Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Model-Driven Development of Reactive Information Systems

From Graph Transformation Rules to JML Contracts

Reiko Heckel[1], **Marc Lohmann**[2]

[1] Department of Computer Science, University of Leicester, United Kingdom
e-mail: reiko@mcs.le.ac.uk
[2] Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Germany
e-mail: mlohmann@uni-paderborn.de

Received: date / Revised version: date

Abstract. The Model-Driven Architecture focuses on the evolution and integration of applications across heterogeneous platforms by means of generating implementations from platform-independent models. Most of the existing realizations of this idea are limited to static models.

We propose a model-driven approach to the development of reactive information systems, like dynamic Web pages or Web services, modeling their typical request-query-update-response pattern by means of graph transformation rules.

Rather than generating executable code from these models we focus on the verification of the consistency between different sub-models and an implementation that may have been produced manually. The main technical tool for achieving this goal is a mapping of graph transformation rules to contracts expressed in the Java Modeling Language (JML).

1 Introduction

Most business applications developed today depend on a specific middleware platform providing services for communication, persistence, security, etc. while supporting interoperability across different kinds of hardware and operating systems. If such systems have to interact over the Web, for example to provide integrated services, we face the problem of the interoperability of these platforms. Solutions at different levels have been proposed to overcome this problem.

At implementation level, standards like Web Services provide a collection of languages and protocols to support interoperability by interchanging XML-documents representing, e.g., remote procedure calls. At the level of design, the OMG has proposed the Model-Driven Architecture (MDA) [35,36] to achieve interoperability among models. Starting from standard UML models [29] specifying the intended functionality, the MDA approach is largely concerned with the vertical structure of the mappings required to implement this functionality on any given platform. The idea is to distinguish between *platform-independent models* (PIMs) that are refined into *platform-specific models* (PSMs) which carry annotations for the generation of platform-specific code.

The idea seems realistic enough for mappings to data type or interface definition languages like SQL DDL, XML Schemata, or CORBA IDL, and for static aspects of Java and EJB, thanks to the relative simplicity and stability of these more mature languages. Siegel [34], for example, describes the use of MDA for applications based on Web services, focusing on the integration of service interfaces in applications developed with the MDA approach. However, the integration does not take into account dynamic information about how the services should be used.

Other approaches, mostly based on statecharts [14], focus on reactive behavior, ignoring the aspect of data states and their evolution. In this paper, we are interested in what we call *Reactive Information Systems*, i.e., applications like Web services or HTML-based interactive Web applications, which follow the basic *request-query/update-response* pattern. An external request is answered (or causes further requests) in combination with a non-trivial query and update of the data state of the application or its associated database. Thus, data state transformations are an important aspect of our models.

Our approach to integrate data state transformations and reactivity at the level of models is based on *graph transformation rules*. Such a graph transformation rule is a transformation rule on the internal object structure of the server which is applied in response to a *request*, i.e., an incoming operation call or event corresponding

to the name and parameters of the rule. This rule-based way of specifying reactive behavior goes back to object-oriented graph transformation approaches, like the one used in the FUJABA CASE tool [1]. In this approach, graph transformation rules represent method implementations from which stand-alone and fully executable code can be generated.

This complete implementation of the MDA paradigm, however, is not always realistic. A standardized mapping of behavioral models to implementations could be too inflexible, especially when project- or enterprise-specific technologies, libraries, patterns, or coding conventions have to be used. Moreover, automated generation of methods requires a fairly complete and low-level model to start with. The production of such low-level models can become very expensive, e.g., when specialized experts have to be employed. Finally, performance issues may create further obstacles.

Hence, for the near future, we do not expect that method implementations are generated fully automatically from models. Rather, in order to verify the consistency of models with manually derived code, we aim to support the model-based monitoring and testing of implementations by translating graph transformation rules to contracts in JML, the Java Modeling Language [5, 26]. Thus, in contrast to Fujaba, which facilitates a rule-based specification of system behavior, our behavior specification with graph transformation rules is rather declarative.

In the following sections we show how graph transformation rules are embedded into a typical development process of reactive information systems, starting from requirements expressed in terms of use cases and sequence diagrams in Section 2 via architectural and detailed design in Section 3 to the integration with platform-specific code in Section 4. Much of this is standard, except for the way in which graph transformation rules are used, but we intend to demonstrate that, beside the technical advantages graph transformations have to offer (and which are discussed in detail later in the paper) they also fit into mainstream development processes. The technical core of the paper is Section 5 where we describe how to generate JML contracts from our models. In Section 6, we discuss in more detail some applications of these contracts to the vertical consistency problems arising between the implementation and models at different levels.

2 Requirements Specification

Following the Unified Process [25], the functional requirements of Web-enabled applications are captured as use cases where interesting scenarios are detailed by means of sequence diagrams.

As a running example throughout the paper we use the model of an online shop. As shown by the use case diagram in Figure 1, a client of the online shop can *query products*, *order a product*, or *pay for an order*. The use case diagram structures the fundamental actor's goals, while abstracting from necessary subtasks, of Web-application and user to reach these goals.

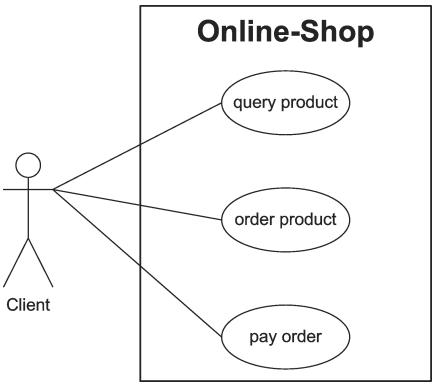

Fig. 1. Use case diagram of the shop example

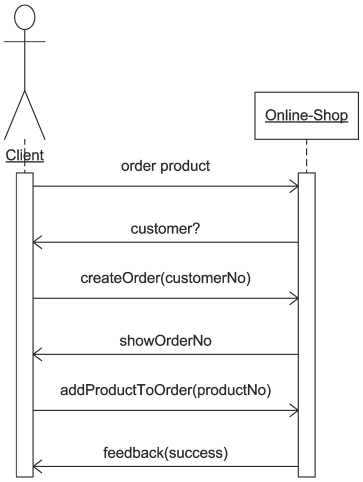

Fig. 2. Sequence diagram describing one scenario for the use case *order product*

Sequence diagrams are used to model scenarios for use cases in a more formal way as sequences of messages exchanged, in our example, between the client and the online shop. Variants can be expressed by different sequence diagrams associated with the same use case. Figure 2 shows one scenario detailing the use case *order product*. The client who triggers the use case is asked by the online shop to enter his customer number for identification. Then the online shop sends information about the actual order to the client. The user can add a product to the actual order by transferring a product number to the online shop. The product is added to the order and client gets a positive feedback.

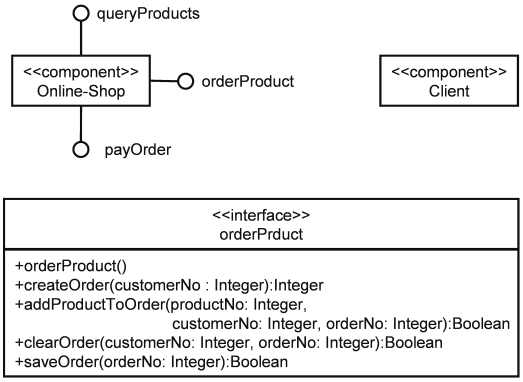

Fig. 3. Components of the online shop example

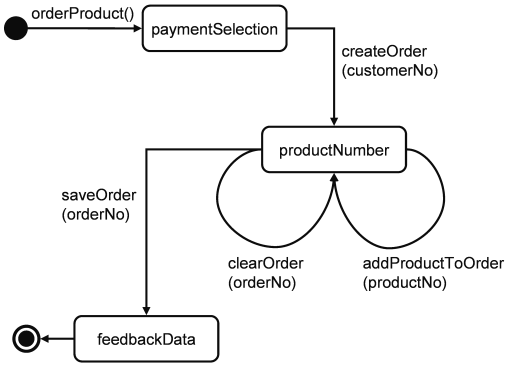

Fig. 4. Protocol statechart for the online shop component

3 Platform-Independent Design

The requirements specification presented in Section 2 focuses on specific scenarios of the externally visible behavior of the application. In this section, we will complement this view, first moving from specific scenarios to general specifications and then from external requirements to the internal realization. The first step, called *architectural design*, describes the involved components and their possible interactions. This model is refined in the *detailed design* adding internal data state transformations.

3.1 Architectural design

The structural view of the architectural design describes the components and interfaces of the online shop. We provide an interface for every use case, which is later implemented by a control class to execute the use case. Figure 3 shows the two components of our online shop system. The online shop itself is a component with three interfaces for the three use cases in the diagram of Figure 1. For each interface we can list the operations of the online shop that can be called by a client through that interface. In Figure 3 we have only detailed the interface for the use case *order product*. This interface contains operations for creating a new order for existing customers, adding products to an existing order, removing all products from an order, and for saving an order.

Whereas sequence diagrams are used to describe single scenarios from a global point of view, protocol statecharts are used to specify the sequences of requests individual components are willing to accept. Figure 4 shows the statechart for the interface `orderProduct` corresponding to the use case *order product*.

3.2 Detailed design

After having described components from an outside perspective, in this section their data structures and computations are modeled.

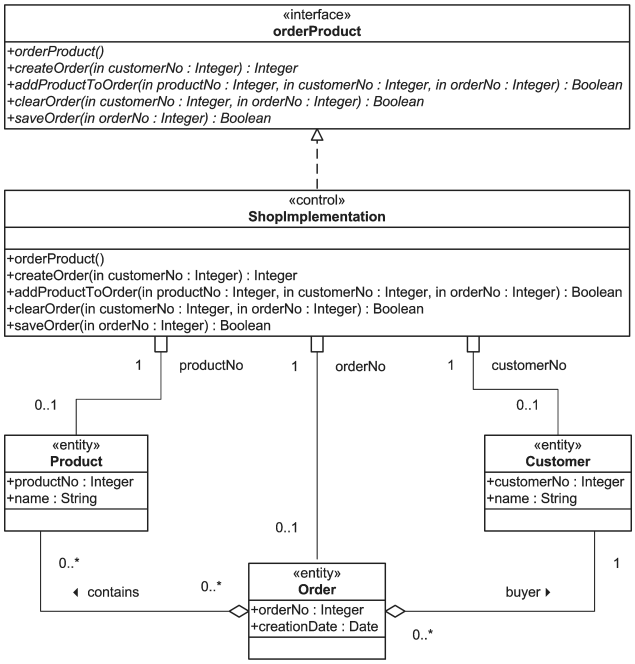

Fig. 5. Platform-independent class diagram for the online shop component

Class diagrams are used to represent static aspects. Figure 5 shows the result of detailing the use case *order product*. We use the stereotypes `control` and `entity` as introduced in the Unified Process [25]. Each of these stereotypes expresses a different role of a class in the implementation. Instances of control classes coordinate other objects, encapsulating the control related to a specific use case. Entity classes model long-lived or persistent information. The control class `ShopImplementation` implements the interface `orderProduct`. The control class is connected to the entity classes of the system via qualified associations. A qualified association defines an attribute of the referenced class as a qualifier and uses it as a key to get direct access to a specific object.

In addition to the static and dynamic diagrams of our models we now introduce a *functional view* integrating

the other two by describing the effect of an operation on the data structure of the system. This requires moving the focus both from sequences to single operations and from the externally visible behavior to its internal realization. Therefore, we take an operation-wise view on the internal behavior.

To describe the effect of an operation we use graph transformation rules as shown in Figure 6. Structurally, a graph transformation rule consists of two graphs, the left- and the right-hand side, here visualized by UML object diagrams. Both graphs are typed over the design-level class diagram, see e.g., [15] for technical details. The basic intuition is that every object or link which is only present in the right-hand side of the rule is newly created, and every object or link which is present only in the left-hand side of the rule is deleted. Objects or links which are present in both sides are not affected by the rule. If there is only one object of a given type, it can remain anonymous. Otherwise, different objects should be distinguished by names, separated from their type by a colon. Beside the two graphs, transformation rules specifying operations of classes include the name of the operation as well as a list of parameters and a return value. The parameter list is an ordered set of variables, while the return value is a single variable. These variables can be used in the graph transformation rule as place holders for attribute values or object identities.

Let us have a closer look at the example rule in Figure 6 describing the behavior of the operation `createOrder`. The link between the objects on the left-hand side of this rule requires that the object `this:ShopImplementation` references an object `c:Customer` whose value for the attribute `customerNo` is given as specified by the input parameter `cNo`. That means, only existing customers can execute this operation. As a result, the operation creates a new object of type `Order` and two links, between the objects `this:ShopImplementation` and `o:Order` as well as between `o:Order` and `c:Customer`. Additionally, the value of the attribute `orderNo` of `o:Order` becomes the return value of the operation `createOrder`, as indicated by the variable `oNo`.

The benefit of this form of specification is the formal integration of static models, in the form of class diagrams, providing the types of the objects within rules, and dynamic models (in the form of statecharts specifying the possible order of invocations of the operations that are specified by the rules). Due to this detailed description, transformation rules allow for automatic code generation [12] as well as model-based testing [17,18].

The idea of specifying an operation in terms of pre- and post-conditions is known from Design by Contract [28]. Contracts are an design technique sharing similarities with establishing a legal contract, i.e., the relationship between a class and its clients is viewed as a formal agreement, expressing each party's rights and obligations. At the level of components, developers use

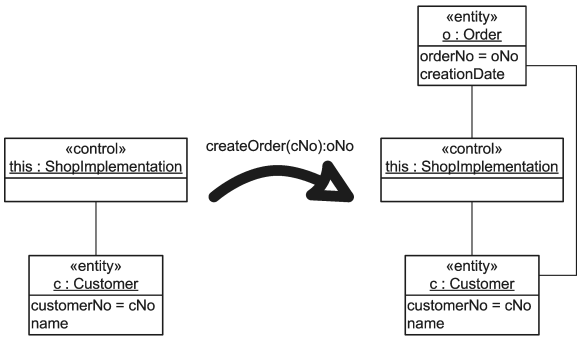

Fig. 6. Graph transformation rule describing internal behavior of operation *createOrder*

design by contract to specify interfaces. The contract describes what the component expects of its clients and what clients can expect of it, independently of how it is accomplished.

Figure 7 shows a more complex graph transformation rule specifying the operation `addProductToOrder`. This operation adds a product to an existing order. For a successful execution of the operation, the object `this:ShopImplemenation` must know three different objects with the following characteristics: an object of type `Customer` which has an attribute `customerNo` with the value `cNo`, an object of type `Order` which has an attribute `orderNo` with the value `oNo`, and an object of type `Product` which has an attribute `productNo` with the value `prNo`. The concrete argument values are bound when the client calls the operation. If these objects are found, a further prerequisite is that there is a link between the objects `c:Customer` and `o:Order`. The negative application condition additionally requires that the object `p:Product` must not be previously connected to the order. As a result, the operation adds the identified object `p:Product` to the order (by adding a link). Additionally, we can support negative application conditions on the right-hand side of a rule to express that some structure must not exist after the application of the rule.

Universally quantified operations, involving a set of objects whose cardinality is not known at design time, can be modeled using multiobjects. An example is shown in Figure 8. This graph transformation rule specifies an operation which removes all products from an existing order. The multiobject `p:Product` in the pre-condition indicates that the operation is executed if there is a set (which maybe empty) of objects of type `Product`. After the execution of the operation, all objects conforming to `p:Product` (as well as the corresponding links) are deleted, i.e., the order is cleared.

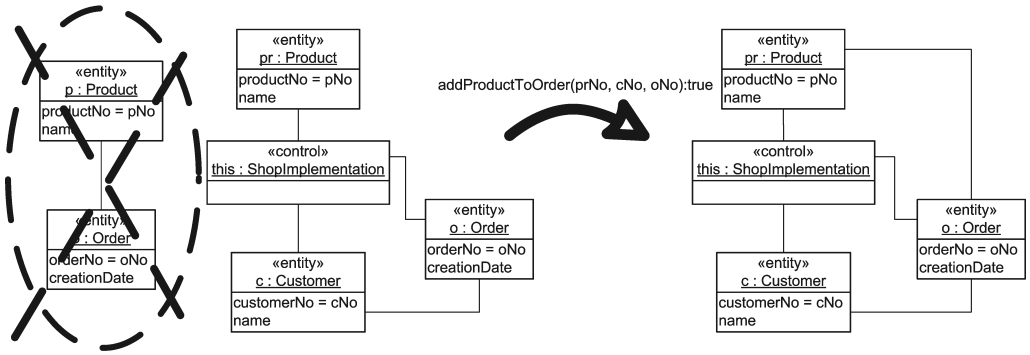

Fig. 7. Graph transformation rule describing internal behavior of operation *addProductToOrder*

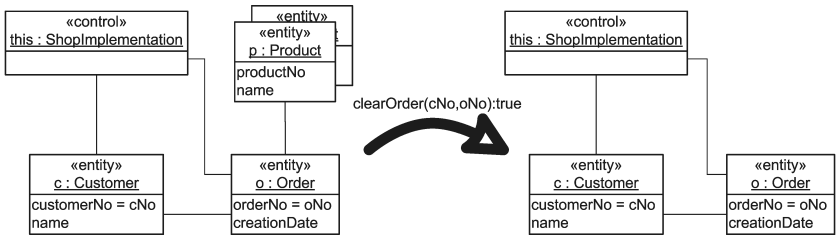

Fig. 8. Graph transformation rule describing internal behavior of operation *clearOrder*

4 Integrating Platform-Specific Aspects

In the next two subsections, we show how to map the platform-independent models developed in the previous sections on platforms like HTML or SOAP which realize the request-query/update-response pattern. The aim is to deploy code generated from platform-independent models on a specific middleware while keeping it separate from platform-specific code.

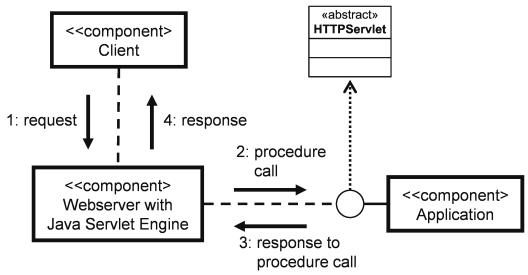

Fig. 9. Using a servlet (abstract overview)

4.1 Platform-specific design

In most reactive information systems, some middleware serves as a link between clients and back-end services. This middleware is normally responsible for the provision of basic information and communication services, like a Web server implementing the Java Servlet technology [37] to support an HTML-based application or a SOAP server to implement a Web service [38]. In both cases, clients request are filtered through the middleware via pre-defined interfaces with the application.

Figure 9 shows an abstract overview of the operation of applications realized with the Java Servlet technology. A user submits HTML form data to the server (1). The server locates the requested application class, which must implements the Servlet interface, and invokes a method to handle the request (2). The application processes the data and calculates a response (3). The Servlet engine transfers this response back to the client (4). The Servlet interface declares methods to manage the servlet and its communications with clients. These methods have to be implemented by the application class. Other Internet-based communication platforms, like the SOAP implementation of Apache Axis [31], work in a similar way.

4.2 Implementation

In this section we describe the integration of the application logic generated from the platform-independent model with the platform-specific technology.

Figure 10 shows the integration for the Java Servlet technology. Classes that are specific to the technology are shown in grey. The new platform-independent abstract class `StatemachineHandler` has the task of an object controller. For every use case, an implementation for the abstract method of this class is generated implementing the protocol statechart of the corresponding use case by filtering incoming requests according to the ordering specified. As a result, the methods of the class `PayOrder` are never called directly. Instead, every time a method of the class `PayOrder` has to be called, the

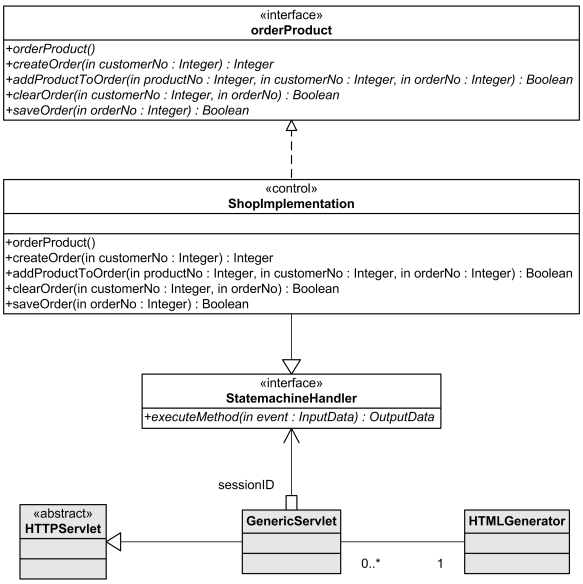

Fig. 10. Integration of PIM and PSM on implementation level

method `executeMethod` is called, which calls the correct method depending on the current state and the the incoming data. This design follows the *Command* design pattern [13] in order to reduce the dependencies between the sender and the receiver, thus allowing their independent development.

On the platform-dependent side the elementary class `GenericServlet`, which has to be implemented by a software developer, inherits from an abstract class `HTTPServlet`, to allow for integration in a Java Servlet-based server. This platform-dependent class calls the platform independent classes realizing the application logic. The class `GenericServlet` is called when a client has submitted a form. It calls the method determined by the protocol statechart. Auxiliary data encoded into the form provides control information like `sessionID`s to obtain the correct statemachine object for each request from a client.

Figure 10 shows another platform-specific class, which has to be implemented manually. The `HTML-generator` is responsible for the creation of the HTML forms sent to the client in response to their requests.

To use alternative platforms or other kinds of user interfaces, only the platform specific classes of Figure 10 have to change. For example, to use XForms [39], a new technology more powerful than HTML forms and based on XML, only the `HTML-Generator` has to be changed to create this new kind of user interface. Further you have to ensure, that the middleware calling the servlet is able to evaluate XForms. For other middleware, one may have to replace the class `GenericServlet` by another class which implements the required interface of the technology.

5 Generation of executable contracts

Structural aspects of Java programs, like interfaces, classes, fields, and signatures of methods, can be generated from class diagrams by most commercial CASE tools. A programmer can complete the implementation by filling in the behavioral code between the generated Java fragments. Ideally, this code will conform to the behavioral design given by the graph transformation rules by creating all the conditions and effects that are specified. However, we do not assume that the specifications are complete, additional effects are allowed on public classes, as well as new classes and methods.

Semantically, this assumption coincides with a loose interpretation of graph transformation rules which are seen as minimal descriptions of the data state transformation to be implemented [16]. Thus, a graph transformation specifies only what at least has to happen on a systems state, but it allows the programmer to implement additional effects. This loose interpretation is necessary to leave the opportunity for integrating additional aspects, like security checks, statistics about system usage, etc.

To validate the consistency of models with the manually derived code, we want to support the monitoring and testing of the behavior modeled by graph transformations. Therefore, we show in the following the transformation of our graph transformation rules into JML constructs.

The Java Modeling Language (JML) [5,26] is a behavioral specification language for Java classes and interfaces. The JML assertion language is based on Java expressions following the example of Eiffel [27]. However, JML is more expressive than Eiffel, supporting constructs such as universal and existential quantifications. The approach of JML is model-based like VDM and Z, which means that JML specifications can be given semantics in terms of mathematical models such as sets and relations. However, since these models are expressed in terms of the concepts of Java, programmers can deal with it at an intuitive level without having to learn another specification language. Thus, the same language concepts can be used throughout specification and implementation. Additionally, JML supports specification-only variables and model methods which enhance the expressiveness of specifications. Different tools are available to support, e.g., runtime assertion checking, testing, or static verification based on JML.

With the mapping of graph transformation rules into JML contracts these options become available to us, adding important new applications to graph transformation-based models.

The semantic idea of the mapping from graph transformation rules to JML is the following: each rule specifies a set of transitions according to its loose semantics, allowing for additional, unspecified effects. When we call a method m specified by a rule r while being in a state

```
public class Product {
  private String name;
  private int no;
  public String getName() {
    return name;
  }
  public int getNo() {
    return no;
  }
  public void setName(String string) {
    name = string;
  }
  public void setNo(int i) {
    no = i;
  }
}
```

Fig. 11. Fragment of Java class Product

s, the method must terminate successfully (without exception) if there exists a corresponding graph transition from s via r in the model. If the state resulting from the execution of m in the implementation is t, there must be a model transition from s to t via r, too. In terms of transition system this means that the implementation can have more transitions than the model, but whenever there is a model transition matching the implementation, they must agree on the successor state.

5.1 Translation of UML class diagrams to Java

Given a UML class diagram, each class is translated into a corresponding Java class. All private or protected attributes of the UML class diagram are translated to private and protected Java class attributes with appropriate types and constraints, respectively. According to the Java coding style guides [32], we translate public attributes of UML classes to private Java class attributes that are accessible via appropriate `get-` and `set`-methods. Standard types may be slightly renamed according to the Java syntax. Attributes with multiplicity greater than one map to a reference attribute of some container type. Further, each operation specified in the class diagram is translated to a method declaration in the corresponding Java class up to obvious syntactic modifications according to the Java syntax. If a class `B` is a direct subclass of a class `A` in the UML model then the Java class `B` is a direct subclass of the Java class `A`. A UML class `Product` representing products of our online shop is shown in Figure 5 with two attributes `productNo` and `name` representing the identification number and name of a product. Figure 11 shows a fragment of the corresponding Java code. The attributes are manipulated by the corresponding `get-` and `set`-methods (c.f. [11]).

Associations are translated by adding an attribute with the name of the association to the respective classes. For example, association `buyer` of Figure 5 is translated to a private variable `buyer` of type `Customer`. Again, appropriate access methods are added to the Java class. Because the UML association `buyer` is bidirectional, we additionally add an attribute named `revBuyer` to the class `Customer`. For associations that have multiplicities with an upper bound bigger than one, we use classes implementing the standard Java interface `Collection`. A collection represents a group of objects. Especially we use the class `TreeSet` as implementation of the subinterface `Set` of `Collection`. A set differs from a collection in containing no duplicate elements. For qualified associations, we use the class `HashMap` implementing the standard Java interface `Map`. An object of type `Map` represents a mapping of keys to values. A map cannot contain duplicate keys; each key can map to at most one value. For these container class attributes, we provide appropriate access methods that allow adding and removing elements. Examples are the access methods `addProduct` or `removeProduct`. To check the containment of an element we add operations like `hasProduct`. In case of qualified attributes, we access elements via keys by adding additional methods like `getProductByNumber`. As described in [11], in order to guarantee the consistency of the pairs of references that implement an association, the respective access methods for reference attributes call each other.

Based on this mapping of class diagram, below we start to develop the translations of rules into JML contracts. We proceed in two iterations.

5.2 From graph transformation rules to JML: First attempt

For the transformation of our visual contracts introduced in Section 3.2 to JML, we assume a translation of UML class diagrams into Java as described above. For each method specified by a graph transformation rule, we generate a corresponding method signature and JML specification of the pre- and post-condition as shown in Figure 12. The example shows a declaration of a Java method. Using JML it is possible to describe the interface of a method by its signature and the behavior of a method by pre- and post-conditons. The behavioral information is specified in annotation text (first four lines of method `m` in Figure 12). Due to the embedding into Java comments, the annotations are ignored by a normal Java compiler. The keywords `public normal_behavior` are used to specify that the specification is intended for clients and that when the pre-condition is satisfied a call must return normally, without throwing an exception [26]. Normally, in such a public specification, only names with public visibility may be used. However, JML allows the formulation of public specifications where private variables are regarded as public for specification purposes. JML pre-conditions follow the keyword `requires`, and post-conditions follow the keyword

```
public class A implements {

  ...

  /*@ public normal_behavior
    @ requires JML-PRE;
    @ ensures JML-POST;
    @*/
  public Tr m(T1 v1, ... Tn vn) {...}

  ...

}
```

Fig. 12. Template for specifying pre- and post-conditions by JML

```
generateJML(rulepart : ObjectGraph)
// rulepart is the pre- or post-condition
// of a graph transformation rule

set B := this;
// set of bound runtime objects
set U := rulepart/this;
// set of unbound objects
// representing one side of rule

while (U != empty) {
  Set temp = B;
  while (Temp != empty) {
    Object ob = Temp.removeNext();
    Set linkedObjects = ob.getLinkedObjects();
    while (linkedObjects != empty) {
      Object testLink = linkedObject.removeNext();
      if (U containsObjectOfType testLink) {
        Object uo = U.getObjectOfType;
        if (testLink isSimilarTo uo) {
          B.add(testLink);
        }
        U.remove(testLink);
      }
    }
  }
}
```

Fig. 13. Pseudo-code for translation of left-hand side of graph transformation rule into JML

ensures. JML-PRE and JML-POST represent Boolean expressions. The pre-condition states properties of the method arguments and the current state of the systems. If the pre-condition is true, the execution of the method must lead to a state satisfying the post-condition.

If in the class diagram a subclass A* of A redefines the method m with an additional contract described by a graph transformation rule, then the method m of the subclass A* must satisfy the pre- and post-conditions of both classes, i.e., it inherits the contract of the superclass. In JML this is indicated by adding the keyword also at the beginning of a JML construct. This is the basic structure to describe a contract between a caller of a method and the implementer of the method.

For a meaningful mapping of graph transformation rules, JML-PRE and JML-POST must interpret their left- and right-hand side, respectively. That means, evaluating a JML pre-condition should correspond to finding an occurrence of the left-hand side of the rule in the current system state. To find such an occurrence, a JML pre-condition (post-condition) implements a search starting from the object this, i.e., the one executing the method. If a JML pre-condition (post-condition) succeeds in finding an occurrence, it returns true, otherwise it returns false. Our algorithm for generating the JML assertions applies the same matching strategy for both pre- and post-conditions.

The algorithm for translating graph transformation rules into JML is shown in pseudocode in Figure 13. We distinguish between two disjoint sets B and U. Set B contains objects that have already been processed by the translation, generating the corresponding JML code. This set is initialized with the object this:ShopImplementation representing an instance of the class of the method called. The set U contains the objects that have not yet been processed by the algorithm. This set is initialized with all objects except for the object this. In every iteration of our algorithm we choose one object of the set B that is connected via a link to an object u of the set U. We check every outgoing link of u that is connected to an object in B. If u satisfies all attribute conditions as well as all conditions related to connections with objects in B, it is added to the set B and we proceed with the next iteration.

The efficiency of the conditions, in terms of the number of navigation steps required to evaluate them, depends on the order in which the objects are selected for testing. The algorithm described above abstracts from this ordering, leaving the choice of the next object to be matched non-deterministic. Heuristics suggest that in order to detect a mismatch as soon as possible, one should prefer objects that are referenced by to-one or qualified associations [40]. Such strategies have to be taken into account in an implementation of the algorithm, but do not affect its correctness.

Next we explain the translation in more detail by means of an example, the method addProductToOrder of Figure 7. The starting point for the creation of the pre-condition is the node this. Figure 14 shows how the algorithm traverses the object diagram of the rule's left-hand side. Next, we explain the code generated during theses steps.

In the first iteration, we have to consider the outgoing edges of the object this:ShopImplementation. At first, we test if there is an object of type Product with attribute value prNo for the attribute productNo. For this purpose, JML supports universal and existential quantifications, extending Java expressions by the logical operators \forAll and \exists. The general syntax of quantified expressions is

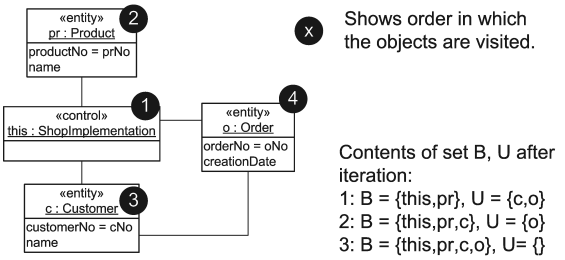

Fig. 14. Order in which algorithm traverses the pre-condition of Figure 7

```
/*@ public normal_behavior
  @ requires (\exists Product p;
  @                   product.contains(p);
  @                   p.getNo() == productNo);
  @*/
public boolean addProductToOrder(int productNo,
           int customerNo, int orderNo) { ... }
```

Fig. 15. First iteration

`(\forAll T x; r; p)` and `(\exists T x; r; p)`. The expression `(\forAll T x; r; p)` is true when every object `x` of type `T` that satisfies `r` also satisfies `p`. The expression `(\exists T x; r; p)` is true if there is at least one object `x` of type `T` that satisfies `r` and `p`. Using these expressions we get the JML expression of Figure 15. The keyword `requires` is followed by an `exist`-quantifier that is satisfied if there is at least one object of type `Product` that is reachable from the object `this:ShopImplementation` and has an attribute value of `prNo` for the attribute `productNo`. The object `pr:Product` is directly added to `B` because it does not contain any further outgoing edges.

In the second iteration, the algorithm creates the code for testing if there is an object of type `Customer` with an attribute value `cNo`. Again, this is achieved by creating a corresponding `exists`-expression. Nothing more has to be tested, because `c:Customer` does not contain any further outgoing links to objects in `B`. In the third iteration, the object `o:Order` has to be added. It is accessible from the set `B` by two links. Our algorithm takes the link starting from the object `this:ShopImplementation`, which identifies the object `o:Order` uniquely, due to the qualified association in the class diagram. Again, we create a corresponding `exists`-expression. However, in this iteration we require additional code because the object `o:Order` is connected to other objects in set `B`. In this case, the generated JML code tests if an object of type `Order` is linked to the object of type `Customer` that was found in the second iteration. After the JML code for the pre-condition is created, our algorithm creates the JML code for testing the negative application conditions. Figure 16 shows the entire code generated from the graph transformation rule of Figure 7. The last line of the pre-condition

```
/*@ public normal_behavior
  @ requires (\exists Product p;
  @           product.contains(p);
  @           p.getNo() == productNo);
  @           && (\exists Customer c;
  @           customer.contains(c);
  @           c.getNo() == customerNo
  @        && (\exists Order o;
  @        order.contains(o);
  @        o.getOrderNo() == orderNo
  @        && o.getCustomer() == c
  @        && o.containsProduct(p) == false)));
  @
  @ ensures  (\exists Product p;
  @            product.contains(p);
  @            p.getNo() == no &&
  @   (\exists Customer c;
  @     customer.contains(c);
  @     c.getNo() == customerNo
  @     && (\exists Order o;
  @          order.contains(o);
  @          o.getOrderNo() == orderNo
  @          && o.getCustomer() == c
  @          && o.getProduct().contains(p))));
  @*/
public boolean addProductToOrderWrong(
               int productNo,
               int customerNo,
               int orderNo) {
```

Fig. 16. Operation *addProductToOrder* with pre- and post-conditions

tests the negative application condition. It tests whether the object `o` knows the object `p`, i.e., if the product has already been added to the order. The creation of the post-condition from the rule's right-hand side follows the same algorithm.

5.3 Shortcomings of the algorithm and second attempt

The JML contract of Figure 16 interprets the left- and right-hand side of the rule in Figure 7 individually, each as a complex boolean expression with no relation between them. This is in contrast to the semantics of graph transformation rules, where variables for attribute values and object identifiers have a scope which includes both sides of the rule. JML's binding of universal and existential quantifications, however, does not ensure that the same objects are bound to the same variables in the pre- and post-condition. For example, in Figure 16, the object bound to the variable `p` in the pre-condition does not need to be the same as the one bound to the variable `p` in the post-condition—bindings of quantified variables are not reusable in the post-condition.

Fortunately, JML provides an alternative way of binding using the `old` expression. Figure 17 shows an example where the `old` clause is used like a `let`-expression evaluated in the pre-state. It adds a binding to a variable

```
/*@ public normal_behavior
  @
  @ old Product p = getProductByNo(productNo);
  @ old Customer c = getCustomerByNo(customerNo);
  @ old Order o = c.findOrderByNo(orderNo);
  @
  @ requires p != null;
  @ requires c != null;
  @ requires o != null;
  @ requires o.getCustomer() == c
  @        && o.containsProduct(p) == false;
  @
  @ ensures p != null;
  @ ensures c != null;
  @ ensures o != null;
  @ ensures \not_modified(p, c);
  @ ensures o.getCustomer() == c;
  @ ensures o.getProducts().contains(p);
  @*/
public boolean addProductToOrder(int productNo,
                         int customerNo,
                         int orderNo) { ... }
```

Fig. 17. Relation between pre- and post-conditions

```
/*@ normal_behavior
  @ requires (\exists Order o; order.contains(o);
  @            o.getOrderNo() == orderNo);
  @ ensures \result.getOrderNo() == orderNo;
  @
  @public pure model Order findOrderByNo
  @           (int orderNo) {...}
  @*/
```

Fig. 18. model method findOrderByNo of class Customer

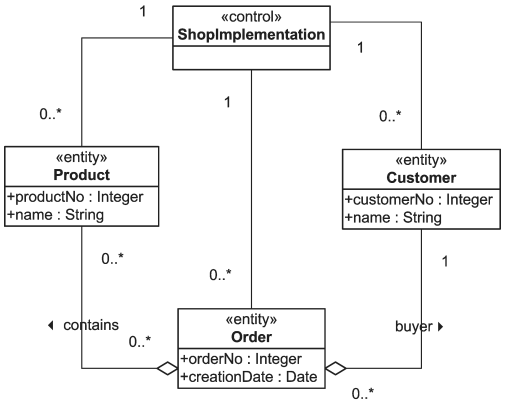

Fig. 19. Modified platform independent class diagram

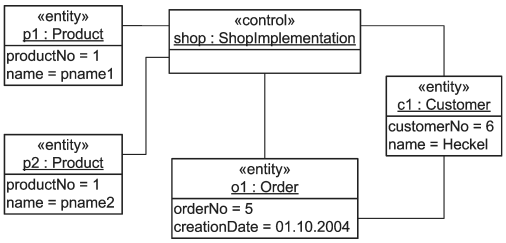

Fig. 20. Pre-state with two possible embeddings

`p` in the specification. In that way the same object can be referenced in both the pre- and the post-condition. The method `getProductByNo` is obtained from the qualified association between class `ShopImplementation` and class `Product` in the class diagram.

If a link between two objects is not qualified, we cannot rely on methods generated from the class diagram. For example, in order to write the JML code for handling the association between the class `Customer` and `Product` we need additional methods searching for an object on the other end of the association. As a solution we can use JML *model methods*. Model methods are Java methods declared within comments that can only be used in specifications, not in regular Java code. They offer the full expressiveness of Java. We have defined a set of model method templates to support our approach, defining operations that allow us to find objects with specific properties, like particular attributes values. These templates are instantiated by binding their parameters to model elements from the class diagram in the course of the translation into JML. An example for such a model method is the method `findOrderByNo` in Figure 18. It allows to find an order that is identified by an `orderNo` with the result being bound to the variable `c`. Also, these model methods can be annotated by JML contracts.

After, we have used the let expression to create a binding that can be used in both pre- and post-condition, we can test in the pre-condition whether objects with the given characteristics have been found (see first 3 requires-statements of Figure 17). Then we need to test if the objects are linked as described by the left- and right-hand side of the graph transformation rule. The operation `o.getCustomer()` is a method generated from the class diagram. It allows to test if the object bound to variable `o` knows an object `c` of type `Customer`. The operation `getProducts()` called on variable `o` is also generated from the class diagram and returns a list over the Java interface `Set`. We can use the operations from that interface to test whether the product `p` is part of the order `o`. The JML statement `\not_modified` asserts that the values of objects are the same in the post-state as in the pre-state.

The above translation works well if, given the input parameters of the method, all objects in our graph transformation rules have a unique match in the system state, e.g., by means of unique attribute values, qualified or to-one associations. Let us modify our model as shown in Figure 19 (disregarding operations) such that classes `Product`, `Order` and `Customer` are no longer connected to class `ShopImplementation` via qualified associations.

Figure 20 shows an instance of our online shop example representing a possible state of the online shop. If the method `addProductToOrder` is executed at this

```
/*@ public normal_behavior
  @
  @ old Vector embeddings = this.findEmbeddings();
  @
  @ requires embeddings != null;
  @
  @ ensures embeddings != null;
  @ ensures this.testEmbedding(embeddings) == true;
  @*/
public boolean addProductToOrder(int productNo,
              int customerNo, int orderNo) {...}
```

Fig. 21. Handling non-deterministic embeddings with JML

state, we can find two possible embeddings of the pre-condition of the corresponding graph transformation rule, namely `ep1= {{pr,p1},{cr,c1}, {o, o1}}` and `ep2= {{pr, p2}, {cr,c1}, {o, o1}}`. On the one hand, using the JML specification of Figure 17 it may be possible that during the test of the pre-condition the embedding `ep1` is stored in the variables `p`, `c` and `o` as the methods pre-state. Then, according to the contract of Figure 17 it is assumed that there is a link between `p1:Product` and `o1:Order`. On the other hand, the implementation might add the object `p2:Product` to the order. In this case, the post-condition of the contract evaluates to false even if the method is implemented correctly. The problem is that the implementation has chosen another embedding than the pre-condition of the JML contract, consequently adding a link between the objects `o1:Order` and `p2:Product`, but not between the objects `o1:Order` and `p1:Product`.

We can handle such non-deterministic embedding problems by storing all possible embeddings of the pre-condition into the pre-state before the operation is executed. For doing so, we generate corresponding model methods that find all possible embeddings and return them in an array, which is bound to a variable using the JML `old`-expression. Figure 21 shows an example. All embeddings are bound to the variable `embeddings` by calling the model method `findEmbeddings`. This method is similar to the graph pattern matching algorithms as used in Fujaba [11]. In contrast to the code in Figure 17, the `old`-expression binds a set of embeddings rather than a single object. After the execution of the method, we have to test if there is an embedding that satisfies the post-condition. This is also achieved by a model method that receives the embeddings generated in the pre-condition as input.

Even if we can handle such a non-deterministic embedding, we are of the opinion that a developer should avoid such problems by a more detailed modeling of the class diagrams or the graph transformation rules. Otherwise, the behavior of the application would be non-deterministic, which could lead to unexpected and very hard to trace errors.

```
/*@ public normal_behavior
  @
  @ old Customer c = findCustomerByNo(customerNo);
  @ old Order o = c.findOrder(orderNo);
  @ old Vector p = o.getProduct();
  @
  @ requires p != null;
  @ requires c != null;
  @ requires o != null;
  @ requires order.contains(o);
  @
  @ ensures p != null;
  @ ensures c != null;
  @ ensures o != null;
  @ ensures \not_modified(c, o);
  @ ensures order.contains(o);
  @ ensures (\forall Product pr; p.contains(pr);
  @   pr.getOrder() == null &&
  @   pr.getShopImplementation() == null);
  @*/
public boolean clearOrder(int customerNo,
            int orderNo) {... }
```

Fig. 22. Handling multiobjects and object deletions with JML

To generate JML contracts for rules with multiobjects, as in Figure 8, we must be able to handle sets of objects whose cardinality is not known at design time. Again, this is achieved by a combination of the `old`-expression and the use of model methods. Figure 22 shows an example of how to translate the graph transformation rule of Figure 8. We add a binding to a variable `product` of type `Collection`. This variable stores all products that are connected to the object `o` of type `Order`. In the last line of the post-condition we use the `forall`-expression to check whether all objects of type `Product` that were recorded in the pre-condition have indeed been deleted. This is done by testing whether the objects of type `Product` have a link to other objects. If they do not reference other objects, then no object references them. This is ensured by our code for handling references generated from the class diagram. In order to guarantee consistency of pairs of references that implement an association, the respective access methods for reference attributes call each other (see Section 5.1). If another object no longer references an object in Java, it will be deleted by the Java garbage collection. Thus, we can assume that the objects are deleted. Since sets are unordered and the tests are performed only before and after the operation, the programmer can implement the removal of these links in an arbitrary order.

6 Consistency

In a model-driven approach, the quality of the system developed depends on the quality of the model and its correct implementation. One important quality aspect of models is consistency. In general, consistency problems

occur if different views of the same system are redundant or dependent on each other. Depending on whether the views are at the same or at different levels of abstraction, we distinguish horizontal and vertical consistency problems, respectively.

Vertical consistency is a property of the transitions between requirements, design, and implementation. As such, it is at the heart of the MDA approach. Therefore, in this section we concentrate on vertical consistency, in particular, of behavioral models and their implementations. In this category, we face three different consistency problems.

1. Requirements expressed in terms of scenarios in Section 2 have to be consistent with the allowed sequences of operations between service provider and client as specified by the protocol statecharts of the architectural design in Section 3.1.
2. These protocols have to be fulfilled by the components providing the services, as described by the graph transformation rules of the detailed design in Section 3.2.
3. This consistency relation has to be preserved by a programmer of the platform independent code with a correct implementation of the behavioral models.

The first consistency requirement relates sequence diagrams and protocol statecharts. This is a standard model checking problem and is addressed by quite a number of publications. One possibility to validate such a requirement is to translate statecharts and sequence diagrams into a common semantic domain, like CSP [19]. This allows for expressing and checking the requirements as explained in the following (cf. [9]). The sequence of requests (incoming messages in a sequence diagram) received by a component instance must be acceptable by the corresponding statechart diagram (or they must be subsequences of an acceptable sequence if we want to allow for sequence diagrams showing only a part of the possible behavior). This is the case in our simple online shop (see Figure 2 for a sequence diagram and the corresponding protocol statechart in Figure 4). If we remove, e.g., the *createOrder(customerNo)* message, the consistency requirement is violated. We have to ensure that for every sequence diagram there exists a corresponding path through the protocol statechart.

The second problem is one of realisability. All sequences of requests accepted by the protocol statecharts must be executable based on the operations as specified by the graph transformation rules. This is true if, in each situation, the current internal data state satisfies the pre-conditions of all graph transformation rules that may be applied at this stage according to the protocol. For example, the rule in Figure 7 requires in its pre-condition the presence of an object of class `Order`. If this is not available, this rule is not applicable and the execution of the whole sequence fails.

To make this notion of consistency precise, we need to exercise our graph transformation rules in all possible ways prescribed by the protocol statechart. This is possible because of the formal background of graph transformation, which provides us with an operational semantics for such rules, i.e., a notion of graph transformation, which, abstractly speaking, defines a binary relation on states induced by rule applications. (See, e.g., [33] for different ways of formalizing this concept.) The transformation relation thus defined must produce a superset of the traces obtained from the statechart.

Such a condition can be verified through testing by executing the models, either by means of a model interpreter or through a model compiler, which translates models into executable code. Examples of the former include statechart simulators like [20], but also graph transformation engines like AGG [10]. Compilers of statecharts can be found, for example, in the Rhapsody [14] and Fujaba [12] CASE tools, while the latter also transforms graph transformation rules into executable Java code.

The third problem is concerned with the correct implementation of the graph transformation rules by the programmer. This can be ensured in two different ways. One can run tests to validate that the implementation is correct according to its specification or verify statically that the code satisfies its specification.

For both alternatives, we can use our translation of the graph transformation rules into Java classes annotated with JML specifications as described in Section 5. For testing purposes the JML compiler [6], an extension to a standard Java compiler, translates JML specifications into Java bytecode for checking runtime assertions like pre- and post-conditions. Since the JML expressions generated by our translation are side effect-free, the execution of such checks is transparent in that, unless an assertion is violated, the behavior of the original program is unchanged.

In addition, the JMLUnit tool combines JML with the popular unit testing tool JUnit for Java. The basic idea of this tool is that a specification of a method by pre- and post-conditions can be viewed as a test oracle [30,2], and a runtime assertion checker can be used as the decision procedure for the test oracle [7]. The JMLUnit tool frees the programmer from writing most unit test code and significantly automates unit testing of Java classes and interfaces.

Another crucial activity of testing is the generation of test cases. For testing object-oriented software against state diagrams, this problem has been discussed, for example, in [3,4].

Generally more ambitious than testing is the static verification of methods against their contracts. There are several tools for verifying JML assertions which provide different levels of automation and support different levels of expressivity in specifications (see [5] for an overview). One interesting tool is the LOOP tool [24,21],

developed at the University of Nijmegen, that translates JML-annotated code to proof obligations that one can try to prove interactively using the theorem prover PVS. The verification of the proof obligations is accomplished using a Hoare logic [23] and a weakest pre-condition calculus [22] for Java and JML.

7 Conclusion

In this paper, we have presented a model-driven approach to the development of reactive information systems, e.g., based on Web technologies like HTML or SOAP. We have discussed the separation of technology-independent from platform-specific aspects of the implementation in order to focus on the platform-independent part.

Here, graph transformation rules have been proposed to specify reactive behavior in combination with data state transformation. By translating the graph transformation rules into executable contracts in JML, we can ensure that models and manually derived code are consistent. We have provided a discussion of the vertical consistency problems arising from our approach, and how they can be addressed, in particular, through graph transformation and JML tools.

Although graph transformation rules are not part of the mainstream UML methodology, our experience with the use of this concept in a course on Web-based application development at the University of Paderborn suggests that the higher level of integration of static and dynamic aspects adds to the understandability of models.

To support our methodology we have been implementing an editor that allows developers to coherently model class diagrams and graph transformation rules. This implementation extends the Eclipse tool and uses the Eclipse Modeling Framework. Currently, we are working on an extension of this editor with the described code generation facilities for generating executable JML assertions.

References

1. From UML to Java and Back Again: The Fujaba homepage. `www.upb.de/cs/isileit`.
2. Sergio Antoy and Richard G. Hamlet. Automatically checking an implementation against its formal specification. *Software Engineering*, 26(1):55–69, 2000.
3. Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Object Technology Series. Addison Wesley, 1999.
4. L. Briand and Y. Labiche. A UML-based Approach to System Testing. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools., 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, LNCS, pages 194–208. Springer-Verlag, 2001.
5. Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.
6. Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Software Engineering Research and Practice (SERP 2002)*, pages 322–328, 2002. To appear in SERP 2002.
7. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255. Springer-Verlag, 2002.
8. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools.* World Scientific, 1999.
9. G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proc. European Software Engineering Conference (ESEC/FSE 01), Vienna, Austria*, volume 1301 of *Lecture Notes in Comp. Science*, pages 327–343. Springer Verlag, 2001.
10. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and tool environment. In Engels et al. [8], pages 551 – 601.
11. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.
12. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. volume 1764, pages 112–121. Springer-Verlag, 2000.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addsion-Wesley, 1994.
14. D. Harel and E. Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
15. R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. A view-based approach to system modelling based on open graph transformation systems. In Engels et al. [8].
16. R. Heckel, M. Llabrés, H. Ehrig, and F. Orejas. Concurrency and loose semantics of open graph transformation systems. 12:349–376, 2002.
17. Reiko Heckel and Marc Lohmann. Towards model-driven testing. volume 82 of *Electr. Notes Theor. Comput. Sci.*, 2003.
18. Reiko Heckel and Leonardo Mariani. Automatic conformance testing of web services. In *FASE*, pages 34–48, 2005.
19. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
20. B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. In *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques, Geneva, Switzerland.* Carleton Scientific, 2000.

21. Marieke Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, Holland, February 2001.
22. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 2003.
23. Bart Jacobs and Erik Poll. A logic for the java modeling language jml. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 284–299. Springer-Verlag, 2001.
24. Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, U. Hensel, and H. Tews. Reasoning about java classes: preliminary report. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 329–340. ACM Press, 1998.
25. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
26. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, 2000.
27. Bertrand Meyer. Lessons from the design of the eiffel libraries. *Commun. ACM*, 33(9):68–88, 1990.
28. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.
29. OMG. *UML 2.0 Superstructure Specification*, 2003.
30. Dennis K. Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Trans. Softw. Eng.*, 24(3):161–173, 1998.
31. The Apache XML Project. Axis user's guide. `http://xml.apache.org/axis/`, 2002.
32. Achut Reddy. Java coding style guide. `wwws.sun.com/software/sundev/whitepapers/java-style.pdf`, May 2000.
33. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
34. Jon Siegel. Using omg's model driven architecture (MDA) to integrate web services, 2002. `http://www.omg.org/mda/mdafiles/MDA-WS-integrate-WP.pdf`.
35. Jon Siegel and OMG Staff Strategy Group. Model driven architecture, revision 2.6, November 2001. `ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf`.
36. Richard Soley and OMG Staff Strategy Group. Model driven architecture, draft 3.2, November 2000. `ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf`.
37. Sun Microsystems Inc. Java(tm) servlet specification 2.3. `http://java.sun.com/products/servlet`, 2001.
38. W3C. Soap version 1.2 part 1: Messaging framework. `http://www.w3.org/TR/2002/WD-soap12-part1-20020626/`, 2002.
39. W3C. Xforms 1.0 W3C candidate recommendation. `http://www.w3.org/TR/2002/CR-xforms-20021112/`, November 2002.
40. Albert Zündorf. Graph pattern matching in progres. In *Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science*, pages 454–468. Springer-Verlag, 1996.