Towards Model-Driven Unit Testing

Gregor Engels[1,2], Baris Güldali[1], and Marc Lohmann[2]

[1] Software Quality Lab
[2] Department of Computer Science
University of Paderborn, Warburgerstr. 100, 33098 Paderborn, Germany
engels@upb.de, bguldali@s-lab.upb.de, mlohmann@upb.de

Abstract. The Model-Driven Architecture (MDA) approach for constructing software systems advocates a stepwise refinement and transformation process starting from high-level models to concrete program code. In contrast to numerous research efforts that try to generate executable function code from models, we propose a novel approach termed *model-driven monitoring*. On the model level the behavior of an operation is specified with a pair of UML composite structure diagrams (visual contract), a visual notation for pre- and post-conditions. The specified behavior is implemented by a programmer manually. An automatic translation from our visual contracts to JML assertions allows for monitoring the hand-coded programs during their execution.
In this paper[1] we present how we extend our approach to allow for *model-driven unit testing*, where we utilize the generated JML assertions as test oracles. Further, we present an idea how to generate sufficient test cases from our visual contracts with the help of model-checking techniques.

Keywords Design by Contract, visual contracts, test case generation, model checking

1 Introduction

Everyone who develops or uses software systems knows about the importance of software qualities, e.g. correctness and robustness. However, the growing size of applications and the demand for shorter time-to-market hampers the development of high-quality software systems. To get a better handle on the complexity, the paradigm of model-driven development (MDD) has been introduced. In particular, the Object Management Group (OMG) pushed its Model-Driven Architecture (MDA) [1] initiative based on the Unified Modeling Language (UML) that provides the foundation for MDA. However, the MDA is still in its infancy compared to its ambitious goals of having a (semi-)automatic, tool-supported

[1] This paper is an abbreviated version of our same-titled contribution to MoDeV^{2}a 2006. For related work, refer to the longer version in the MoDeV^{2}a 2006 workshop proceedings or to our web page at http://www.upb.de/cs/ag-engels.

stepwise refinement process from vague requirements specifications to a fully-fledged running program. A lot of unresolved questions exist for modeling tasks as well as for automated model transformations.

Nevertheless, in today's software development processes models are an established part for describing the specification of software systems. In principle, models provide an abstraction from the detailed problems of implementation technologies. They allow software designers to focus on the conceptual task of modeling static as well as behavioral aspects of the envisaged software system. Unfortunately, abstraction naturally conflicts with the desired automatic code generation from models. To enable the latter, fairly complete and low-level models are needed. Today, a complete understanding of the appropriate level of detail and abstraction of models is still missing. Thus, in today's software development processes developers are normally building an application manually with respect to its abstract specification with models.

In our work, we introduced a new modeling approach. We do not follow the usual approach that models should operate as source for an automatic code generation step that produces the executable function code of the program. Rather, we restrict the modeling task to providing structural information and minimal requirements towards behavior for the subsequent implementation. We expect that only structural parts of an implementation are automatically generated, while the behavior is manually added by a programmer. As a consequence it can not be guaranteed that the hand-coded implementation is correct with respect to the modeled requirements. Therefore, we have shown in previous publications [2–4] how models can be used to generate assertions which monitor the execution of the hand-coded implementation. Herewith, violations of the modeled requirements will be detected at runtime and reported to the environment. We call this novel approach *model-driven monitoring*. It is based on the idea of Design by Contract (DbC) [5], where so-called contracts are used to specify the desired behavior of an operation. Contracts consist of pre- and post-conditions. Before an operation is executed, the pre-condition must hold, and in return, after the execution of an operation, the post-condition must be satisfied. The DbC approach has been introduced at the level of programming languages. For instance, the Java Modeling Language (JML) extends Java with DbC concepts [6] which are annotated to the source code. During the execution of such an annotated Java program, the assertions are monitored. An exception is raised as soon as a violation of the assertions is detected. With the concepts of *visual contracts* [2] we have lifted the idea of contracts to the level of models. A visual contract allows for specifying a contract by pairs of UML composite structure diagrams for the pre- and post-conditions. A transformation of our visual contracts into JML allows for monitoring a system that is implemented manually.

Now we want to extend our approach to allow for *model-driven unit testing*. The visual contracts respectively the generated JML assertions are viewed as test oracles to decide whether the results calculated by a hand-coded implementation are correct. Additionally, we want to generate test cases from our models with the help of model-checking techniques.

2 Overview of the Approach

Test-driven development [7] is an important part of agile processes. E.g. Extreme Programming (XP) [8] emphasizes the test-first approach. When handling a programming task, programmers always begin writing unit tests. This tests formalizes the requirements. If all tests run successfully then the coding is complete. To accent the agile part of our model-driven monitoring approach we want to support the test-driven development by enabling model-driven unit testing. Therefore, beside the generation of runtime assertions we want to automatically generate test cases from our models. Figure 1 shows our development process enabling model-driven monitoring and model-driven unit testing.

On the design level, a software designer has to specify a model of the system under development. This model consists of class diagrams and visual contracts. The class diagrams describe the static aspects of the system. Each visual contract specifies the behavior of an operation. The behavior of the operation is given in terms of data state changes by pre- and post-conditions, which are modeled by a pair of UML composite structure diagrams as explained in Sect. 3.

In the next step, we generate code fragments from the design model. This generation process consists of two parts. First, we generate Java class skeletons from the design class diagrams. Second, we generate JML assertions from every visual contract and annotate each of the corresponding operations with the generated JML contract. The JML assertions allow us to check the consistency of models with manually derived code at runtime. The execution of such checks is transparent in that, unless an assertion is violated, the behavior of the original program remains unchanged.

Then, a programmer uses the generated Java fragments to fill in the missing behavioral code in order to build a complete and functional application. His programming task will emanate from the design model of the system. Particularly, he will use the visual contracts as reference for implementing the behavior of operations. He has to code the method bodies, and may add new operations to existing classes or even completely new classes, but he is not allowed to change the JML contracts. If new requirements for the system demand new functionality then the functionality has to be specified with visual contracts before the programmer can start programming. Using our visual contracts this way in a software development process resembles agile development approaches.

When a programmer has implemented the behavioral code, he uses the JML compiler to build executable binary code. This binary code consists of the programmer's behavioral code and additional executable runtime checks which are generated by the JML compiler from the JML assertions. The manual implementation of a programmer leads to system state changes. The generated runtime checks monitor the pre- and post-conditions during the execution of the system.

To further integrate agile approaches in our process we additionally want to integrate *model-driven unit testing* in our development process. Therefore, we have to address the following three problems of model-driven testing [9]:

1. the generation of test cases from models,

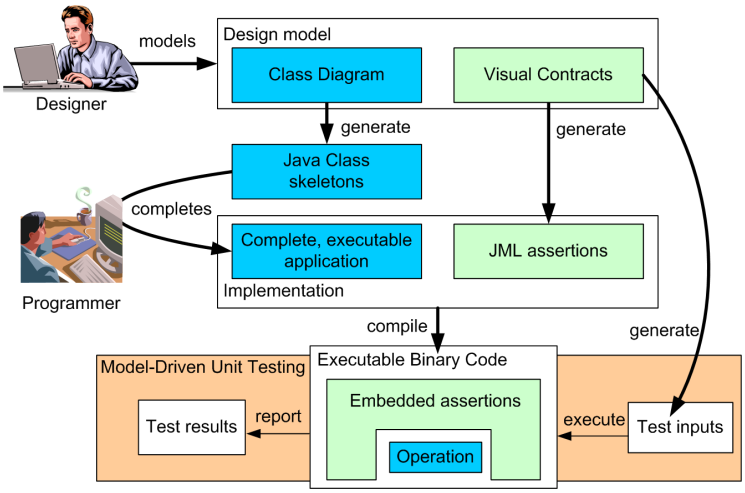

Fig. 1. Overview of the testing approach

2. the generation of a test oracle to determine the expected results of a test,
3. the execution of tests in test environments.

The basic idea of our testing approach is that the specification of an operation by a pre- and post-conditions (visual contract) can be viewed as a test oracle [10] and runtime assertion checking can be used as a decision procedure. Thus, our visual contacts can be viewed as test oracles since the JML assertions are generated from our visual contracts. Still, we need to answer the problem of how to generate test cases from models. Therefore, we want to combine well-known testing techniques for the generation of test input parameters and model checking to be able to create concrete system states. The idea how to create test cases is described in detail in Sect. 5.1.

3 Modeling with Visual Contracts

We show how to specify a system with visual contracts by the example of an online shop. We distinguish between a static and a functional view. UML class diagrams are used to represent the *static view* of a system specification. Figure 2 shows the class diagram of the sample online shop. We use the stereotypes `control` and `entity` expressing a different role of a class in the implementation. Instances of control classes encapsulate the control related to a specific use case and coordinate other objects. Entity classes model long-lived or persistent information. The control class `OnlineShop` is connected to the entity classes of the system via qualified associations. A rectangle at an association end with a qualifier (e.g. `productNo`) designates an attribute of the referenced class. The qualifier allows us to get direct access to specific objects.

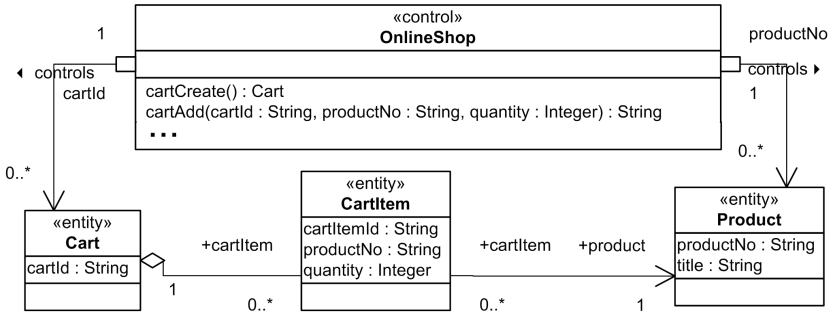

Fig. 2. Class diagram specifying static structure of online shop

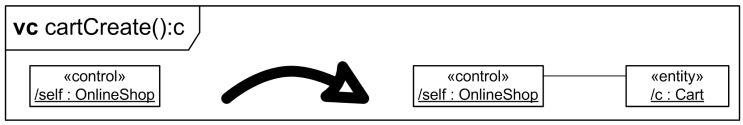

Fig. 3. Visual contract for operation *cartCreate*

Class diagrams are complemented by visual contracts that introduce a *functional view* integrating static and dynamic aspects. Visual contracts allow us to describe the effects of an operation on the system state of the system. Thus, for our visual contracts we take an operation-wise view on the internal behavior.

In the following, we want to explain our visual contracts by two examples. The operation `cartCreate` of the control class `OnlineShop` creates a new cart. Figure 3 shows a visual contract that describes the behavior of the operation. The visual contract is enclosed in a frame, containing a heading and a context area. The keyword `vc` in the heading refers to the type of diagram, visual contract in this case. The keyword is followed by the name of the operation that is specified by the visual contract. The operation name is followed by a parameter-list and a return-result if they are specified in the class diagram. The parameter-list is an ordered set of variables and the return-result is also a variable. The variables of the parameter-list and the return-result are used in the visual contract.

The visual contract is placed in the context area and consists of two UML composite structure diagrams [11], representing the pre- and the post-condition of an operation. Each of them is typed over the design class diagram. The semantics of our visual contracts is defined by the loose semantics of open graph transformation systems [12]. The basic intuition for the interpretation of a visual contract is that every model element, which is only present on the right-hand side of the contract, is newly created, and every model element that is present only on the left-hand side of the contract, is being deleted. Elements that are present on both sides are unaffected by the contract. Additionally, we may extend the

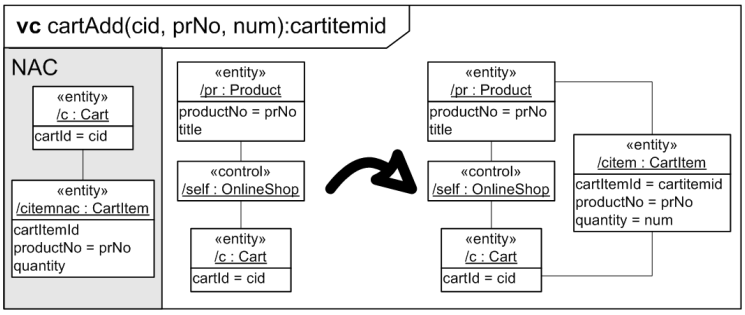

Fig. 4. Visual contract for operation *cartAdd*

pre- or post-condition of a visual contract by negative pre-conditions (i.e., negative application conditions [13]) or respectively by negative post-conditions. A negative condition is represented by a dark rectangle in the frame. If the dark rectangle is on the left of the pre-condition, it specifies object structures that are not allowed to be present before the operation is executed (see Fig. 4). If the dark rectangle is on the right of the post-condition, it specifies object structures that are not allowed to be present after the execution of the operation.

The contract as described in Fig. 3 expresses that the operation `cartCreate` can always be executed, because the pre-condition only contains the model element `self`, i.e. the object executing the operation. As an effect, the operation creates a new object of type `Cart` and a link between the object `self` and the new object. Additionally, the object `c:Cart` is the return value of the operation `cartCreate` as indicated by the variable `c` used in the heading.

Figure 4 shows a more complex contract specifying the operation `cartAdd`. This operation adds a new `CartItem`, which references an existing `Product`, to an existing `Cart`. In contrast to the visual contract of Fig. 3, the variables of the parameter-list and the return-value are now used to specify values of attributes of the objects. For a successful execution of the operation, the object `self` must know two different objects with the following characteristics: an object of type `Cart` that has an attribute `cartId` with the value `cid`, and an object of type `Product` that has an attribute `productNo` with the value `prNo`. The concrete argument values are bound when the client calls the operation. The same `Cart` object is reused in the negative pre-condition. The negative pre-condition extends the pre-condition by the requirement that the `Cart` object is not linked to any object of type `CartItem` that has an attribute `productNo` with the value `prNo`. This means, it is not permitted that the product is already contained in the cart. As a result, the operation creates a new object of type `CartItem` with additional links to previously identified objects. The return value of the operation is the content of the attribute `cartItemId` of the newly created object.

4 Translation to JML

After describing the modeling of a software system with visual contracts, we now present how the model-driven software development process continues from the design model. A transformation of visual contracts to JML constructs provides for model-driven monitoring of the contracts. The contracts can be automatically evaluated for a given state of a system, where the state is given by object configurations. The generation process as well as the kind of code that is generated from a class diagram and the structure of a JML assertion that is generated from a visual contract are described in detail in [2, 4]. Here we only describe the transformation more generally and from a methodical perspective.

Each UML class is translated to a corresponding Java class. For attributes and associations, the corresponding access methods (e.g., `get`, `set`) are added. For multi-valued associations we use classes that implement the Java interface `Set`. Qualified associations are provided by classes that implement the Java interface `Map`. We add methods like `getProduct(int productNo)` that use the attributes of the qualified associations as input parameters. Operation signatures that are specified in the class diagram are translated to method declarations in the corresponding Java class.

For each operation specified by a visual contract, the transformation of the contract to JML yields a Java method declaration that is annotated with JML assertions. The pre- and post-conditions of the generated JML assertions are interpretations of the graphical pre- and post-conditions of the visual contract. When any of the JML pre- and post-conditions is evaluated, an optimized breadth-first search is applied to find an occurrence of the pattern that is specified by the pre- or post-condition in the current system state. The search starts from the object `self` which is executing the specified behavior. If the JML pre- or post-condition finds a correct pattern, it returns true, otherwise it returns false.

5 Test Case Generation and Test Execution

In the previous sections we explained how a software designer develops a design model and how Java class skeletons and JML assertions can be generated from them. We also explained how a programmer can complete the generated code fragments to build a complete executable application. After these steps we want to test our application. In Sect. 2 we explained the three tasks of model-driven testing. In this section we will explain how we handle the first and the third task, i.e. the generation of test cases and the execution of a test. The second task (the generation of a test oracle) is described in Sect. 4 since we can interpret the JML assertions as test oracles. Similar to classical unit-testing, our test items are operations. The behavior of an operation is dependent of the input parameters and the system state. Thus, a test case has to consider the parameter values of an operation and a concrete system state.

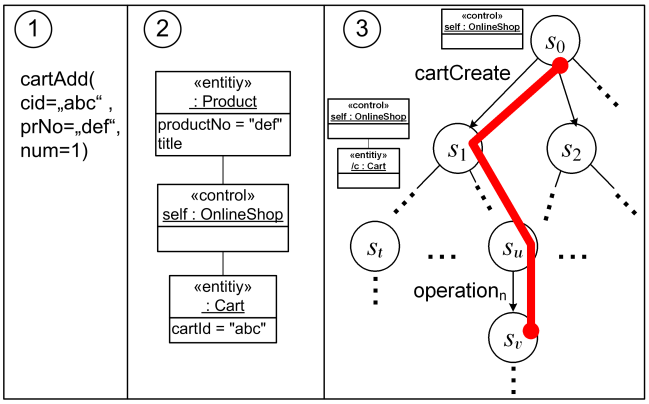

Fig. 5. Three steps of test case generation

5.1 Test Case Generation

A test case for an operation consists of concrete parameter values and a concrete system state. We can generate a test case for an operation from our model in three successive steps. In the following, we explain how to generate a sample test case for the operation `cartAdd` (Fig. 4). Figure 5 illustrates the three steps.

In the first step, we generate values for the input parameters of an operation as specified in the class diagram. In Fig. 5 we generated the parameter values for the operation `cartAdd` randomly. For the parameter `cid` the value “abc” is generated. The parameter `prNo` gets the value “def” and the variable `num` gets the value “1”. Beside a the random generation of input parameters, we could also use other techniques for test data generation, e.g. equivalence-class partitioning or boundary value analysis (see e.g. [14]).

To generate a sufficient system state for testing, we have to execute two further steps. Since the visual contracts specify system state requirements, we use them as source for generating the system state. Therefore, we initialize the pre-condition of a visual contract with the parameter values generated in step one. The variables in the parameter-list are used to restrict the attribute values of objects in the pre-condition as explained in Sect. 3. Thus, the initialization gives an object structure. In this object structure some of the attributes have concrete values. Figure 5 shows how the attributes `productNo` and `cartId` of the classes `Product` and `Cart` are initialized with the parameter values of step one according to the pre-condition in Fig. 4. It is important to notice that this object structure describes a system state only partially.

In the last step of our test case generation, we have to find out how to generate a system state which contains the object structure found in step two. Due to the fact that the object structure in the previous step defines a system state only partially, we cannot just build a system state by creating the known objects

and attribute values. Such a system state would be incomplete and it would be artificial in a sense that the application would never create such a system state at runtime. Additional objects or attribute values can be created during the execution of the systems at runtime and these may have side-effects on the execution of an operation. Thus, tests should work on realistic system states. To avoid these artificial system states it would be useful to build a system state by using the control operations of the system itself. We assume that each operation call leads to a state change of the system. Thus, we have to find a sequence of operation calls that starting from the initial system state lead to a sufficient system state which contains this object structure. As a visual contract describes the system state change of an operation, we can use these contracts to compute all possible states of the system. Therefore, we consider a system state as a *graph* and the visual contracts constitute *production rules* of a *graph transition system*. Figure 5 illustrates how we want to generate a transition system. Initially the system state comprises just an instance (`self`) of the controller class `OnlineShop`. Executing, e.g., the operation `cartCreate` makes the in Fig. 3 specified changes on the system state. Thus, a new object of type `Cart` is generated and linked to the control object `self`. Executing further operations brings the system to a state s_v which contains the object structure generated in step two. Knowing all visual contracts and an initial state, we can compute the graph transition system and search for a production sequence that creates a system state which contains the object structure found in step two. These computations can be done automatically with *model checking* techniques [15]. The computed production sequence directly refers to an operation sequence which brings the system state to some desired state containing the object structure computed in step two. If no sufficient production sequence is found in the graph transition system (the searched object structure cannot be constructed using the existing operations), our test case generation approach has to backtrack to step one and generate other test data.

5.2 Test Execution with Embedded Oracles

After test cases are generated, the test execution can start. Test execution comprises two main steps as shown in Fig. 6. First, the operation sequence determined by the test case generation must be executed in order to set the system state. Second, the operation under test is called with the test input parameters also generated by the test case generation.

The embedded assertions lead to a run-time behavior of an operation call as shown in Fig. 6. When the operation under test is called, a pre-condition check method evaluates the method's pre-condition and throws a pre-condition violation exception if it does not hold. If the pre-condition holds, then the original, manually implemented operation is invoked. After the execution of the original operation, a post-condition check method evaluates the post-condition and throws a post-condition violation exception if it does not hold. If the embedded assertions throw an exception then the implementation does not behave according to its specification. Thus, we have found an error.

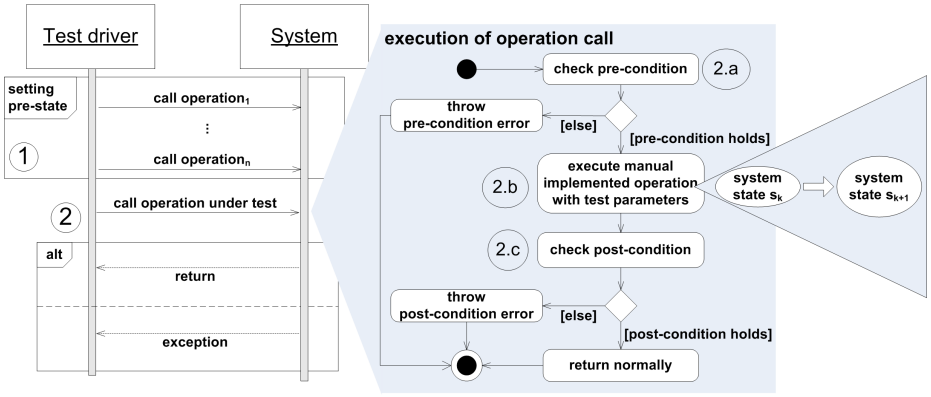

Fig. 6. Run-time behavior of test execution

6 Tool Support

Most of the steps of our approach can be supported by tools. In former publications we have reported on our Visual Contract Workbench, an integrated development environment for using visual contracts in a software development process [16]. This development environment allows software designers to model class diagrams and specify the behavior of operations by visual contracts. It further supports automatic code generation as described in Sect. 4.

The most challenging task of our test generation approach is finding an operation sequence for setting a system state as explained in Sect. 5.1. This task can be automatically solved by *model checking* tools. A candidate for our purposes is GROOVE [17], a model checker for attributed graph transition systems. The test execution can be implemented by a test driver as shown in Fig. 6. In the context of JML, we can use the JMLUnit tool [18] for this purpose.

7 Conclusion

We have developed an approach that lifts the Design by Contract (DbC) idea, which is usually used at the code level, to the model level. Visual contracts are used as a specification technique. They are used to specify system state transformations with pre- and post-conditions which are modeled by UML (composite) structure diagrams. Further, we presented how to use the visual contracts in a software development process. A translation of the visual contracts into the Java Modeling Language, a DbC extension for Java, enables the model-driven monitoring. To support model-driven monitoring, we provide a visual contract workbench that allows developers to model class diagrams and visual contracts. Further the workbench supports automated code generation.

In this paper, we have shown how we want to extend our approach with model-driven unit testing. In our testing approach, a test case consists of parameter values and a concrete system state. The visual contracts – respectively the generated JML assertions – serve as test oracles to decide whether a manual implementation is correct according to its specification. In future work we will concretize our testing approach and extend our workbench with testing facilities.

References

1. Meservy, T.O., Fenstermacher, K.D.: Transforming software development: An MDA road map. IEEE Computer **38** (2005) 52–58
2. Lohmann, M., Sauer, S., Engels, G.: Executable visual contracts. In Erwig, M., Schürr, A., eds.: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05). (2005) 63–70
3. Engels, G., Lohmann, M., Sauer, S., Heckel, R.: Model-driven monitoring: An application of graph transformation for design by contract. In: International Conference on Graph Transformation (ICGT) 2006. (2006) accepted for publication.
4. Heckel, R., Lohmann, M.: Model-driven development of reactive informations systems: From graph transformation rules to JML contracts. International Journal on Software Tools for Technology Transfer (STTT) (2006) accepted for publication.
5. Meyer, B.: Applying "Design by Contract". IEEE Computer **25** (1992) 40–51
6. Leavens, G., Cheon, Y.: Design by Contract with JML (2003)
7. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional (2002)
8. Beck, K.: Extreme Programming Explained. Embrace Change. The XP Series. Addison-Wesley Professional (1999)
9. Heckel, R., Lohmann, M.: Towards model-driven testing. Electr. Notes Theor. Comput. Sci. **82** (2003)
10. Antoy, S., Hamlet, D.: Automatically checking an implementation against its formal specification. IEEE Transactions on Software Engineering **26** (2000) 55–69
11. OMG (Object Management Group): UML 2.0 superstructure specification - revised final adopted specification (2004)
12. Heckel, R., Ehrig, H., Wolter, U., Corradini, A.: Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. APCS (Applied Categorical Structures) **9** (2001) 83–110
13. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
14. Binder, R.V.: Testing Object-Oriented Systems. Addison-Wesley (2000)
15. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: International Conference on Graph Transformation (ICGT) 2004. (2004) 226–241
16. Lohmann, M., Engels, G., Sauer, S.: Model-driven monitoring: Generating assertions from visual contracts. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE) 2006 Demonstration Session. (2006)
17. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE) 2003. (2003) 479–485
18. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: European Conference on Object-Oriented Programming (ECOOP) 2002. (2002) 231–255