Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract

Gregor Engels[1,2,3], Marc Lohmann[1], Stefan Sauer[1,3], and Reiko Heckel[4]

[1] University of Paderborn, Department of Computer Science
Warburger Str. 100, 33098 Paderborn, Germany
{engels, mlohmann}@uni-paderborn.de
[2] sd&m AG, software design & management
Am Schimmersfeld 7a, 40880 Ratingen, Germany
[3] Software Quality Lab, University of Paderborn,
Warburger Str. 100, 33098 Paderborn, Germany
sauer@s-lab.upb.de
[4] University of Leicester, Department of Computer Science
University Road, Leicester, United Kingdom
reiko@mcs.le.ac.uk

Abstract. The model-driven development (MDD) 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*. Here, models are used to specify minimal requirements and are transformed into assertions on the code level for monitoring hand-coded programs during execution.
We show how well-understood results from the graph transformation community can be deployed to support this model-driven monitoring approach. In particular, models in the form of visual contracts are defined by graph transitions with loose semantics, while the automatic transformation from models to JML assertions on the code level is defined by strict graph transformation rules. Both aspects are supported and realized by a dedicated Eclipse plug-in.

1 Introduction

Object-oriented technology provided us with a better handle on complexity than previous technologies. Nevertheless, the growing size of applications and the demand for shorter time-to-market entail that many issues remain. In recent years, the paradigm of a model-driven development (MDD) approach has been introduced and discussed heavily. In particular, the Object Management Group (OMG) favored a model-driven approach to software development and pushed its Model-Driven Architecture (MDA) [1] initiative as well as standards such as the Unified Modeling Language (UML) that provides the foundation for MDA.

However, model-driven development is still in its infancy compared to its ambitious goals of having a (semi-)automatic, tool-supported stepwise refinement process from vague requirements specifications to a fully-fledged running program. A lot of unresolved questions exist for horizontal *modeling* tasks as well as for vertical *model transformation* tasks.

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.

Horizontal modeling levels are interrelated by vertical model transformations. Here, too, a complete understanding is missing how such a transformation might be specified and implemented. A number of model transformation approaches have been proposed and discussed, in particular, as answer to the Query-View-Transformation (QVT) RFP of the OMG [2].

The graph transformation community has been investigating and discussing since years graph-based approaches for specifying structure and behavior of software components as well as for specifying transformations.Thus, graph transformation provides well-defined and well-investigated candidate solutions for the mentioned open issues in the MDD realm.

In our work, we employ results from research on graph transformation to offer solutions for horizontal modeling as well as vertical model transformation problems. In particular, we introduce a novel 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 structure 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. Yet, we will show 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.*

Model-driven monitoring (MDM) is based on the idea of Design by Contract (DbC) [3], 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, it has to be guaranteed that the post-condition is satisfied.

The DbC approach has been introduced for textual programming languages and is supported by appropriate tools, e.g. for the Eiffel language [4]. Recently, the same approach has been put into effect for the Java programming language. For instance, the Java Modeling Language (JML) extends Java with Design by Contract concepts [5]. JML assertions are based on Java expressions and are annotated to the source code. During the execution of such an annotated Java program, any violation of an assertion is monitored. An exception is raised as soon as a violation is detected.

We lift this idea of contract specifications to the level of visual models and reuse the concept of graph transformation to specify pre- as well as post-conditions of an operation in a graphical, UML-like way. As those *visual contracts* define minimal requirements towards an operation, the semantic concept of loose graph transitions, formalized by the double-pullback (DPB) approach [6], is deployed to provide the semantics of the contract-based approach.

Besides this novel modeling approach, we deploy graph transformation results for defining the automatic transformation step from visual contract specifications to textual JML assertions. In contrast to the modeling of minimal requirements illustrated above, we provide a complete specification of this transformation step here. Thus, the semantic concept of strict graph transformations is deployed which is formalized by the double-pushout (DPO) approach [7].

The complete approach of contract-based modeling of a software system is supported by a tool chain that we implemented as Eclipse plug-in. The presented method for specifying software components by contracts has been studied in an industrial setting [8, 9].

We give an overview of the model-driven monitoring approach in the following section. Section 3 explains our method of modeling with visual contracts based on the concepts of graph transitions. The translation from visual contracts to JML assertions is described in Sect. 4. There we use graph transformation rules for specifying the translation. The tools that we provide to support our method are introduced in Sect. 5. Finally, we summarize the achievements and sketch future perspectives.

2 Towards Model-Driven Monitoring

Model-driven monitoring (MDM) constitutes a novel strategy for model-driven software development beyond the classical idea of model-driven development (MDD) centered upon the automatic generation of function code and model-driven testing (MDT) focussing on automatically deriving test cases from models. We enable model-driven monitoring by embedding visual contracts in a model-driven software development process according to Fig. 1. Visual contracts are interpreted as models of behavior from which code for testing and runtime assertion checking can be generated. The visual contracts also specify the behavior which is then manually implemented by programmers.

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. Both the pre- and post-condition of a visual contract are typed over the class diagram.

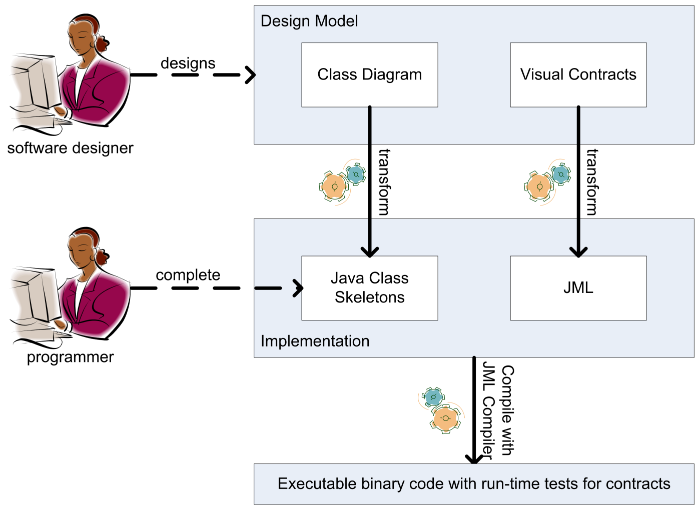


Fig. 1. Towards Model-Driven Monitoring

In the next step, we generate Java code 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 has to be transparent in that, unless an assertion is violated, the behavior of the original program remains unchanged. Thus, our transformation rules (see Sect. 4) for generating JML assertions from the UML design model only generate assertions that behave accordingly.

Then, a programmer uses the generated Java fragments to fill in the missing behavioral code in order to build a complete and functional application. Her programming task will emanate from the design model of the system. Particularly, she will use the visual contracts as reference for implementing the behavior of operations. She has to code the method bodies, and may add new operations to existing classes or even completely new classes, but she is not allowed to change the JML contracts. The latter guarantees that the JML contracts remain consistent with the visual contracts. Integrity of visual contracts can be technically assisted by separating Java class skeletons and JML assertions into two different files and prohibiting access to the JML assertions file. Programmers do not need to see the JML annotations; rather they should use the more intuitive visual contracts as the starting point for their programming.

When a programmer has implemented the behavioral code, she 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

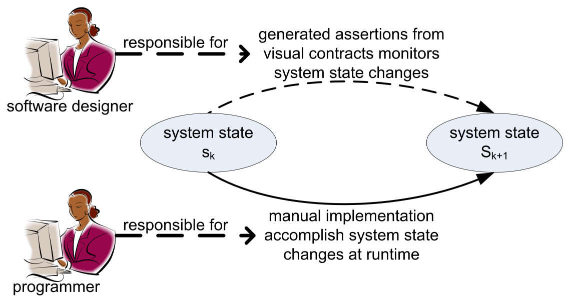

responsible for
generated assertions from visual contracts monitors system state changes
software designer
system state s_k
system state S_{k+1}
responsible for
manual implementation accomplish system state changes at runtime
programmer

Fig. 2. Behavior at runtime

generated by the JML compiler from the JML assertions. This leads to a runtime behavior as shown in Fig. 2. The manual implementation of a programmer leads to a system state change. The generated runtime checks monitor the pre- and post-conditions during the execution of the system. They monitor whether the manually coded behavior of an operation fulfills its JML specification. Thus, we indirectly monitor whether the system state change performed by the manual implementation complies with the visual contract specification of the design model since the JML annotations are purely generated from the visual contracts. Thus, we support model-driven monitoring of implementations by transforming our visual contracts into contracts in JML.

Our visual contracts are given in a UML-like notation of graph transformation rules. However, the classical interpretation of graph transformation rules based on the double-pushout approach (DPO) [7] is not adequate for the representation of a contract. In this approach it is assumed that during the execution of an operation nothing is changed beyond the specification in the rule. This would mean that we have to describe the behavior of an operation completely on the model level, which would lead to the drawbacks mentioned in the introduction. Rather, our method builds upon the loose semantic interpretation of visual contracts. They are interpreted as a minimal description of the data state transformation which has to be implemented by the programmer. Thus, a visual contract specifies only what at least has to happen on a system state, but it allows the programmer to implement additional effects. This loose interpretation is necessary both to give the programmer the opportunity for optimizing her code, e.g. by adding new classes or methods, and to generate assertions from partial, incomplete models. Therefore, we have to interpret our visual contracts as graph transitions. In the double-pullback (DPB) [6] approach graph transitions allow additional changes that are not encoded in the transformation rules.

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.

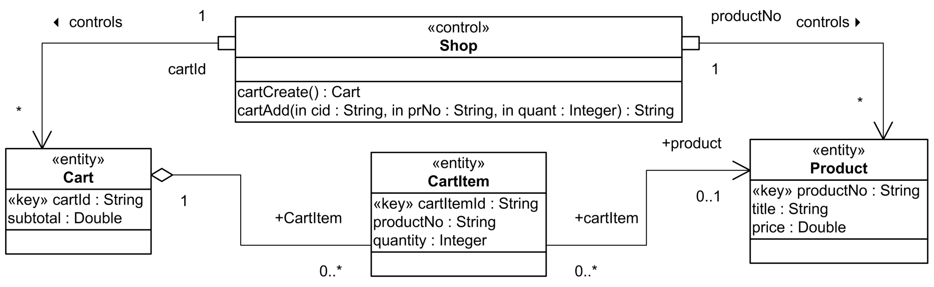


Fig. 3. Class Diagramm specifying static structure of online shop

UML class diagrams are used to represent the *static view* of a system specification. Figure 3 shows the class diagram of the sample online shop. We use the stereotypes `control` and `entity` as introduced in the Unified Process [10]. The stereotype `key` indicates a unique identifier for each of a set of objects of the same type. Qualifiers on associations (e.g. `productNo` on the association `controls` between `Shop` and `Product`) designate an attribute of the referenced class and provide direct access to a specific object.

The *functional view* of the system is described by visual contracts (i.e., graph transformations) for a selected set of operations. It integrates static and dynamic aspects to describe the effect of an operation on the data state of the system. Therefore, visual contracts take an operation-wise view on the internal behavior.

On the functional level a designer has different degrees of freedom to decide how detailed a model is. At first a designer can decide which of the operations to specify by visual contracts. If an operation is not detailed by a visual contract then the only consequence is that the operation is not monitored at runtime.

Further, if a designer describes an operation by a visual contract, she has the freedom to decide how detailed the specification shall be. The less detailed an operation is specified by a contract, the more freedom has a developer in implementing an operation. This is possible due to the assumption that the contracts are an incomplete description of the system state changes by an operation. A contract only specifies what at least has to happen, but it allows a developer to implement additional effects. For example, the implementation of the visual contract of Fig. 4 can additionally calculate the total costs of a cart and assign this value to the attribute `subtotal` of `Cart`. This interpretation is supported by the loose semantics of open graph transformation systems [6].

Structurally, a visual contract consists of two graphs, representing the pre-condition and the post-condition, respectively, like the left- and a right-hand side of a graph transformation rule (compare Fig. 4). The graphs are visualized by UML composite structure diagrams. Each of the diagrams is typed over the design class diagram.

Additionally, we may extend the pre- or post-condition of a visual contract by negative pre-conditions (i.e., negative application conditions [11]) or respectively

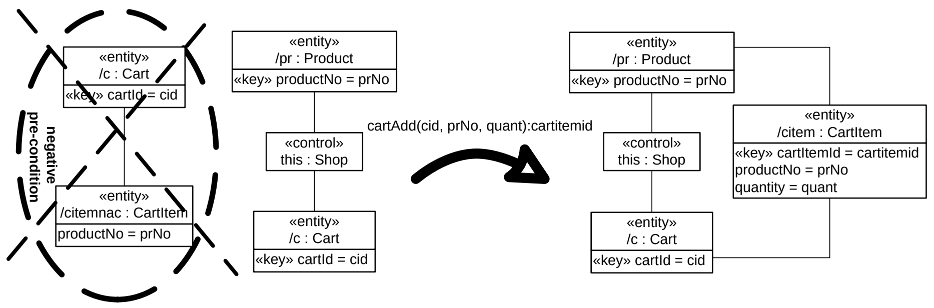


Fig. 4. Visual contract for operation `cartAdd`

by negative post-conditions. A slashed ellipse marks them. The negative pre-condition specifies object structures that are not allowed to be present before the operation is executed. The negative post-condition identifies object structures that are not allowed to be present after the execution of the operation.

Beside the different graphs, a visual contract contains the operation name, a parameter-list and a return-result. The variables of the parameter-list and the return-result are used in the visual contracts to further qualify the objects.

The visual contract in Fig. 4 specifies the operation `cartAdd`. This operation adds a new `CartItem`, which references an existing `Product`, to an existing `Cart`. The variables of the parameter-list and the return-value are used to determine values of attributes of different objects. For a successful execution of the operation, the object `this` must know two objects: 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 actual argument values are bound when the client calls the operation. The `Cart` object is reused in the negative pre-condition (compare object identifiers). 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 Transformation of the Design Model to Java Code with JML Assertions

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 [12, 13]. Here we describe the transformation more generally and from a methodical perspective and explain the formalization by graph transformation rules which underlies the transformation.

4.1 Transformation of Class Diagrams to Java

Each UML class is translated to a corresponding Java class. Attributes and associations are complemented by the corresponding access methods (e.g., `get`, `set`). 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.

4.2 Transformation of Visual Contracts to JML

For operations that are specified by a visual contract, the transformation of the contract to JML yields a Java method declaration that is annotated with a JML assertion. 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 (compare [14]) is applied to find an occurrence of the pattern that is specified by the pre- or post-condition in the current system data state. The search starts from the object `this` which is executing the specified behavior. If the JML pre-condition or post-condition finds a correct pattern, it returns true, otherwise it returns false.

4.3 Specifying the Contract Transformation

After demonstrating the transformation in principle, we explain in the following how we have defined a precise specification of the transformation from visual contracts to JML. The declarative specification in [12] abstracts from representation details of the visual contracts and leaves out different details of the mapping between visual contracts and JML. In contrast, we present an operational specification of the transformation from visual contracts to JML here. The provision of the operational model transformation is the prerequisite for an automated translation of visual contracts to JML as implemented in our development tools.

The operational specification is the second application of graph transformation concepts in our method towards model-driven monitoring. Other than the first application for specifying visual contracts that state minimal requirements on a single horizontal modeling level, we need a complete specification of the transformation behavior to support the automation of the model transformation in the vertical direction. The operational specification is based upon an extension of the UML 2 metamodel for visual contracts. The metamodel represents the source language of the model transformation and provides the type graph on which the graph transformation rules operate, i.e., the graph transformation rules are specified on the metamodel level, and the concrete models are viewed as metamodel instances when they are transformed.

4.4 Extended UML 2 Metamodel

Our visual contracts integrate with the UML 2 metamodel. Mainly we use elements from the UML 2 metamodel packages *InternalStructures* and *Collaborations*. The *InternalStructure* subpackage provides mechanisms for specifying structures of interconnected elements, representing runtime instances, which collaborate over communication links to achieve some common objectives. A *collaboration* represents how elements of the model cooperate to perform some essential behavior. Among others, the participating elements may include classes and objects, associations and links as well as attributes and operations. Collaborations allow us to describe only the relevant aspects of the cooperation of a set of instances by identifying the specific roles that the instances will play.

Figure 5 provides a view on the metamodel for our visual contracts. *VisualContract* specializes *Collaboration*. A collaboration defines a set of cooperating entities to be played by instances (its roles) as well as a set of connectors that define communication paths between the participating instances. The roles are represented by *ConnectableElement*s, which are referenced by a *Collaboration*. *ConnectableElement* is a *TypedElement*, which references a *Type*. *Class* is a *Classifier*, which is a *Type*. Consequently, the *ConnectableElement* can define a role that classes have to play in order to accomplish the behavior of a collaboration (visual contract, respectively). *ConnectableElement*s are linked by a *Connector* with *ConnectorEnd*s. A *Connector* specifies a link that enables communication between two or more instances. This link may be an instance of an association. In contrast to associations, which specify links between any instance of the associated classifiers, connectors specify links between instances playing the roles of the connected parts only. Additionally, the UML 2 metamodel offers specializations of *ConnectableElement* for representing parameters and variables.

We also define attribute values that an instance must provide in order to play one of the defined roles. According to the UML metamodel, you cannot specify the content of the features (properties) of a role in more detail. Therefore, we have introduced a specialization of a *ConnectableElement* named *VCElement* and a class *Constraint* to restrict possible attribute values. The class *Constraint* groups a feature (which represents an attribute of a class) and a permitted value. The permitted value of a feature can be a simple value (*ValueSpecification*) or another *VCElement*. Since the value of a feature can change from the pre- to the post-condition, we distinguish in the meta-model by association whether the reference value belongs to the pre- or post-condition.

We have to define whether a *VCElement* is part of the pre- or post-condition. To specify the absence of certain structures, both pre- and post-conditions may

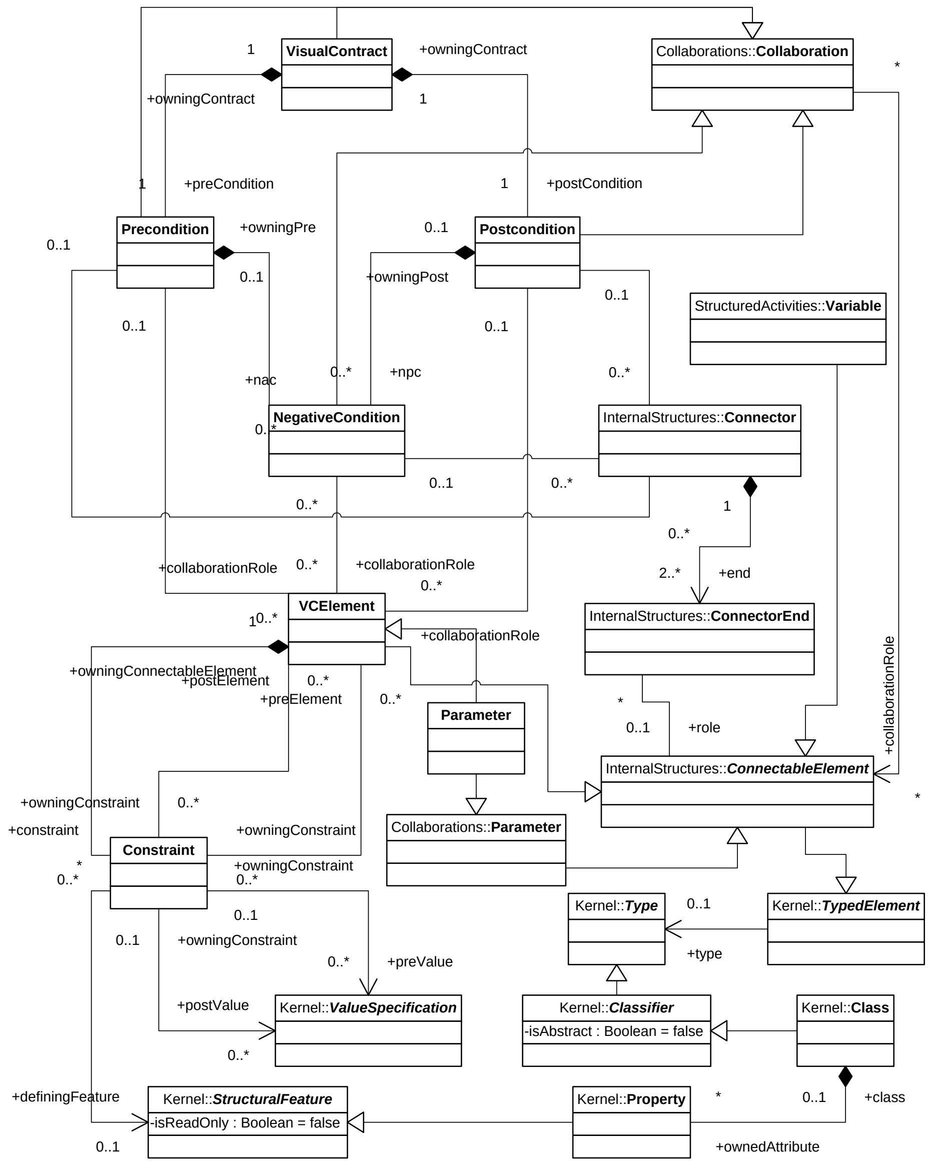
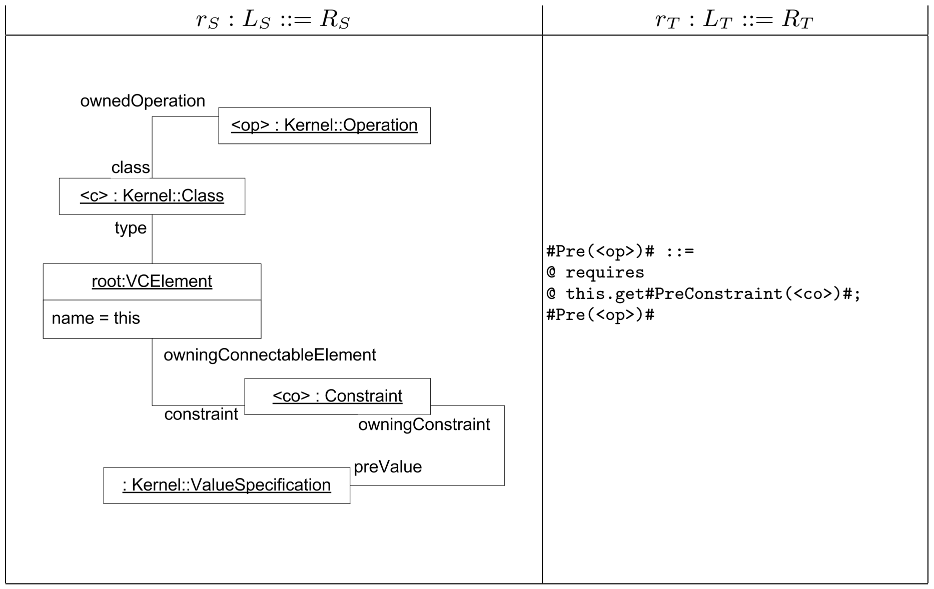


Fig. 5. Extract of the UML 2 metamodel extension for visual contracts



$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

```
#Pre(<op>)# ::=
@ requires
@ this.get#PreConstraint(<co>)#;
#Pre(<op>)#
```

Fig. 6. Compound rule that starts the generation of JML assertions for checking attribute values of object *this*

contain negative conditions. Therefore, we have added three metamodel classes to the UML metamodel: *Precondition*, *Postcondition*, and *NegativeCondition*.

4.5 Operational Transformation with Compound Rules

For the operational specification of our transformation from visual contracts to JML, we assume that the source model is syntactically correct according to our metamodel. We define the transformation by a set of compound rules as introduced in [15].

The basic idea of compound rules is that a model transformation from a source language to a target language can be defined by a synchronized model transformation on the source and the target language. Such a synchronized model transformation can be specified by a set of model transformation rules, consisting of two parts for transforming both the source and target model.

Figure 6 depicts a sample compound rule that starts the generation of the JML assertions for checking the attribute values of the object `this` in the precondition. A compound rule $r : (r_s, r_t)$ consists of two parts, a UML part and a JML part. Both r_s and r_t can be viewed as graph transformation rules. In general, the source transformation rule $r_s : L_S ::= R_S$ describes the transformation of the source model, the target transformation rule $r_t : L_T ::= R_T$ specifies the transformation of the target model. Note that in Fig. 6 r_s is an identical transformation with $L_S = R_S$, which is visualized by the left-hand side only. Source and target rules are coupled by the ability of using shared variables. Such variables are denoted by $\#variable\#$.

When applying a compound rule for the transformation of a source to a target model, at first an occurrence of the left-hand side L_s of the source transformation rule is searched within the source model (source match). In Fig. 6 the left-hand side of the source rule matches, if a `VCElement` (part of a visual contract added to an operation) `this` has a constraint with a value specification. If a source match is found, the variables are instantiated. This means, that a value is assigned to each variable according to the source match. Then, an occurrence of the left-hand side of the target transformation rule L_T (using the instantiated variables—in our example there is only one variable `op`) is searched within the target model (target match). Then the target match is replaced by the right-hand side R_T of the target transformation rule. In our example, the target transformation rule prepares the code for testing the content of an attribute of the object `this`.

In order to specify model transformations with control, the approach in [15] provides support for assembling compound rules into transformation units. Such units consist of a set of compound rules with control. Each compound rule is contained in a rule set. The rule sets are then organized in a sequence of rule sets where each rule set can be considered as a layer. Within a rule set, rules may be applied non-deterministically. A transformation unit consists of a set of compound rules together with a control expression specifying the organization of rules into rule sets, layers and determining whether a rule should be applied once or as long as possible.

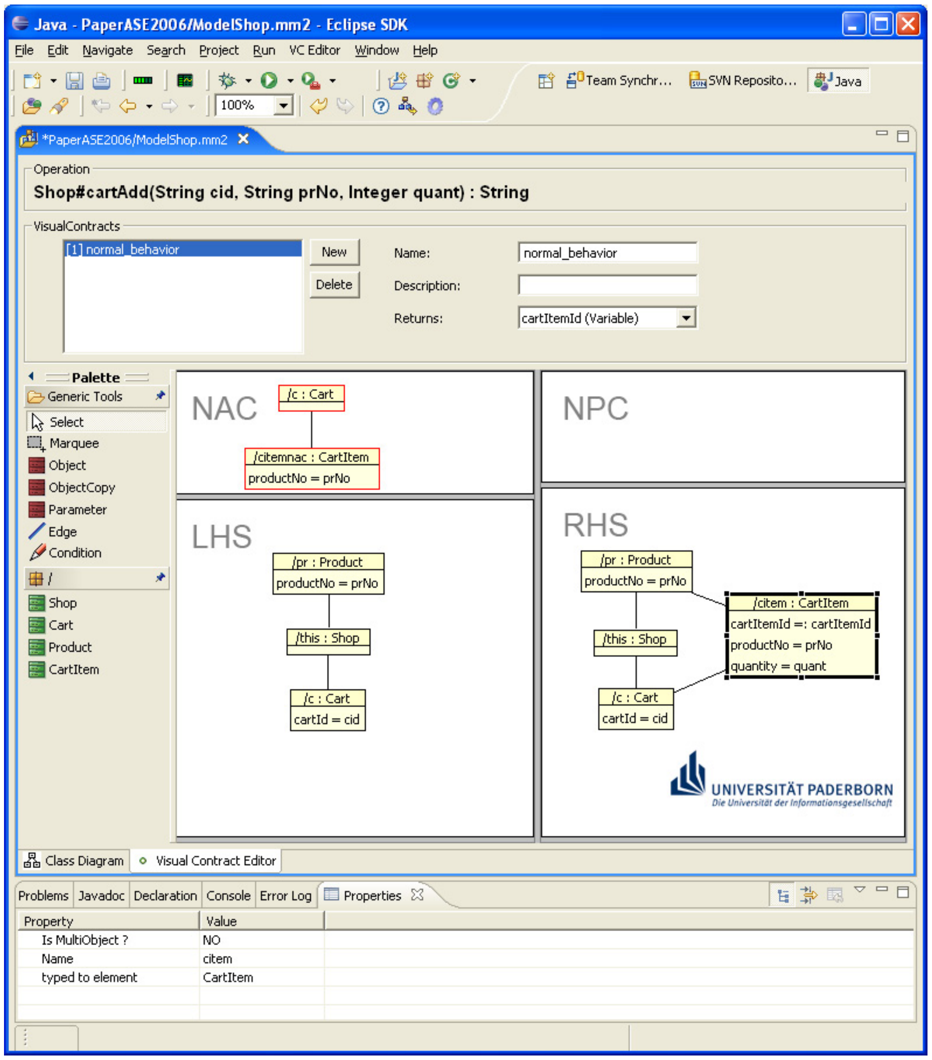
For defining the transformation of our models consisting of class diagrams and visual contracts to Java classes and JML we need round about 95 compound rules. These compound rules have to be organized in approximately 25 transformation units.

5 Tool Support

In the previous sections, we have shown how to use visual contracts in models of software systems for specifying operations and how to translate visual contracts to JML. This enables model-driven monitoring. We can monitor the correctness of a manual implementation with respect to its specification.

Existing CASE tools or graph transformation tools do not support the use of visual contracts for specifying software systems as described in our approach. As a proof of concept and for showing the practical feasibilty of our approach, we have developed an integrated development environment for using visual contracts in a software development process. This development environment allows software developers 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 manual implementation to get a complete application, and the compilation of the generated assertions by a JML compiler.

Figure 7 shows the user interface of our tool for modeling visual contracts. The central workspace of the visual contract editor is divided into four sectors.



Operation
Shop#cartAdd(String cid, String prNo, Integer quant) : String

VisualContracts
[1] normal_behavior

Name: normal_behavior
Description:
Returns: cartItemId (Variable)

Property	Value
Is MultiObject ?	NO
Name	citem
typed to element	CartItem

Fig. 7. Tool support for modelling visual contracts

A software designer can specify the pre- and post-condition in the bottom-left sector (labeled with *LHS*) and the bottom-right sector (labeled with *RHS*), respectively. An object `this` (the active object executing the operation) is added automatically in both sectors when a new visual contract for an operation is created. Every object added to the pre- or post-condition must be within reach of the object `this` by links. Additionally, the top sector allows for specifying negative conditions. The top-left sector (labeled with *NAC* for negative application condition) is for specifying object structures that are not allowed to be present before the operation is executed. The top-right sector (labeled with *NPC* for negative post-condition) is for specifying object structures that are not allowed to be present after the execution of the operation.

The development environment is implemented as an Eclipse plug-in. We mainly used the Graphical Editor Framework (GEF) [16] and the Eclipse Modeling Framework (EMF) [17] for the implementation of the plug-in. The code generation was implemented using Eclipse JET [18], which is a part of the EMF.

6 Conclusion

We have shown in this paper, how we have been employing results from research on graph transformation in model-driven software development processes. Addressing horizontal modeling issues, we have lifted the Design by Contract idea to the visual model level. Visual contracts use graph transformation concepts for the specification of pre- and post-conditions of operations. Since they only define minimal requirements towards the implementation of an operation, we use the loose semantics of graph transitions of the double-pullback approach.

For the vertical direction of model transformations, we use compound (graph transformation) rules to define a transformation of our visual contracts to the Java Modeling Language JML, a Design by Contract extension for Java. To automate this model transformation, we need the strict semantic interpretation of graph transformation rules as formalized by the double-pushout approach.

Altogether, we have introduced model-driven monitoring as a new and practically useful amalgamation of graph transformation and Design by Contract concepts. In contrast to the automatic generation of function code, we generate assertions from contracts that are monitored and automatically checked while the actual and manually implemented function code is executed.

To support our model-driven monitoring method, we provide an editor that allows developers to coherently model class diagrams and visual contracts. The editor is complemented by code generation facilities for Java classes with JML assertions for their operations.

In an industrial case study [9, 8], we have successfully applied visual contracts for specifying the interfaces of Web services. Our method and tools are currently considered by an industrial partner software company of the Software Quality Lab (s-lab) for deployment in their software development projects.

References

1. Meservy, T., Fenstermacher, K.D.: Transforming software development: An MDA road map. Computer **38**(9) (2005) 52–58
2. OMG (Object Management Group): Request for proposal: Mof 2.0 query / views / transformations rfp (2002)
3. Meyer, B.: Applying "Design by Contract". IEEE Computer **25**(10) (1992) 40–51
4. Meyer, B.: Eiffel: The Language. second printing edn. Prentice-Hall (1992)
5. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Department of Computer Science, Iowa State University (2005)
6. Heckel, R., Ehrig, H., Wolter, U., Corradini, A.: Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. APCS (Applied Categorical Structures) **9**(1) (2001) 83–110
7. Ehrig, H., Pfender, M., Schneider, H.: Graph grammars: an algebraic approach. In: 14th Annual IEEE Symposium on Switching and Automata Theory, IEEE (1973) 167–180
8. Engels, G., Güldali, B., Juwig, O., Lohmann, M., Richter, J.P.: Industrielle Fallstudie: Einsatz visueller Kontrakte in serviceorientierten Architekturen. In Biel, B., Book, M., Gruhn, V., eds.: Software Enginneering 2006, Fachtagung des GI Fachbereichs Softwaretechnik. Volume 79 of Lecture Notes in Informatics., Köllen Druck+Verlag GmbH (2006) 111–122
9. Lohmann, M., Richter, J.P., Engels, G., Güldali, B., Juwig, O., Sauer, S.: Abschlussbericht: Semantische Beschreibung von Enterprise Services - Eine industrielle Fallstudie. Technical Report 1, Software Quality Lab , Unversity of Paderborn (2006)
10. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley Professional (1999)
11. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26**(3,4) (1996) 287–313
12. 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
13. 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.
14. Zündorf, A.: Graph pattern matching in progres. In Cuny, J., Ehrig, H., Engels, G., Rozenberg, G., eds.: 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science. LNCS 1073 (1996)
15. Heckel, R., Küster, J.M., Taentzer, G.: Towards automatic translation of UML models into semantic domains. In Kreowski, H.J., Knirsch, P., eds.: Proceedings of the Appligraph Workshop on Applied Graph Transformation. (2002)
16. Eclipse Consortium: Eclipse graphical editing framework (GEF) - version 3.1.1. http://www.eclipse.org/gef/ (2006)
17. Eclipse Consortium: Eclipse modeling framework (EMF) - version 2.1.2. http://www.eclipse.org/emf/ (2006)
18. Eclipse Consortium: Java emitter templates (JET). Eclipse Modeling Framework (EMF) - Version 2.1.1, http://www.eclipse.org/emf/ (2006)