7

A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services

Marc Lohmann[1], Leonardo Mariani[2] and Reiko Heckel[3]

[1] University of Paderborn, Department of Computer Science Warburger Str. 100, 33098 Paderborn, Germany `mlohmann@uni-paderborn.de`
[2] Università degli Studi di Milano Bicocca – DISCo via Bicocca degli Arcimboldi, 8, 20126 Milano, Italy `mariani@disco.unimib.it`
[3] University of Leicester, Department of Computer Science University Road, LE1 7RH Leicester `reiko@mcs.le.ac.uk`

Abstract. Service-oriented computing is distinguished by its use of dynamic discovery and binding for the integration of services at runtime. This poses a challenge for testing, in particular, of the interaction between services.

We propose a model-driven solution to address this challenge. Service descriptions are promoted from largely syntactical to behavioural specifications of services in terms of contracts (pre-conditions and effects of operations), expressed in a visual UML-like notion. Through mappings to semantic web languages and the Java Modelling Language (JML) contracts support the automatic discovery of services as well as the derivation of test cases and their execution and monitoring.

We discuss an extended life cycle model for services based on the model-driven approach and illustrate its application using a model of a hotel reservation service.

7.1 Introduction

Service-oriented computing is becoming the leading paradigm for the integration of distributed application components over the Internet. Besides its implementation, the life cycle of a service includes the creation and publication of a service description to a registry. Clients will query the registry for service descriptions satisfying their requirements before selecting a description, binding to the corresponding service and using it.

Established technology for providing, querying and binding to services is largely based on syntactic information. From UDDI registries, e.g., services can only be retrieved by inspecting interface descriptions and associated keywords [50]. The lack of semantic information in service descriptions prevents reliable automatic integration of services. For instance, if an application interacting with a shopping cart assumes that the `addItem(item,qt)` operation adds `qt` to the quantity of the target item, interactions will fail with all carts that implement an `addItem(item,qt)` operation overwriting the quantity instead of increasing it [26]. To mitigate semantic problems, natural language specifications can be associated with interface descriptions. However, these descriptions cannot be automatically processed by clients and are often ambiguous and incomplete. For instance, according to [17], more than 80% of Web services have descriptions shorter than 50 words and more than 50% of service descriptions are even shorter than 20 words.

In addition to the danger of binding to incompatible services, problems can be caused by services which fail to correctly implement their specifications, i.e., their public service descriptions. A client application has only limited capacity to verify the quality of a remote Web service because it cannot access the service implementation. Moreover, owners of services can modify their implementations at any time without alerting existing clients. Hence, clients can neither rely on the quality of a service at the time of binding nor on its behavioural stability over time.

Several testing and analysis techniques for Web services and service-based applications have been developed [10], addressing the verification of functional and non-functional requirements, interoperability and regression, but they focus on the technical verification problem, failing to provide a sound embedding in the life-cycle of services. For example, many approaches focus on testing entire applications, which is obviously insufficient because it reveals faults of single services too late to be effectively fixed and does not consider dynamic changes.

In this chapter, we present a framework for developing high-quality service-based applications addressing both the verification problem, as well as its embedding in the service life-cycle. In line with the current best practice, this includes a model-driven approach for developing service specifications with automated mappings to languages for service description and matching, as well as monitoring at the implementation level. We focus on functional service specifications aimed at the interoperability of services.

Model-driven development provides the foundation for (formal) reasoning about the behaviour of services and their compositions. Models allow developers to focus on conceptual tasks and abstract from implementation details. Moreover, models are often represented with a high-level visual language that can be understood more intuitively than source code and formal textual descriptions and are effective for communication between developers [15].

We describe the data types visible at a service interface with a class diagram. We specify the behaviour of its operations by graph-transformation rules [13], describing the manipulation of object structures over the class diagrams. Graph transformation rules combine a number of advantages which make them particularly suitable for the high-level modelling of operations: (1) they have a formal semantics; (2) they address the transformation of structure and data, an aspect that would otherwise be specified textually in a programming- or logic-oriented style; (3) they form a natural complement to mainstream visual modelling techniques like state machines or sequence diagrams; and (4) they can easily be visualised themselves in a UML-like notation, supporting an intuitive level of understanding beyond the formal one [3].

Our approach aims to guarantee high-quality service-oriented applications by refining the classical life-cycle of service registration, discovery and usage as follows.

- Service registration: Only tested Web services should be allowed to participate in high-quality service-based applications. For this, we propose to extend the functionality of UDDI registries to automatically generate test cases that are executed when a Web service either adds or updates its behavioural description. Registration is allowed only if all test cases have been passed [26].
- Service discovery: Based on the extension of service descriptions to include behavioural specifications, service discovery can match descriptions against behavioural requirements [23]. For instance, a client can explicitly query for a cart that implements an `addItem` operation that overwrites the quantity of items already present in the cart.
- Service usage: Since clients can access services over a period of time, it is important that their behaviour remains consistent with their specifications. Service models are used to automatically generate monitors that are able to continuously verify the behaviour of Web service implementations [16].

In summary, our framework provides discovery mechanisms based on behavioural descriptions, supports continuous monitoring of web services at the provider and client sides and allows registration and composition of high-quality Web services only.

The rest of the chapter is organised as follows. Section 7.2 describes the life-cycle of high-quality service-based applications. Section 7.3 introduces graph-transformation as the formal language used to describe the behaviour of services, along with a running example used throughout the chapter to present our framework. Testing, discovery and monitoring techniques, which are the core of the approach, are presented in Sects. 7.4–7.6, respectively. Sects. 7.7 and 7.8 discuss empirical validation of our framework and related work, respectively. Finally, Sect. 7.9 outlines conclusions and future work.

7.2 Life-Cycle of High-Quality Service-Oriented Applications

High-quality service-oriented applications are systems obtained from *reliable composition* of *high-quality web services*. Reliable composition is achieved by automatic service discovery and binding based on the matching of behavioural

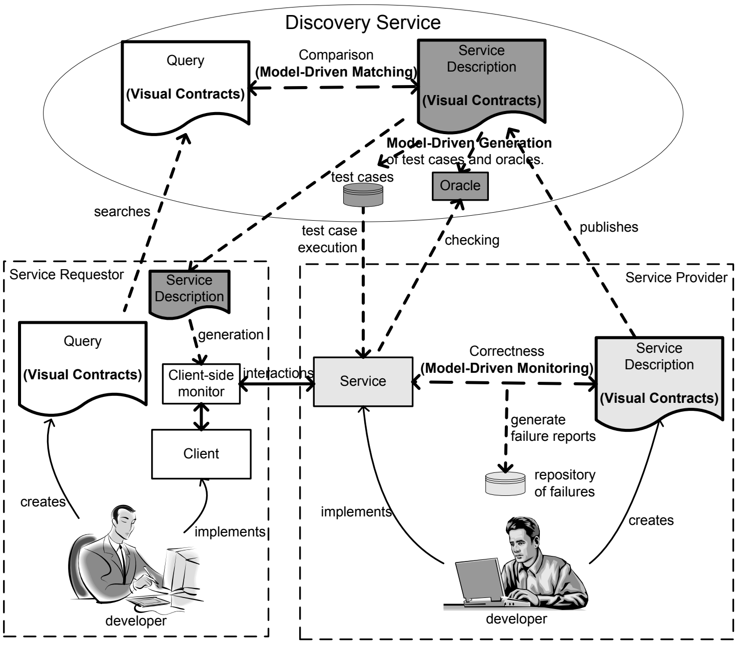

Fig. 7.1. The development process for high-quality service-based applications

specifications. High-quality services are tested before their registration and monitored throughout their life time.

Figure 7.1 shows the entities that participate in service development, publication and discovery of high-quality service-oriented applications. Different shades of grey are used to mark entities involved in different steps of the process. Items associated with *service development* are indicated in light grey, artefacts and activities related to *service registration* are shown with a dark grey background, and *service discovery* is indicated with a white background.

During *service development* software developers design and implement single services. The development methodology associated with our framework is based on the integrated use of UML diagrams to describe static aspects and graph transformations to describe dynamic properties of the service under development. The dynamic properties of a service are given in terms of pre- and post-conditions, both instances of the design class diagram as explained in Sect. 7.3. These visual descriptions are used both internally and externally.

Internal use consists of translating these visual descriptions into JML monitors that are embedded into the implementation.[4] External use consists in

[4] The Java Modelling Language (JML) [9, 33] is a behavioural specification language for Java classes and interfaces. JML assertions are based on Java expression and annotated to the Java source code. Thus, JML extends Java with concepts of Design by Contract following the example of Eiffel [37]. However, JML is more expressive than Eiffel, supporting constructs such as universal and existential

uploading these descriptions to the discovery service to support the generation of test cases and monitors required in the next phases. Violations detected by JML monitors are collected in a repository and examined by developers who use this information to fix bugs in the implementation.

The *translation process* from visual descriptions to JML monitors consists of two parts: first, Java class skeletons are generated from the design class diagrams; second, JML assertions are derived from graph transformation rules. The assertions allow us to validate the consistency of the models they are derived from with the manually produced code. The execution of such checks must be transparent in that, unless an assertion is violated, the behaviour of the original program remains unchanged. This is guaranteed since JML assertions are free of side-effects. (See Sect. 7.6 for details on the translation.)

Programmers use the generated Java fragments to fill in the missing behavioural code in order to build a complete and functional application according to the design models and visual contract of the system. They are not allowed to change the JML assertions, thus ensuring that they remain consistent with the visual contracts. If new requirements for the system demand new functionality, the functionality has to be specified using visual contracts first, in order to derive new assertions for implementation.

When behavioural code has been implemented, programmers use a JML compiler to build the executable binary code. This binary code includes the programmer's behavioural code and additional executable runtime checks that are generated from the JML assertions. The runtime checks verify if the manually coded behaviour of an operation fulfils its JML specification, i.e., pre- and post-conditions. Since the JML annotations are generated from the visual contracts, we indirectly verify that the behavioural code complies with the visual contract of the design model.

During *service registration* service developers publish specifications based on graph transformations to make services available to potential clients. When specifications are uploaded, discovery services automatically generate test cases and oracles, to verify that services satisfy their expectations. Test cases are executed against target services and oracles evaluate results. In some cases, services under test may need to implement special testing interfaces to let oracles inspect their internal states, to verify the correctness of the results. These interfaces implement a *get* operation that returns the current state of the web service, according to its behavioural descriptions. If necessary, these interfaces can include a *set* operation that assigns a given state to the target web service. Since both the signature of getter/setter methods and the structure of the web service state are known a priori, these interfaces can be automatically generated.

If a service does not pass all test cases, registries generate a report which is sent to service developers and they cancel service registration. Otherwise, if a

quantifications. Different tools are available to support, e.g., runtime assertion checking, testing, or static verification based on JML.

service passes all tests, registries complete registration and inform service developers that they can turn off any testing interface. This protocol guarantees that only high-quality Web services can register. If service providers modify the behaviour of their services, they must provide a new specification and repeat registration and testing. Service providers are discouraged to change the behaviour of services without publishing updated specifications because clients would discover services by referring to outdated specifications, and thus they would be unable to interact with these services. Moreover, client-side monitors can automatically detect anomalous behaviour to prevent clients from interacting with unsound implementations of a published specification.

During *service discovery* clients retrieve and bind to services. To this end, service requestors submit queries to discovery services. Discovery services automatically process requests and if any of the specifications satisfies the queries, references to the corresponding services are returned to clients. In our framework, both queries and specifications are visually expressed by graph transformations. Thus, developers use a coherent environment at both client and service provider sites.

While *using a service*, clients can download the service description (service specifications) and generate a client-side monitor to verify the correctness of its behaviour. A client-side monitor is similar to a client-side proxy, which can be generated from Web service engines like Apache Axis [1]. Additionally, the client-side monitor embeds JML assertions and checks if requests and results exchanged between clients and services satisfy expectations by wrapping the invocation of the service of the client.

Thanks to these refinements of the standard life cycle, clients have the guarantee of interacting with web services that have been verified against their requirements. Moreover, any deviations from the expected behaviour are revealed by client-side monitors.

The life-cycle of a web service described in this section allows the development of reliable web services. It helps developers of services to produce reliable services by generating test cases and JML assertions, which can be used to monitor the implementation at the provider side, e.g., during testing. During registration, automatic generation of test cases is able to detect and reject incorrect services. Client-side monitors help to ensure that the behaviour of a service remains consistent with its description.

Even if we only present how to generate Java and JML code from our models in this chapter, the overall framework does not mandate a specific programming language. The service descriptions exchanged between the service provider, service requester and the discovery service are not platform-specific, and the communication between them can also be based on platform independent XML-dialects like SOAP [39]. To be completely platform independent, we only need to adjust our code and test generators to support multiple programming languages. A translation of graph transformation rules to Microsoft's Spec# [4] (adds the idea of contracts to Microsoft's C# [32]) is also possible as shown in [44].

7.3 Web Service Specification

In this section, we describe the specification-related concepts underlying our modelling approach using the example of a hotel reservation system. This system is able to create a number of reservations for different hotels and manages them in a reservation list for each customer. For example, this allows John to organise a round trip, while visiting different hotels and celebrating the purchase of his new car. After John has finished planning his trip he can commit the reservation list. Later activities such as payment or check-in are not part of our example.

In our approach, a design model consists of a static and a functional view.

7.3.1 Modelling Static Aspects

UML class diagrams are used to represent the static aspects in our design model. Figure 7.2 shows the class diagram of our hotel reservation system. We use the stereotypes `control`, `entity` and `boundary`. Each of these stereotypes expresses a different role of a class in the implementation. Instances of control classes encapsulate the control flow related to a specific complex activity, coordinating simpler activities of other classes. Entity classes model long-lived or persistent information. Boundary classes are used to model non-persistent information that is exchanged between a system and its actors. The stereotype `key` indicates key attributes of a class. A key attribute is a

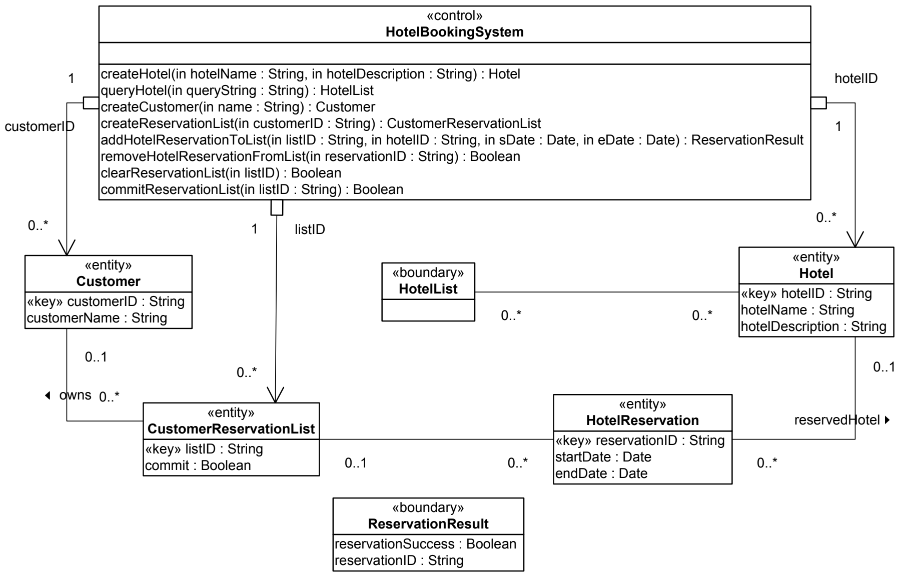
«control»
HotelBookingSystem
createHotel(in hotelName : String, in hotelDescription : String) : Hotel
queryHotel(in queryString : String) : HotelList
createCustomer(in name : String) : Customer
createReservationList(in customerID : String) : CustomerReservationList
addHotelReservationToList(in listID : String, in hotelID : String, in sDate : Date, in eDate : Date) : ReservationResult
removeHotelReservationFromList(in reservationID : String) : Boolean
clearReservationList(in listID) : Boolean
commitReservationList(in listID : String) : Boolean
1
customerID
0..*
hotelID
1
0..*
1
listID
«entity»
Customer
«key» customerID : String
customerName : String
«boundary»
HotelList
0..*
0..*
«entity»
Hotel
«key» hotelID : String
hotelName : String
hotelDescription : String
0..1
0..1
◂ owns
0..*
0..*
«entity»
CustomerReservationList
«key» listID : String
commit : Boolean
0..1
0..*
«entity»
HotelReservation
«key» reservationID : String
startDate : Date
endDate : Date
reservedHotel ▸
0..*
«boundary»
ReservationResult
reservationSuccess : Boolean
reservationID : String

Fig. 7.2. Class diagram specifying the static structure of the hotel reservation system

unique identifier for a set of objects of the same type. A small rectangle associated with an association ending with a qualifier (e.g. `hotelID`) designates an attribute of the referenced class. In combination with the attributes, the qualifier allows us to get direct access to a specific object. For instance, the control class `HotelBookingSystem` is connected to the entity classes of the system via qualified associations.

7.3.2 Modelling Functional Aspects

Class diagrams are complemented by graph transformation rules that introduce a *functional view*, integrating static and dynamic aspects. They allow us to describe the pre-conditions and effects of individual operations, referring to the (conceptual) data state of the system. Graph transformation rules are formed over the classes of the design class diagram and are represented by a pair of UML object diagrams, specifying pre- and post-conditions. The use of graph transformations to specify the functional view of services is a key aspect of our approach because they enable specification of the service behaviour, automatic generation of test cases, automatic generation of monitors and specification of visual queries.

In particular, the functional view is used to (formally) match the behaviour required by the client and the behaviour offered by the server, at the discovery service side. When a client uploads a description of the required behaviour, the service discovery analyses all available specifications and responds with the list of all compatible services. The required behaviour is visually defined by the client as a set of graph transformation rules that represent the operations that must be implemented by returned services. The functional view of a service is also used during the registration phase to automatically generate test cases and oracles. Test cases are generated by the discovery service that executes them and rejects the requests for registration of services that do not pass all test cases. Finally, the modelling of the functional aspect is used both from the server and the client to automatically generate monitors that verify at runtime if the observed interactions satisfy expectations. Any violation is signalled to the client (server) that, in case of problems, can bind to another service (can search and repair the fault).

In the following, we will introduce graph transformation rules through a number of examples. The operation `createReservationList` of the control class `HotelBookingSystem` creates a new reservation list for an existing customer. Figure 7.3 shows a graph transformation rule that describes the behaviour of the operation. The rule is enclosed in a frame, containing a heading and a context area. The frame notation originates from UML 2.0, providing a portable context for a diagram. The heading is a string enclosed in a rectangle with a cutoff corner, placed in the upper left corner of the frame. The keyword `gtr` refers to the type of diagram, in this case a graph transformation rule. The keyword is followed by the name of the operation specified by the rule, in turn followed by a parameter list and a return parameter, if declared in

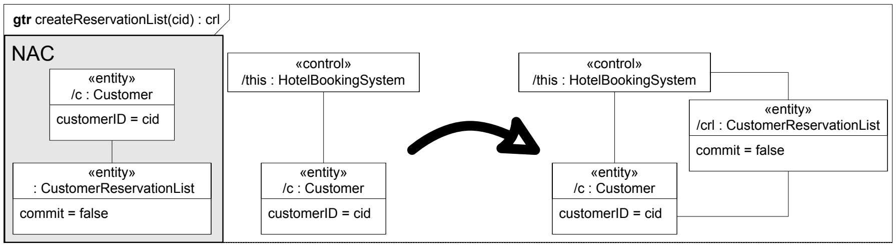

Fig. 7.3. Graph transformation rule for operation `createReservationList`

the class diagram. All parameters occur in the graph transformation rule. An extension of the UML 2.0 metamodel for graph transformation rules of this form can be found in [16].

The graph transformation rule itself is placed in the context area. It consists of two object diagrams, its left- and right-hand side, both typed over the design class diagram. The basic idea is to consider the left-hand side as a pattern, describing the objects, attributes and links that need to be present for the operation to be executable. Then, all items only present in the left- but not in the right-hand side of the rule are removed, and all present only in the right-hand side are newly created. Objects present in both sides are not affected by the rule, but required for its application. If there is only one object of a certain type, it can remain anonymous; if a distinction between different objects of the same type is necessary, then there must be an object identifier separated from the type by a colon.

We may extend the pre- or post-conditions of a rule by negative preconditions [20] or post-conditions. A negative condition is represented by a dark rectangle in the frame. If the dark rectangle is on the left of the precondition, it specifies object structures that are not allowed to be present before the operation is executed (negative pre-condition). If the dark rectangle is on the right of the post-condition, it specifies object structures that are not allowed to be present after the execution of the operation (negative post-condition). A detailed explanation of graph transformation rules can be found in [13].

The graph transformation rule as described in Fig. 7.3 expresses the fact that the operation `createReservationList` can be executed if the `HotelBookingSystem` object references an object of type `Customer`, which has an attribute `customerID` with the value `cid`. The concrete values are specified when the client calls the operation. The negative pre-condition additionally requires that the object `c:Customer` be not connected to an object of type `CustomerReservationList` that has the value `false` for the attribute `commit`. That means, the system only creates a new reservation list for an existing customer if there is no reservation list for the customer, which is not yet committed. As an effect, the operation creates a new object of type `CustomerReservationList` and two links between the objects of types

HotelBookingSystem and CustomerReservationList as well as Customer and CustomerReservationList. As indicated by the variables used in the heading, the object crl:CustomerReservationList becomes the return value of the operation createReservationList. The active object, executing the method, is designated by the variable this.

Figure 7.4 shows a functional specification of the operation addHotelReservationToList by two graph transformation rules. If the operation is successfully executed, it adds a new hotel reservation to the reservation list of the customer. If the operation is not successfully executed, it does nothing. The pre-conditions of both rules are identical. That means, from an external point the resulting behaviour is non-deterministic. A client does not know whether the hotel reservation system will create a new reservation or not. The reason is that the decision depends on the availability of the hotel, which is not known in advance. For a successful execution of the operation, the object this must know two different objects with the following characteristics: an object of type Hotel which has an attribute hotelID with the value hid, an object of type CustomerReservationList which has an attribute listID with the value lid and an attribute commit with the value false. If the requested hotel is available, the operation creates a new object HotelReservation and initialises its attributes startDate and endDate

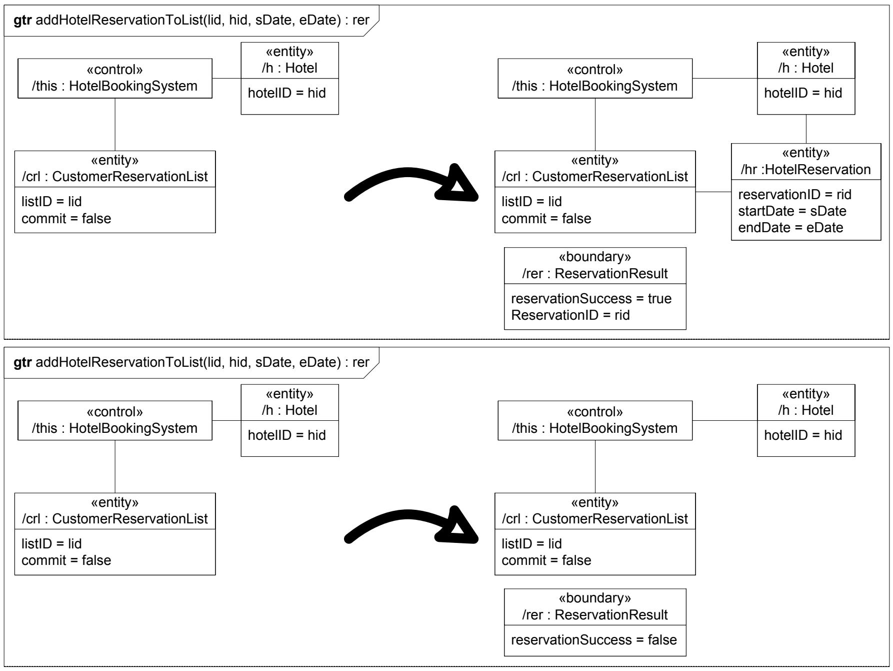
gtr addHotelReservationToList(lid, hid, sDate, eDate) : rer
«control»
/this : HotelBookingSystem
«entity»
/h : Hotel
hotelID = hid
«entity»
/crl : CustomerReservationList
listID = lid
commit = false
«entity»
/hr :HotelReservation
reservationID = rid
startDate = sDate
endDate = eDate
«boundary»
/rer : ReservationResult
reservationSuccess = true
ReservationID = rid
gtr addHotelReservationToList(lid, hid, sDate, eDate) : rer
reservationSuccess = false

Fig. 7.4. Graph transformation rule for operation addHotelToList

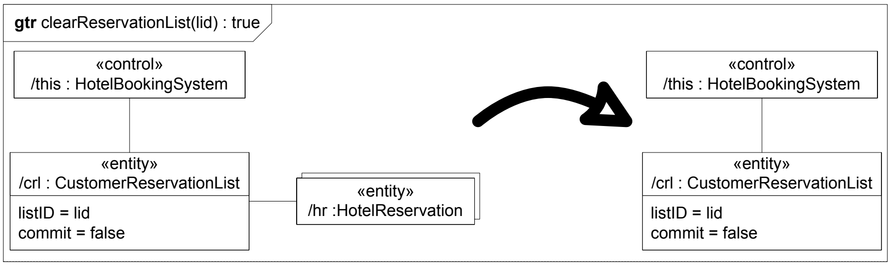
gtr clearReservationList(lid) : true
«control»
/this : HotelBookingSystem
«entity»
/crl : CustomerReservationList
listID = lid
commit = false
«entity»
/hr :HotelReservation
«control»
/this : HotelBookingSystem
«entity»
/crl : CustomerReservationList
listID = lid
commit = false

Fig. 7.5. Graph transformation rule for operation `clearReservationList`

according to the parameter values (see top rule in Fig. 7.4). This new object is linked to the objects `h:Hotel` and `crl:CustomerReservationList` identified in the pre-condition. Additionally, the object creates a new boundary object of type `ReservationList`, initialises its attributes and uses this object as return value. Generally, the boundary object is used to group different return values into one return object. If the requested hotel is not available, the service only creates a boundary object `rer:ReservationResult` and sets the value of its attribute to `false` (see bottom rule in Fig. 7.4). This allows to show the client that the reservation has not been successful.

Universally quantified operations, involving a set of objects whose cardinality is not known at design time, can be modelled using multi-objects. An example is shown in Fig. 7.5. This rule specifies an operation which removes all `HotelReservations` from an existing, not committed `CustomerReservationList`. The multi-object `hr:HotelReservation` in the pre-condition indicates that the operation is executed if there is a set (which maybe empty) of objects of type `HotelReservation`. After the execution of the operation, all objects conforming to `hr:HotelReservation` (as well as the corresponding links) are deleted, i.e., the reservation list is cleared.

Figure 7.6 shows the remaining graph transformation rules for the operations of the hotel reservation system.

7.4 Web Service Registration

As outlined in Sect. 7.2, registration of web services includes a testing phase where registries automatically generate, execute and evaluate test cases. Only if all test cases are passed, the registration phase is successfully completed; otherwise, registration is aborted. In both cases, a report is sent to service owners.

Execution of test cases can require the implementation of ad hoc interfaces that are used by registries to set and reset the state of Web services, when normal interfaces do not support all necessary operations. In particular, test case execution usually requires a reset operation to clean the current state, a

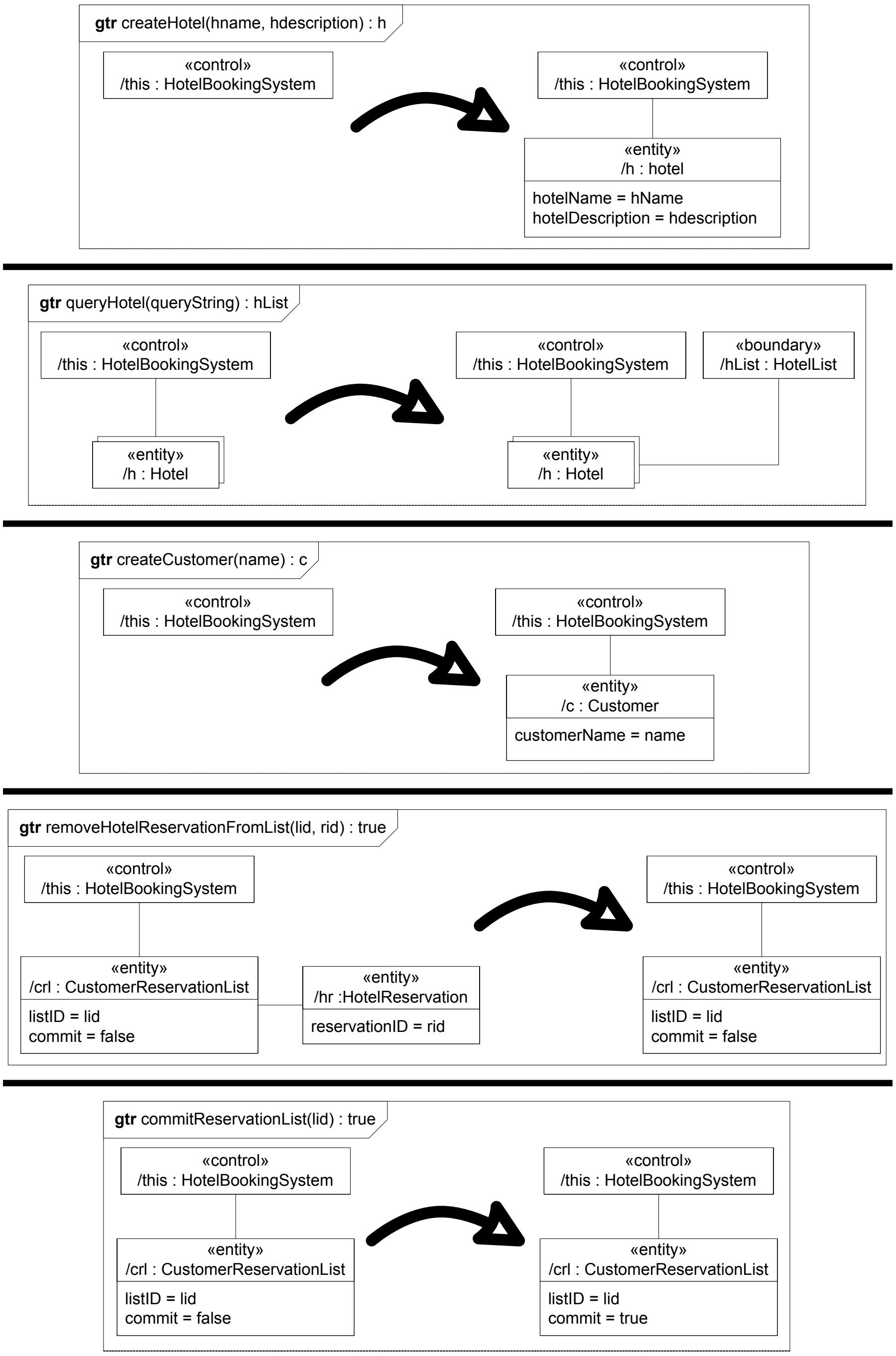
gtr createHotel(hname, hdescription) : h
«control»
/this : HotelBookingSystem
«control»
/this : HotelBookingSystem
«entity»
/h : hotel
hotelName = hName
hotelDescription = hdescription
gtr queryHotel(queryString) : hList
«control»
/this : HotelBookingSystem
«entity»
/h : Hotel
«control»
/this : HotelBookingSystem
«boundary»
/hList : HotelList
«entity»
/h : Hotel
gtr createCustomer(name) : c
«control»
/this : HotelBookingSystem
«control»
/this : HotelBookingSystem
«entity»
/c : Customer
customerName = name
gtr removeHotelReservationFromList(lid, rid) : true
«control»
/this : HotelBookingSystem
«entity»
/crl : CustomerReservationList
listID = lid
commit = false
«entity»
/hr :HotelReservation
reservationID = rid
«control»
/this : HotelBookingSystem
«entity»
/crl : CustomerReservationList
listID = lid
commit = false
gtr commitReservationList(lid) : true
«control»
/this : HotelBookingSystem
«entity»
/crl : CustomerReservationList
listID = lid
commit = false
«control»
/this : HotelBookingSystem
«entity»
/crl : CustomerReservationList
listID = lid
commit = true

Fig. 7.6. Remaining graph transformation rule for operations of the hotel reservation system

creational interface to transform the target service into a given state (similarly to a setter method) and an inspection interface, to access the internal state of Web services (similarly to a getter method). Once testing has been passed, the testing interface can be disabled.

Automatic test case generation validates the behaviour of a given Web service by addressing two aspects: correctness of single operations, e.g., booking a hotel room, and correctness of multiple operations, e.g., fully managing a reservation list [26].

The registration phase also includes the generation of client-side monitors. The following sections present techniques for generation of test cases and monitors.

7.4.1 Test Cases for Single Operations

The result of an operation depends on both its inputs and the current state of the service. Admissible inputs are defined by operation signatures, which constrain each variable with a type, while states are defined by class diagrams limiting the types and relations of objects. This information is complemented by transformation rules that specify the pre-conditions and effects of the operations.

Testing single operations means executing them on samples from their domains to evaluate the general correctness of their behaviour. Since transformation rules provide information that allows the identification of different sets of “equivalent” inputs, we generate test cases by a domain-based strategy [54] known as partition testing. The rationale is that test cases can be suitably selected by dividing operations’ domains into (possibly overlapping) subsets and choosing one or more elements from each subset [53]. Inputs from each domain should trigger sets of equivalents behaviours, according to Web service specifications.

Usually, input domains are identified by following fault-based guidelines, identifying small partitions, where several insidious faults can be present, and large partitions, where little assumptions about specific implementation threats can be made [53]. In case of operations specified by graph transformations, if op_i are the transformation rules that define the behaviour of an operation and the pre-condition of each rule is indicated with pre_i, we can identify the following domains:

- completeDomain: Each pre_i is a domain. Selecting at least one input from each pre_i guarantees the execution of all transformation rules.
- multiRules: Any $pre_i \cap pre_j \neq \varnothing$ is a domain. It represents the case of an input that can potentially trigger either of two rules. The choice is internal to the Web service, and can eventually be non-deterministic. The identification of the rule that must be triggered is a potential source of problems, coverage of these domains guarantees execution of all possible decision points.

- boundaryValues: Any pre_i can specify conditions on node attributes. Since many faults are likely to arise when attributes assume values at boundaries of their domain, a separate domain is represented for each input where at least one attribute assumes a boundary value.
- multiObjects: pre_i can contain multi-objects, which are satisfied by inputs with any cardinality of nodes. The operation must be able to suitably manage any input. We identified three domains that must be covered when a multi-object is part of a pre-condition: inputs with 0 elements, with 1 element and with more than 1 element.
- unspecified: Input values that do not satisfy any pre_i, but conform to the constraints represented by the operation signature. In these cases, a target Web service should respond by both signalling incorrectness of the input and leaving its state unchanged.

Note that domains are not disjoint because the same input can reveal a failure for multiple reasons.

Given an operation *op* and its rules op_i, we derive test cases by generating a set of triples (in, seq, out), where *in* is an input that belongs to one of the domains associated with *op*, *seq* consists of an invocation to *op*, and *out* is the expected result, which can be any $op_i(in)$, where *in* satisfies the pre-condition of op_i. Test cases must cover all domains. Moreover, different domains can be covered with different numbers of test cases, according to the tester's preference. For instance, we can cover the "completeDomain" with four test cases, and the "multiObjects" domain with one test case. Concrete attribute values are randomly generated taking into account constraints associated with rules. We only consider linear constraints; however, extensions to non-linear constraints can be incorporated as well [30]. Values from the *unspecified* domain are obtained by considering a transformation rule, and generating inputs that preserve the structure of the pre-condition of the rule, but includes attribute values that violate at least one constraint associated with the pre-condition.

For example, if we generate test cases for the rule `clearReservationList(lid)` shown in Fig. 7.5, we can identify the following domains:

- completeDomain: The operation is specified with one rule, thus the technique identifies only one domain that corresponds to the pre-condition of `clearReservationList(lid)`.
- multiRules: Since we have only one rule, there is no input that can potentially trigger multiple rules. Thus no domain is selected.
- boundaryValues: The rule includes only one unspecified attribute value, which is `listId`. The type of this attribute is `String`. Thus, two domains with `String`s of minimum and maximum length are considered (the maximum length can be either defined by the tester or inherited from the specification of the String type).

- multiObjects: The rule includes one multi-object. Thus the technique generates three domains: one with an empty set of `HotelReservation`, one with a single `HotelReservation` and one with multiple `HotelReservations`.
- unspecified: The only constraint about attribute values that can be violated is `commit=false`. Thus, the technique generates a domain with `commit=true`. This is an interesting test case because Web service developers may erroneously assume that the `clearReservationList` operation can be executed on commited reservation lists.

The exact number of test cases depends on the amount of samples that are extracted from each domain. For instance, if we extract 4 samples from *completedomain*, 1 sample from *boundaryValues*, 1 sample from *multiObjects* and 1 sample from *unspecified*, we obtain $4 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 1 = 10$ test cases.

7.4.2 Test Cases for Operation Sequences

To test the effect of sequences of operations, we analyse the relation between transformation rules. A sequence of rule applications leads to a sequence of transformations on the data state of the service. Typically, when state-dependent behaviour has to be tested, data-flow analysis is used to reveal state-dependent faults by exercising variable definitions and uses [19]. A similar idea can be applied to graph transformation rules.

In particular, given two transformation rules p_1 and p_2, p_1 may disable p_2 if it deletes or adds entities that are required or forbidden by p_2, respectively. In this case, we say that a conflict between p_1 and p_2 exists. Given two transformation rules p_1 and p_2, p_1 may cause p_2 if it deletes or adds state entities that are forbidden or required by p_2, respectively. In this case, we say that a dependency between p_1 and p_2 exists. For example, a dependency between rule `addReservationToList`, shown in Fig. 7.4, and rule `createReservationList`, shown in Fig. 7.3, exists. This is because the former rule can be applied only if the state includes a `CustomerReservationList` node, which can be created by the latter rule. Moreover, a conflict between rule `clearReservationList`, shown in Fig. 7.5, and rule `removeHotelReservationFromList`, shown in Fig. 7.6, exists. This is because the former rule deletes the `HotelReservation` node, which is required by the latter rule.

Our technique addresses testing of sequences of operations by covering dependencies and conflicts between rules. Sequences of operations that do not include any dependency or conflict are likely to be sequences of independent operations. Thus, they have been already covered by testing of the single operations.

To turn the test requirement to cover a sequence of two rules $\langle p_1, p_2 \rangle$ into an executable test case, we must solve a search problem. In particular, the pre-condition of rule p_1 may not be satisfied by the initial state of the target web service (this happens also for testing of single operations). Moreover, the state that results from the execution of rule p_1 may not allow the immediate execution of rule p_2. Thus, we need to identify two sequences of operations: seq_{pre}, which brings the Web service from the initial state to a state that satisfies the pre-condition of p_1, and seq_{betw}, which brings the web service from the state that results from the execution of p_1 to a state that satisfies the pre-condition of p_2. The sequence seq_{betw} should not modify the entities that are part of the conflict/dependency between p_1 and p_2, otherwise the dependency between the two transformations would be removed.

Conflicts and dependencies can automatically be identified by the AGG tool [51], while the search for sequences seq_{pre} and seq_{betw} can be supported by tools like PROGRESS [47] and GROOVE [46]. The existence of testing interfaces can simplify the solution to the search problem. Early experience presented in [26] shows that the problem is feasible at the level of complexity of several common Web service APIs.

For example, if we focus on rule `createReservationList`, shown in Fig. 7.3, we can automatically identify the following dependencies

- ⟨`createCustomer, createReservationList`⟩, because `createCustomer` creates the `Customer` node, which is required by the pre-condition of `createReservationList`.
- ⟨`commitReservationList`, `createReservationList`⟩, because `commit-ReservationList` modifies the value of the `commit` attribute from `false` to `true`, and the `createReservationList` rule forbids the presence of a `CustomerReservationList` with `commit` equals to `false`.

and conflicts:

- ⟨`createReservationList, createReservationList`⟩, because the first `createReservationList` creates a `ReservationList` with `commit` equals to `false`, which is forbidden by the second `createReservationList`.

Examples of concrete test cases that can be generated from the test requirements above are (attribute values are omitted):

- $TC1 =$ `createCustomer; createReservationList`
- $TC2 =$ `createCustomer; createReservationList;`
  `commitReservationList; createReservationList`
- $TC3 =$ `createCustomer; createReservationList;`
  `createReservationList`

7.4.3 Test Oracles for Services

Oracles evaluate the correctness of the test result by comparing the expected return values and post states with those produced by the test of the service. Graph transformation rules can be translated into JML assertions that verify consistency of runtime behaviour and specification. The mapping for generating JML assertions is presented in detail in Sect. 7.6.

Clients can use this technology to create client-side monitors. A client-side monitor is a stub with embedded assertions. In this case, JML assertions can only check data values sent and received by clients, and cannot inspect the web service state. Verification of internal behaviour of Web services is possible using server-side monitors. For example, a client-side monitor can verify that the `createCustomer` operation, shown in Fig. 7.6, returns a `customer` object with a name equal to the string passed as parameter, but cannot verify if the same object is part of the internal state of the web service.

All violations revealed by the client-side monitors are recorded, to be accessed by the developers of the service or prospective clients, to identify and fix problems, to adapt client applications or to select new web services.

7.5 Web Service Discovery

An important part of our approach, albeit not the focus of this book chapter, is the discovery of services based on their semantic descriptions. Current standards already enable much of the discovery process, but they concentrate largely on syntactic service descriptions. However, service requestors can be assumed to know what kind of service they need (i.e. its semantics), but not necessarily how the service is actually called (i.e. its syntax). Thus, a provider must be able to formulate a semantic request and a discovery service must be able to match a semantic service description to a corresponding request.

We will give only a brief overview of how our approach enables the semantic discovery of services. The interested reader is referred to previous publications on the discovery of services specified by graph transformation rules [23, 24].

In our approach, graph transformation rules serve as both description of an offered service and formulation of a request. From a provider's point of view, the left-hand side of the rule specifies the pre-condition of the provider's service (i.e. the situation that must be present or the information that must be available for the service to perform its task). The right-hand side of the rule depicts the post-condition (i.e. the situation after the successful execution of the web service). From a requester's point of view, the left-hand side of the rule represents the information the requester is willing to provide to the service, and the right-hand side of the rule represents the situation the requester wants to achieve by using the service.

Matching the rules of a provider with those of a requestor means deciding whether a service provider fulfils the demands of a service requestor and vice versa. Informally, a provider rule matches a requestor rule if (1) the requestor is willing to deliver the information needed by the provider and in turn (2) the provider guarantees to produce the results expected by the requestor.

We have formalized this informal matching concept using contravariant subgraph relations between the pre- and post-conditions of the rules of the service provider and the requestor [24]. In short, the requester is willing to deliver the information needed by the provider if the latter's pre-conditions is a subgraph of the requestor's pre-conditions. The provider produces the results expected by the requestor if the provider's post-condition is a subgraph of the requestor's post-condition. That means, the requestor is allowed to offer more information than needed by the provider and the provider can produce more results than needed by the requestor.

A prototypical implementation of our approach is available [23] using DAML+OIL [12] as semantic web language for representing specifications at the implementation level. Matching is based on the RDQL (RDF Data Query Language) [48] implementation of the semantic web tool Jena by HP [27]. RDQL is a query language for specifying graph patterns that are evaluated over a graph to yield a set of matches. A visual editor for graph transformation rules has been implemented, to support the creation of models [34, 16].

7.6 Web Service Monitoring

The loose coupling of services in a service-oriented application requires the verification that a service satisfies its description not only at the time of binding, but also that it continues to do so during its life time. We propose to use a monitoring approach to continuously verify services at runtime.

Monitors are derived from models, with class diagrams describing the structure and graph transformations describing the behaviour of services. In the following, we describe a translation of models into JML constructs to enable a *model-driven monitoring* [16, 25, 36].

7.6.1 Translation to JML

Class diagrams are used to generate static aspects of Java programs, like interfaces, classes, associations and signatures of operations. The transformation of graph transformations into JML constructs makes the graph transformations observable in the sense that they can be automatically evaluated for a given state of a system, where the state is given by object configurations. In the following, we will concentrate on the code generation for the service provider. The code generation for the client side monitors works similar and will not be discussed in detail in this book chapter. The requestor side monitors can be obtained by restricting the generation of JML assertion to the ones that include only references to parameters. Thus, any assertions with references to any other state variables will not part of the client-side monitor.

Translation of UML Class Diagrams to Java

Given a UML class diagram, we assume that each class is translated to a corresponding Java class. In the following, we will focus on the characteristics of such a translation that we need for explaining our mapping from graph transformation rules to JML.

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 [45], 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. Furthermore, 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.

Associations are translated by adding an attribute with the name of the association to the respective classes. For handling, e.g., the association `owns` of Fig. 7.2, a private variable `owns` of type `Customer` is added to the class `CustomerReservationList`. Again, appropriate access methods are added to the Java class. Because the UML association `owns` is bidirectional, we additionally add an attribute named `revOwns` 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. In particular, we use the class `TreeSet` as implementation of the sub-interface `Set` of `Collection`. A set differs from a collection in that it contains 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. In addition, we provide access methods for adding and removing elements. Examples are the access methods `addCustomerReservationList` or `removeCustomerReservationList`. To check the containment of an element, we add operations like `hasCustomerReservationList`. In case of qualified attributes, we access elements via keys by adding additional methods like `getCustomerReservationListByID`. As described in [18], 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.

Translation of Graph Transformation Rules to JML

For the transformation of graph transformation rules into JML, we assume a translation of design class diagrams to Java as described above. Listing 7.1 shows how a method is annotated with a JML specification. The behavioural information is specified in the Java comments. Due to their embedding into Java comments, the annotations are ignored by a normal Java compiler. The keywords `public normal_behavior` state that the specification is intended for clients, and that if the pre-condition is satisfied, a call must return normally, without throwing an exception. JML pre-conditions follow the keyword

```
public class A {

  ...

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

  ...

}
```

Listing 7.1. Format for specifying pre- and post-conditions by JML

`requires`, and post-conditions follow the keyword `ensures`. Both `JML-PRE` and `JML-POST` are Boolean expressions. The pre-condition states what conditions must hold for the method arguments and other parts of the state of the systems. If the pre-condition is true, then the method must terminate in a state that satisfies the post-condition.

If a JML construct represents a visual contract, the JML's pre- and post-conditions must be interpretations of the graphical pre- and post-conditions. When a JML pre-condition (post-condition) is evaluated, figuratively an occurrence of the pattern that is specified by the pre-condition of the corresponding graph transformation rule has to be found in the current system data state. To find the pattern, a JML pre-condition (post-condition) applies a breadth-first search starting from the object `this`. The object `this` is the object that is executing the behaviour. If a JML pre-condition (post-condition) finds a correct pattern, it returns true, otherwise it returns false.

In Listing 7.2 the JML contract for verifying the visual contract of Fig. 7.3 is shown. Mainly, we test the pre- and post-conditions by nesting existence or universal quantifications that are supported by JML. Additionally, the negative application condition is nested into the pre-condition. The general syntax of JML's quantified expressions is given as `(\forAll T x; r ; p)` and `(\exists T x; r; p)`. The `forAll` expression is true if every object `x` of type `T` that satisfies `r` also satisfies `p`. The `exists` expression is true if there exists at least one object `x` of type `T` that satisfies `r` also satisfies `p`.

Next, we explain the JML-contract of Listing 7.2 in more detail. The pre-condition including the negative application condition is tested in lines 2–11. Lines 3–5 check if the active object (object `this` of type `HotelBookingSystem`) knows an object of type `Customer` with the value `cid` (parameter of the operation `createReservationList`) for the attribute `customerID`.

In lines 6–11 the negative application condition is checked. It is checked whether the previously identified customer (`c`) references an object of type

```
 1 /*@ public
 2 normal_behavior
 3   @ requires
 4   @ (\exists Customer c;
 5   @    this.getCustomer.values().contains(c);
 6   @    c.getCustomerID().equals(cid) &&
 7   @    !(
 8   @     (\exists CustomerReservationList crlNAC;
 9   @        c.owns.contains(crlNAC);
10   @        crlNAC.getCommit() == false
11   @      )
12   @     )
13   @   );
14   @
15   @ ensures (
16   @ (\exists Customer c;
17   @    this.getCustomer.values().contains(c);
18   @    c.getCustomerID().equals(cid) &&
19   @    (\exists CustomerReservationList crl;
20   @       this.customerReservationList.values().contains(crl);
21   @       crl.owns == c &&
22   @       crl.getCommit() == false &&
23   @      \result == crl
24   @     )
25   @   );
26   @*/
27 public CustomerReservationList createReservationList
28                                                        (String cid);
```

Listing 7.2. JML contract of operation `createReservationList` of Fig. 7.3

`CustomerReservationList` with the value `false` for the attribute `commit`. If such an object is found, then lines 6–11 return `false` (see `!` in line 7).

The post-condition is tested in lines 14–23. The objects of the post-condition are tested in the following order by the JML expression: `c:Customer` and `crl:CustomerReservationList`. Therefore, two JML-exists expressions are nested into each other. In line 22 whether the object `crl` is returned by the operation is tested. The JML-keyword `result` is used to denote the value returned by the method.

7.6.2 Runtime Behaviour

For enabling model-driven monitoring, we have bridged the gap between the model and the implementation level by the definition of a transformation of our visual contracts into JML assertions. On the implementation level we can take advantage of existing JML tools: The JML compiler generates assertion check methods from the JML pre- and post-conditions. The original, manual implemented methods are replaced by automatically generated *wrapper methods* and the original methods become a private method with a new name. The

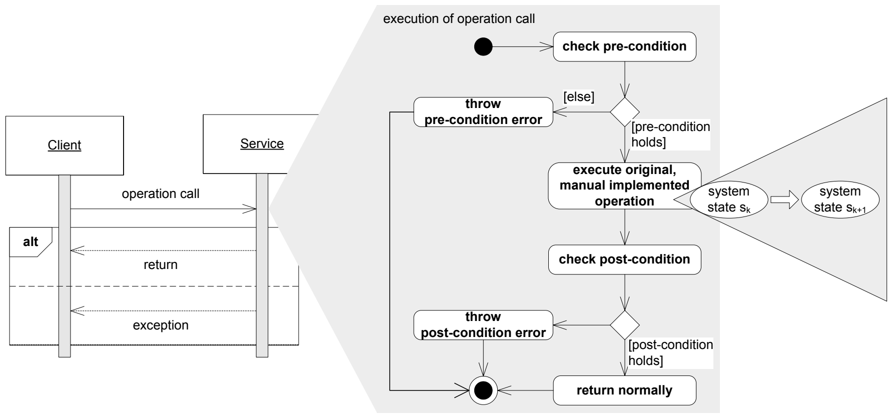

Fig. 7.7. Runtime behaviour of operation, model-driven monitoring approach

wrapper methods delegate client method calls to the original methods with appropriate assertions checks.

This leads to a runtime behaviour of an operation call as shown in Fig. 7.7. When a client calls an operation of a service, a pre-condition check method evaluates a method's pre-condition and throws a pre-condition violation exception if it does not hold. If the pre-condition holds, then the original operation is invoked. After the execution of the original operation, a post-condition check method evaluates the post-condition and throws a post-condition violation exception if it does not hold.

If an exception is thrown during the pre-condition test, then the client (routine's caller), although obligated by the contract to satisfy a certain requirement, does not satisfy it. This is a bug in the client itself; the routine is not involved. A violation of the post-condition means that the manual implemented operation was not able to fulfil its contract. In this case, the manual implementation contains a bug, the caller is innocent.

If the contracts are not violated at runtime, then automatic monitoring is transparent. The system state is only changed by the original operation. Our transformation of the visual contracts into JML ensures that the assertion checks generated by the JML compiler do not have any side effects on the system state. That is, except for time and space measurements, a correct implementation's behaviour is unchanged.

With the generated assertions, we can monitor the correctness of an implementation. If we want to take full advantage of our model-driven monitoring approach, a system needs to react adequately. As introduced before, an exception is thrown if a pre- or a post-condition is violated at runtime. The JML tools introduce the exception classes `JMLPreconditionError` and `JMLPostconditionError` to catch these exceptions. Listing 7.3 shows how to use these classes in an implementation. The operation at the beginning of the

```
try {
  shop.cartAdd(item, cartId);
} catch (JMLPreconditionError e) {
    System.out.println("Violation of precondition "
                                + e.getMessage());
} catch (JMLPostconditionError e) {
  System.out.println("Violation of postcondition "
                                + e.getMessage());
} catch (Error e) {
  System.out.println("Unidentified error!"
                                + e.getMessage();
}
```

Listing 7.3. Exceptions handling at development time

```
Violation of pre-condition by method OnlineShop.cartAdd
regarding specifications at
File "de\upb\dbis\amazonmini\OnlineShop.refines-java",
line 34, character 18 when
    'cid' is Cart_1
    'item' is de.upb.dbis.amazonmini.Item@ecd7e
    'this' is de.upb.dbis.amazonmini.OnlineShop@1d520c4
```

Listing 7.4. Example of an exception

`try-catch` block is an operation detailed by visual contracts on the design level. A programmer on the client side can now use these exception handling mechanisms to catch pre- and post-condition violations and implement an adequate reaction. Listing 7.4 shows the example of a message if a pre-condition is violated.

To summarise, with our model-driven monitoring approach we can build reliable (correct and robust) software systems. Correctness is the ability of a software system to behave according to its specification. Robustness is the ability to react to cases not included in the specification. At runtime, the generated JML assertions allow for the monitoring of the correctness. The generated exceptions allow a programmer to make a software system robust if it does not behave according to its specification.

7.7 Empirical Validation

The framework presented in this chapter has been used with several case studies: test case generation has been applied to publicly available web services and the monitoring technology has been used with web services provided by industrial partners. In the following, we summarize the results obtained so far. Details about these experiences are available in [26, 35, 14].

The technique for *automatic testing* has been applied to both a selection of web services available from `www.xmethods.com` and the Amazon web service. Since the GT-based specifications of these web services are not available, we manually specified the behaviour expected from these web services, and we used automatic testing to check if web service implementations conform with our expectations.

Testing showed that all web services, with the exception of the Kayak Paddle Guide web service, behave according to our expectations. The Kayak Paddle Guide web service returns the recommended length of a paddle given the height of the person who will use it. Testing revealed a fault that consists in suggesting a kayak of a maximum length, even if the input represents an incorrect height for a person.

Test case generation applied to the Amazon Web Service also revealed an incompatibility between its specification and the actual behavior of the web service. However, the incompatibility was due to our misinterpretation of the operation for adding items. Once the specification had been fixed, test cases did not identify any incompatibility.

This experience showed that the technique for test case generation scales well even with web services of non-trivial complexity, like the Amazon Web Service.

In an industrial case study [35, 14], we successfully applied graph transformations for specifying the interfaces of web services. In this case study, we used the web services of a business process for ordering new insurance contracts. We have been able to replace almost all previously created textual descriptions of web services by descriptions with graph transformations to allow for an efficient administration and monitoring of web services.

Moreover, we demonstrated the feasibility of the monitoring technology by implementing an in-house version of the Amazon Web service and generating its server-side monitors. The monitoring components worked fine and this experience demonstrated the feasibility of the technology.

7.8 Related Work

The vision of service-oriented architectures is that a program in need of some functionality (that it cannot provide itself) queries a central directory to obtain a list of service descriptions of potential suppliers of this functionality. The notion of service description is a central one in service-oriented architectures. A service description describes the functionality of a service. An important fact to note at this point is that a service requestor must know the *syntax* of a service to be able to call the service and additionally the service requestor must know the *semantics* of a service to be able to call the service correctly.

In this section, we will focus on the service descriptions, the usage of models and model-driven testing approaches in service-oriented architectures.

7.8.1 Service Descriptions

An interface definition is a technical description of the public interfaces of a web service. It specifies the format of the request and the response from the web service. The Web Service Description Language (WSDL) [11] proposed by the World Wide Web Consortium (W3C) is an XML (Extensible Markup Language) format for describing interfaces offered by a web service as a collection of operations. For each operation, its input and output parameters are described. The W3C refers to this kind of service description as the "documentation of the mechanics of the message exchange" [5, 7]. While these mechanics must be known to enable binding, a semantic description of services based on WSDL is not possible. WSDL only encodes the syntax of the web service invocation; it does not yield information on the service's semantics. Of course, human users might guess which service an operation (e.g. `orderBook(isbn:String)`) provides, but such explicit operation names are technically not required.

UDDI (Universal Description, Discovery, and Integration) [50], the protocol for publishing service descriptions, allows users to annotate their WSDL file with information about the service in the form of explanatory text and keywords. Using Keywords is one way of supplying semantics but not a reliable one, as there has to be a common agreement between requestors and providers about the meaning of the different keywords. The current state of web service technology is such that a developer solves these semantic problems by reading additional textual service descriptions in natural language.

Trying to describe a service with keywords also ignores the operational nature of services. When executing a service, one expects certain changes, i.e., the real world is altered in some significant way (e.g. an order is created, a payment is made or an appointment is fixed). Service descriptions should reflect this functional nature by providing a semantic description in the form of pre- and post-conditions (a style of description also known from contract-based programming) [38]. For a service-oriented architecture, this kind of semantic description can be found in [42, 49]. In both approaches the pre- and post-conditions are expressed in terms of specialised ontologies. While [42] shows the matching only for single input and output concepts, [49] combines a number of pre-defined terms to express the pre- and post-conditions (e.g. `CardCharged-TicketBooked-ReadyforPickup`). Using this style of description, it is possible to distinguish between rather similar services (e.g. booking a ticket, which is sent to the customer vs booking a ticket, which has to be picked up) without coining special phrases for each individual service in the ontology.

While this latter approach addresses human users, all previously mentioned solutions are directed towards machine-readable descriptions only. An important characteristic of our approach is its usability by mainstream software engineers.

7.8.2 Models

Models provide abstraction from the detailed problems of the implementation technologies and allow developers to focus on the conceptual tasks. In particular, visual structures can often be understood more intuitively than source code or formal textual descriptions. Thus, they are usually more effective for the communication between a service provider and a service requestor. Software engineers have long recognised this feature of visual languages and they make use of it. Especially, the diagrams of the industry standard Unified Modelling Language (UML) [41] have become very successful and accompanying software development tools are available. Further, models are an established part of modern software development processes. They are becoming more crucial with the advent of the Model Driven Architecture (MDA), since the MDA promotes generating implementation artefacts automatically from models, thus saving time and effort. Thus, a visual representation of pre- and post-conditions is a promising amalgamation that can be easily integrated into today's model-driven software development processes.

Since version 1.1, the UML standard comprises the Object Constraint Language (OCL) [40] as its formal annotation language for UML models. In contrast to the commonly used graphical diagrammatic UML sub-languages, OCL is a textual language. As a consequence, OCL expressions have a completely different notation for model elements than the diagrams of the UML. The types of constraints that can be expressed in OCL include invariants on classes and types as well as pre- and post-conditions of operations. However, OCL is of limited use even in organisations which employ UML diagrams. The limited readability of OCL and the difficulty of integrating a purely textual language like OCL with diagrams are important reasons for this situation.

Other proposals provide a visual counterpart of OCL by exploiting visualisations with set-theoretic semantics. Kent and Howse define a visual formalism for expressing constraints in object-oriented models. They first proposed constraint diagrams [31] that are based on Venn diagrams and visualised the set-theoretic semantics of OCL constraints. Later, Howse et al. advanced this approach towards Spider Diagrams [28] to support reasoning about diagrams. Visual OCL [8] is a graphical representation of OCL. Both proposals embed textual logic expressions in the diagrams, which leads to a hybrid notation of OCL constraints. In addition, the diagrams differ from the diagrams that are commonly used in organizations employing UML in software development and, thus, developers have to learn another visual language.

We rather prefer to represent practically relevant concepts of object constraints by using graph transformation rules. A graph transformation rule allows for the reuse of UML's object diagram notation. Thus, we have chosen a notation that is familiar to software developers and it easily integrates into today's software development processes.

7.8.3 Testing

There are several techniques for testing applications that include web services, see [10] for a survey on this topic. A few of these techniques investigated the development of enhanced UDDI servers that validate the quality of web services by executing test cases during the registration phase [6, 52]. In particular, Tsai et al. investigated the use of UDDI servers that execute test scripts developed by web service providers [52], and Bertolino et al. investigated the use of UDDI servers that generate test cases from Protocol State Machine diagrams [6]. The former technique focuses on validating a limited set of scenarios identified by testers at design-time, while the latter technique focuses on validating interaction protocols. These techniques follow a complemental point of view with respect to the one presented in this chapter, which focuses on validating if the effect of single invocations and invocation sequences modify the conceptual state of an external web service according to its specification. Moreover, existing techniques do not address the methodology aspect of the development.

We believe that high-quality applications can be obtained with thorough design and modelling of systems. UML is a well-known design language, and some of its diagrams, e.g., sequence diagrams and state charts, can be used for test case generation [43]. However, the generated test cases often fail to capture the concrete complexity of the exchanged parameters that are often restricted to few simple types, see for instance [22]. Moreover, due to the lack of a precise semantics, UML diagrams often need to be extended with some formalism that unambiguously defines the semantics of their elements.

Graph-transformations naturally integrate with UML diagrams, because they can be easily derived from UML design artefacts [3], they allow reasoning about behaviours of target applications and are suitable for test case generation. Moreover, graph transformations naturally address the complexity of both objects that can be exchanged between components and state objects.

A few approaches for test case generation from graph transformations exist [2, 55]. We advance these approaches in three ways: (1) we extend and apply domain-based testing and data-flow techniques to the case of graph transformations, (2) we generate executable test oracles from graph transformation rules and (3) we automatically test and validate web services. Finally, the idea of using registries which automatically test web services before registering seems to be original.

Automatic generation of assertions from models has been addressed in other works. For instance, different approaches show how to translate OCL constraints into assertions that are incorporated into Java code. Hamie [21] proposes a mapping of UML design class diagrams that are annotated with

OCL constraints to Java classes that are annotated with JML specifications. OCL is used to precisely describe the desired effects of the operations in terms of pre- and post-conditions. Invariants on the classes of a design model are also expressed using OCL. The Dresden OCL Toolkit [29] supports parsing, type checking and normalisation of OCL constraints. An application of the toolkit is a Java code generator that translates OCL constraints into Java code fragments that can compute the fulfilment of an OCL constraint. Even if different approaches facilitate the translation of OCL into executable contracts, OCL still lacks an easy-to-use representation.

7.9 Conclusions

Service-oriented applications are characterised by loosely coupled and dynamic interactions among participants. In particular, to obtain reliable and high-quality systems, service discovery, binding and integration must be extensively validated. Several approaches address quality problems in isolation, failing to provide a sound embedding of quality techniques into the life-cycle of services.

In contrast, our framework addresses coherent and model-driven development of high-quality service-based applications and its embedding into the service life-cycle. The resulting service-oriented architecture

- allows participation only of high-quality web services, i.e., web services that passed automatic testing;
- continuously monitors the behaviour of web services and instantaneously signals any violation of specifications;
- supports discovery based on behavioural descriptions rather than syntactical descriptions of interfaces;
- provides client-side monitoring facilities, which support clients in discovering integration problems and re-selecting new services if current services do not behave correctly.

We demonstrated the feasibility of our approach by applying the technologies that are part of the framework for high-quality service-based applications to several real web services.

References

1. Apache. Axis. `http://ws.apache.org/axis/`.
2. P. Baldan, B. König, and I. Stürmer. Generating test cases for code generators by unfolding graph transformation systems. In *proceedings of the 2nd International Conference on Graph Transformation*, Rome, Italy, 2004.
3. L. Baresi and R. Heckel. Tutorial introduction to graph transformation: a software engineering perspective. In *proceedings of the International Conference on Graph Transformation*, volume 1 of *LNCS*. Springer, 2002.

4. M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer-Verlag, 2004.
5. B. Benatallah, M.-S. Hacid, C. Rey, and F. Toumani. Semantic reasoning for web services discovery. In *proceedings of the WWW 2003 Workshop on E-Services and the Semantic Web (ESSW' 03)*, 2003.
6. A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. *Architecting Systems with Trustworthy Components*, chapter Audition of Web Services for Testing Conformance to Open Specified Protocols. Number 3938 in Lectures Notes in Computer Science Series. Springer, 2006.
7. D. Booth, H. Haas, F. McCabe, E. Newcomer, C. Michael, C. Ferris, and D. Orchard. Web services architecture - W3C working group note 11 february 2004. Technical report, W3C, 2004.
8. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. In M. Gogolla and C. Kobryn, editors, *proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes In Computer Science*, pages 257–271. Springer-Verlag, 2001.
9. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, February 2005.
10. G. Canfora and M. D. Penta. Testing services and service-centric systems: Challenges and opportunities. *IEEE IT Pro*, pages 10–17, March/April 2006.
11. R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language - W3C working draft 10 may 2005, May 2005.
12. D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. DAML+OIL (march 2001) reference description - W3C note 18 december 2001, March 2001.
13. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Chapter 3: Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars of Computing by Graph Transformation*. World Scientific, 1997.
14. G. Engels, B. Güldali, O. Juwig, M. Lohmann, and J.-P. Richter. Industrielle Fallstudie: Einsatz visueller Kontrakte in serviceorientierten Architekturen. In B. Biel, M. Book, and V. Gruhn, editors, *Software Enginneering 2006, Fachtagung des GI Fachbereichs Softwaretechnik*, volume 79 of *Lecture Notes in Informatics*, pages 111–122. Köllen Druck+Verlag GmbH, 2006.
15. G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A view-oriented approach to system modelling based on graph transformation. In *proceedings of the 6th European Conference held jointly with the International Symposium on Foundations of Software Engineering*, pages 327–343. Springer-Verlag, 1997.
16. G. Engels, M. Lohmann, S. Sauer, and R. Heckel. Model-driven monitoring: An application of graph transformation for design by contract. In *proceedings of the Third International Conference on Graph Transformations (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2006.
17. J. Fan and S. Kambhampati. A snapshot of public web services. *SIGMOD Record*, 34(1):24–32, March 2005.

18. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT)*, volume 1764 of *Lecture Notes In Computer Science*, pages 296–309. Springer Verlag, 1998.
19. P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.
20. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.
21. A. Hamie. Translating the object constraint language into the java modeling language. In *proceedings of the 2004 ACM symposium on Applied computing*, pages 1531–1535. ACM Press, 2004.
22. J. Hartmann, C. Imoberdorf, and M. Meisinger. Uml-based integration testing. In *proceedings of the 2000 international symposium on Software testing and analysis (ISSTA)*, pages 60–70. ACM Press, 2000.
23. J. H. Hausmann, R. Heckel, and M. Lohmann. Model-based discovery of Web Services. In *proceedings of the International Conference on Web Services (ICWS)*, 2004.
24. J. H. Hausmann, R. Heckel, and M. Lohmann. Model-based development of web services descriptions enabling a precise matching concept. *International Journal of Web Services Research*, 2(2):67–84, April-June 2005.
25. R. Heckel and M. Lohmann. Model-driven development of reactive informations systems: From graph transformation rules to JML contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006.
26. R. Heckel and L. Mariani. Automatic conformance testing of web services. In *proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer-Verlag, 2005.
27. Hewlett-Packard Development Company. Jena - a semantic web framework for Java. `http://jena.sourceforge.net/`.
28. J. Howse, F. Molina, J. Tayloy, S. Kent, and J. Gil. Spider diagrams: A diagrammatic reasoning system. *Journal of Visual Languages and Computing*, 12(3):299–324, June 2001.
29. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. *Science of Computer Programming*, 44:51–69, 2002.
30. B. Jeng and E. Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology*, 3:254–270, 1994.
31. S. Kent and J. Howse. Mixing visual and textual constraint languages. In R. France and B. Rumpe, editors, *proceedings of International Conference on The Unified Modeling Language (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 1999.
32. A. Kühnel. *Visual C# 2005.* Galileo Computing, 2006.
33. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Department of Computer Science, Iowa State University, February 2005.
34. M. Lohmann, G. Engels, and S. Sauer. Model-driven monitoring: Generating assertions from visual contracts. In *proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 355–356, September 2006.

35. M. Lohmann, J.-P. Richter, G. Engels, B. Güldali, O. Juwig, and S. Sauer. Abschlussbericht: Semantische Beschreibung von Enterprise Services - Eine industrielle Fallstudie. Technical Report 1, Software Quality Lab (s-lab), Unversity of Paderborn, May 2006.
36. M. Lohmann, S. Sauer, and G. Engels. Executable visual contracts. In M. Erwig and A. Schürr, editors, *proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 63–70, 2005.
37. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.
38. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, second edition, 1997.
39. N. Mitra. SOAP version 1.2 part 0: Primer - W3C recommendation 24 june 2003, Juni 2003.
40. OMG (Object Management Group). UML 2.0 OCL final adopted specification, 2003.
41. OMG (Object Management Group). UML 2.0 superstructure specification - revised final adopted specification, 2004.
42. M. Paolucci, T. Kawmura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. A. Hendler, editors, *proceedings of the First International Semantic Web Conference on the Semantic Web*, volume Lecture Notes In Computer Science; Vol. 2342, pages 333–347, Sardinia, Italy, 2002. Springer-Verlag.
43. M. Pezzè and M. Young. *Software Test and Analysis: Process, Principles and Techniques.* John Wiley and Sons, 2007.
44. M. Raacke. Generierung von spec#-code aus visuellen kontrakten, October 2006. Bachelor Thesis at the University of Paderborn.
45. A. Reddy. Java coding style guide. Technical report, 2000.
46. A. Rensink. The GROOVE simulator: A tool for state space generation. In *2nd Intl. Workshop on Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 479–485. Springer, 2004.
47. A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: language and environment. In *Handbook of graph grammars and computing by graph transformation: vol.2: applications, languages, and tools*, pages 487–550. World Scientific, 1999.
48. A. Seaborne. RDQL - a query language for RDF - W3C member submission 9 january 2004. Technical report, W3C, 2004.
49. K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding semantics to web services standards. In L.-J. Zhang, editor, *proceedings of the International Conference on Web Services, ICWS '03*, pages 395–401, Las Vegas, Nevada, USA, 2003. CSREA Press.
50. O. U. S. TC. UDDI version 3.0.2. OASIS standard, Organization for the Advancement of Structured Information Standards, 2004.
51. Technical University Berlin. The attributed graph grammar system (AGG). `http://tfs.cs.tu-berlin.de/agg/`.
52. W. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao. Verification of web services using an enhanced UDDI server. In *proceedings of the IEEE International Workshop on Object-oriented Real-time Dependable systems*, 2003.
53. E. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17:703–711, 1991.
54. L. White and E. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6:247–257, 1980.

55. J. Winkelmann, G. Taentzer, K. Ehrig, and J. Küster. Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. In *proceedings of the International Workshop on the Graph Transformation and Visual Modeling Techniques*, Electronic Notes in Theoretical Computer Science, 2006.