Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2015,4
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Realizing Change-Driven Consistency for Component Code, Architectural Models, and Contracts in Vitruvius

Max E. Kramer, Michael Langhammer, Dominik Messinger,
Stephan Seifermann, Erik Burger

2015

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Realizing Change-Driven Consistency for Component Code, Architectural Models, and Contracts in Vitruvius

Max E. Kramer
Karlsruhe Institute of Technology
Karlsruhe, Germany
max.e.kramer@kit.edu

Michael Langhammer
Karlsruhe Institute of Technology
Karlsruhe, Germany
michael.langhammer@kit.edu

Dominik Messinger
Microsoft Canada Development Centre
Vancouver, Canada
domessin@microsoft.com

Stephan Seifermann
FZI – Research Center for Information Technology
Karlsruhe, Germany
seifermann@fzi.de

Erik Burger
Karlsruhe Institute of Technology
Karlsruhe, Germany
burger@kit.edu

ABSTRACT

During the development of component-based software systems, it is often impractical or even impossible to include all development information into the source code. Instead, specialized languages are used to describe components and systems on different levels of abstraction or from different viewpoints: Component-based architecture models and contracts, for example, can be used to describe the system on a high level of abstraction, and to formally specify component constraints. Since models, contracts, and code contain redundant information, inconsistencies can occur if they are modified independently. Keeping this information consistent manually can require considerable effort, and can lead to costly errors, for example, when security-relevant components are verified against inconsistent contracts. In this technical report, we present details on realizing an approach for keeping component-based architecture models and contracts specified in the Java Modeling Language (JML) consistent with Java source code. We use change-driven incremental transformations and the Vitruvius framework to automate the consistency preservation where this is possible. Using two case studies, we demonstrate how to detect and propagate changes and refactoring operations to keep models and contracts consistent with the source code.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.2.11 [**Software Architectures**]: Languages

Keywords

Model-Driven Engineering, Formal Specification, Co-Evolution

1. INTRODUCTION AND MOTIVATION

Component-based software systems are often designed and realized using heterogeneous development artefacts, because it is inefficient to perform all development tasks in general-purpose programming languages. To obtain a more abstract representation, Architecture Description Languages (ADLs), for example, can be used to model components and their relationships while omitting implementation details. If architecture models or natural language specifications are considered too informal or too inefficient for a precise description of component interfaces, formal languages, such as the Java Modeling Language (JML) can be used to specify contracts. These special languages can be helpful to design and maintain component-based software, but they introduce redundancy. Names and parameters of services that are provided by a component, for example, may appear in the code, the architecture model, and the contracts. Such redundant information becomes inconsistent as soon as the code, model, or contracts are changed in isolation during the development and maintenance of components and systems. If consistency is restored manually after a change, modifications have to be performed in each of the three artefacts, requiring manual effort and possibly still leading to costly inconsistencies that are only discovered in later phases of the development process. For security-relevant components that are verified against contracts, such inconsistencies can lead to incorrect verifications and therefore insecure systems and are therefore intolerable.

In this technical report, we present details on a semi-automated approach and a tool for component-based software systems development to keep architecture models, formal component contracts, and source code consistent. We describe change detection and propagation by change-driven incremental transformations. We have used the Vitruvius framework [12] to implement the change detection and consistency approach in a research prototype[1]. Our prototype processes architectural models that use the Palladio Component Model (PCM) [3], contracts defined in JML, and Java source code but it can be adapted to other ADLs and specification languages.

With two case studies, we have evaluated whether all kinds of code changes can be detected, and whether contracts and

[1] sdqweb.ipd.kit.edu/wiki/Vitruvius/Development

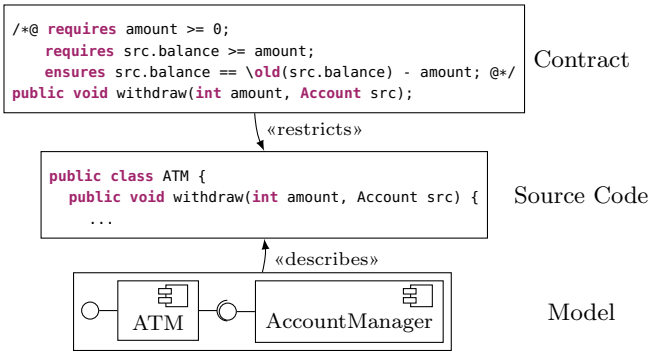

Figure 1: Extracts from the contract, source code, and model for an example banking system

models are kept consistent accordingly. Our evaluation shows that it is possible to keep models, contracts, and the implementation of components automatically consistent in most of the change and refactoring scenarios that we have checked with our evaluation case studies.

The remainder of this report is structured as follows: After foundations (section 2) and an overview (section 3), we discuss change monitoring (section 4). Then, we present our concept, implementation, and evaluation for code and contract consistency (section 5), followed by related work (section 6), and a conclusion with a discussion of future work (section 7). This technical report is an extension of a conference paper [13], and all code and tests are freely accessible.[1]

2. BACKGROUND AND FOUNDATIONS

In this chapter, we provide background information that is fundamental for our approach and tool.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [22] is a software development paradigm that strives for the use of modelling languages with precise semantics for all development artifacts. In order to be used in automated transformations and generation steps, all model instances have to conform to a so called metamodel, which describes static and dynamic constraints of a modelling language. A metamodel defines the abstract syntax of a modelling language and plays a role similar to an abstract syntax tree in a programming language: The exact graphical or textual representation of model instances, also called concrete syntax, is separated from it to ease transformations. The semantics of model instances in MDE can be explicitly defined in a formal way, or implicitly defined by the transformations into other models or programming languages. New modelling languages can be created by defining a new metamodel that itself conforms to a meta-modelling language or meta-metamodel.

The Eclipse Modeling Framework (EMF) defines the metamodelling language Ecore, based on the Meta-Object Facility (MOF) ISO 19508 standard by the Object Management Group (OMG). It is built on top of the popular Eclipse IDE and used in many open-source projects. The Java Model Printer and Parser (JaMoPP) [10], for example, defines an Ecore-based metamodel for Java programs. It can be used to parse Java source code as a regular model instance so that it can be analyzed and transformed like every other model before it is printed back to source again.

2.2 Palladio Component Model

The *Palladio Component Model* [21] is the Ecore-based metamodel of the component-based ADL that is used in the Palladio Bench[2]. It features reusable components, which provide and require services at interfaces, which are both defined in a system-independent component repository. A specific system is modelled by assembling instances of these components in so-called assembly contexts. These assembly contexts are linked using assembly connectors and are also used in composite components, which delegate services internally using delegation connectors.

2.3 Java Modeling Language

The *Java Modeling Language (JML)* [16] is a behavioral interface specification language for Java. It can be used to define contracts for interfaces and classes, which are often just called specifications. JML contracts can be defined in usual Java source files, or in a separate JML file that repeats all Java declarations of the specified interface or class. The contracts are noted inside Java comments directly before the declaration of the corresponding Java element.

JML contracts consist of statements and modifiers. A statement starts with a JML keyword and is followed by a regular or extended Java expression. *Extended* means that additional operators and keywords can be used. For instance, `\old(expr)` provides the expression result from the time before executing the method. Essential JML statements are preconditions, postconditions, and invariants. These statements start with the keywords `requires`, `ensures`, and `invariant`. Modifiers also specify elements. For instance, the `pure` modifier marks a method side-effect free, which is required when using the method inside contract specification statements. An example for a JML contract is shown in the upper box of Figure 1: It defines two preconditions and a postcondition for a withdrawal operation of an Automatic Teller Machine (ATM).

JML contract specifications are divided into cases that must hold in specific contexts. A method can have exactly one lightweight or at least one heavyweight specification case. Lightweight specification cases have to hold in all contexts. Heavyweight specification cases only have to hold if the precondition holds.

3. FRAMEWORK OVERVIEW

Our monitoring and consistency approach for component-based code, architecture models, and contracts is part of our work on the generic Vitruvius framework for multi-view modeling [12]. The Vitruvius framework is based on the central idea of Orthographic Software Modeling [2]: all information of a software system is represented in a single underlying model and can be accessed solely by views. It tries to combine the advantages of projective and synthetic approaches as defined by the ISO 42010 standard by providing a method for constructing and maintaining a modular, Virtual Single Underlying Model (VSUM): The VSUM consists of individual models for different modeling languages in order to support existing languages and tools. The virtual model instance is dynamically managed by the framework and restricted by the metamodels that are added to a so-

[2]http://www.palladio-simulator.com

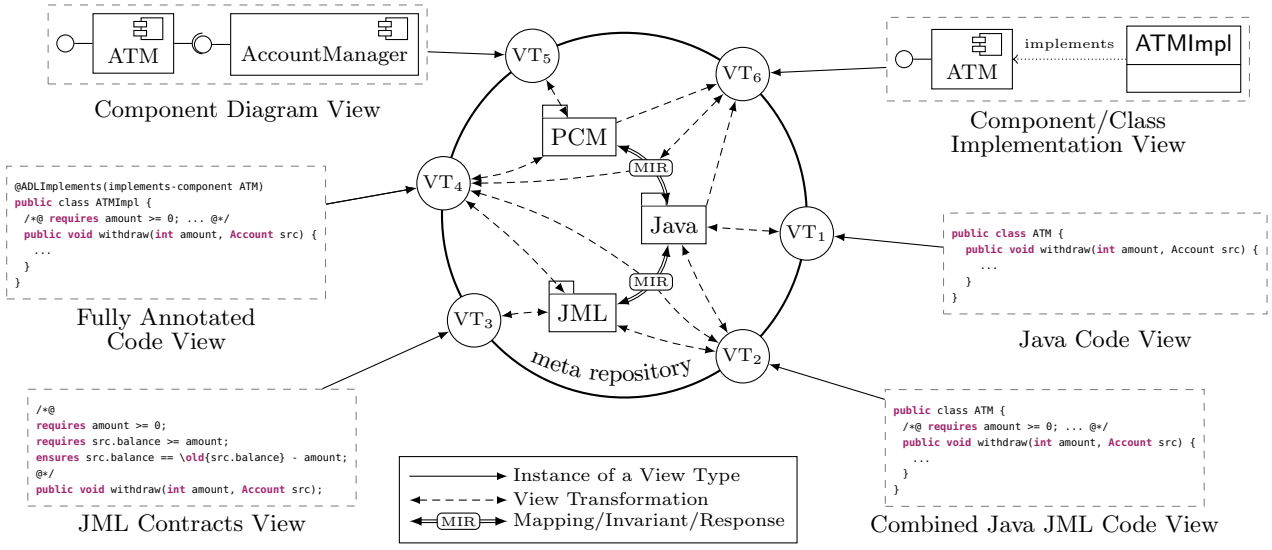

Figure 2: Meta repository and possible views for component-based software engineering with contracts

called meta repository. The relations between the individual metamodels of the meta repository are encapsulated and expressed using a domain-specific language for mappings, invariants, and responses (MIR). Therefore, views can focus on relevant elements and relations and do not have to consider all internal details of all modeling languages. This decouples the inner modular structure of the VSUM from its representation in the views. To make this possible, all views have to report all changes to the framework so that it can propagate them within the VSUM to sustain consistency.

The prototype described and evaluated in this paper uses the concepts and infrastructure of the Vitruvius framework to implement change monitoring in external editors and internal consistency using change-driven model transformations. The VSUM consists of models representing the Java code, architecture models of the PCM, and JML contracts. To this end, the metamodels for Java, PCM, and JML are added to the meta repository as shown in Figure 2. Currently, we only support the three well-known standard views for Java (VT_1), JML (VT_3), and PCM (VT_5). In future work, we will develop and evaluate views that combine information of several models: a combined view for Java source code and JML (VT_2) that can display contracts of component services together with their implementation; a fully annotated code view (VT_4) to display information of architectural models and JML contracts as annotations in the Java source code; and a component/class implementation view (VT_6) to display which classes in the Java code implement which components of the architectural models. Our prototype is based on EMF and processes Java source code, PCM models, and JML contracts as instances of metamodels that are defined in Ecore.

4. ARCHITECTURAL MODELS AND CODE

In this section, we describe how Vitruvius can be used to keep a component-based architectural model consistent with its source code. The goal of keeping architecture model and code consistent during the development of a software system is to avoid architecture drift, and to help developers to find and reduce architecture erosion [19]. Furthermore, an up-to-date architecture model has many advantages: it is, for example, possible to make more accurate decisions for the further evolution of the software system. In our scenario, the Palladio Component Model also allows architects to make performance predictions for the system under development. To apply the Vitruvius process to architecture models and code, we have to specify

a) an Ecore-based architecture model, and an Ecore-based representation of a general purpose language;

b) bidirectional mapping rules for code and architecture;

c) monitors that report atomic architecture and code changes

d) a method to clarify the intent of developers and architects in the case that ambiguous changes are made.

For our prototype, we chose the EMF-based PCM as the architectural model because we plan to use the PCMs performance prediction capabilities in future work. However, it would also be possible to use other component-based architecture models, for example, UML composite diagrams, as the architecture model. As an EMF representation of the source code, we use JaMoPP (Java Model Printer and Parser) [10]. JaMoPP allows us to treat Java source code like an EMF model. For instance, it is possible to execute model transformations on JaMoPP instances and to print the java source code for the transformed model. For our prototype, we defined the following mapping rules (cf. [14]) between PCM elements and code Elements: A PCM repository maps to 1) a main package that represents the repository, 2) a contracts package in the main package that will contain all interfaces, and 3) a data types package in the main package that contains all data types. Every PCM component maps to a package within the main package and a public component realization class within the component package. Every PCM interface maps to a Java Interface in the contracts package. PCM Signatures with its parameters and return types map to Java Methods with corresponding parameters and return types. A PCM Datatype maps to a class within the datatype

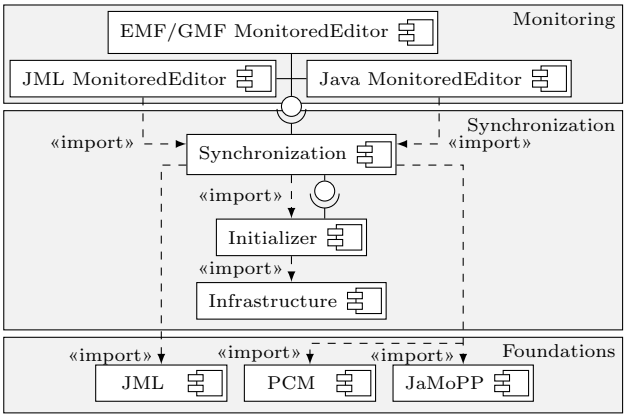

Figure 3: Simplified architecture of the implementation for synchronization between code and contracts and code and architecture.

package that contains getters and setters for the inner types of the datatype. A required role maps to a member typed with the required interface and setter for required interface in the main class of the requiring component. For every PCM provided role, the main class of the providing component implements the provided interface. These mapping rules are a refined version of the mapping rules that we have presented in [14].

Using Vitruvius, domain experts, e.g., architects, can create specific mapping rules for their projects. One of our ongoing research efforts is to complete the development of a domain-specific language for bidirectional mapping rules, which can be used inside the Vitruvius framework.

In order to monitor code changes, we implemented a monitor that observes the Eclipse Java code editor (4.1.1). To monitor changes on components and interfaces, we implemented a monitor that is able to observe changes in arbitrary EMF and GMF models that can be used for all graphical and tree-based PCM editors (4.1.2).

4.1 Monitoring and Propagating Changes

In this section, we describe the implemented monitors for code and architecture that are part of the prototype that can be used to keep architecture and code consistent. Two monitors have been implemented: *JavaMonitoredEditor* and *EMF/GMF MonitoredEditor*. Their interaction with the *Synchronization* components is depicted in Figure 3.

4.1.1 Code Changes

In order to keep the architecture consistent with architectural relevant source code changes, we implemented a mechanism that notifies the Vitruvius framework as soon as a change in code was made. Since we want to keep the advantages of a powerful code editor and also leverage developers' experience with familiar environments, we decided to implement our code monitor as an extension of the standard Eclipse Java code editor. We use the notification mechanism of the Eclipse AST (Abstract Syntax Tree) to get notified when a change occurs in the code. This mechanism notifies our code monitor every time the Eclipse change reconciliation mechanism is executed and the code is compiled. We use this notification to build semantic code changes. A semantic code change can be a simple *rename method*, but we also support more complex changes, such as *move method*. Based on the semantic changes, we build instances of a change metamodel and pass them to consistency preservation transformations, which implement the mapping rules described above.

If developers or architects make ambiguous changes, i.e., changes that cannot be propagated automatically to code or architecture, we have to clarify their intent. The intent clarification mechanism lets the transformations that keep the architecture and code consistent interact directly with the developers or architects, and displays a dialog to clarify the intent [15]. In future work, we plan to implement more interaction options, e.g., postponing decisions, which are collected in a task list. Hence, processing of these tasks can be delayed. This has the advantage that developers who are unfamiliar with the architecture do not have to make the decision by themselves. Instead, they can ask the system architect to get involved in the decision process. Another advantage is that developer interruption is limited due to the possibility to postpone changes.

Code change propagation can range from trivial updates to more sophisticated transformation decisions. Consider this simple example: A developer changes the name of an architecturally relevant interface method. According to the mapping rules defined above, we can automatically rename the name of the signature of a PCM interface.

However, because arbitrary changes can be made in code, we have to face the problem that developers can make ambiguous changes that cannot be propagated to the architecture without getting more information from the developers or architects. For this scenario, consider the following example: A software developer creates a new package as well as a new class within this package. This would lead to the two changes *package created* and *class created*. However, using the above mentioned mapping rules, it is unclear whether the new package should become a component at the architectural level. If the package should be a component, it is also unclear whether the newly created class should be the component-realization class of the component. Therefore, we have to ask the developer whether his or her intent is to create a new component on architecture level or not. If the answer is yes, we can create the component on architectural level and ask whether the class is the component-realization class of the component. If the second answer again is yes, we can use the class as the component-realization class. If the answer is no, we automatically create a new component-realization class for this component. The intent clarification mechanism is realized within the mapping transformations and therefore can be defined project-specific.

Figure 4 illustrates the basic workflow of change propagation from code to architecture. The first step is that developers edit the source code (1). After the code modification, the Monitored Editor is notified (2). The Code monitor generates and retrieves a JaMoPP Model (3), and submits the changes to the Change Synchronization component (4). The Change Synchronization component uses the elements from the correspondence instance and decides whether the change was an ambiguous (5a) or unambiguous change. The correspondence instance contains the information which elements from the JaMoPP model correspond to which elements from the PCM. If the change was an unambiguous change, the Palladio Component Model can be updated directly (6). If it was an ambiguous change, we have to clarify the intention of the developers. Therefore, the Monitored Editor is notified

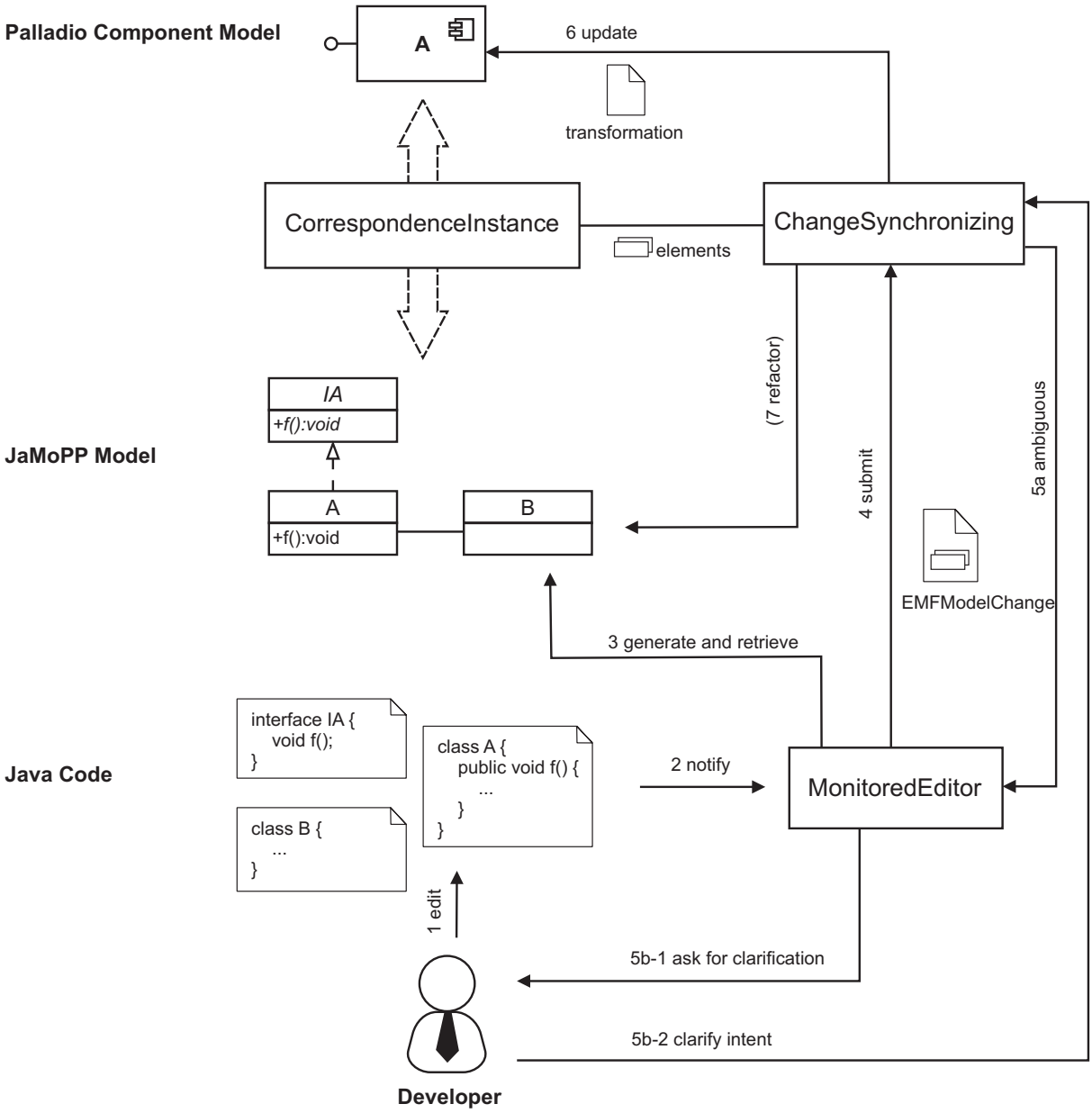

Figure 4: Illustration of the basic workflow of change propagation from code to architecture within the prototypical implementation.

(5a), and the developers are asked for intent clarification (5b1). After the intent is clarified, the Change Synchronization is notified again (5b2). Based on the intent clarification, the Change Synchronization component can now update the PCM accordingly (6). The Change Synchronization component can also refactor the JaMoPP model respectively the source code(7).

4.1.2 Architecture Model Changes

In this section, we describe how changes made in an architectural model are propagated to the source code. As mentioned above, we have implemented a change monitor to track changes in all EMF- or GMF-based PCM editors. We use the change recorder mechanism of EMF to receive change notifications. After a save action, all changes are propagated in the order they were conducted. During the propagation, the following steps are executed for each monitor change:

1. find the correct transformation for the change
2. execute the transformation
3. create/update/delete the correspondences
4. save/delete the changed models

Consider the following example: An architect adds a new component in the PCM and changes the component from the default name of a new component to *ATM*. Afterwards he saves the editor. The change monitor records two changes. The first change is that a new component with the (default) name *aName* has been created. The second change is that the component has been renamed to *ATM*. These changes are propagated as soon as the save is triggered. The framework automatically executes the transformations for creating a new component and gives it the name *aName*. Using the mapping rules explained in section 4, it will create a new package named *aname* in the package that corresponds to the repository. Also a new Java class named *aNameImpl* will be created. For the second change the transformation for renaming a component will be executed. Hence, the package as well as the class are renamed to *atm* respectively *ATMImpl*.

4.2 Code Monitoring Performance

Monitoring of code changes is a background process in the IDE and therefore does not directly block the developer's flow of producing and altering source code. Ambiguous changes, however, lead to intent clarification requests. Those requests have to appear before a next change occurs. Therefore, it is crucial that change monitoring is time-efficient. We measured performance of our change monitor in terms of time consumption for the creation of change objects, i.e. in-memory representations of the detected semantic change. Our monitoring mechanism has two stages: First, the detection and classification of a change in the AST and, second, the conversion of the AST change object into a description of modifications of JaMoPP code models. Time performance evaluation was conducted by applying three different types of changes to source files of different LLOC sizes, ranging from small files with limited functionality to very large auto-generated files. We performed and observed code changes on the open-source Java code base of the commercially used *Apache Hadoop Distributed File System (HDFS)*[3].

[3]hadoop.apache.org

LLOC	Rename Method	Replace Method Modifier	Add or Remove Field
28	57 (0.94)	50 (0.34)	57 (0.33)
350	292 (0.22)	278 (0.32)	324 (0.33)
1045	832 (0.09)	856 (0.16)	865 (0.10)
2050	1,776 (0.17)	1,676 (0.16)	1,954 (0.16)
15812	14,683 (0.09)	14,334 (0.09)	14,880 (0.10)

Table 1: Average total monitoring time in ms for edit operations on different-sized HDFS source files with the coefficient of variation (in parentheses)

The following three changes were applied to the source code: 1) Rename a method, 2) replace a method modifier, 3) add or remove a field. Every change was repeated 100 times on each source file, except for the largest source file where the number of repetitions was 25. The experiments were conducted on a 3.40 GHz quad-core desktop PC with 8 GB RAM running a 64-bit Eclipse 3.5 on a 64-bit Windows 7 system.

Table 1 shows the average total time consumption per file and change scenario. In every experiment, the first change observation took significantly longer than the succeeding changes, thus we consider the first measurement value an outlier. Consequently, the cells in Table 1 contain the average of the second to 100th – or 25th – measurements. The average is given as the arithmetic mean, and the table's time unit is milliseconds. The value in parentheses gives the coefficient of variation, i.e., the standard deviation divided by the average.

Our performance evaluation indicates a linear increase of the monitor's time overhead with the LLOC size of source code files. It also shows that the time increase is independent from the change that is conducted (see Figure 5). The sample correlation coefficient between the average time consumption in milliseconds and LLOC size is 0.9995. In addition, our measurements showed that the creation of AST change objects required at maximum only 4% of the combined time consumption for AST and Vitruvius change creation. Therefore, the creation of JaMoPP-based change objects determines our monitor's time performance. The explanation of the observed linear dependency lies in one implementation detail: After every change, the affected compilation unit is entirely parsed into a JaMoPP model. We do not cache JaMoPP models and instead rely on parsing on-demand due to JaMoPP's large memory footprint. Although our monitor needs less than one

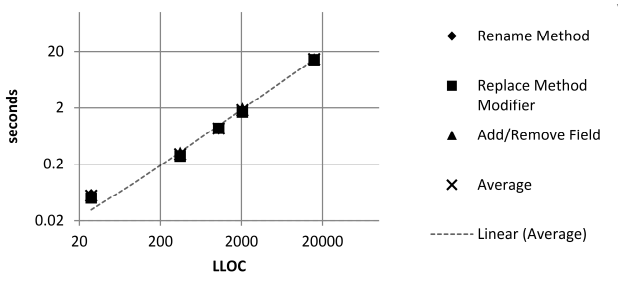

Figure 5: Average time overhead per change observation [ms]. Both axes are logarithmic to base 10.

second for 1045 logical lines of code, a single change may have side-effects on large, generated files, and thus result in large delays. The improvement of our code change monitor's performance is part of our future work. We plan to build partial JaMoPP models that only contain change-affected code elements, and thus decouple the monitor's processing work from the compilation unit sizes.

5. COMPONENT CONTRACTS AND CODE

Our overall objective is to support component developers and system architects by keeping component contracts written in JML and component implementations written in Java consistent after changes. We achieve this with a consistency concept in which the overlapping parts of code and contracts as well as reactions on changes affecting them are defined. The overlap is described by relations between relevant elements and by constraints for these relations. Change reactions restore these constraints after a change has occured. We do not limit changes to refactorings, but consider any possible change performed by a developer. In the evaluation of our consistency concept, we show that the implemented change reactions restore consistency after changes.We covered identifiers, visibility, types, `pure`, `helper`, `nullable`, default behaviors regarding null, `assignable`, generic and exception specifications. In Table 2 we provide a short overview of the covered JML constructs.

5.1 Contract Consistency Concept

We have developed a consistency concept for code and contracts, which defines the overlapping parts between these artifacts, and describes what reactions are needed for which changes. We make the following assumptions to focus our efforts on interesting and common situations:

a) We cover JML elements of language levels 0 and 1, because they are supported by most JML tools [17, chap. 2.9].

b) We restrict ourselves to constructs that are properly supported by OpenJML, which is the JML compiler used in the evaluation.

c) We do not cover change reactions that are already realized in IDEs, such as updating callers when renaming a method, but we update the contracts.

d) We assume that every line of code is reachable.

e) We only cover structural code element changes that are performed in the code and not in the JML files.

We refer to *directions* in the description of the following change reactions. A direction $A \rightarrow B$ means that a change occurred in artifact A and has an effect on artifact B. We consider the artifacts code C and specifications (contracts) S. Directions that are not mentioned in an element's concept have not to be considered because they are already handled by the IDE or they are not relevant.

We divided our consistency concept into three parts based on the affected JML elements. For every part, we define the relevant overlap, constraints, and change reactions.

5.1.1 Basic Syntax Elements

We call identifiers, visibility modifiers, and types *Basic Syntax Elements*, because they form the basis for many expressions and statements and can appear in a large number of syntactical legal locations.

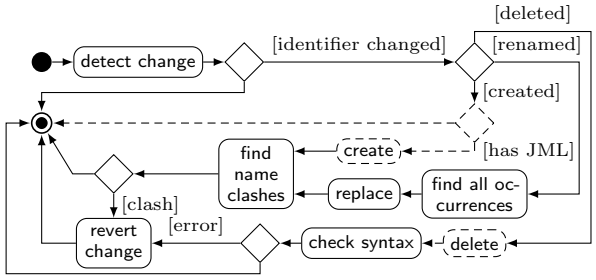

Figure 6: Reaction on an identifier change in the direction specification to specification and code to specification. The dashed elements are only relevant for the latter.

Identifiers are used in Java and JML to reference elements within code and specifications and in between. Therefore, identifiers and their uses have to be kept consistent across both artifacts: Figure 6 illustrates the necessary reactions for $C \rightarrow S$ and $S \rightarrow S$. We have to consider the latter, because there is no refactoring support for JML to our knowledge. Identifier changes have to be propagated to the other artifacts by replacing or removing their occurrences, e.g., using static code analysis and refactorings. If an identifer clash or unresolved reference is detected, we simply revert the identifier change. The handling for $S \rightarrow C$ is identical to $C \rightarrow S$ but we consider specification-only methods and fields only. We do not consider other elements because structural changes have to be performed in the code as noted in the restrictions mentioned above.

Visibility modifiers for structural elements, such as methods or fields, have to be the same in Java and JML. The accessibility rules for JML are based on the Java rules. A referenced element has to be at least as visible as its specification. For invariants and history constraints, there is a special requirement: They have to be at least as visible as the referenced fields. After a visibility change in Java, we have to copy the new visibility modifier to the JML contract and execute further reactions based on the change: If the visibility has been decreased, we have to enforce the old visibility with specification-only modifiers to keep the element accessible from existing specifications. Such modifiers override the regular visibility in specifications but do not affect the visibility in code. If the visibility of a field has been increased, we have

Element	Coverage		Evaluation	
Identifier	C	P	1	2
Visibility	C			
Type	C			
pure	C	P	1	
helper	C	P	1	
nullable	C			
null defaults	C			
assignable	C			
Generic Specifications	C			
Exception Specifications	C			

Table 2: Overview of coverage of JML constructs in consistency concept *C*, prototypical implementation *P*, and evaluation test suits.

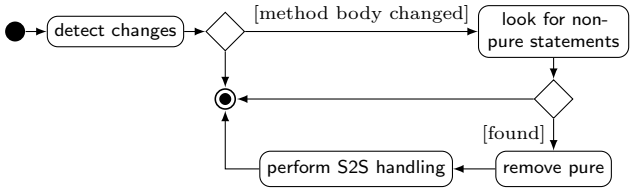

Figure 7: Reaction on an method change in the direction code to specification with respect to `pure`.

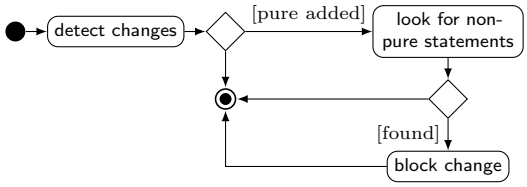

Figure 8: Reaction on a `pure` change in the direction specification to code.

to increase the visibility of invariants and history constraints that reference the field to meet the special visibility requirement mentioned above. After a visibility change in JML, we have to block the change if the visibility has been decreased in such a way that there are syntax errors.

JML elements have to be type-checked in accordance with the Java type check: The type check of the compiler has to succeed in each artifact and corresponding elements have to have the same types. If a return type or a field type is replaced with a subtype of the original type, the change can be processed automatically. Any other change can modify the program behavior and has to be processed manually.

5.1.2 Specification-Only Elements

Elements that are only definable and usable in specifications are called *Specification-Only Elements*. We cover specification-only fields, methods, and imports.

Specification-only fields can be separated into model fields and ghost fields. The value of a model field is determined by a typed expression, which is evaluated when accessing it. Assignments are not possible. The type of the field has to match the type of the expression. In contrast, the value of a ghost field is determined by explicit assignments only. For both field types the constraints for *Basic Syntax Elements* have to hold in addition. Hence, the corresponding change reactions have to be applied. Changes that affect the semantics have to be processed manually. Such changes can be type changes to non-subtypes or implementation changes that affect the assignments to ghost fields.

Model methods have the same syntax as Java methods and can be used in the same way inside specifications. Therefore, the same constraints have to hold. Model methods are treated like Java methods in some change handlings, such as the ones for `pure`, `assignable`, exception specifications, and `nullable`.

Model imports are equal to Java imports but only have effects on specifications. Every identifier that is only used in the specification has to be included into the compilation using a model import statement so that the specification can be compiled without syntax errors. After removing a regular import statement, we have to add a model import for the same identifier if a syntax error in the specification occurred. After adding an import statement for an identifier that has already been imported by a model import, we have to remove the model import. Additionally, we have to compile the specification after every change of the specification to ensure that no additional import is required. If there is a syntax error, we can either try to find the correct import or ask the user to provide it.

5.1.3 Method and Type Specifications

Method or Type Specifications are statements and modifiers that can be used to specify a method or a type. In our context, the modifiers `pure`, `helper`, `nullable`, and `non_null` are relevant. Relevant statements are default behaviors regarding `null`, `assignable`, generic behavior specifications, and exception specifications.

The `pure` modifier marks methods as free of side-effects. Therefore, the modifier may only be added to methods that neither contain assignments to fields nor calls to methods with side-effects. Methods used in specifications have to be marked `pure` because specifications have to be evaluated without side-effects. Figure 8 shows the handling for $S \rightarrow C$: If a user adds the modifier to a method that has side-effects, we have to block the change. $C \rightarrow S$ is shown in Figure 7: If a user changes the body of a pure method so that is has side-effects, we have to remove the modifier. Removing the modifier also has effects on $S \rightarrow S$. As can be seen from Figure 9, if a method that lost its pure status is used in specifications, we have to block the change. The same procedure has to be performed for all methods that call this changed method, because they also lose the pure status. The whole change has to be blocked if at least one of the methods that loses its pure status is used in specifications.

The `helper` modifier suppresses the invariant and constraint check when entering and leaving the annotated method. Otherwise, the evaluation of such specifications can lead to infinite-loops depending on the implementation. Figure 9 shows a conservative handling of this possiblity: As a precaution, we have to add the `helper` modifier to a method as soon as it is mentioned in an invariant or a constraint. The method cannot break these specifications anymore because it has to be marked pure by the above change reactions. We can remove `helper` if the corresponding method call has been removed from the expression.

Fields, parameters, and methods can be annotated with the `nullable` and `non_null` modifiers, to specify whether `null` can be assigned or returned. The implementation has to conform to the specified behavior, which can be checked with verification techniques. For undecidable situations caused by the halting problem, assistance from the user is necessary, however. If the implementation does not conform to the specification, we have to clarify the intention of the user. If the user did not intend to make an element `nullable` or `non_null`, we have to block the change. Otherwise, we must add the modifier to the element and keep on checking where the modifier has to be added until the user does not confirm additions anymore or until no errors remain. We have to apply the same reaction after changing the modifier in the specification explicitly instead of changing it implicitly via the code.

If neither `nullable` nor `non_null` is explicitly specified, the implicit default is `non_null`. This default can be changed to `nullable` with a statement. After such a change, there are two options: If the user did not intend to change the semantics

of the specification, we have to add the modifier that is the opposite of the new default to all elements without modifier. If the user did intend to change the semantics, the code has to be fixed manually. We can perform a verification of the system and suggest changes, which can be used by the user during manual fixing. In any case, we have to clarify the user's intention.

Exception specifications define throwable exceptions as well as preconditions for throwing them and postconditions that must hold afterwards. A method with an exception specification has to throw only exceptions of the specified types. Changes to the preconditions and postconditions of throws specifications are covered by the change reactions for *Generic behavior specifications*. After removing the last `throw` statement for an exception, we have to remove the exception type and its conditions from the specification. If an exception specification is removed, we have to ensure that it cannot be thrown anymore. Using static code analysis for this task only works for checked exceptions or for projects in that the source code of all transitively called methods is available. If a method has an exception specification that is defined using the `signals_only` keyword, then all unmentioned exception types are forbidden. Therefore, we have to add the exception type to such a specification if a `throw` statement is added. This approach only works for lightweight specifications because we cannot determine the preconditions for a single statement and match it to a heavyweight specification case reliably. Instead we suggest verifying the system and asking the user to fix the issues manually. After changing the exception specifications in JML, we also have to verify the system and block the change if an error occurred.

We call specification statements that can take arbitrary expressions *Generic behavior specifications*. This group includes preconditions, postconditions, and invariants. The syntactical aspects of changes to these elements are covered by the change reactions of the *Basic Syntax Elements*. To recover the semantics after a code change, we have to infer new contracts for the changed implementation and merge it with the existing ones. Suitable approaches for these two tasks are subject of current and future research activities [20] but to our knowledge no technique for specification merging is ready to be used. After a contract change, we have to apply approaches for automated fixing of programs, such as [24], which repairs the code with respect to its specification. Many of them, however, need too much time or cannot fix complex situations. Because the necessary techniques are not available yet, the user has to fix the issues manually after errors are detected.

A method can only change the locations that are mentioned in the `assignable` clause. Here, we only describe how to handle changes to object field locations: After a change of the method body, we have to calculate the difference between the specified set of assignments and the real assignments and modify the existing specification accordingly. Static code analysis can determine the assignments by looking for assigned fields and the `assignable` clause of called methods. After changing the clause for a changed method, we have to perform the same procedure recursively for all methods that call the changed method. For heavyweight specification cases, we suggest asking the user for the `assignable` clauses that shall be modified. With this handling for method body changes, the assignable fields are always derived from the implementation

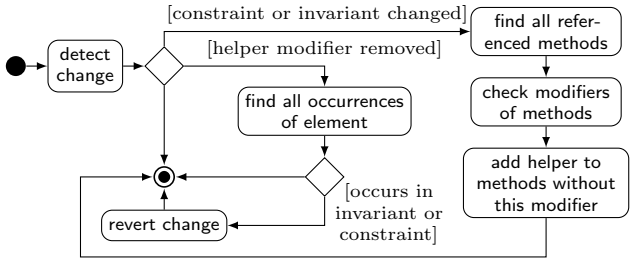

Figure 9: Reaction on a `helper` change in the direction specification to specification.

of the method. Therefore, we have to block changes of the assignable clause by the user.

5.2 Generalization of Concept

The previously introduced consistency concept is tailored to JML and Java, because they are mature languages, and are commonly used. Nevertheless, the concept is applicable to other concrete representations of code and contracts. As long as the programming language is object-oriented, most of the code handling does not have to be changed. In contrast to programming languages, contract specification languages are more heterogeneous with respect to features and concepts. Therefore, we focus on the generalization of our consistency concept with respect to the contract handling. In general, handlings become obsolete if elements do not exist or the handling is already done by an IDE.

Basic syntax elements include identifiers, visibility and types. The identifier handling remains the same as long as identifiers are used to reference elements. If the specification languages use the visibility modifiers of the programming language, these visibilities have to be adjusted instead of the not-existing specification visibilities. The type handlings remain the same for strongly typed languages.

Method and type specifications can be generalized as well. The concept of query methods is an integral part of contract specification even if no special keyword for them exists. Therefore, instead of relying on the modifier, we have to perform static code analysis to detect the query property. The `nullable` and `non_null` modifiers have to be mapped to equivalent statements such as `x != null`. The default values regarding `null` and `helper` cannot be mapped because they are implementation specific.

Generic behavior specifications include preconditions, postconditions and invariants - amongst others. The corresponding mappings can be applied as they are because other specification languages contain the same elements. The same holds true for languages that support frame rules. Otherwise these handlings can be ignored. The analysis of the exception handling remains the same but if the programming handling does not allow declaring exceptions, all exceptions have to be treated as runtime exceptions.

Specification-only elements are implementation specific and cannot be generalized.

5.3 Evaluation

The objective of our evaluation was to show that our developed concept can be realized to keep component specifications and implementations consistent after changes. So far, we implemented and evaluated all parts of our concept that have to

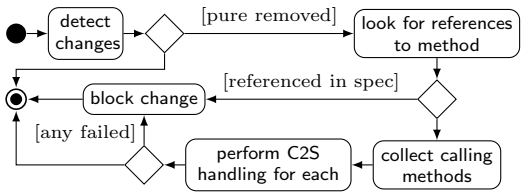

Figure 10: Reaction on a `pure` change in the direction specification to specification.

Property	Test Suite 1	Test Suite 2
Coverage	path	context
Type	system	system
Selection	manual	automatic
Syntax Check	yes	yes
Semantics Check	yes	no
Validation Data	JavaCard API	JavaCard API
# Tests	32	1085

Table 3: Overview of test suits used for evaluating the synchronization between code and contracts.

do with identifiers, pure and helper methods. We performed 32 manual tests and 1085 automated tests using a real world case study to check the semantic and syntactic correctness of our implementation and the concept behind it.

We embedded our concept in the Vitruvius framework using the layers of Figure 3. In the monitoring layer, the change detection mechanism for Java (section 4) reports code changes. Currently, JML changes are not detected automatically but injected. The synchronization layer contains the model-specific parts of the synchronization logic. Model transformations that realize our consistency concept are the most important part of *Synchronization*, which provides the initialization data for Vitruvius. To support all identifier modifications, we implemented transformations for the create, delete, and rename operations on fields, parameters and methods. To support `pure` and `helper`, we implemented transformations for adding and removing these modifiers and for changing method bodies and invariants. *Vitruvius Initializer* starts the framework. Together with *Vitruvius Infrastructure*, e.g., for storing and retrieving model correspondences, it is located in the synchronization layer. The model printers and parsers for Java and JML and the PCM metamodel are located in the foundations layer. We use the Java Model Printer and Parser (JaMoPP) to obtain models from Java source code and to serialize them again. For JML we developed a model-based printer and parser using Xtext. Appendix A contains a more detailed description of this model printer and parser.

With our evaluation, we have answered the following research question: Is our implemented consistency concept capable of keeping specifications consistent after changes in code or specifications for a specific case study? As evaluation case study we used a verified implementation of the JavaCard API [18], which can be used to program smart cards with Java. The project is often used as a case study for research on JML. Our implementation does not support all language features of JML that are used in the JavaCard API project. Therefore, we had to modify the original specification in the following way: We removed all specifications that contained `assignable`, `signals`, `signals_only`, interface constants, bit operations, casts and nested `\forall` statements. These modifications changed the semantics of the specifications. For our evaluation this is, however, no problem because we only examined differences between an initial code and specification state with the state after the synchronization. Only these differences and not the question whether the complete program fulfills the specification is relevant for us. Additionally, we replaced some syntactic sugar that we do not support in our prototype without modifying the specifications semantics. A more detailed description of these moficiations is given in Appendix B.

We created two test suites to evaluate our prototype and concept based on the case study. A short overview of them is given in Table 3. Both suites are implemented as automatic system tests, which makes the tests reproducible and evaluates the whole process reaching from change detection to change processing. Appendix B provides a more detailed description of the test environment. The two test suites are checked using two test oracles: The first oracle is a semantic check that compares the delta obtained after change detection and transformation with a manually checked reference delta. It is only used for the first test suite. The second oracle is a syntactic check of the specification after transformation using a JML compiler. It is used both for the first and the second test suite. For the first test suite, we manually created 32 tests to improve path coverage. For the second test suite, we used a selection algorithm to cover all possible contexts for the implemented rename operations. This led to 1085 tests for all fields, methods, and parameters. We only checked the syntax because manually creating and checking reference deltas for all tests would have required too much effort.

All tests of the first test suite succeeded: The syntax and semantics were correct after each transformation. We missed, however, some paths in the transformations because not all JML constructs that we support in our implementation are used in the case study. For instance, we could not test name clashes of Java methods with JML model methods because there are no such methods in the case study. Testing the remaining paths with another case study is part of our future work.

In the second test suite, 95 % of the 1085 tests succeeded with a correct syntax after the transformations. Additionally, we ensured that the delta between the original and changed state was not empty. 4.7 % of all tests in the second suite failed because of limitations of our current implementation. Contracts for interfaces, for example, are not yet implemented. Only 0.3 % of all tests in the second suite failed because of implementation errors. These errors, however, do not stem from our concept or the transformations but revealed an error in the static code analysis, which resolves references. Therefore, we consider the transformations and the underlying concept for rename operations correct in the tested contexts.

Altogether, the JavaCard API case study showed that the implemented and tested parts of our consistency concept are correct. We tested all paths of the implemented transformations that were reachable given the limitations of the case study. For rename operations, we tested all possible contexts within the case study. The representative case study covers most contexts, but not all of them, and the tests did not reveal errors of the concept for those contexts. Whether our concept

and implementation also works in the remaining possible contexts has to be investigated in further case studies.

6. RELATED WORK

In the following, we present related work on the coevolution of code and architectures or specifications.

6.1 Code Architecture Coevolution

De Silva and Balasubranamiam [5] classify approaches for controlling architecture erosion in a survey. The category *Architecture to implementation linkage* is closest to the VITRUVIUS approach. Many linkage concepts merge architecture and implementation information into a single entity, which contradicts our notion of separation of concerns. In addition, information merging becomes more complex and produces uncomfortably large artifacts when other models besides architecture specifications are considered. ArchJava [1] is an example of linkage through information merging. It introduces new language constructs, such as *component* and *port* to Java to include architectural information directly into the source code. Our goal, however, is to maintain separate artifacts and allow the use of existing editors and compilers.

There are also commercial solutions that support model and code synchronization. IBM Rational Rhapsody[4], for example, supports round-trip engineering of code and UML as well as other languages, e.g., SysML. Rhapsody's round-trip mechanism is based on code annotations that explicitly associate code elements with model elements. Consequently, it is possible to maintain unassociated elements, i.e., non-code-related model elements and non-model-related code elements. Enterprise Architect[5] allows forward and reverse engineering for initial code or model generation. Enterprise Architect's synchronization mechanism enables the user to add new code elements which are then added to the model and vice versa through partial generation in contrast to full file generation. Modification of existing elements is, however, not reflected on the respective other side. Borland Together[6] and UML Lab[6] support the round-trip engineering of UML class diagrams and source code. Both of the approaches use information in the source code to generate the class diagram. Hence, information that is not included in the source code cannot be displayed using these approaches.

Structure101[7] is an architecture development environment. It can be used to organize classes and packages into a hierarchical compositional model. Refactorings on the compositional model are transferred to the code base. Lattix[8] uses DSM (Design Structure Matrix) to manage software architecture. It calculates and displays dependencies within a software system and can be used to automatically calculate subsystems based on the dependencies. However, neither Structure101 nor Lattix, support a fully integrated co-evolution of component-based architecture and source code.

6.2 Code Specification Coevolution

Feldman [6] gives a high-level overview of code changes and their effects on contracts. 68 Fowler refactorings [8] are inspected and grouped into three categories: Refactorings that a) have only syntactic effects on contracts, b) require additional contracts, and c) may violate existing contracts. The specification effects of code refactorings are described and three specification refactorings are introduced. It is explained why some refactorings, such as *Extract Method*, cannot be processed automatically using theorem provers. Unfortunately, the full analysis and the change handling code are not made public.

Crepe is a tool based on these findings [9], which uses the Eclipse refactoring engine for Java to adjust contracts. It parses contracts defined in JavaDoc comments as Java code to respond to syntactical refactorings, such as renamings. JML contracts cannot be processed because they are defined in regular comments. Crepe simplifies existing contracts, e.g., for superclasses, using Mathematica and creates new contracts with Discern [7], for example, when a new method is added. There is, however, no implementation available.

Feldman et al. [7] presented an approach for inferring preconditions and postconditions by propagating weakest preconditions in an iterative semi-automated process. The goal is that developers are relieved from defining simple contracts that can be inferred. They only provide additional invariants or complex restrictions. Unfortunately the prototype does not cover postconditions and is not publicly available. 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.

A proof-preserving approach for the *Extract Method* refactoring was presented by Cousot et al. [4]. It can be used if the old and the extracted method have to be verified.

The specification refactorings *Pull Up Specification* and *Push Down Specification* were implemented by Hull [11]. The goal is to move specifications before code refactorings to simplify those. A trial-and-error heuristic is used and adjustments for callers of changed methods are still an open issue. The implementation is available as open source.

7. CONCLUSIONS AND FUTURE WORK

In this technical report, we have presented details on a solution for the problem that redundant information in code, contracts, and models can become inconsistent during the development of component-based systems. We have explained how we monitor changes in the Java source code editor of the Eclipse workbench to trigger incremental model transformations on architectural models, which are based on these changes. We have evaluated the approach in a case study and have shown that this is an efficient way to keep component

[4] ibm.com/software/products/ratirhapfami
[5] sparxsystems.com
[6] borland.com/products/together and uml-lab.com
[7] structure101.com
[8] http://lattix.com

models consistent in the case of source code changes. For contracts, we have discussed what is necessary and possible to maintain them if the source code or contract is modified. We have presented a prototypical implementation, which we applied successfully to the realistic JavaCard API case study. Altogether, we have demonstrated that semi-automated consistency can be achieved for redundant information in the implementation, contracts, and architecture of component-based software systems. In the future, we will finish our work on transformations that keep the component implementation consistent after model changes. We will implement and test the remaining contract change scenarios with additional case studies. Finally, we will improve the performance of our research prototype by employing incremental parsing techniques.

Acknowledgments

This work was partially funded by the German Federal Ministry of Education and Research under grant BMBF 01BY1172 (KASTEL) and by the German Research Foundation in the Priority Programme SPP1593 Design For Future.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. "ArchJava: connecting software architecture to implementation." In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE. 2002, pp. 187–197.

[2] C. Atkinson, D. Stoll, and P. Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development." In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by L. Maciaszek, C. González-Pérez, and S. Jablonski. Vol. 69. Communications in Computer and Information Science. Springer, 2010, pp. 206–219.

[3] S. Becker, H. Koziolek, and R. Reussner. "The Palladio component model for model-driven performance prediction." In: *Journal of Systems and Software* 82 (2009), pp. 3–22.

[4] P. M. Cousot et al. "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, pp. 213–232.

[5] L. De Silva and D. Balasubramaniam. "Controlling software architecture erosion: A survey." In: *Journal of Systems and Software* 85.1 (2012), pp. 132–151.

[6] 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, pp. 261–270.

[7] 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, pp. 90–99.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 4th ed. Addison-Wesley, 1999, p. 431.

[9] M. Goldstein, Y. A. Feldman, and S. Tyszberowicz. "Refactoring with Contracts." In: *Proceedings of AGILE Conference (2006)*. 1011. 2006, pp. 53–64.

[10] F. Heidenreich et al. "Closing the Gap between Modelling and Java." In: *Software Language Engineering*. Ed. by M. van den Brand, D. Gašević, and J. Gray. Vol. 5969. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 374–383.

[11] I. Hull. "Automated Refactoring of Java Contracts." Master's Thesis. University College Dublin, 2010, p. 61.

[12] 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*. VAO '13. ACM, 2013, 5:1–5:6.

[13] M. E. Kramer et al. "Change-Driven Consistency for Component Code, Architectural Models, and Contracts." In: *Proceedings of the 18th International ACM Sigsoft Symposium on Component-Based Software Engineering*. CBSE '15. accepted, to appear. ACM, 2015.

[14] M. Langhammer. "Co-evolution of component-based architecture-model and object-oriented source code." In: *Proceedings of the 18th international doctoral symposium on Components and architecture*. ACM. 2013, pp. 37–42.

[15] M. Langhammer and M. E. Kramer. "Determining the Intent of Code Changes to Sustain Attached Model Information During Code Evolution." In: *Fachgruppenbericht des 2. Workshops "Modellbasierte und Modellgetriebene Softwaremodernisierung"*. Vol. 34 (2). Softwaretechnik-Trends. Gesellschaft für Informatik e.V. (GI), 2014.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. "JML: A Notation for Detailed Design." In: *Behavioral Specifications of Businesses and Systems*. Ed. by H. Kilov, B. Rumpe, and I. Simmonds. Vol. 523. The Springer International Series in Engineering and Computer Science. Springer US, 1999, pp. 175–188.

[17] G. T. Leavens et al. *JML Reference Manual – Draft Revision 2344*. Tech. rep. Department of Computer Science, Iowa State University, 2013.

[18] W. Mostowski. "Fully Verified Java Card API Reference Implementation." In: *Proceedings of the 4th International Verification Workshop (VERIFY 07), Workshop at CADE-21*. 2007.

[19] D. E. Perry and A. L. Wolf. "Foundations for the Study of Software Architecture." In: *ACM SIGSOFT Software Engineering Notes* 17.4 (1992), pp. 40–52.

[20] N. Polikarpova et al. "What Good Are Strong Specifications?" In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. IEEE Press, 2013, pp. 262–271.

[21] R. Reussner et al. *The Palladio Component Model*. Tech. rep. KIT, Fakultät für Informatik, 2011.

[22] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering." In: *Computer* 39.2 (2006), pp. 25–31.

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

Table 4: Summary of JML language features supported by the newly created model printer and parser.

[23] H. Shimba et al. "Bidirectional Translation between OCL and JML for Round-Trip Engineering." In: *Proceedings of the 20th Asia-Pacific Software Engineering Conference*. APSEC'13. 2013, pp. 49–54.

[24] Y. Wei et al. "Automated Fixing of Programs with Contracts." In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. ACM, 2010, pp. 61–72.

APPENDIX

A. JML MODEL PRINTER AND PARSER

To our knowledge, no model printer and parser for JML exists except for an Xtext-based implementation mentioned in a synchronization approach for OCL and JML [23]. Unfortunately, the implementation is not available. Therefore, we constructed a simple model printer and parser for a subset of JML.

The objective of the model printer and parser is a fast and simple conversion from model to JML and vice versa to be used in our evaluation. Therefore, we did not aim for soundness, error-checking or extensive supporting techniques. Instead, we assume that the given JML contracts are syntactically correct.

We implemented the JML model printer and parser using the Xtext framework. The grammar is based on a user defined Java 5 grammar[9]. We replaced the defined expressions with Xbase[10] expressions and added Java-style variable declarations. Xbase is a partial programming language for Java-like expressions that can be used in Xtext grammars. Using Xbase allows the user to define illegal syntax with respect to Java and JML. This is, however, fine with our assumption of correct JML contracts. We extended the Xbase expressions and the Java grammar with the JML language elements show in Table 4. The model printer and parser is seamlessly integrated in Eclipse and EMF.

[9] `www.eclipse.org/forums/index.php/t/251746/`

[10] `https://www.eclipse.org/Xtext/documentation/305_xbase.html#xbase-language-ref-introduction`

B. TEST ENVIRONMENT FOR JML

We use system tests to evaluate the consistency concept for code and contracts. These tests check the whole processing reaching from change dection to the serialization of the synchronized artifacts. The tests use an adjusted version of the JavaCard API project. We had to modify the project to circumvent limitations of our prototype and test environment. The adjustments are separated into syntactic and semantic changes. The former do not change the meaning of the contracts or the code while the latter do. Because we focus on the change from one artifact state to another, this is no limitation for our tests. An overview of the syntactic changes is given in Table 5. Table 6 illustrates the semantical changes.

A evaluation test consists of five steps: First, the framework is initialized with a fresh copy of the JavaCard API project. Second, the test looks for artifact elements that can be changed in the test. Third, an editor manipulator opens the editor and performs the change. Thereby, the code monitor detects a change and propagates it to the synchronization engine. Changes in JML are injected in the synchronization engine directly. In the fourth step, the test waits for the synchronization to finish and performs a syntax check by compiling the whole project with an OpenJML compiler. In the last step, the test calculates the differences between the synchronized and the original project and verifies them with reference differences.

We separate our tests into two categories based on the test coverage: Path and context. The former group tries to cover as much paths as possible. We omit the second step for these tests, because we select the changed elements manually to cover a certain path. The context tests try to test one single path with as much contexts as possible. We omit the last step for these tests, because creating reference differences for all of them requires too much effort.

C. OPENJML PITFALLS

We used OpenJML in our code and contract synchronization evaluation for checking the syntax of the results. We encountered the following problems and pitfalls:

- When compiling via command line, we were unable to change the specification visibility by adding modifiers 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 worked.
- 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 [17, chap. 17.3]. Therefore, we had to remove the fields completely from the JML files.

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 [17, chap. 2.4]	24
added missing throws declarations	required [17, chap. 17.3]	21
removed implementations from JML	not allowed [17, chap. 17.3]	13
converted ghost to regular field	simplified handling in prototype	13
sum of all syntactic changes		267

Table 5: Overview of syntactic changes applied to the JavaCard 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
sum of all removed specification items		440

Table 6: Overview of semantic changes applied to the JavaCard API project to make it usable with our prototype. The project contained ca. 1410 specification items in total.