Model-Based Development of Web Applications Using Graphical Reaction Rules

Reiko Heckel and Marc Lohmann

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
D–33098 Paderborn, Germany
{reiko,mlohmann}@upb.de

Abstract. The OMG's Model-Driven Architecture focusses on the evolution and integration of applications across heterogeneous middleware platforms. Presently available instances of this idea are mostly limited to static models.
We propose a model-driven approach to the development of web-enabled applications, seen as reactive information systems on an HTTP-based communication platform, which covers both static and dynamic aspects. To support the separate change of both platform and functionality we separate at model and implementation level the platform-independent application logic from classes specific to technologies like HTML or SOAP.
We discuss a notion of consistency between models at different abstraction levels based on a concept of graphical reaction rules, i.e., graph transformation rules which integrate data state transformation and reactive behavior.

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, the approach of web services provides a collection of languages and protocols to support interoperability at the level of text-based HTTP by interchanging XML-documents representing, e.g., remote procedure calls. At the level of design, the OMG has proposed the Model-Driven Architecture (MDA) [16,17] to achieve interoperability through models. Starting from standard UML models [12] 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 all relevant annotations for the generation of platform-specific code.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 170–183, 2003.

The idea seems realistic (and has in part been implemented) 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 [15], for example, describes the use MDA for applications based on web services, focussing 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.

For behavioral models like statecharts and collaboration diagrams, transformations to code are more sophisticated, but largely platform-independent [3,5]. If we want to use such diagrams to model the business logic of our application, an integration with the platform-specific models and their mappings is required.

According to the OMG's proposal this integration should be done by augmenting, throughout the platform-independent models, model elements with annotations in terms of stereotypes and tagged values. Then, in the likely case of a change in the platform-independent model (induced, for example, by an evolution of the functional requirements), the platform-specific annotations have to be re-iterated for all relevant target platforms. This shows an imbalance of the MDA proposal which focuses entirely on evolution and integration across platforms, disregarding the evolution of the functional aspect.

In this paper, we discuss a model-driven approach to the development of web-enabled applications which avoids the above obstacles by separating platform-independent and platform-specific models. This separation of concerns is preserved at the implementation level as well as in the mapping.

The approach aims at both interactive (HTML-based) web applications and web services which share the basic request-query/update-response pattern, where an HTTP-request is answered (or causes further requests to other servers) in combination with a query or update of the internal data state of the server or its associated data base. Rather than with other kinds of reactive systems, like in the embedded domain where data is often abstracted from at the level of models, web-based business applications are primarily information systems, and data state transformations are a crucial aspect of their behavior.

Our approach to integrate data state transformations and reactivity at the level of models is based on a special format of graph transformation rules that we call *graphical reaction rules*. Such a reaction rule is a transformation rule on the internal object structure of the server which is triggered by a *request*, i.e., a special kind of vertex modeling an incoming message that is consumed when the rule is applied. This rule-based way of formalizing reactive behavior goes back to actor grammars [10], a model of actor systems by graph transformation, and has since then been adopted by several authors.

Based on this concept we outline a model-based development approach starting from requirements expressed in terms of use cases and sequence diagrams in Sect. 2 via architectural and detailed platform-independent design in Sect. 3 to platform-specific design and implementation in Sect. 4. Then, in Sect. 5, we provide a discussion of the vertical consistency problems arising between these

different levels and argue for the necessity of a model-driven testing approach in Sect. 6.

2 Requirements Specification

Following the Unified Process [9], the functional requirements of web-enabled applications are captured by 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 Fig. 1, a client of the online shop can query products, order a product, or pay for an order. If the client wants to pay, for example by credit card, his credit card data has to be verified before he gets an acknowledgement. Therefore, the online shop uses the service of a credit card company to verify credit card data.

Sequence diagrams are used to model scenarios like this in a more formal way as sequences of messages exchanged, in this case, between the client, the online shop, and the credit card company. Variants can be expressed by different sequence diagrams associated with the same use case. Figure 2 shows two sequence diagrams detailing the use case *pay order*. The initial segments of both scenarios are identical: The client who triggers the use case is asked by the online shop to enter his preferred method of payment, e.g., by automatic debit from the client's bank account or by credit card. In our sample scenarios the client chooses to pay by credit card, which requires the transmission of the credit card data, e.g., the name of the credit card company, the credit card number, etc. The online shop sends the data to a credit card company for validation. The client gets a positive or negative feedback, depending on whether the credit card check has been successful or not. That means, we have one success and one failure scenario.

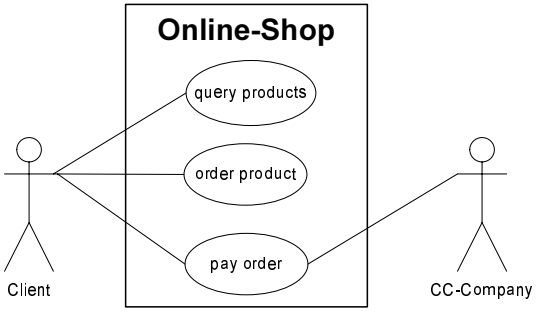

Fig. 1. Use case diagram of the shop example

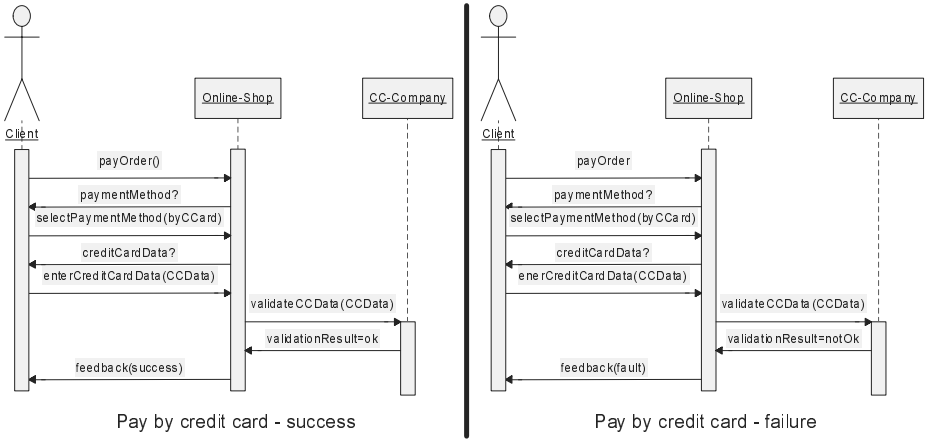

Fig. 2. Two scenarios describing the use case pay order

3 Platform-Independent Design

The requirements specification presented in Sect. 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 examples 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 three components of our online shop. The online shop itself is a component with three interfaces for the three use cases in the diagram of Fig. 1. For each interface we can list the operations of the online shop that can be called by a client through that interface. In Fig. 3 we have only detailed the interfaces for the use case *pay order* and for the *card validation* use case of the credit card company.

The data objects exchanged as parameters of operation calls between client and online shop as well as between online shop and credit card company are specified in a class diagram. We are qualifying the corresponding classes with stereotypes, one of the extensibility mechanisms of UML. The general notation for the use of stereotype is to enclose it in guillemots (≪≫). We are using the stereotype *boundary* defined within the Unified Process [9] which is intended to designate GUI classes or forms that model the interaction between the system and its actors.

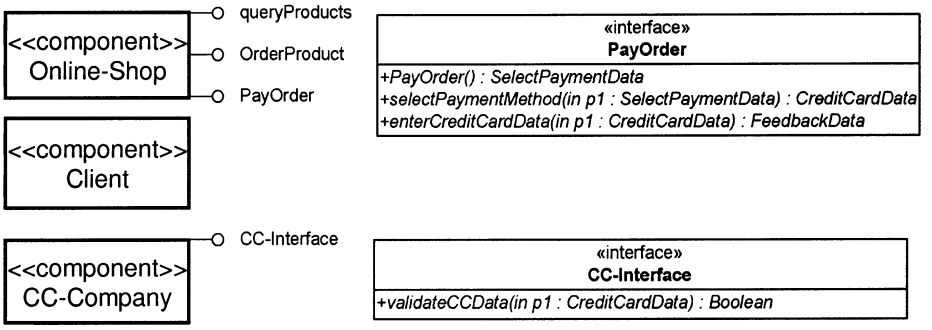

Fig. 3. Components of the online shop example

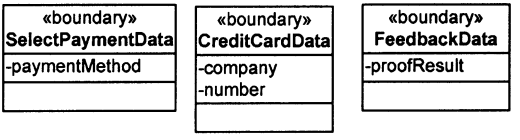

Fig. 4. Boundary classes describing the exchanged data

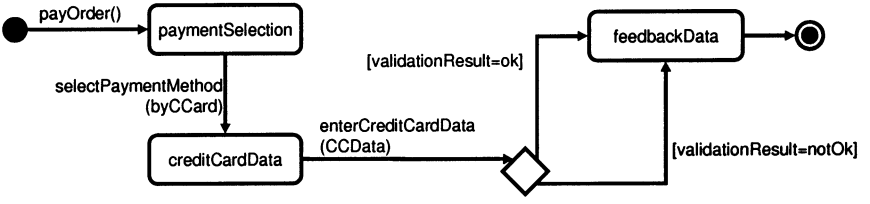

Fig. 5. Protocol statechart for the component online shop

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 5 shows the statechart for the interface *payOrder* corresponding to the use case pay order. The states are named according to the boundary classes whose instances are transferred to the client in response to the previous request.

3.2 Detailed Design

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

Class diagrams are used to represent static aspects. Figure 6 shows the result of detailing the use case *pay order*. Beside the stereotype *boundary* introduced above, *control* and *entity* stereotypes are used (cf. again [9]): Each of these

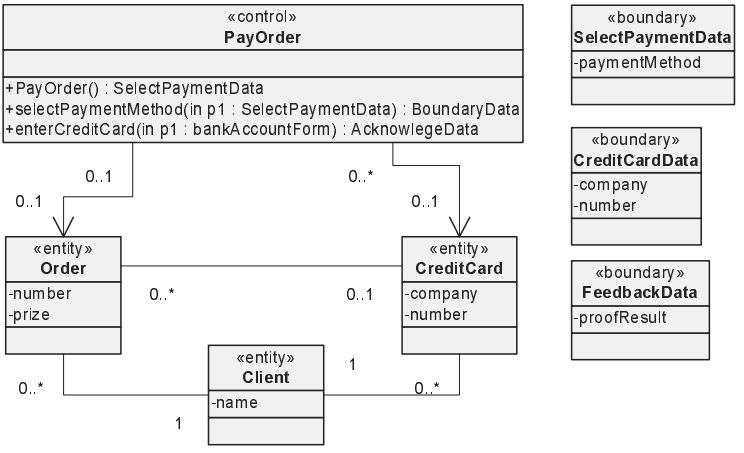

Fig. 6. Platform-independent class diagram for online shop

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. Boundary classes are used to model interactions between the system and its actors. Entity classes model long-lived or persistent information.

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. This requires to move the focus both from sequences to single operations and from the externally visible behavior to its internal realization. To simplify this transition, we first take an operation-wise view on the externally visible behavior. To this aim we introduce for each operation *abstract reaction rules*, which are derived from the sequence diagrams. For example, Fig. 7 shows the abstract reaction rules for the success scenario of Fig. 2. The left-hand side of the rule contains the method call on the component as part of the precondition of the operation. The right-hand side shows the visible effects, i.e., a message sent to another component.

Next, the internal data state transformation associated with this operation is described. Figure 8 shows the refinement of the lower right abstract reaction rule of Fig. 7. The left-hand side of the diagram represents the precondition of the rule, i.e., that the validation of the credit card has been successful. The right-hand side shows the desired effect of the execution: A new boundary object with an acknowledgement is created.

One benefit of this form of specification integrating static and dynamic models is that it provides a detailed and precise enough specification to allow automatic code generation (cf. [3]) which is an essential for the goal of model-driven development.

To stick with standard UML notation, reaction rules can be expressed as collaboration diagrams where pre- and post-conditions are jointly represented in one diagram. In order to distinguish objects that are created or deleted, the stan-

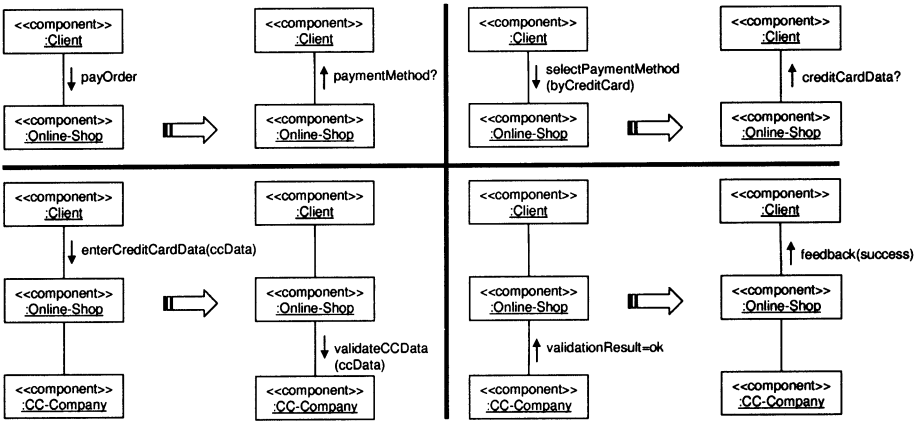

Fig. 7. Abstract reaction rules

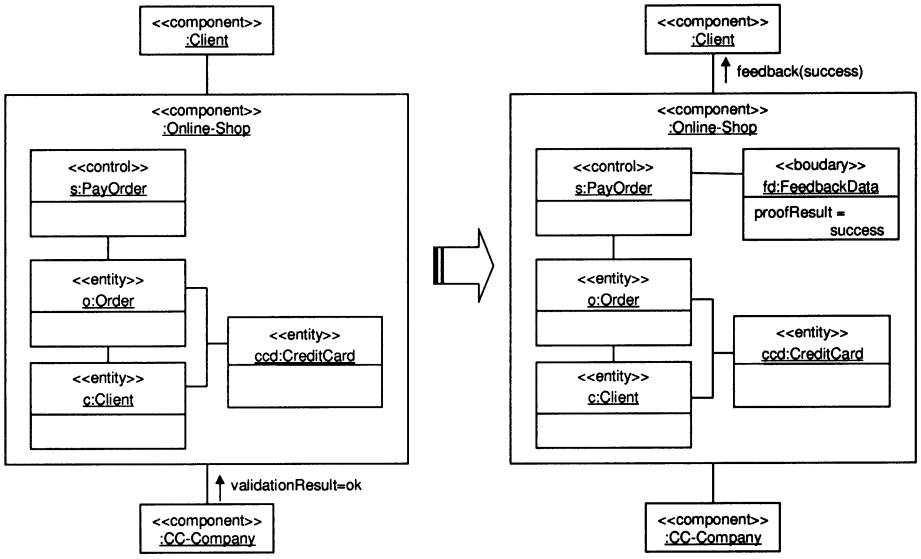

Fig. 8. Refinement of lower right abstract reaction rule of Fig. 7

dard constraints *new* and *destroyed* are used. As an example, Fig. 9 shows the corresponding representation of the reaction rule of Fig. 7. This correspondence between collaboration diagrams and graph transformation rules can be extended to more sophisticated cases where complex scenarios are modelled within one diagram [6]. In this case, rather than a single rule, a *graph process* is required, i.e., a partially ordered set of interrelated rules each modeling a single step of the interaction.

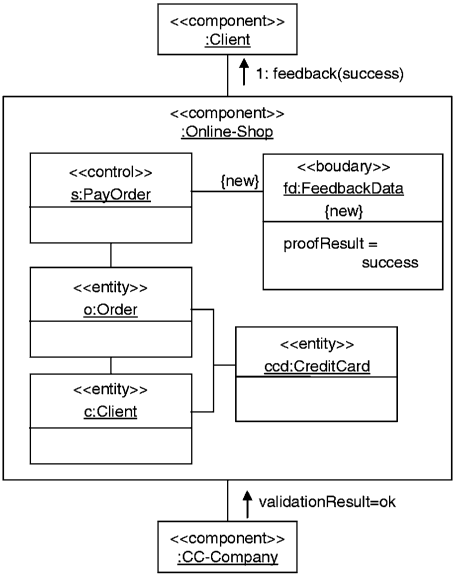

Fig. 9. Reaction rule of Fig. 8 expressed with collaboration diagram

4 Integrating Platform-Independent and Platform-Specific Models

In the next two subsections we show how to map the platform-independent models developed in the previous section on platforms like HTML or SOAP which realize the request-query/update-response pattern. Model information required for this mapping is captured in the *platform-specific design*. The aim is to deploy code generated from platform-independent models on a specific middleware while keeping platform-independent and platform-specific models separated.

4.1 Platform-Specific Design

In many web-enabled applications a middleware serves as a link between clients and back-end services. This middleware is normally responsible for the implementation of the chosen base technology. For example you can use a web server which implements the Java Servlet technology [18] to implement an HTML application or you can use a SOAP server to implement a SOAP-based [19] application. In both cases a client doesn't send a request directly to the application. Instead a client addresses the middleware and this middleware requests the application with pre-defined interfaces.

Figure 10 shows an abstract overview how a web application realized with the Java Servlet technology works. A user normally fills in an HTML form on the client and by clicking the Submit button the form data is send to the server (1). The server locates the requested application, more exactly he locates an

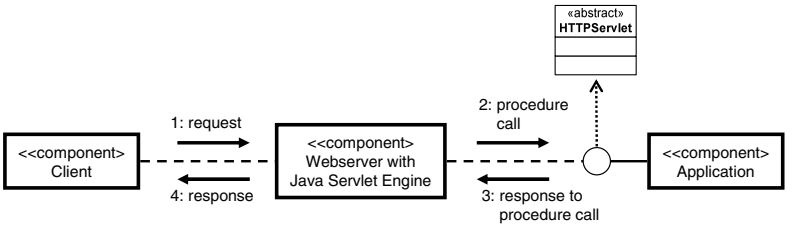

Fig. 10. Using a servlet (abstract overview)

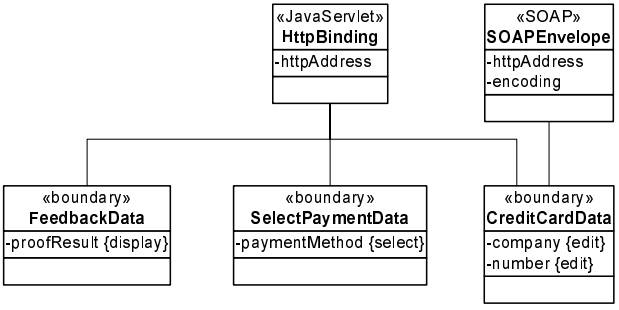

Fig. 11. Annotated boundary classes for interactive HTML application and for a SOAP Service

application which implements the servlet interface. The application is called via the servlet interface by a normal method call (2). The application processes the data and calculates a response (3). The middleware (servlet engine) transfers this response back to the client (4). That means to implement a Servlet, a servlet API is used, especially the Servlet interface. The Servlet interface declares, but does not implement, methods that manage the servlet and its communications with clients. These interfaces have to be provided when developing a servlet [18]. Other Internet-based communication platforms, like the SOAP implementation Apache Axis [13], work in a similar way.

To use different kinds of middleware, we have to annotate the boundary classes with information needed by the middleware. Figure 11 shows how this annotation could look like. For using the Java servlet technology we annotated the boundary classes with an additional class *HTTPBinding*. This class contains an attribute for the address where the application is to be found. Further, we have to annotate the attributes of the boundary classes to show, which information is displayed to the client and which information can be edited by the client. Therefore, we use property strings, marked with curly brackets. Property strings indicate property values that apply to an attribute and an attribute can contain more than one property value. *Edit* means that the client can edit the data freely, *select* that the client can choose one value from many and *display* that data is only transferred to the client. To access the credit card institute via SOAP the

class *CreditCardData* also needs some SOAP specific annotations. Objects are encoded in a special way in a SOAP message and the message has to be sent to an appointed web address.

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 12 shows the integration for the Java Servlet technology. Classes that are specific to the technology are shown in grey. Let us at first explain the changes of the platform-independent models made during the code generation. In Figure 12 a new platform-independent abstract class *StatemachineHandler* is shown which has the task of an object controller. For every use case, an implementation for the abstract method of this class is generated. The method implements the statechart of the corresponding use case, filtering incoming requests according to the protocol. 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 method *executeMethod* is called, which calls the correct method depending on the state and the type of the incoming boundary data. This design is following the *Command* design pattern [4], which allows a decoupling between the sender and the receiver. Decoupling means that the sender has no knowledge of the receiver's interface.

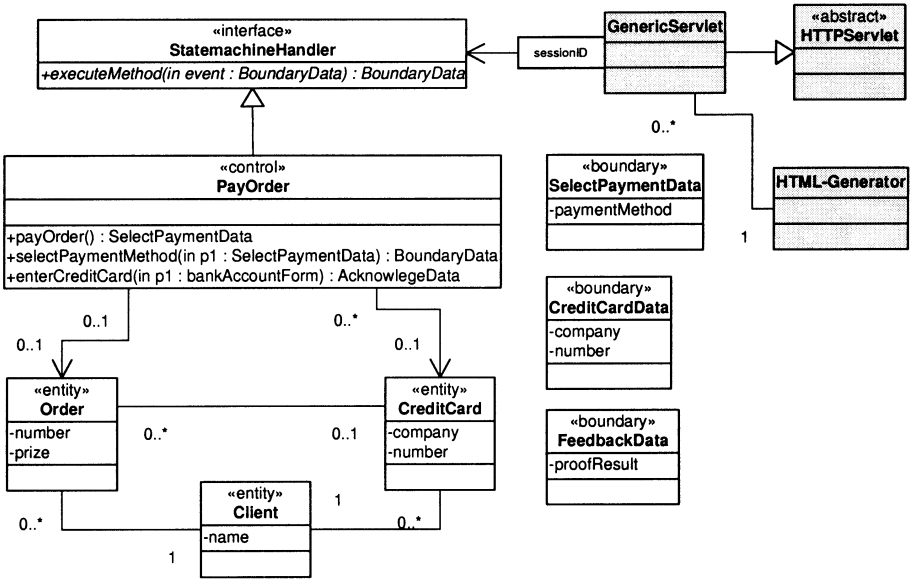

Fig. 12. Integration of PIM and PSM on implementation level

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 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 filled in a form, for example an HTML representation of the platform-independent boundary class *selectPaymentForm*. It then calls the corresponding method of a control class of the platform-independent model. Therefore, some auxiliary information has to be encoded in the forms transmitted to the client, because no control information of the application shall be encoded in the platform-dependent classes. In our example, we need a *sessionID* to obtain the correct session object for each request from a client.

In Fig. 12 we have one more platform-specific class which has to be implemented by a software engineer. The class *HTML-generator* is responsible for the creation of an HTML form from the boundary classes, which is sent to the client. To create an HTML form from a specific boundary class the platform specific annotations (see Fig. 11) are needed.

To use alternative platforms or other kinds of user interfaces, only the platform specific classes of Fig. 12 have to change. For example, to use XForms [20], 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 Vertical Consistency

The quality of models is a prerequisite for the quality of the resulting system. One important aspect of quality (and one of the few that can be sufficiently formalized) 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 transition from requirements to design models and their implementation. As such, it is a property responsible for the transition between platform independent and platform-specific models in the MDA approach. Therefore, in this paper we concentrate on this aspect, and in particular on the consistency of behavioral models. In this category we face two vertical consistency problems.

1. Requirements expressed in terms of scenarios in Sect. 2 have to be consistent with the contracts between service provider and client as specified by the protocol statecharts of the architectural design in Sect. 3.1.

2. These contracts (protocols) have to be fulfilled (correctly implemented) by the components providing the services, as described by the reaction rules of the detailed design in Sect. 3.2.

The first consistency requirement relates the sequence diagrams in Fig. 2 and the protocol statechart in Fig. 5: The sequence of requests (incoming messages) received by the Online Shop component instance in any of the two diagrams must be acceptable by the 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 both examples, but if we remove, e.g., the *selectPaymentMethod(byCCard)* message, the consistency requirement is violated.

To validate this notion of consistency, it may be phrased as a model checking problem by translating statecharts and sequence diagrams into a common semantic domain, like CSP [7], which allows to express and check the condition that every trace of requests to a service contained in a sequence diagram is (an instance of) a subsequence of a trace of call events generated by the protocol statechart of that service (cf. [1]).

The second problem is concerned with the correct implementation of the protocols by the internal data state transformations of reaction rules. Given an initial data state for a component, we may ask if all sequences of requests are indeed executable in the sense that the current internal data state satisfies the precondition of all reaction rules that may be applied at this step according to the protocol. For example, the rule in Fig. 8 requires in its precondition the presence of an object of class *CreditCardData*. 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 reaction 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., [14] for different ways of formalizing this concept.)

To implement the protocol defined by the statechart of the component, the transformation relation thus defined must be able to produce a superset of the traces obtained from the statechart. This linear-time condition could be replaced by more sophisticated notions, e.g., using the failure and divergence model [7] or various notions of simulation on transition systems. A detailed study of what is a semantically convincing and at the same time feasible approach is outside the scope of this paper. One advantage of the simpler, linear condition is that it can be validated by testing.

6 Towards Model-Driven Testing

In order to test the consistency between models, an execution of models is required either by a model interpreter or through a model compiler which translates models into executable code. Examples of the former include statechart

simulators like [8], but also graph transformation engines like AGG [2]. Compilers of statecharts can be found, for example, in the Rhapsody [5] and Fujaba [3] CASE tools, while the latter also transforms graph transformation rules (denoted as UML collaboration diagrams) into executable Java code.

In the context of MDA, a model shall be mapped on multiple platforms, thus reusing the effort of coding and design, but not the amount of testing required because implementations obtained from the same model may behave differently on different platforms. Hence, we require what could be called an approach to model-driven testing. By this we mean the testing of consistency properties, among models or between models and code, while reusing the results of platform-independent tests (or of tests performed on an "ideal" platform, like a single Java virtual machine) for implementations of the same models on different or heterogeneous target platforms. The idea is that recording platform-independent test results, we determine the expected results of platform-specific tests, which can then be automatically executed and compared. This idea is independent of the question whether the original tests are performed automatically or by hand.

7 Conclusion

In this paper we have presented a model-driven approach to the development of reactive information systems based on web technology which refines the OMG's MDA in order to separate better the technology-independent from the platform-specific aspects. We have used graphical reaction rules to specify reactive behavior in combination with data state transformation and discussed the consistency issues arising with more abstract protocol specifications by means of statecharts and requirements expressed by sequence diagrams.

Although graphical reaction 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.

Future work shall include the development of tool support for automated consistency checks based on the construction of labeled transition in the previous section, as well as the exploration of the idea of model-driven testing.

References

1. 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* `http://www.springer.de/comp/lncs`, pages 327–343. Springer Verlag, 2001.
2. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and tool environment. In G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, pages 551–601. World Scientific, 1999.

3. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *Lecture Notes in Comp. Science* `http://www.springer.de/comp/lncs`. Springer-Verlag, 2000.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addsion-Wesley, 1994.
5. D. Harel and E. Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
6. R. Heckel and St. Sauer. Strengthening UML collaboration diagrams by state transformations. In H. Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy*, volume 2185 of *Lecture Notes in Comp. Science* `http://www.springer.de/comp/lncs`. Springer-Verlag, April 2001.
7. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
8. 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.
9. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
10. D. Janssens and G. Rozenberg. Actor grammars. *Mathematical Systems Theory*, 22:75–107, 1989.
11. R. Milner. Bigraphical reactive systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *Proc. 12th Intl. Conference on Concurrency Theory (CONCUR 2002), Aalborg, Denmark*, volume 2154 of *Lecture Notes in Comp. Science* `http://www.springer.de/comp/lncs`, pages 16–35. Springer-Verlag, August 2001.
12. Object Management Group. UML specification version 1.4, 2001. `http://www.celigent.com/omg/umlrtf/`.
13. The Apache XML Project. Axis user's guide. `http://xml.apache.org/axis/`, 2002.
14. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
15. 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`.
16. 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`.
17. 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`.
18. Sun Microsystems Inc. Java(tm) servlet specification 2.3. `http:://java.sun.com/products/servlet`, 2001.
19. W3C. Soap version 1.2 part 1: Messaging framework. `http://www.w3.org/TR/2002/WD-soap12-part1-20020626/`, 2002.
20. W3C. Xforms 1.0 W3C candidate recommendation. `http://www.w3.org/TR/2002/CR-xforms-20021112/`, November 2002.