different meta-modeling languages.

The target of the newly created JML model printer and parser is to transform JML statements into models and vice versa. By this, we can easily reuse Vitruvius and use the models inside transformations. We do not aim for comprehensive editor support, syntax

Grammar Element	JML Keyword	Grammar Element	JML Keyword
Type	axiom	Modifier (method)	spec_public
	invariant		spec_protected
	constraint		helper
	model		pure
	ghost	Modifier (spec	instance
Method	behavior	element)	static
	normal_behavior	Expression	\old
	exceptional_behavior		\fresh
	requires		\result
	ensures		\forall

Table 5.1.: Summary of JML language features supported by the newly created model printer and parser.

checking, validation or soundness. We assume the JML code to be valid and to be stored in a separate specification file, which is officially supported by the JML specification. This eases serialization and parsing and frees us from the obligation to be able to parse every Java statement, which would be necessary for mixed artifacts.

We created the model printer and parser for JML with the Xtext framework. We could have used EMFText as well, but maintaining a meta-model and the concrete syntax separately was considered too much work for a prototype. Additionally extending JaMoPP is not trivial because of the complexity of Java. Xtext uses a single grammar file, which defines the concrete syntax and implicitly the meta-model too. If the excluded objectives (e.g. soundness) should be achieved, it would be better to extend JaMoPP to support JML constructs but this is not the objective of this thesis.

The created JML grammar is based on a non-official grammar for Java 1.5[1]. As it does not support some very basic expressions, we replaced this part of the grammar by Xbase expressions (see Subsection 2.1.2). Xbase is a partial programming language with a Java-like syntax. Especially declaring variables and casting is different. We adjusted the Xbase grammar to support Java variable declarations to create a model printer and parser, which can process most Java syntax. Of course, this also allows illegal syntax but this has not to be considered because of our restrictions and assumptions about valid code. Again, we do not aim for comprehensive tool support for developers. To adjust the Xbase grammar we extended the expressions with the supported JML constructs. Table 5.1 subsumes the supported JML language features, which were introduced by the mentioned modifications.

The JML model printer and parser is seamlessly integrated in Eclipse and EMF. Therefore, models can be read and written without the need for specialized handling. JML files can be used with the Vitruvius framework just like the Java files.

5.2.2. Referencing between Model Elements

Correspondences describe the relationship between elements of a model. In contrast, mappings describe the relationships for meta-models (see Subsection 2.3.1). Storing references between model elements in Ecore-based models is simple because elements in other files can be referenced. Unfortunately, this does not work anymore as soon as models are not updated but completely regenerated. The latter is done in the change monitoring approach. Thus, the received change models do not contain the same object instances as the

[1] http://www.eclipse.org/forums/index.php/t/251746/

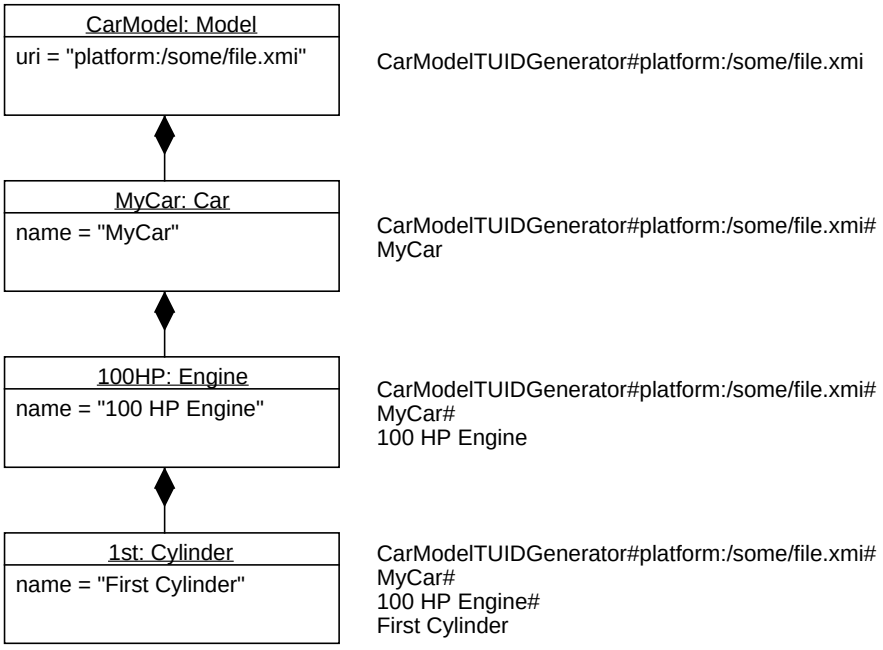


Figure 5.2.: Example of the hierarchical TUID for a simple model. The *MyCar* element is the root object.

instances in the VSUM. As a result of this, changed elements cannot be matched to the corresponding model element in the VSUM by using the element's instance. Vitruvius uses an identifier mechanism for referencing models to circumvent this problem:

An identifier is stored instead of the element itself. This identifier only depends on the contents of the model element instead of its identity. In Vitruvius, this is called a temporarily unique identifier (TUID). Such an identifier is generated by meta-model specific calculators if a reference is about to be stored. When resolving a reference the identifier is used to look up the wanted element in a given model. Updating such a reference is possible by adjusting the identifier. Because it consists of hierarchical segments, only a specific part has to be exchanged. Thereby, the complexity of the operation is reduced. The whole process is fully transparent to the caller of reference handling modules.

The TUID calculators are meta-model specific. If the meta-model already has an identifier attribute, a default implementation provided by Vitruvius can be used. Otherwise, the calculator has to be developed manually. A TUID consists of several hierarchical segments. Each segment identifies an element on the specified level of the containment hierarchy. The example in Figure 5.2 illustrates how the TUID is built. The first two segments contain the name of the generator and the location of the model. The former is necessary to map a TUID to a meta-model and the latter is used to load the model. The following segments define the elements on the containment hierarchy inside the model. In the example, the name attribute is always used to infer the segment. Obviously, this is not correct for all meta-models because we have to build a segment, which uniquely identifies an element on the given hierarchy level. For instance, it is legal to have multiple methods with the same name as long as the ordered parameter types are different.

As a part of this thesis, the TUID calculators for the JaMoPP and JML meta-models have been implemented. We chose an approach, which minimizes the maintenance effort by using a generic algorithm for constructing and resolving a TUID. The specific meta-model information is introduced by segment providing methods. Such methods take an element

and create a single TUID segment, which uniquely identifies the element on every given hierarchy level. The segment can only be used to identify the element if the matching parent object has already been found. Therefore, finding a single segment is much easier than finding the whole TUID. For instance, one of the segments of a parameter could be its name. If its parent method is known, this segment uniquely identifies the parameter in the set of parameters of the parent method. The algorithm for constructing the TUID uses these segments and the containment hierarchy of the given element: The algorithm follows the containment hierarchy from the element to the root element. The segments of the elements on the various levels are calculated and concatenated in inverse order. Thus, the first segment represents the root element and the last segment the element. Now the meta-information is prepended, which includes the name of the TUID calculator and the model location.

The resolution algorithm is shown in Algorithm 1. It takes the root element of the model and the TUID of the wanted element. The first two segments are stripped off because they only contain the generator identifier and the location of the model. They have already been used before the resolution to find the model and the matching TUID calculator. On every hierarchy level, the segments of all elements of the previously matched element are compared with the given segment to find the new matching object and descend the containment hierarchy by one level.

The advantage of both algorithms is extensibility and easy maintenance. A new element can be supported by just adding a single method that provides a single segment. Neither the calculation nor the resolution of the TUIDs has to be adjusted. The complexity of the calculation is $\mathcal{O}(n)$ and the complexity of the resolution is $\mathcal{O}(n * m)$ where n is the number of segments and m is the sum of contained objects, which have to be compared on each level. In the worst case, m is the total number of elements in the model, but it is much smaller in practice. The calculation cannot be done with less effort as long as the elements do not already have a unique identifier and there shall be one segment for each level in the hierarchy. The resolution could be done with a complexity of $\mathcal{O}(n)$, but this is only possible by adding knowledge about the meta-model to the algorithm, which makes it harder to maintain and extend.

Algorithm 1 Algorithm for finding an element by using its TUID.

```
candidate = rootObject
for i = 2 to tuid.length - 1 do
  newCandidate = false
  for all candidate.containedObjects do
    if calculateSegment(containedObject) == tuid[i] then
      candidate = containedObject
      newCandidate = true
    end if
  end for
  if not newCandidate then
    return null
  end if
end for
return candidate
```

5.2.3. Calculation of Correspondences

The correspondence model keeps track of relations between model elements. Vitruvius offers a meta-model and utility classes to use correspondence models. Such a model has to

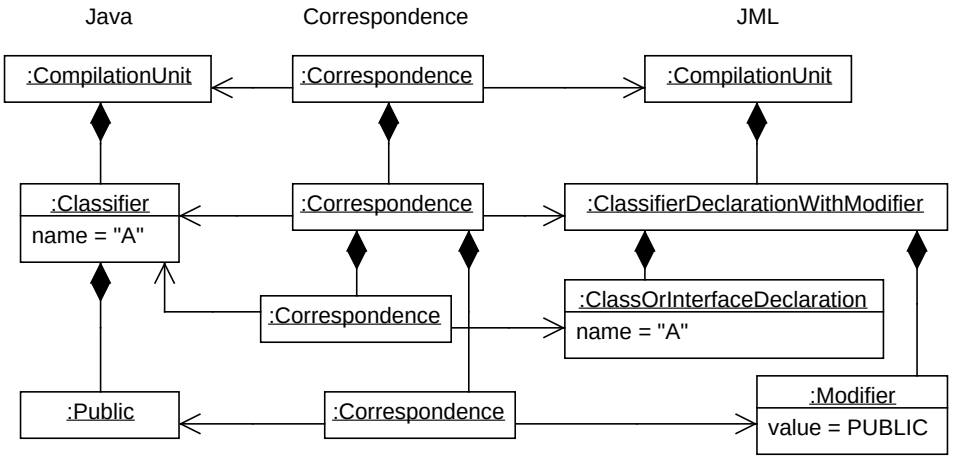


Figure 5.3.: Example for correspondence between a Java and JML classifier.

be built before the first transformation can take place because it is used to find the target of the change in other artifacts. Therefore, the correspondences are calculated during the initialization of the framework. The only input to this process are lists of models grouped by meta-model.

In our prototype, we search for matching models based on the location and name of the model first. For the Java and JML artifacts, this can be easily done because the file path is constructed by the full-qualified name, which is the same for both languages. Therefore, we match files by determining the relative path with respect to the source directory and testing the result for equality. We do not track correspondences between additional files (e.g. a reference to an imported class), because this is not necessary for our prototype and the correspondence meta-model has no construct for expressing this. Instead, we can look for such references during the synchronization.

We calculate the correspondences for every pair of matched files. We cannot use generic approaches for finding correspondences because they only work for very limited examples such as structural equal meta-models or conventionally named elements. The calculation traverses the containment hierarchy from the root object to the leafs. Like the containment hierarchy, the correspondences are hierarchical, so containment relations exist between them. In fact, the containment hierarchy of the model and the correspondences is the same, but there can be more or less correspondences on a single hierarchy level from the models point of view. This is illustrated in Figure 5.3. On the left hand side there is an excerpt of a Java model and on the right hand side an excerpt of a JML model. In the middle, the resulting correspondences are shown. An element can be part of multiple correspondences, which is necessary because a Java classifier is represented by two JML elements. This has to be considered for the dependency between the correspondences. It would be wrong to make the correspondence between the modifiers depend on the correspondence that involves the ClassOrInterfaceDeclaration. The parent correspondence should always be the correspondence that references both parents of the model objects. The dependencies are useful as soon as elements and their related correspondences shall be deleted. Hence, deleting the Java classifier would require deleting the top most correspondence referencing this classifier. Contained elements are deleted automatically.

Finding correspondences involves comparing the structure of the meta-models and matching the objects. The latter is similar to the task of creating a TUID calculator: In both situations, one has to find a set of attributes, which uniquely identifies an object on the

current hierarchical level. Good knowledge about the meta-model is required to achieve this. For instance, one has to know that the set of a method name and all parameter types in correct order identifies a Java and JML method. This example also shows that using only direct attributes of the object to match does not always provide the necessary information. One could not identify a method by just looking at the name. Depending on the meta-models, this is complex, but because structural elements are represented in the same way in Java and JML, often only directly contained objects have to be considered for the correspondence calculation.

5.3. Synchronization Mechanism

The previously described preparation results are used in the synchronization. They are the input of the synchronization process besides a change, which shall be processed. During the change processing, the given change is applied to the preparation results. Especially, the models of the VSUM and the correspondence models are updated. Hence, the output of the synchronization process is a set of changed prepared elements, which are saved afterwards. These steps are executed by just a few classes, which are shown in Figure 5.4. The used utility classes are omitted in this figure.

The next subsection deals with the synchronization process in more detail. Afterwards, we give an overview of the changes supported by the prototype and justify why this subset has been selected for implementation. The implementation of global transformations is the topic of the third subsection. A basic technique for syntactic changes is introduced and its usage in the transformations is explained by some concrete examples. The last subsection describes the synchronization by virtually processing a concrete change on concrete models.

5.3.1. Synchronization Process

In the following paragraphs, we describe the synchronization process of Vitruvius in general. Because the transformations that apply the change are not part of Vitruvius but extensions of the framework, we explain the usual implementation of such a transformation. Additionally, we cover the differences between atomic and composite changes.

After the detection of a change, Vitruvius starts the synchronization process. First, it uses Definition 5.1 to determine the pairs of meta-models for which a synchronization is necessary. For instance, a change on a Java file is related to the Java meta-model. Because there is a mapping between Java and JML, the instances of the JML meta-model have to be synchronized with the Java models. Therefore, a matching transformation executor (EMFModelTransformationExecuting) for Java and JML models is selected.

Definition 5.1 (Selection of meta-model pairs for synchronization.)**.** *After receiving a change of a model element with a given meta-model m the set of meta-model pairs for the synchronization is determined by $\mathcal{P}_{sync}(m)$ with*

- $\mathcal{M}$:= *set of all meta-models*
- $\mathcal{P} := \mathcal{M} \times \mathcal{M}$ *set of all mappings*
- $\mathcal{P}_{def} \subseteq \mathcal{P}$ *set of all defined mappings*
- $\mathcal{P}_{sync}(m) := \{(m, q) \mid q \in \mathcal{M} \wedge (m, q) \in \mathcal{P}_{def}\}$ *with* $m \in \mathcal{M}$

Second, the change is passed to the found executors. These executors are not part of the Vitruvius framework but have been implemented during this thesis. We implemented one executor (CSTChangeSynchronizer), which can synchronize Java to JML and vice versa. The executor delegates the change to a transformation (EObjectMappingTransformation) which can process the changed model element. For instance, a rename operation on a

method would result in a call to the transformation for methods. The transformation has distinct operations for each type of change, which can be an update operation, for instance. The operation distinguishes between the changed features of the object, which could be the name attribute.

The third part is the processing of the change. The steps for doing this are quite similar for the various change types: The corresponding objects have to be found in order to apply the change on them. All of their TUIDs are calculated and stored. They are used to update the previously used correspondences between model elements later. After that, the change is applied. This might be a simple update of an attribute or a quite complex operation affecting many models and algorithms, which is the case for refactorings. Afterwards, the existing correspondences are updated. This is necessary because the TUID that is used to reference model elements might have been indirectly changed by the transformation. Renaming a method would lead to a changed TUID because the name is used as one component of the TUID, for instance. These updates are crucial: Without up-to-date correspondences, finding the model elements to change would fail and we could not perform the transformation.

All changed elements are reported back to the executor (CSTChangeSynchronizer) in a final step. It performs the necessary operations on the files, which represent the models. This includes serializing, loading, creating and deleting of files. We did not implement creating or deleting complete models in our prototype, however.

Composite changes cannot be handled like this because the mentioned transformations can process atomic changes only. In our prototype, they are preprocessed by change refiners (CompositeChangeRefiner), which convert the composite change a) to a set of atomic model changes or b) to a custom model transformation. The refiners encapsulate their results in an executable result object. The former is processed by executing every single change in sequence like described above. For the latter the custom transformation is executed. Such transformations perform the same steps as regular transformations but can only process a single specific type of change. For instance, a custom transformation can perform the rename refactoring on all possible methods but does not support changing their return type. In contrast, regular transformations distinguish between changed objects and features as well as change operations and cover a wider range of changes. Only the first refiner that matches the composite change is used to avoid duplicate changes. Matching means that the given composite change is matched by the stored pattern in the refiner. Using only one refiner is sufficient as long as one composite change represents only one logical change. This is the case for the implemented change transformations mentioned in Subsection 5.3.2. If there are multiple logical changes in one composite change, all refiners have to be called and a consuming mechanism has to be established to remove already processed parts. Additionally, a fall through mechanism extracts all regular changes and processes them in sequence if no refiner matches the change.

5.3.2. Supported Changes

Implementing detection and processing routines for all possible changes is not possible in the context of this thesis. Therefore, we have chosen a subset of these changes and created implementations for them. The main objective is to enable evaluation of our synchronization concepts described in Chapter 4. We started implementing basic structural changes because these are the base for all other changes and are a good start to get to know Vitruvius and the change monitoring approach. These changes include creating and deleting methods, adding and deleting imports and so on. Afterwards, we considered global changes. We do not aim for comprehensive support for the developers but consider heavily used changes with a higher priority. Therefore, we selected the rename refactoring

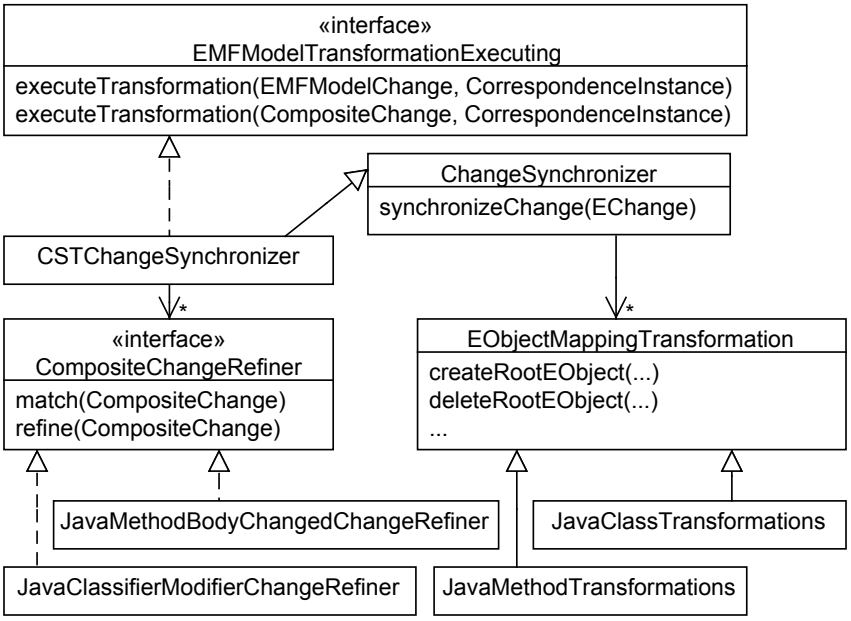


Figure 5.4.: Static structure of the synchronizer for code, tests and specifications.

as example for global changes from Java to other artifacts. According to the Eclipse usage data shown in Appendix A, the rename refactoring is one of the most heavily used refactorings besides the formatting and commenting operations. Obviously, formatting has no influence on other artifacts because no information is changed. Commenting can be considered as add and delete operations with respect to the effects on other artifacts, so it does not have to be processed in a special way. Organizing imports is used very often as well but it is not global as long as no model import is required as replacement for a deleted import. Because we assume that the technique for processing such changes is quite simple, we favored the rename refactoring over organizing imports as an example for global changes. We also selected the rename operation for global changes from JML to other artifacts. Additionally, we implemented some modifier changes that can be considered global too. Because there is no comprehensive source of information for the frequency distribution of changes in JML, we made a random choice.

An overview of the supported changes for the direction from Java to JML is given in Table 5.2 and for the direction from JML to JML in Table 5.3. A partial detection support means that the change detection approach [Mes14] is able to detect the change but does not offer fine-grained changes. Because this is necessary for the synchronization, we had to either extend it or implement composite change refiners that reduce the changes to a more concise representation.

In general, all changes are supported that are necessary to keep the structure of the Java and corresponding JML files consistent. This only includes applying the change to the corresponding element and not checking any further constraints that are described in the mappings in the first step. Indeed, some structural changes are not supported: All changes that affect the path of the file or its locator respectively are omitted. Hence, we do not process rename operations on classes, interfaces, enumerations and packages. At the moment, no concept for processing changes of the model location has been established in Vitruvius. Because extending Vitruvius is not the goal of this thesis, we postponed the implementation of such changes until a universal concept exists. In fact, interfaces and enumerations are not processed at all because of a limited amount of time. There is no technical or conceptual reason for this.

We also implemented complex synchronization logic in addition to the basic structural changes mentioned above. Renaming methods, fields and parameters is treated as refactoring, so all references are updated in JML. This includes the structure (e.g. rename the method in JML) and the specification (e.g. rename the method call in a precondition). Java does not have to be adjusted because we assume that the developer used the rename refactoring of Eclipse and all references have already been updated. Anyway, initiating the Java refactoring before our transformations can be integrated with less effort. Before performing the change, there is a check for name clashes with model methods or fields. Additionally, before deleting those elements, the transformation makes sure that there are no references to the element anymore. Otherwise, removing the element would lead to syntax errors.

Effects on the pure and helper modifiers are processed too. As soon as a method body is changed in Java, a transformation checks whether the method is free of side effects. Based on the result, the pure modifier is transitively removed or added. The same process is applied after adding or removing the modifier by hand. For helper the only relevant elements are invariants. As soon as an invariant calls a method, the helper modifier has to be added to prevent possible infinite loops during evaluation. Therefore, after every change of the invariant expression, the referenced methods are calculated and the modifier is added or removed. The same process is used after removing the modifier by hand.

5.3.3. Techniques for Handling Global Changes

Some of the supported changes that are described in the previous subsection are classified as global changes. Global changes can affect elements that do not directly correspond to the changed element. For instance, the rename refactoring can be considered global because all references to the renamed method have to be updated even if there is no tracked correspondence. In contrast, the creation of a parameter is not global because it affects only the parent method, which directly corresponds to the method in the other artifact.

Performing global transformations often requires some sort of code analysis. We need static code analysis for the implemented transformations. It is called static because it is not necessary to run the analyzed code. Precisely, we need information about references between elements, the type of statements in the method body or lists of similar elements, for instance. Obviously, we need this information for code and contracts. Such analysis can be done on the concrete representations (e.g. Java code) or on abstract representations (e.g. models).

JaMoPP already includes a cross-reference resolution mechanism, so references can be analyzed easily. Other analysis can be performed as well by using some sort of model queries. Several solutions like EMF-IncQuery [BURV11] exist for performing such queries but because our queries are quite simple we estimated that it is easier to implement them by ourselves than introducing a new technology with different concepts (EMF-IncQuery uses graph patterns). By using JaMoPP, we can do analysis for Java-based artifacts without introducing a new tool. Unfortunately, the implemented JML meta-model does not have the ability to resolve cross-references, so it cannot be used for all analysis. We consider implementing reference resolution for JML to be expensive. Instead, we reuse the mechanisms of JaMoPP by transforming the JML model to a JaMoPP representation with similar syntactic properties. We do this on the fly and only for analysis purposes. The so-called *shadow copy* is temporary and is never serialized or visible to the user.

The simplified approach for transforming the JML to a JaMoPP model is as follows: We create a copy of the existing JaMoPP models. Thereby, the structural elements (classes, methods, ...) of the JML models are already represented and only the specification

Element	Operation	Support	Comment
Class	Create	■ ◒	only classes not defining compilation unit name
	Delete	■ ◒	only classes not defining compilation unit name, ignoring unresolved symbols
	Rename	■ ○	
	Change Modifier	◪ ◒	only syntactic change, no checks
Interface	all	■ ○	no operation is supported
Enum	all	□ ○	no operation is supported
Import	Create	■ ●	
	Delete	■ ◒	no check for unresolved symbols in JML
Package	all	■ ○	no operation is supported
Method	Create	■ ●	
	Delete	■ ●	
	Rename	■ ●	
	Change Modifier	◪ ◒	only syntactic change, no checks
	Change Type	■ ◒	type only replaced for this element, no further checks
	Change Exceptions	◪ ◒	only throws declarations are synchronized, no further checks
	Change Body	◪ ◒	effects on pure
Field	Create	■ ●	
	Delete	■ ●	
	Rename	■ ●	
	Change Modifier	◪ ◒	only syntactic change, no checks
	Change Type	■ ◒	type only replaced for this element, no further checks
Parameter	Create	◪ ●	
	Delete	◪ ●	
	Rename	◪ ●	
	Change Modifier	◪ ◒	only syntactic change, no checks
	Change Type	◪ ◒	type only replaced for this element, no further checks
Variable	all	□ ○	not relevant
Comment	all	□ ○	not relevant

Table 5.2.: Collection of Java changes, which can be applied to JML by our prototype. Detection support: ■ native, ◪ modified, □ no. Synchronization support: ● full, ◒ partial, ○ no.

Element	Operation	Support	Comment
Method	Rename	□ ●	model methods
	Change Modifier	□ ◒	pure, helper
Field	Rename	□ ●	model, ghost fields
Invariant	Change Expression	□ ●	add/remove helper

Table 5.3.: Collection of JML changes, which can be applied to JML by our prototype. Detection support: ■ native, ◪ modified, □ no. Synchronization support: ● full, ◒ partial, ○ no.

specific parts still have to be transformed. For method specifications, we simply copy every expression that occurs in a specification statement into the corresponding method body and keep track of the relation between the original statement and the copied expression in the shadow copy. The expression can still contain JML specific keywords like \old(), which cannot be processed by JaMoPP. Therefore, we add dummy elements that can replace those constructs with respect to the syntax. Instead of \old() we insert a method call to a created dummy method oldReplacement() which returns the same type as the method. This is equivalent to the original statement with respect to the static analysis of the syntax. For type specifications such as invariants or constraints, we add a dummy method that is used to insert the expressions in the same way as for method specifications. For every model method or model/ghost field we create a regular method or field in the shadow copy first and perform the same operations as for regular methods afterwards.

The shadow copy is used to find references and perform simple refactorings such as renaming. To rename an element we simply rename the element in the shadow copy after all references have been resolved. Now, all expressions that have been copied to the method bodies are updated automatically. In a post-processing step, we copy all expressions back to their corresponding specification statements. This is possible because we stored the correspondence between them during the construction of the shadow copy. Afterwards, we replace keyword replacements by the original keywords. For instance, oldReplacement() is replaced by \old(). Now, the rename refactoring has been applied to the JML specifications.

Analysis of references and statements without changing the JML specifications is also possible. The transformation for handling the pure modifier only looks for calls to non-query methods and assignments to fields, for instance. This is carried out on the shadow copy by looking at all statements and searching for assignments to fields and method calls. If a method call is found, then it is checked whether the called method is marked pure. If not, the method is no query method. These simple rules even cover changes on parameters because the object can only be modified by assignments to fields or calls to non-query methods. If the pure modifier has been added or removed, the same procedure is applied to all callers of this method because they could have lost or gained the query status by this change.

We recorded some processing times for transformations that use the shadow copy to get an idea of the performance. The average duration of a synchronization was about 20 s on a Dell Precision T1650 desktop computer with an Intel Xeon E3-1240 V2 CPU and 8 GB RAM. This certainly is too much time for real-time applications. More than the half of that time is needed to initialize the shadow copy, which is used for reference resolving. Therefore, another approach for reference resolving in JML has to be found when developing a production-ready synchronization solution. For our prototype this is, however, sufficient.

5.4. Exemplary Execution

After the high-level description of the synchronization process in the sections above, we describe the synchronization by example now. We cover the preparation of elements such as the correspondences, the algorithms used during the synchronization and the output. The processed change will be the addition of a statement with side-effects to a query method, because this is considered a global change and we can cover most parts of the synchronization by this example.

The synchronization framework has to be started before the change can be processed. In a first step, our Vitruvius factory loads the meta-models of Java and JML and establishes

```
public class Test {                  ⟷  public class Test {
  private int i;                     ⟷    private int i;
  public void pureMethod() {         ⟷    public /*@ pure */ void pureMethod();
    int q = 1;
  }
  public void pureMethod2() {        ⟷    public /*@ pure */ void pureMethod2();
  }
  public void nonPureMethod() {      ⟷    public void nonPureMethod();
    i = 0;
  }
}                                          }
```

Figure 5.5.: Initial artifacts for the exemplary executions (Java left, JML right) with simplified correspondences.

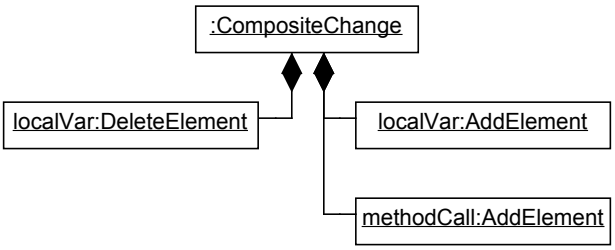


Figure 5.6.: Detected change after adding a method call to the pureMethod() method.

a mapping between those two models, which means that elements of Java are related to JML and vice versa. Hence, a synchronization between them is necessary. Loading all Java and JML files as models is the second step. We assume that there are only the two files shown in Figure 5.5 in the project – one Java and one JML file. During loading, the content of the files is transformed to meta-model conform models by the model printers and parsers. In the third step, the factory establishes the correspondences. For our example, only correspondences between the structural elements are important. Figure 5.5 illustrates only the relevant correspondences by using bidirectional arrows. The last step is constructing and registering the transformation executer for Java and JML. At this stage, the VITRUVIUS and our synchronization system are initialized and all necessary preparations are done. The code monitors watch the artifacts to detect changes now.

Now, the user adds the method call nonPureMethod() into the pureMethod() method. The monitored editor for Java detects this by using an extension for the classifier plug-in which compares method bodies and reports changed ones. We consider determining the minimal change of the body, which correctly reflects the performed changes, just by comparing the previous and new state to be hard. Moreover, this is not be necessary for our implemented transformation. Therefore, the composite change shown in Figure 5.6 is created. For every statement of the previous state, a delete change is added. For every statement of the new state an add change is added. This change is not minimal but contains all necessary information to reach the new state from the previous one.

The VITRUVIUS framework propagates the change shown in Figure 5.6 to the transformation executer. It detects a composite change, which has to be preprocessed in order to apply the contained changes. A matching composite change refiner is found by matching their patterns to the composite change. The pattern of the refiner for method body changes matches changes that contain only add and delete changes for statements belonging to the same method. Because this is true for our composite change, the refiner matches the change. The result of the refinement is an instance of a transformation for method body changes that has already been initialized with the changed method and the previous

and new state. The synchronizer can execute this result.

The transformation for the method body change is executed now. In general, we have to determine whether the query status of the method has been changed and whether we have to adjust the method and other methods. We need to resolve references in order to do this, so we create a shadow copy of the Java models. We replace the statements of the changed method with the new statements in the shadow copy. Thereby, we can determine the query status for the changed method by collecting all method calls and field assignments in the body. If we find a field assignment, the method is no query. For a method call, we check whether it is marked as pure by analyzing the corresponding JML method. As soon as we detect a non-pure method call, the method is no query. Because we added a non-pure statement (call to the non-query method nonPureMethod()), the method is no query. By comparing the new query status with the old one, we detect that the query status is lost, so we have to make adjustments.

We have to remove the pure modifier from the changed method and all methods that transitively call it, because it lost the query property. First, we collect all transitively calling methods, which will be pureMethod2 only. Before removing the modifier for all found methods, we make sure that none of them is used in a specification. If we remove the modifier for such a method, the specification could not be evaluated without side-effects anymore. This would be harmful. In our context, no specifications refer to the methods that we want to change. Therefore, we remove the modifier from pureMethod and pureMethod2.

Usually, we would have to update the correspondences, which track the relation between model elements (e.g. pureMethod in the Java model belongs to pureMethod in the JML model). In this case, we do not have to because the modifier is not tracked inside the correspondences at all and no TUID, which is used to reference elements inside correspondences, depends on the modifier. Instead, we skip this step and simply return the changed elements pureMethod and pureMethod2 to the transformation executing. The models of the changed elements are determined to save the changes. In our case, the JML file of the Test class is serialized. Now, the change has been completely processed and the change monitor can look for new changes again.

6. Evaluation

We evaluated the implemented prototype to prove our concepts developed in Chapter 4. In this chapter, we first explain the chosen evaluation procedure and the goals that we aimed for. Second, we introduce the real-world project that we used during our evaluation. Additionally, we explain how we modified this project to make it usable with our prototype. In the last section, we discuss the results of our evaluation.

6.1. Procedure

The goal of our evaluation was showing that the concepts described in Chapter 4 are correct and that the implementation works. Systematic testing using a validation set that was not used during the design and development of our approach showed both. During the evaluation, we used automated system tests to make the results reproducible and to perform many tests in little time by using an automated selection algorithm for change subjects. We tested the whole process reaching from change detection to change application by using system tests. These tests took place in a real-world project, which is introduced in the next section. Only a subset of the consistency concept that we developed and described in Chapter 4 is covered by this evaluation because we can only test concepts with an implementation. Evaluating the remaining concepts is left for future work. In addition to the system tests, we used unit tests for transformations and correspondence mechanisms during the development. An overview of the test suites is given in Table 6.1. The remainder of this chapter deals with the system tests only.

In order to test systematically we need changes that shall be processed. The concrete change and its context determines which change handling will be executed. We selected

Property	Test Suite 1	Test Suite 2	Test Suite 3
Coverage	path	context	–
Type	system	system	unit
Selection	manual	automatic	manual
Syntax Check	yes	yes	(yes)
Semantics Check	yes	no	yes
Validation Data	JavaCards API	JavaCards API	dummies
# Tests	32	1085	134

Table 6.1.: Overview of test suits used for evaluating the prototype.

two criteria for change selection, which produced two test suites for two sets of changes. The test suites are described in the following two paragraphs. The limitations for the test suites are described afterwards.

The first criterion was coverage of all available paths in the implemented change handling. For example, we had to create two tests if a change handling checked name clashes before applying the change: One test covered the name clash and the other one the regular change application. We selected the changes manually in order to cover as much paths as possible. This lead to a concise test suite and allowed us to verify the results in detail: After the change had been applied, we performed a type check of the whole project to reveal errors in the syntax. We used the OpenJML compiler to do this. Interested readers can have a look at Appendix B for a more detailed description of the test setup and the parameters used for the type check. After this syntax check, we calculated the delta between the old and the new state of all files in the project and compared it with a stored reference delta. We created this delta in a previous test run, checked it for errors in the syntax and semantics and manually set it as reference delta. The test failed if the automatic syntax check revealed errors or if the produced delta did not match the reference delta.

The second criterion was coverage of all elements that could be changed. We call these elements *change targets*. All named elements are targets for the rename change, for instance. An algorithm selected the targets automatically. For rename operations, this algorithm collected all elements of a specified type and filtered unsupported elements. The results were passed to the testing framework, which created separate tests from it. After the change had been processed, we performed a syntax check only because rename operations only have effects on the syntax. So creating reference deltas for every test is not useful.

Both test suites did not cover all change types and targets because of the following limitations of our prototype and the project used for evaluation. In the first test suite, we omitted changes that are not handled completely by our prototype and therefore would most probably lead to syntax errors. According to Subsection 5.3.2, such changes are modifier and type changes on all Java elements. Furthermore, the evaluation project did not contain several elements that are necessary for covering all paths. An overview of these elements and the paths which could not be covered anymore is given in Table 6.2. In the second test suite, we omitted interfaces and constructors completely. The former are not supported by the prototype at all. The latter cannot be processed because of a technological gap between the change monitoring approach and the prototype: The prototype uses the latest JaMoPP version, which treats constructors as separate elements. The change monitoring uses the release version of JaMoPP instead, which treats constructors as methods.

6.2. Investigated Software System

The evaluation was done using a real-world project to test the concepts and implementation in a realistic context. The requirements on this project were quite restrictive: It had to meet our restrictions formulated in Section 4.1 (JML language levels 0 and 1, no dead code, separate specification files, no specification file without code file) and had to contain a considerable amount of various JML statements. We looked for such projects in open source repositories, publications and the JML examples located in the OpenJML repository. Unfortunately, the source code of only a few projects is available, although JML can be considered the most popular behavioral specification language for Java [CJLK10]. Additionally, those projects are often teaching projects and therefore have a very limited amount of lines of code (LOC). Our research revealed only two eligible projects: An implementation of the JavaCards API for smart card development and a remote voting

Project Limitation	Change	Untested Path
no model methods	Rename Model Method	rename model method
	Rename Model Method	name clash with JML
	Rename Model Method	name clash with Java
	Rename Method	name clash with JML
	Add Method	name clash with JML
	Change Invariant	removed call of method not used in other places
	Change Invariant	removed call of method used in other places
	Change Invariant	add call of method used in other places
	Change Invariant	add call of method not used in other places
no model fields	Rename Model Field	rename model field
		name clash with JML
		name clash with Java
no unused imports	Remove Import	remove import
no unused field	Remove Field	remove field
no second level class	Remove Class	remove class

Table 6.2.: Limitations of the JavaCards API project and resulting untested paths in the first test suite.

system called KOA[1]. Both systems were developed using ESC/Java, which is a language and a tool for extended static checking [FLL+02]. Although JML and ESC/Java share the same syntax, some differences exist. For instance, ESC/Java(2) does not check the visibility modifier, but JML-based tools do. [LPC+13, appx. G] We chose the JavaCards API as evaluation project, because it is more self-contained than the KOA project and it has already been used in case studies of related research [Mos07].

JavaCards is a technique for developing applications for smart cards. It consists of the JavaCards API, which encapsulates the access to the JavaCards virtual machine and native methods on the smart card. A new API is necessary because the virtual machine has been adjusted to fit the needs of the smart card environment. The most prominent restriction of this environment are the limited resources. The advantages of using the JavaCards API are that applications become platform independent, can make use of Java language features and are protected against other applications on the same smart card. [HMGM02, chap. 8.1] In [Mos07] the author has implemented and specified the JavaCards API for verification purposes using the KeY system [BGH+07]. It can be used to verify any application using JavaCards. The sources do not contain unit tests because a complete verification makes them superfluous as long as the specifications used to verify the implementation are correct.

Unfortunately, we could not use the JavaCards API without modifications because there is a gap between ESC/Java and JML [LPC+13, appx. G]. The former was used to specify the project and the latter was the target language for our prototype. Additionally, our prototype did not support all JML constructs. Therefore, we had to adjust the project to make it usable with our prototype. The necessary changes can be separated into two groups: The first group is shown in Table 6.3 and contains changes that affect the syntax only but do not change the semantics of the contracts. Changes of this group do not

[1]Kiezen op Afstand (Dutch: remote voting)

Change	Reason	#
converted $a \implies b$ to $\neg a \vee b$	restriction of prototype	46
moved specifications from Java to JML	restriction of prototype	43
converted some //@ to /**/	restriction of prototype	23
removed comment specifications	restriction of prototype	5
added missing fields	assumption of prototype	23
added missing constructors	assumption of prototype	14
added missing import statements	assumption of prototype	13
added missing methods	assumption of prototype	1
adjusted specification visibility in Java	bug in OpenJML, Appendix C	28
adjusted specification visibility in JML	required [LPC+13, chap. 2.4]	24
added missing throws declarations	required [LPC+13, chap. 17.3]	21
removed implementations from JML	not allowed [LPC+13, chap. 17.3]	13
converted ghost to regular field	simplified handling in prototype	13

Table 6.3.: Overview of syntactic changes applied to the JavaCards API project to make it usable with our prototype. The project contained ca. 1410 specification items in total.

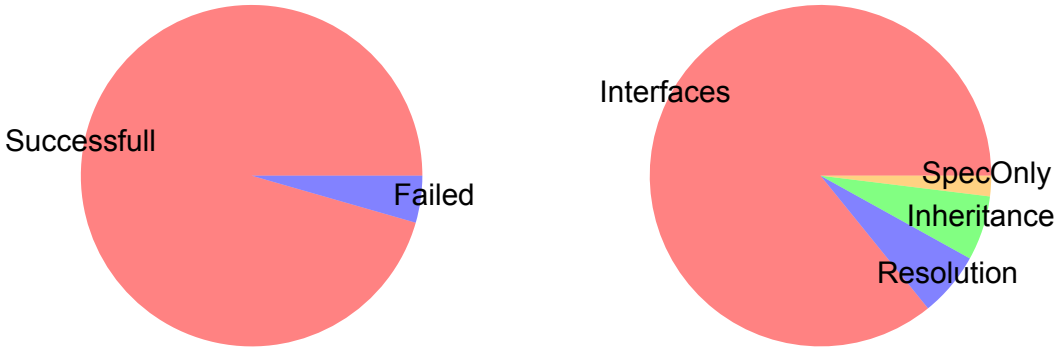
Change	Reason	#
removed assignable	restriction of prototype	344
removed static final fields from JML	bug in OpenJML, Appendix C	39
removed signals and signals_only	restriction of prototype	30
removed specifications from Java	assumption of prototype	16
removed bit operations from JML	restriction of prototype	8
removed casts from JML	restriction of prototype	2
removed \forall in nested expressions	restriction of prototype	1

Table 6.4.: Overview of semantic changes applied to the JavaCards API project to make it usable with our prototype. The project contained ca. 1410 specification items in total.

weaken the specification or change the implementation and therefore have no effect on our evaluation of the correctness. The second group is shown in Table 6.4 and contains changes that affect the semantics too. We had to apply the changes of both groups in order to be able to parse and process the artifacts in our prototype. We had to remove ca. 31 % of the specification items, which were assignable clauses mostly. Although this weakened the contracts, this did not affect our evaluation because we only wanted to compare the state before a change with the state after the change to determine the syntactic and semantic changes. Therefore, the only requirement was a working initial state.

6.3. Results and Discussion

The selection criteria for test data described in Section 6.1 lead to 1117 tests. We created 32 tests for the first test group that covered as much paths of the supported change handlers as possible. Some paths were not covered as described in the previous section. The second group contained 1085 tests and covered the renaming of all fields, methods and parameters of the JavaCards API project. All tests of the first group succeeded which means that the result of the change handling was correct with respect to the syntax and the semantics. In the second group, 95 % of the tests produced a correct result with respect to the syntax as can be seen in Subfigure 6.1(a). The reasons for failing 49 tests are classified in Subfigure 6.1(b): 42 tests did not succeed because interfaces were involved.



(a) Overview of succeeded and failed tests.

(b) Partitioning of the reasons for failed tests.

Figure 6.1.: Test results of the second test suite of the evaluation tests.

We do not process interfaces at all, so references of specifications inside interfaces are not updated. This can lead to syntax errors. Three tests failed because we tried to change an inherited method. At the moment, inheritance is not considered. Problems during the resolution of the shadow copy made another three tests fail. We still have to investigate the reason for these problems. The remaining test did not succeed because we produced a syntax error in a specification file without a corresponding Java file. We do not support this at the moment. If we consider the tests that do not meet our requirements as ignored, we only failed the three tests that had problems in the resolution process. These tests are 0.3 % of the whole second test suite.

The results of the first test group show that our concept and the implementation work for real-world setups. This applies to simple and complex operations. Obviously, this test group does not prove the correctness of the handling in general because we only tested a very limited amount of concrete contexts. Nevertheless, we covered all types of supported contexts in the JavaCards API project. The evaluation of possibly missed contexts in other projects is left for future work. The second test group showed that our implementation produces syntactical correct results in most cases, but we found a weakness of the resolution process that is used to find references. We still have to investigate the reason for the problems and find a solution. So apart from this bug, the handling of the rename refactoring worked in our case study as long as the assumptions and requirements were met. We covered all supported change targets during the tests, so the results are representative.

7. Conclusion and Future Work

7.1. Conclusion

The focus of this thesis was the development of a concept for co-evolution of code, contracts and unit tests. Co-evolution is important because a contradictory information in one of the artifacts leads to a malfunction of the composed software system. So when applying a change to one of these artifacts, the other ones have to be evolved too. Because this process is error-prone and can be extensive, we developed a concept that can be used to apply changes without potentially omitting an important step.

To develop our concept we first defined the overlap between the artifacts and the constraints for the relation between the corresponding elements. We used Java code, JML contracts and JUnit tests as concrete examples because they are popular, mature and feature-rich. In addition to the overlap, we developed reactions for changes. These reactions apply the change to other artifacts and maintain the constraints of the overlap. Changes can be simple changes such as adding an import or more complex changes such as refactorings. The combination of the overlap and the reactions is our consistency concept. We generalized the language specific concept by mapping the language specific features to generic features that usually occur in the corresponding artifact types. We used a comparison of several specification languages to determine common features. Regarding the code language, we assumed that an object-oriented language is used.

We implemented our concept for Java code and JML contracts and omitted tests because of a limited amount of time. Our solution uses model-driven techniques such as the Vitruvius synchronization framework or a change monitoring approach [Mes14]. To make use of the features of Vitruvius we convert the artifacts into models using model printers and parsers for Java and JML. The latter has been implemented as part of this thesis. We could not implement the whole consistency concept in the given time, so we focused on the most basic structural changes and some more complex changes such as the rename refactoring and the handling of two JML modifiers. All supported changes could be handled by using static code analysis, which has been implemented by model queries. These queries are carried out on a so-called shadow copy, which allows resolving references between Java and JML.

Our implementation has been evaluated in a case study using the JavaCards API project [Mos07]. It contains a considerable amount of code and specifications but we had to removed constructs that are not supported by our prototype. We executed two test suites

on the project: The first test suite aimed for coverage of all possible paths in the tested change transformations and the second one covered all elements that can be changed by the rename refactoring. All tests of the former test suite succeeded whereas the latter revealed some errors: 0.3 % of the tests failed. In general, the tested implementations and concepts worked.

7.1.1. Limitations

At the moment, not all of the change reactions of the concept can be applied because there are no convenient approaches for contract inferring, merging and fixing. The inferred contracts often are weaker than the manually written contracts. Even if there was a merging technique for contracts, the gap between inferred and manually written contracts could lead to problems. Automatic bug fixing techniques exist as well but take too much time and are limited to quite restrictive contexts. Therefore, changes on generic specification statements such as preconditions, postconditions or invariants cannot be processed automatically at the moment.

We did not prove our whole concept because we were not able to implement and evaluate all parts in the context of this thesis. Therefore, users of the concept have to evaluate the used parts first before relying on its correctness.

The prototype is not production ready yet. First, the change handling which uses static code analysis takes too much time for real-time situations. Second, there are several issues related to Vitruvius and the change monitoring approach. Some further development is required to fix the issues and the clue code provided by our synchronization system.

7.1.2. Benefits

To our knowledge, no precise consistency concept for code, contracts and unit tests exists yet. The definition of the overlap, the constraints for the contained elements and the change reactions can be used in following research projects on a similar topic. The concept can be used to develop refactoring tools for contracts, for instance. Such tools can lead to a higher distribution of a technology, so JML can become more popular in industrial projects too.

Our approach can save the developer from recurring and error-prone modification cycles during development. In fact, the developer does not have to know details about the synchronization system to make use of it because changes in the editor are automatically detected and applied. Even if a change cannot be applied, the developer is informed about problematic parts, which could have been overseen.

7.2. Future Work

The focus of this thesis is a concept to keep code, contracts and unit tests consistent, which can be seen as a foundation for further research. The following paragraphs give a short overview of possible research topics and some open points regarding our implementation and evaluation.

We implemented a *model printer and parser for JML* as part of this thesis. The main objective was to convert JML to models and vice versa while omitting full coverage of the language, validation or a precise syntax. We suggest implementing a tool that includes these aspects before implementing further model-based tools for JML. Especially, cross-references should be included to allow model queries. Extending JaMoPP is considered a promising approach.

The *evaluation of our consistency concept* is not yet finished. Hence, implementing and testing more aspects of our concept is required before it can become a good foundation

for further research. If our prototype shall be used for this, the bugs revealed during the evaluation should be fixed first. Additionally, we suggest investing some time in VITRUVIUS and the change monitoring approach as well because both are prototypes too.

Another open point is the *evaluation of the generalized concept*. We developed the concept with focus on Java, JML and JUnit and generalized it afterwards. We did not evaluate our generalization yet but we are interested in the evaluation of the concept for other code and specification languages.

Based on the concept some *tools for real-time application* of our concept can be developed. Such tools have a high potential of raising the popularity of an approach. As a result, more people would use contracts. This would lead to more feedback and representative projects for case studies. Even user studies become possible.

Bibliography

[AK02] P. Abercrombie and M. Karaorman, „jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java“, *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4, 2002.

[BF01] D. Bartetzko and C. Fischer, „Jass - Java with Assertions“, *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, 2001.

[BGH+07] B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt, „The KeY system 1.0 (deduction component)“, in *Proceedings, International Conference on Automated Deduction*, Springer Berlin Heidelberg, 2007.

[BKR09] S. Becker, H. Koziolek, and R. Reussner, „The Palladio component model for model-driven performance prediction“, *Journal of Systems and Software*, vol. 82, 2009.

[BURV11] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, „A Graph Query Language for EMF Models“, in *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2011.

[CA10] Y. Cheon and C. Avila, „Automating Java Program Testing Using OCL and AspectJ“, in *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*, IEEE Computer Society, 2010.

[CCH08] C.-T. Chen, Y. Y. C. Cheng, and C. C.-Y. Hsieh, „Contract Specification in Java: Classification, Characterization, and a New Marker Method“, *IEICE transactions on Information and Systems*, vol. E91-D, no. 11, 2008.

[CCLB12] P. M. Cousot, R. Cousot, F. Logozzo, and M. Barnett, „An Abstract Interpretation Framework for Refactoring with Application to Extract Methods with Contracts“, in *Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, ACM SIGPLAN, 2012.

[CJLK10] P. Chalin, P. R. James, J. Lee, and G. Karabotsos, „Towards an industrial grade IVE for Java and next generation research platform for JML“, *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 6, 2010.

[CL02] Y. Cheon and G. Leavens, „A Simple and Practical Approach to Unit Testing: The JML and JUnit Way“, in *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Proceedings*, Springer Berlin Heidelberg, 2002.

[Cok14] D. R. Cok, *OpenJML User Guide and Reference*, 2014. [Online]. Available: `http://jmlspecs.sourceforge.net` (visited on 07/03/2014).

[dBLM+04] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J. Lanet, „A Case Study in JML-Based Software Validation“, in *Proceedings of the 19th IEEE international conference on Automated software engineering*, IEEE Computer Society, 2004.

[DD07] P. H. Dave and H. B. Dave, *Design and Analysis of Algorithms*, 1st ed. Pearson Education Canada, 2007.

[DH98] A. Duncan and U. Hölzle, „Adding Contracts to Java with Handshake“, Department of Computer Science, University of California, Tech. Rep., 1998.

[Ecl14a] Eclipse Foundation, *Eclipse documentation - Eclipse Luna*, 2014. [Online]. Available: `http://help.eclipse.org/luna` (visited on 08/20/2014).

[Ecl14b] ——, *Eclipse Usage Data Collector*, 2014. [Online]. Available: `http://eclipse.org/org/usagedata` (visited on 10/28/2014).

[EGL07] G. Engels, B. Güldali, and M. Lohmann, „Towards Model-Driven Unit Testing“, in *Proceedings of the 2006 International Conference on Models in Software Engineering*, Springer Berlin Heidelberg, 2007.

[EKZC14] S. Efftinge, J. Köhnlein, S. Zarnekow, and Contributors, „XText Documentation (2.7)“, Tech. Rep., 2014.

[Fel03] Y. A. Feldman, „Extreme Design by Contract“, in *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*, Springer-Verlag, 2003.

[FG06] Y. A. Feldman and L. Gendler, „Discern: Towards the Automatic Discovery of Software Contracts“, in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, IEEE Computer Society, 2006.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, „Extended Static Checking for Java“, *ACM SIGPLAN Notices*, vol. 37, no. 5, 2002.

[Fow99] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 4th ed. Addison-Wesley, 1999.

[GFT06] M. Goldstein, Y. A. Feldman, and S. Tyszberowicz, „Refactoring with Contracts“, in *Proceedings of AGILE Conference (2006)*, 2006.

[Gup10] A. Gupta, „An Approach for Class Testing from Class Contracts“, in *Proceedings of the 8th International Conference on Automated Technology for Verification and Analysis*, Springer Berlin Heidelberg, 2010.

[HGS05] T. Hammel, R. Gold, and T. Snyder, *Test-Driven Development: A J2EE Example*, 1st ed. Springer Apress, 2005.

[HJK09] F. Heidenreich, J. Johannes, and S. Karol, „Derivation and Refinement of Textual Syntax for Models“, in *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, Springer Berlin Heidelberg, 2009.

[HJSW10] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, „Closing the Gap between Modelling and Java“, in *Proceedings of the Second International Conference on Software Language Engineering*, Springer Berlin Heidelberg, 2010.

[HMGM02] V. Hassler, M. Manninger, M. Gordeev, and C. Müller, *Java Card for E-Payment Applications*, 1st ed. Artech House, 2002.

[HN01] H. Heinecke and C. Noack, „Integrating Extreme Programming and Contracts“, in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Addison-Wesley Professional, 2001.

[HT04] A. Hunt and D. Thomas, *Pragmatisch Programmieren - Unit-Tests mit JUnit*, 1st ed. Hanser, 2004.

[Hul10] I. Hull, „Automated Refactoring of Java Contracts“, Master's Thesis, University College Dublin, 2010.

[JUn14] JUnit-Team, *JUnit wiki*, 2014. [Online]. Available: `https://github.com/junit-team/junit/wiki` (visited on 08/25/2014).

[KBL13] M. E. Kramer, E. Burger, and M. Langhammer, „View-centric engineering with synchronized heterogeneous models“, in *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, ACM, 2013.

[LBR99] G. Leavens, A. Baker, and C. Ruby, „JML: A Notation for Detailed Design“, in *Behavioral Specifications of Businesses and Systems*, Springer Berlin Heidelberg, 1999.

[LCO+07] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva, „Contract Driven Development = Test Driven Development - Writing Test Cases“, in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ACM Press, 2007.

[LKP02] M. Lackner, A. Krall, and F. Puntigam, „Supporting design by contract in Java“, *Journal of Object Technology*, vol. 1, no. 3, 2002.

[LPC+13] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl, „JML Reference Manual, Draft Rev. 2344“, Iowa State University, Tech. Rep. May, 2013.

[Mes14] D. Messinger, „Incremental Code Architecture Consistency Support through Change Monitoring and Intent Clarification“, Master's Thesis, Karlsruhe Institute of Technology, Germany, 2014.

[MM02] R. Mitchell and J. McKim, *Design by Contract, by Example*, 1st ed. Addison Wesley, 2002.

[Mos07] W. Mostowski, „Fully Verified Java Card API Reference Implementation“, in *In Proceedings, 4th International Verification Workshop (VERIFY'07), Workshop at CADE-21*, 2007.

[OMP04] J. S. Ostroff, D. Makalsky, and R. F. Paige, „Agile Specification-Driven Development“, in *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference*, Springer Berlin Heidelberg, 2004.

[PFN+14] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, „Automated Fixing of Programs with Contracts“, *IEEE Transactions on Software Engineering*, vol. 40, no. 5, 2014.

[PFP+13] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, „What Good Are Strong Specifications?“, in *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013.

[SGM02] C. Szyperski, D. W. Gruntz, and S. Murer, *Component Software Beyond OO Programming*, 2nd ed. Addison-Wesley, 2002.

[SL04] T. Skotiniotis and D. Lorenz, „Conaj: Generating contracts as aspects“, College of Computer and Information Science, Northeastern University, Tech. Rep., 2004.

[Som12] I. Sommerville, *Software Engineering*, 9th ed. Pearson, 2012.

[Sta73] H. Stachowiak, *Allgemeine Modelltheorie*, 1st ed. Springer-Verlag, 1973.

[SV06] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*, 1st ed. Wiley, 2006.

[ZN11] D. M. Zimmerman and R. Nagmoti, „JMLUnit: The Next Generation“, in *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, Springer Berlin Heidelberg, 2011.

Appendix

A. Eclipse Usage Data for Java Refactorings

The Eclipse Usage Data Collector is part of the Eclipse IDE. It is used to collect usage data for various packages. This data is uploaded and made available on the Eclipse website. By now, it has been suspended. The latest available usage data is dated to January 2010.[Ecl14b]

The objective of the usage data analysis is to find the most heavily used refactorings in Java in our context. We chose the data that is available and has the most participants. The command data from November 2009 fulfils these criteria. We selected the data related to the Java plug-in and filtered irrelevant commands like opening dialogs or jumping to locations. Because the data differentiates between various versions, we aggregated the results by summing up the amount of executions and users.

Command	# Executions	# Users
.edit.text.java.format	14602	4875
.edit.text.java.organize.imports	11254	5637
.edit.text.java.toggle.comment	11006	4567
.edit.text.java.rename.element	6817	3850
.edit.text.java.create.getter.setter	2307	1858
.edit.text.java.indent	1794	939
.edit.text.java.add.import	887	433
.edit.text.java.add.javadoc.comment	672	350
.edit.text.java.add.block.comment	486	330
.edit.text.java.override.methods	374	324
.edit.text.java.move.element	336	243
.edit.text.java.extract.local.variable	299	207
.edit.text.java.extract.method	254	172
.edit.text.java.toggleMarkOccurrences	197	121
.edit.text.java.copy.qualified.name	188	159
.edit.text.java.generate.constructor.using.fields	186	162
.generate.javadoc	163	111
.edit.text.java.surround.with.try.catch	157	111
.edit.text.java.clean.up	107	67

Table A.1.: Aggregated Eclipse usage data [Ecl14b] for Java edit operations in 09/2009 with more than one hundred executions.

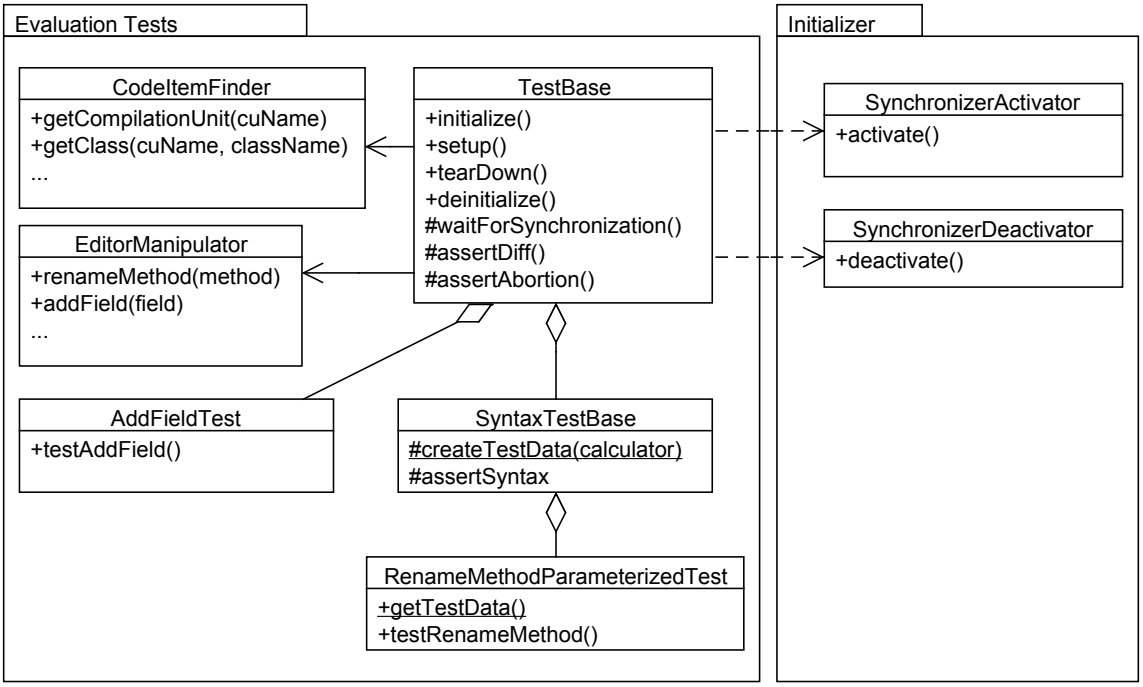


Figure B.1.: Simplified test structure for evaluation tests.

B. Detailed Test Setup Used in the Evaluation

We used automatic plug-in tests to perform the system tests for our evaluation. This section gives a short overview on the test structure, the test data selection for the syntax tests and the test oracles.

An overview of the test structure is shown in Figure B.1. The base class handles the activation and deactivation of the synchronization system as well as the cleanup of the workspace after every test. It provides access to two objects required to perform the change. The CodeItemFinder is used to find structural elements in the evaluation projects and the EditorManipulator is used to perform changes on the found elements. After issuing the change, the test uses two of three methods of the base class. First, it has to wait for the synchronization to finish. Afterwards it can assert that the difference between the old and the new state is the same as the reference delta or it can assert a transformation abortion because of a conflict. Tests for the syntax work in a similar way, but they have to provide the test data first. For instance, the test for renaming methods provides a list of all methods, which shall be tested afterwards. The test procedure is the same as for regular tests but after the synchronization the assertion method for the syntax is always called, so no delta is compared.

The selection of the test data is done as follows: For the rename operation on fields all fields are selected, which belong to a class. For the rename operation on methods all methods are selected, which belong to a class and are not a constructor. The parameters for the rename test are all parameters of the previously selected methods. All these selections are done in Java files.

We use a syntax check and delta comparison as test oracles. The syntax check is done by compiling each Java file with the OpenJML compiler as shown in Listing 7.1. The parameters for the compilation can be seen in the listing. The test fails if the return code is not equal to zero. The delta comparison is a simple string comparison. After the synchronization, we use a diff library[1] to calculate the delta of all files and compose it in

[1] `https://code.google.com/p/java-diff-utils/`

a single delta. Afterwards we compare this delta with a stored delta. The test fails if the strings do not match.

```
1  private int compile(File javaFile, File projectDir, File jmlSpecsLib, File
       jmlRuntimeLib, PrintWriter stdout) {
2    IAPI openjmlAPI = Factory.makeAPI(stdout, null, null, new String[0]);
3    String[] params = new String[]{
4      "-check",
5      "-jml",
6      "-nullableByDefault=false",
7      "-no-internalSpecs",
8      "-no-internalRuntime",
9      "-purityCheck=true" ,
10     "-d", new File(projectDir, "bin"),
11     "-sourcepath", new File(projectDir, "src"),
12     "-specspath", new File(projectDir, "specs"),
13     "-cp", new File(projectDir, "bin").getAbsolutePath() + File.
           pathSeparatorChar + jmlSpecsLib + File.pathSeparatorChar +
           jmlRuntimeLib,
14     javaFile
15   };
16   return openjmlAPI.execute(stdout, null, null, params);
17 }
```

Listing 7.1: Compilation procedure for JML files after performing the synchronization.

C. OpenJML Pitfalls

We used OpenJML in our evaluation for checking the syntax of the results. First we type checked all files via command line and later by calling the library directly via Java. We used the compiler options mentioned in Appendix B. We encountered the following problems and pitfalls:

- When compiling via command line we were unable to change the specification visibility by adding the spec_public or spec_protected modifier to the code element in the JML file. We could change the visibility only by adding the modifier to the code element in the Java file. When compiling via a library call adding the modifier to JML was enough.
- We were unable to declare static final fields in the JML file, because the compiler requested an initializer. Unfortunately, initializers are not allowed in JML files [LPC+13, chap. 17.3]. Therefore, we had to remove the fields completely from the JML files.
- When compiling via a library call, the JML runtime libraries jmlruntime.jar and jmlspecs.jar have to be provided explicitly in the class path.

D. Review Manual for Prototype

A code review is a standard procedure at the chair of Software Design and Quality, IPD, KIT. The author of the code and a group of researchers and students go through the code and discuss interesting parts. A higher code quality and better reusability are the objectives of the review. In this section, the review manual, which is distributed before the code review, is attached.

Code Review Manual for Synchronization System for Code, Contracts and Unit Tests

Stephan Seifermann
stephan.seifermann@student.kit.edu

November 14, 2014

1 Concept

Code, contracts and unit tests are artifacts of the software development process, which have a semantic overlap. Typical representations of them are Java code, JML contracts and JUnit tests. In my thesis, I want to keep these representations consistent after changes occurred.

To achieve this, the artifacts are converted to models and synchronized via the Vitruvius framework. For Java code, JaMoPP is used to create the model. For JML, a simplified Xtext grammar is developed as part of this thesis from which a model parser and printer is created. JUnit code is Java code, so JaMoPP can be used too. Anyway, unit tests are not considered because of a lack of time.

The transformations that perform the synchronization are separated into directions. A transformation might be very simple (e.g. changing an attribute) or might be complex (e.g. a rename refactoring).

2 Foundations

I assume that Java, JUnit and Xtext are well known. JML is a specification language for Java. The specifications are written inside Java comments, so the code can still be compiled with a regular Java compiler. After compiling the code with a special JML compiler, the specifications can be checked during runtime. A violated specification can be indicated by assertions, exceptions or logging messages. It is possible to write specifications inside regular Java files or to use separate specification files, which contain the structural elements of Java (e.g. method declarations without body) and the specifications. In this thesis, the latter is used because of easier model handling. For a subset of JML keywords and a short example, please refer to `http://en.wikipedia.org/wiki/Java_Modeling_Language`

3 Setup

The synchronization system is implemented as Eclipse plug-ins, so you need the Eclipse IDE. I've worked with Luna but older versions might work as well as long as the following required plug-ins are available:

- Eclipse Modeling Tools (use Eclipse MDE edition)
- JaMoPP Trunk $\geq$ 09.10.14 (Update URL: `http://jamopp.org/update_trunk`)
- Xtext $\geq$ 2.6.2 (Update URL: `http://download.itemis.com/updates/releases`)
- Xtend $\geq$ 2.6.2 (Update URL: `http://download.itemis.com/updates/releases`)

After the environment is set up, check out and import all plugins from the following repositories (https://svnserver.informatik.kit.edu/i43/svn has to be prepended):

- Checkstyle (code/Development/de.uka.ipd.sdq.codeconventions/)
- Vitruvius (code/Vitruvius/trunk/)
- MonitoredEditor Fork (stud/StephanSeifermann/dev/vendor/)
- CST Synchronizer (stud/StephanSeifermann/dev/trunk/)

After all plug-ins have been imported into the workspace, there might be some compilation errors due to missing dependencies or broken code in the Vitruvius projects. In that case, just close all open projects and reopen the following plug-in:

- edu.kit.ipd.sdq.seifermann.thesis.initializer

All required dependencies are reopened automatically and the erroneous plug-ins are ignored.

4 Usage

Start an Eclipse instance with all workspace plug-ins and import a project that includes Java and JML files in the launched Eclipse instance. The JML files have to be located in a folder called *specs* in the project root. You can use one of the following projects, for example:

- edu.kit.ipd.sdq.seifermann.thesis.syncexamples
- edu.kit.ipd.sdq.seifermann.casestudies.javacard

Now, activate the synchronization system via *JML - Activate Synchronization* in the main menu and select the wanted project. The activation can take some time depending on the size of the project.

In the runtime Eclipse instance, you can now change the Java code. The changes are propagated to the corresponding JML file. Please note that there are limitations at the moment. a) Because of a bug in the VITRUVIUS framework (update of correspondences) you might only be able to do a single change without restarting the synchronization system. So, if a change is not propagated, then revert it, save the file and restart the system. b) It is crucial that all files are saved before triggering a refactoring. c) Not all changes are fully processed. For instance, a type change is propagated but the effects of it are not checked. This can lead to syntax errors.

If you want to change JML elements, you have to open a JML file and select a change from the *JML* main menu entry. This is necessary because we do not support change detection for JML files at the moment.

5 Overview

5.1 Plug-ins

The synchronization system consists of multiple plug-ins. The most relevant ones and their relationships are illustrated in Figure 1. The following paragraphs describe the plug-ins and their features in short.

Initializer Starts the synchronization system and provides the menu items for doing so.

MonitoredEditorBase Provides base interfaces and registries for monitored editors and projects.

ChangeMonitoringEditor It is located in the *vendor* branch and a forked version of the MonitoredEditor implementation of Dominik Messinger [1]. This collection of plug-ins detects changes in Java code, classifies them into more high level logical changes and propagates them to a synchronization engine. The changes are represented as change models.

Java/JML MonitoredEditor The monitored editors provide changes and the model URIs for the synchronization engine. The JML editor has no support for change detection but only for change injection. The Java editor extends the previously mentioned MonitoredEditor and redirects the changes. The model URIs are determined by looking for files with a specified extension.

CST Synchronisation The plug-in contains all Java and JML specific information for the synchronization engine. It provides the data for initialization, which consists of a) meta-models for Java and JML, b) mappings between

3

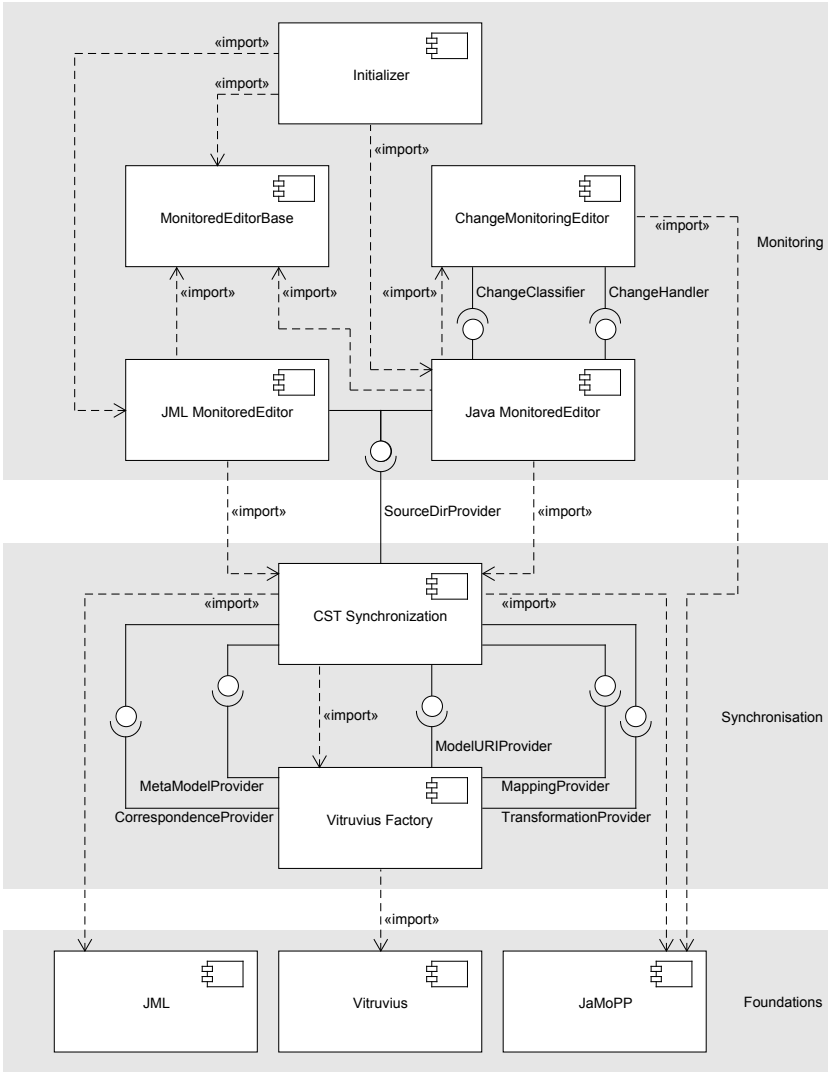


Figure 1: Overview of the synchronization system structure.

these meta-models (Java maps to JML and vice versa), c) model URIs (URIs of all Java and JML files), d) correspondences (relations between elements of the models like a class *Subject* in Java corresponds to a class *Subject* in JML) and e) the transformations for the synchronization.

Vitruvius Factory The factory performs the initialization of the framework. It uses the data provided by the previous plug-in. The framework is not encapsulated completely because especially data types are used all over the layers and plug-ins.

Vitruvius The framework provides the infrastructure and data types for performing synchronizations.

JML Four plug-ins make up the diagrammed JML plug-in. A part of them is generated by the Xtext framework, which is used to create a model parser and printer for the JML language. The grammar is a mix of a non official Java Xtext implementation and the Xbase grammar for expressions with some modifications to include JML constructs.

JaMoPP A model printer and parser for Java code.

5.2 Meta-Model Handling

The package *edu.kit.ipd.sdq.seifermann.thesis.cst.metamodels* contains classes that are necessary for the correct meta-model handling. The provider classes are used by the Vitruvius wrapper to initialize the framework with the meta-models. The namespace URIs, file extensions and a TUID calculator and resolver are passed. The latter is used to reference model elements. This is necessary because JaMoPP and the JML implementation do not update existing models after changes but regenerate them completely. Additionally, there is no identifier attribute.

The calculators use regular attributes of the model elements to calculate a virtual identifier. The identifier represents the containment hierarchy of the element. So, the individual identifiers, which may not be unique itself, are combined with the identifiers of its parents. This must lead to a unique identifier. The resolution process uses the identifier segments to walk through the containment hierarchy.

5.3 Correspondences

The correspondences describe relationships between model elements. Because Java and JML have a similar structure, finding the correspondences is straight forward. Two corresponding compilation units are found by comparing their file paths. For the matched compilation units, all of their elements are compared and correspondences are added by the *Java2JMLCorrespondenceAdder*.

5

5.4 Transformations

The *CSSynchronizer* executes the transformation for a change received from the Vitruvius framework. It can handle atomic and composite changes. The former are processed by a transformation object that matches the changed element. For instance, a rename operation on a Java method would lead to a call of a *JavaMethodTransformations* object. Each transformation class has methods that can handle a single change type.

The composite changes are preprocessed by a refiner that can match its pattern. The refiner creates a set of atomic changes that are handled like regular atomic changes or a transformation object that can be executed. If no refiner matches the change, the composite change is divided into its atomic changes and they are processed separately.

The transformations have two responsibilities: First, they have to apply the change on the target model. Second, they have to update the correspondences after a change. This is necessary after creating and deleting elements as well as after changing properties that have an influence on the identifier.

Refactorings make use of a so called *shadow copy* to apply the change. The shadow copy *ShadowCopyImpl* creates a copy of all Java models. Afterwards, all JML specifications and elements are transformed into dummy statements and inserted into Java. For instance, during a rename refactoring, all proxy references are resolved and the rename operation is carried out on the shadow copy element. Because of the references, the changed name is propagated to all elements. The dummy statements are converted back to JML and replace these elements. Because any operation is carried out on copies, there are no side-effects on the regular Java models.

References

[1] Messinger, D.: Incremental Code Architecture Consistency Support through Change Monitoring and Intent Clarification. Master's Thesis. IPD, Karlsruhe Institute of Technology, Germany. (2014)

List of Tables

List of Figures