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

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
erik.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. Because 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 paper, we present 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

CBSE'15, May 4–8, 2015, Montréal, QC, Canada.

ACM 978-1-4503-3471-6/15/05 ...$15.00.
http://dx.doi.org/10.1145/2737166.2737177.

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 languages. Architecture Description Languages (ADLs), for example, can be used to model components and their relationships while omitting implementation detailsand 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, for example, may appear in the code, the architecture model, and the contracts. Such redundant information becomes inconsistent if changed in isolation. If consistency is restored manually, modifications have to be performed in each artefact, requiring manual effort and possibly still leading to costly inconsistencies. For security-relevant components that are verified against contracts, such inconsistencies can lead to insecure systems and are thus intolerable.

In this paper, we present a semi-automated approach and a tool 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 [10] 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) [2], 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 models are kept consistent accordingly. Our evaluation shows that it is possible to keep these artefacts automatically consistent in most of these change and refactoring scenarios.

The remainder of this paper is structured as follows: After foundations (section 2) and an overview (section 3), we discuss change monitoring (section 4). Then, we present contract consistency (section 5), followed by related work (section 6), and a conclusion with a discussion of future work (section 7). A full version of this paper has been published as technical report [11] and all code and tests are freely accessible.[1]

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

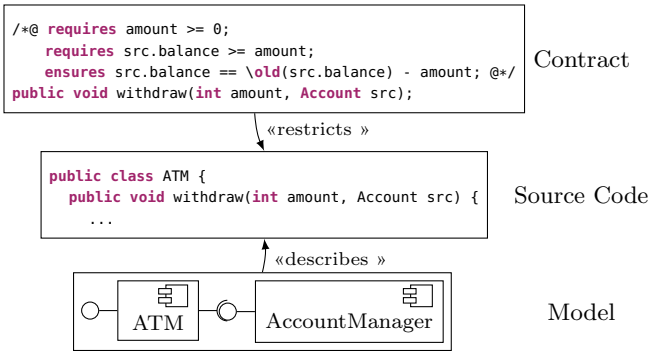

Figure 1: Extracts from the contract, source code, and model for an example banking system

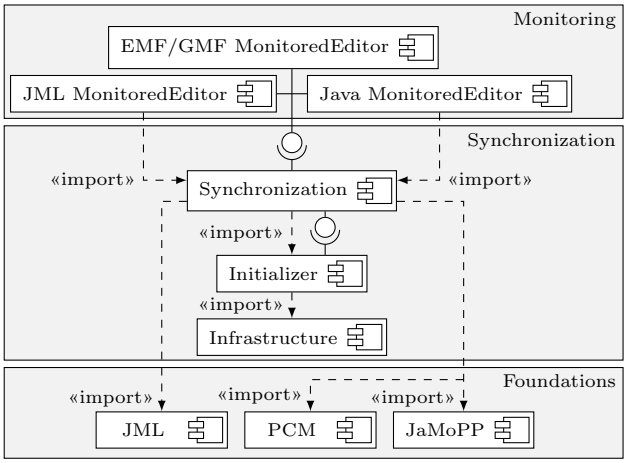

Figure 2: Simplified architecture of the implementation for synchronization between code and contracts and code and architecture.

2. BACKGROUND AND FOUNDATIONS

The *Palladio Component Model* is the Ecore-based metamodel of the component-based ADL that is used in the Palladio Bench. It features reusable components, which provide and require services at interfaces, which are both defined in a system-independent component repository.

The *Java Modeling Language (JML)* [14] 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. Essential JML statements are preconditions (`requires`), postconditions (`ensures`), and invariants. 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).

3. FRAMEWORK OVERVIEW

The Vitruvius framework is based on the central idea of Orthographic Software Modeling: 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-called meta repository. Therefore, views can focus on relevant elements and relations and do not have to consider all internal details of all modeling languages. 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.

Our prototype 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. 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

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 [16]. 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.

In our prototype, we chose PCM as architecture model and JaMoPP [8] to represent Java code and defined the following mapping rules between them (cf. [12]): 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 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.

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.

4.1 Monitoring and Propagating Changes

Two monitors have been implemented: *JavaMonitoredEditor* and *EMF/GMF MonitoredEditor*. Their interaction with the *Synchronization* components is depicted in Figure 2.

4.1.1 Code Changes

In order to keep the architecture consistent with architectural relevant source code changes, we implemented a change notification mechanism. We extended the Java code editor of the Eclipse IDE and listen to changes of the Abstract Syntax Tree (AST) 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, we have to clarify their intent. The intent clarification mechanism lets the transformations, which keep the architecture and code consistent, interact directly with the developers or architects and displays a dialog to clarify the intent [13]. In future work, we plan to implement more interaction options, e.g., postponing decisions, which are collected in a task list.

4.1.2 Architecture Model Changes

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 all changes are propagated in the order they were conducted using the following steps:

1. find the correct transformation for the change
2. execute the transformation
3. create/update/delete the correspondence
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*. 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)*[2].

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 standard deviation (in parentheses)

The following 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. The sample correlation coefficient between the average time consumption in ms 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. Although our monitor needs less than one second for 1045 logical lines of code, a single change may have side-effects on large, generated files and so 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 so 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

[2]hadoop.apache.org

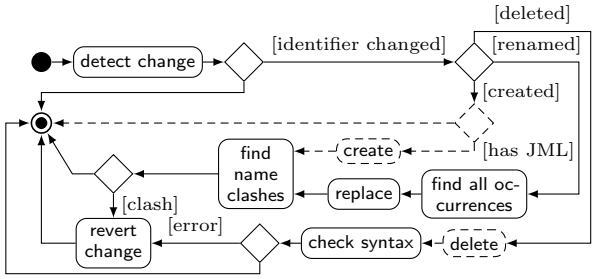

Figure 3: Reaction on an identifier change in the direction specification to specification and code to specification. The dashed elements are only relevant for the latter.

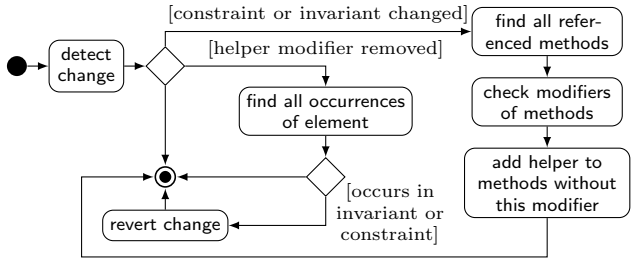

Figure 4: Reaction on a `helper` change in the direction specification to specification.

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 occurs. We do not limit changes to refactorings but consider any possible change performed by a developer. We covered identifiers, visibility, types, `pure`, `helper`, `nullable`, default behaviors regarding null, `assignable`, generic and exception specifications. The following sections focus on three elements that have been evaluated as described in subsection 5.2. The complete consistency concept is described in [11].

5.1 Contract Consistency Concept

We 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 only cover JML elements of language levels 0 and 1, because they are supported by most JML tools. 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 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.

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 3 shows the necessary change reactions for the direction from code to specification and from specification to specification. The latter is relevant because to our knowledge there is no refactoring support for JML. From specification to code we only have to consider specification-only methods and fields as all other structural changes have to be performed in the code as mentioned above: If an identifier clash is detected we simply revert the identifier change.

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. The reaction on a change in the body of a pure method in direction from code to specification checks the body for non-pure statements. If such a statement is found, the pure modifier is removed and the specification to specification reaction is triggered. Otherwise, no action is taken. In direction specification to code the reaction to the addition of a pure modifier is similar: If a non-pure statement is found, the change is blocked. The change reaction shown in Figure 5 handles the effects of a change in the specification on the specification itself.

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. As a precaution, we add the `helper` modifier to a method as soon as it is mentioned in an invariant or a constraint. Methods used in invariants and constraints are query methods. Therefore, they have no influence on the state and cannot break invariants and constraints. Figure 4 illustrates the effects of a specification change on the specification. Other directions are not relevant for the `helper` modifier.

5.2 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 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 2. 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

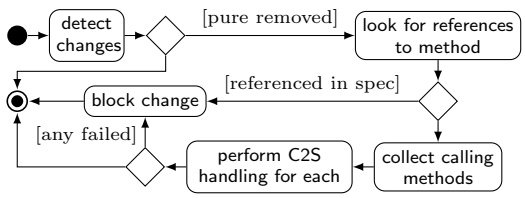

Figure 5: Reaction on a `pure` change in the direction specification to specification.

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

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.

With our evaluation we answered the following research question: Is our implemented consistency concept capable of keeping specifications consistent after changes in the code or specifications for a specific case study? As evaluation case study we used a verified implementation of the JavaCard API [15], 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.

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 2. 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. 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.

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

De Silva and Balasubranamiam [4] 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. ArchJava [1] is an example of linkage through information merging. It introduces new language constructs to Java to include architectural information directly into the source code.

There are also commercial solutions that support model and code synchronization. IBM Rational Rhapsody[3], for example, supports round-trip engineering of code and UML as well as other languages, e.g., SysML. Borland Together[4] and UML Lab[4] 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.

Feldman [5] gives a high-level overview of code changes and their effects on contracts. 68 Fowler refactorings [6] are inspected and grouped into three categories: Refactorings

[3]ibm.com/software/products/ratirhapfami

[4]borland.com/products/together and uml-lab.com

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. The full analysis and the change handling code are not made public.

Crepe is a tool based on these findings [7], 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, and creates new contracts, for example, when a new method is added. There is no implementation available.

A proof-preserving approach for the *Extract Method* refactoring was presented by Cousot et al. [3]. 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 [9]. 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 paper, we have presented 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 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] 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.

[3] 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.

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

[5] 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.

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

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

[8] F. Heidenreich et al. "Closing the Gap between Modelling and Java." In: *Software Language Engineering.* Vol. 5969. LNCS. Springer Berlin Heidelberg, 2010, pp. 374–383.

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

[10] M. E. Kramer, E. Burger, and M. Langhammer. "View-centric engineering with synchronized heterogeneous models." In: VAO '13. ACM, 2013, 5:1–5:6.

[11] M. E. Kramer et al. *Realizing Change-Driven Consistency for Component Code, Architectural Models, and Contracts in Vitruvius*. Tech. rep. 2015.

[12] 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.

[13] M. Langhammer and M. E. Kramer. "Determining the Intent of Code Changes to Sustain Attached Model Information During Code Evolution." In: vol. 34 (2). Softwaretechnik-Trends. GI e.V., 2014.

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

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

[16] 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.