Automatic Conformance Testing of Web Services

Reiko Heckel and Leonardo Mariani

Department of Mathematics and Computer Science,
University of Paderborn,
33095 Paderborn, Germany
{heckel, mariani}@upb.de

Abstract. Web Services are the basic building blocks of next generation Internet applications, based on dynamic service discovery and composition. Dedicated *discovery services* will store both syntactic and behavioral descriptions of available services and guarantee their compatibility with the requirements expressed by clients. In practice, however, interactions may still fail because the Web Service's implementation may be faulty. In fact, the client has no guarantee on the *quality* of the implementation associated to any service description.

In this paper, we propose the idea of *high-quality service discovery* incorporating *automatic testing* for validating Web Services before allowing their registration. First, the discovery service automatically generates conformance test cases from the provided service description, then runs the test cases on the target Web Service, and only if the test is successfully passed, the service is registered.

In this way, clients bind with Web Services providing a compatible signature, a suitable behavior, and a high-quality implementation.

1 Introduction

Internet and the WWW provide a huge amount of services accessible from every connected machine. Most of these services are designed for human users, and only a strict subset can be easily discovered by search engines. This scenario is in contradiction to that of a machine-readable Web that exploits dynamic and automatic composition of services [1].

Web Services and the Service Oriented Architecture (SOA) represent a step toward the Internet as computational infrastructure [2, 3]. Web Services are software applications identified by URIs, whose interfaces and bindings are defined and discovered through XML documents. A Web Service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols [2]. The SOA provides the basic infrastructure for the discovery and dynamic binding of Web Services by defining the roles of *provider*, *requestor* and *discovery service*. A provider offering a service publishes its description at the discovery service. The requestor queries the discovery service in order to find a suitable service it can interact with to perform a certain task. The discovery service provides functions for storing, classifying, and browsing registered services [3].

M. Cerioli (Ed.): FASE 2005, LNCS 3442, pp. 34–48, 2005.

With this basic scenario, several problems remain open. First of all, service description and discovery is largely syntactic, reduced to the signatures of operations and simple classifications. Thus, there is no guarantee that the returned service operates in the way expected by the client. This problem can be overcome by augmenting the syntactic description by a behavioral specification of the service. Rather than logic or algebraic techniques we prefer graph transformation rules for this purpose because they blend well with UML, the standard software modeling language, thus keeping the additional effort manageable [5].

Graph transformation rules have been proposed for modeling both the behavior of the provided service and the client's requirements [4]. The provider uploads (an XML representation of) these models together with the syntactic service description, while the requestor uses a requirements model to specify its query. Then, service discovery includes the matching of these models at the discovery service: If the provided model satisfies the requirements, binding is allowed; otherwise another Web Service must be selected.

This new scenario increases the reliability of the binding between the requestor and the provider. However, another problem still exists. Can the client trust the implementation of the service description? The provider may register a suitable model, but provide a faulty implementation; for instance because of insufficient testing. Moreover, service providers could maliciously provide "models better than services". Since a faulty interaction can affect a distributed computation, clients dynamically binding to faulty Web Services can encounter serious problems, e.g., a complex business transaction may lead to expensive recovery procedures. Therefore, requestors exploiting dynamic and automatic discovery and binding require high-quality Web Services. To reach this goal we foresee the introduction of *High-Quality Service Discovery* agencies, i.e., discovery services with added functionality for behavioral matching and *automatic testing*.

High-quality service discovery automatically tests a Web Service with respect to a provided model consisting of GT rules that specify the individual operations of the service. The registration of services is allowed only if testing is passed, otherwise a report is generated and sent to the service provider. The developer of the Web Service can use the report to refine either the rules of the models or the implementation, depending on the origin of the problem.

Clients that use a high-quality service discovery agency have the guarantee that any discovered Web Service has passed the testing phase, therefore it can rely on both the interface compatibility and the implementation of the service.

2 Registration Scenario

We focus on the scenario taking place during the registration of a new service. The discovery and binding phases have been discussed in [4].

Let us assume the existence of a provider P and a discovery service U. P provides a Web Service ws that is described by means of both a syntactic interface description (e.g., a WSDL descriptor) and GT rules specifying the offered behavior. The registration phase includes the following steps (see Figure 1):

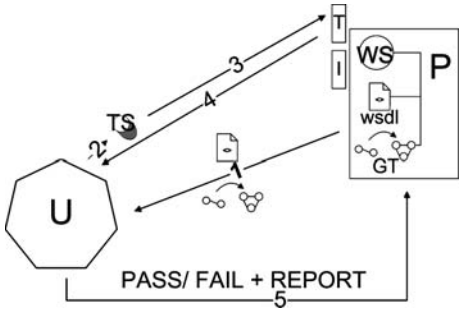

Fig. 1. The registration process for a new Web Service

1. The provider P uploads both the WSDL document and the GT rules to the discovery service U. See Section 3 for details on the specification of Web Services by GT rules.
2. The discovery service U automatically generates a set of test specifications from the GT rules. Tests cover validation of both single operations and sequences. See Section 4 for details.
3. Concrete test cases are generated and remotely executed using a *testing interface* T provided by the Web Service ws. This resembles the normal interface, but includes additional functions facilitating the execution of the test cases. See Section 5 for details.
4. Results of test cases are judged based on the returned results and the conceptual state of the service after completing the operation. The latter is read through the testing interface which provides access to an abstraction of the internal data state. See Section 5 for details.
5. If all test cases have been passed, the discovery service U registers the new Web Service with both the WSDL document and the GT rules for matching against the requirements of requestors. Finally, the service provider replaces the testing interface by the ordinary one. If test cases have failed, the discovery service U generates a report that is sent to the provider P.

3 Specification of Web Services by GT Rules

A Web Service provides a coherent set of operations based on a common data model, i.e., an XML schema. Together with the operation's signatures this makes up the WSDL description of the service. Extending this syntactic description, the behavioral specification of operations by means of GT rules is based on the data model of the service interface as well as a conceptual model of the internal state of the service. At the model level, the state is represented by an attributed graph, visualized as an UML object diagram [6].

A GT rule refines the signature of the service, specifying how parameters and internal data are used and modified. Each service is associated to a set of production rules representing the different computations that can take place

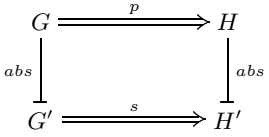

Fig. 2. Relation between conceptual and implementation state transformation

when the service is executed with different input values at different states. A production rule has a precondition and an effect. If the precondition is satisfied, the rule can be applied. The effect consists of objects and links that are deleted and added, and attribute values that are modified. A graph can satisfy the preconditions of multiple rules, in such case the choice is non-deterministic.

In this paper, we visualize graph transformations by the notation proposed in [7], see Figure 3 for an example. The semantics of the adopted graphical notation is the following one: nodes and edges fully contained inside the triangle are part of the pre-condition: they must be present and are not deleted by the application; edges and nodes partially or fully present at the left-hand side of the triangle are part of the pre-condition, too, but they are deleted when the rule is applied; nodes and edges partially or fully present on the right-hand side of the triangle are created by the rule; and finally, edges and nodes partially or fully present below the triangle form a negative application condition: they must not be present in the given graph. Parameters are distinguished from objects in the state of the server by a gray background. Finally, at the bottom of the graphical notation, guard conditions and assignment are provided. The guard is a Boolean expression over attributes: a production rule can take place only if the guard is evaluated to `true`. The assignment is responsible for updating values of attributes.

Graph transformations specify the behavior of a Web Service at the conceptual level. Therefore, the state of the service whose evolution is described does not coincide with the concrete data state, whose representation may involve Java objects and attributes or database tables, depending on the used implementation technology. Testing is performed on the *implementation* of the Web Service, hence it refers to the concrete data state. However, test cases are generated from the *specification*, thus they refer to the conceptual state. Figure 2 captures this situation where G' is the concrete state of the service, and G represents the abstract state that corresponds to G' [8]. G can be obtained from G' by an abstraction function *abs* that extracts a high-level representation of the actual state. A production rule p can be applied to the abstract state to obtain a new abstract state H. In the same way, the corresponding service s can be executed on the concrete state G' to obtain a new concrete state H'.

Testing is used to demonstrate the *conformance* of the specification with the implementation. We define conformance in terms of the following conditions.

Completeness: For each concrete state G' of service s, if $abs(G') = G$ satisfies the precondition of an associated rule p, then there exists a transformation $G' \stackrel{s}{\Longrightarrow} H'$ on the concrete state such that $G \stackrel{p}{\Longrightarrow} H = abs(H')$.

Soundness: A service s does not perform unspecified operations, with the exception of errors leaving the state unchanged, i.e., $G' \stackrel{s}{\Longrightarrow} H'$ implies $G' = H'$ or there exists an associated rule p with $abs(G') = G \stackrel{p}{\Longrightarrow} H = abs(H')$.

We initially validate completeness and soundness by deriving test cases for single rules. Completeness is validated by generating test cases inside the input domain of the rules, while soundness is checked through test cases outside the rules' domain.

Unfortunately, testing single rules is not enough to ascertain the conformance of the specification with the implementation. The execution of sequences of operations can reveal additional implementation faults related to details that are not present in the conceptual state. We account for such cases by defining different types of dependencies among rules and deriving test cases where these dependencies are exercised.

Throughout the paper we consider the example of a Web Service providing simplified banking functions. Figure 3 shows production rules corresponding to the creation of a new account, the withdrawal of money, the charging of an account with a set of payments, the deposit of money, and the closing of the account.

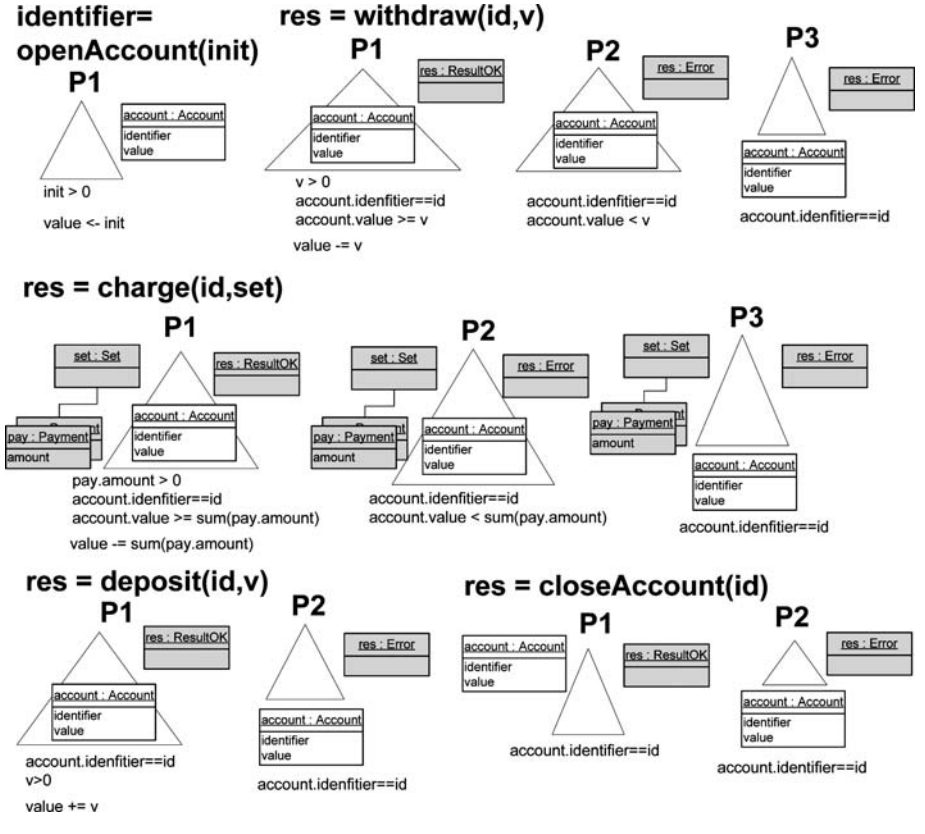

Fig. 3. Production rules for a Web Service managing bank accounts

4 Test Case Generation

We generate test cases for testing conformance of the implementation with individual rules by selecting "promising" inputs. Moreover, we generate test sequences stressing the interaction among rules.

Generation of Test Cases for Single Services. A WSDL description defines the input parameters and domains for the operations of a service. Possible inputs are further constrained by the preconditions of the GT rules. This suggests the derivation of test cases using a domain-based strategy, known as partition testing [9], which is an established technique successfully used in several contexts [10, 11]. The idea is to select test cases by dividing the *input domain* into (possibly overlapping) subsets and choosing one or more elements from each *domain* [10].

Partition testing has been used neither in the context of Web Services nor with respect to GT rules. Hence, our approach reuses standard ideas of testing in a new context. This requires a notion of input domain which combines the concrete input parameters of operations with the conceptual state of the Web Service before it is applied.

Formally, the *input domain* of a service s is the set of all parameter-state pairs $ID(s) = \{\langle par, st\rangle \mid par \text{ satisfies the WSDL description}\}$. The presence of both input and state in the domain is required because services are triggered by a combination of the two. Samples from $ID(s)$ can thus invoke all possible behaviors.

We identified the following fault-based guidelines [10] as strategy for designing the domains from each of which at least one test case should be chosen. The idea is that "small" partitions where several insidious faults can be present, and "larger" partitions where no assumptions about specific implementation threats can be performed are identified. Inside a partition all inputs have the same bias to be faulty. The discovery service administrator can exactly set the number of test cases that are sampled for each domain in a way to find the right balance between coverage and time consumed on testing. Experimental work aiming at finding the best number of test cases that should be selected from each domain is part of future work.

- The input domain of a rule p, given by the set of all parameter-state pairs $\langle par, st\rangle$ satisfying the pre-condition of the rule, defines a domain $D(p)$.
- Parameter-state pairs $\langle par, st\rangle$ simultaneously enabling two different rules p_1, p_2 form an input domain $D(p_1, p_2) = D(p_1) \cap D(p_2)$ because they require an internal decision (possibly non-deterministic) to decide which behavior must take place. This decision may be complex and its implementation incorrect.
- Input parameters and objects in the conceptual states carry attributes that are constrained by types and attribute conditions. Faults are likely when dealing with values at the boundary of their domains [9]. Thus, we define separate domains for inputs where at least one attribute has a boundary value. Note that the same attribute can have multiple boundary values.

- A production rule can also contain multi-objects which, upon application, are expanded to a set of objects whose cardinality depends on the current input. In order to validate this mechanism, we consider the inputs leading to expansions with zero, one, and multiple elements.
- Inputs outside the specification should create a response to notifies the client, but without modifying the state of the server. Failures to check for incorrect inputs can lead to follow-up faults which are very difficult to detect. We therefore consider a domain for values that do not belong to the input of any rule, but are correctly typed with regard to the WSDL description of s, i.e., $ID(s) \setminus \bigcup_{p_i \in s} D(p_i)$.

Note that these domains are not disjoint, but can overlap. It happens because different problems with different probabilities to be faulty can apply to the same concrete input elements.

A test case specification is composed of three parts: the *precondition*, the *test sequence*, and the *expected result*. The *precondition* specifies constraints that are expected to hold for the state of the server when the test case is executed. It is derived from the left-hand sides of the rules that must be tested. Conditions on parameters are not considered as part of the precondition, but contained in the *test sequence* which specifies conditions on input parameters together with the order of service invocations. Conditions on parameters are extracted from the left-hand sides of rules, too. The *expected result* is obtained by executing the rule for the generated input values. Note that a parameter-state pair can trigger multiple rules associated to the same service. In this case, we accept as correct any result produced by any of the applicable rules.

If we apply the defined criteria to the service `charge` in Figure 3, we obtain the following domains.

- The service is specified by three production rules; hence we have domains $D(p_1), D(p_2), D(p_3)$ generated from their left-hand sides.
- There is no non-determinism, i.e., the three domains are pair wise disjoint.
- Considering rule p_1, there are three attributes that can be defined: `payment`, `sum(pay.amount)`[1] and `account.value`. Test cases are generated by fixing a boundary value for at least one of them and randomly generating the other two values. The same applies to rules p_2 and p_3.
- Each multi-object produces domains for zero, one, and many instances. Thus, three test cases are generated for each rule: one with an empty set of payments, one with a set containing one payment, and one with a set containing n payments.
- Incorrect inputs are generated for each rule by choosing attribute values that violate the guard conditions. For instance, p_1 generates a test case with negative payments.

[1] values obtained by the application of common mathematical functions on multi-objects are handled as attributes, thus avoiding the use of problem solvers even for simple cases.

Currently we limit the generation of random values to linear constraints. Extensions to non-linear constraints have already been proposed in [11].

Generation of Tests for Sequences of Operations. The execution of an operation can alter parts of the service's state that are used by other operations. GT rules specify state modifications at a conceptual level. By analyzing these rules we can thus understand dependencies and conflicts between operations without looking into their actual implementation.

Data flow analysis is frequently used to generate test cases. The idea is to exercise paths in the code that include combinations of variable *definition* and *uses* [12]. This problem has been extensively investigated and several coverage criteria have been defined [13]. In particular, a widely used coverage criterion is "all def-use pairs" [14], which requires a test suite that executes all possible pairs of definition and uses for variables in the program under test.

Conceptually, each operation (rule) can add or remove nodes and edges to or from the conceptual state, and can change the values of attributes. The principles of data-flow testing can be reused to test the interaction among production rules if creation of nodes and edges is interpreted as "definition" and deletion as "use". We expect that sequences of operations that include the creation of a (conceptual) entity and its subsequent use are likely to expose (state-based) faults. The formalization of this intuition is given by the relations of conflicts and casual dependencies between rules.

Given two transformations $G \overset{p_1}{\Longrightarrow} H_1$ and $G \overset{p_2}{\Longrightarrow} H_2$ like in Figure 4, they are *parallel independent* if the application of one does not disable the other. That means, one transformation does not delete anything necessary for the application of the other, does not add anything forbidden by the negative application condition of the other, and does not modify attribute values used in the guard condition of the other rule.

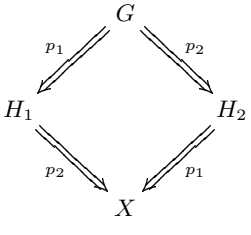

Fig. 4. Independence of transformation steps

Given a sequence of two transformations $G \overset{p_1}{\Longrightarrow} H_1 \overset{p_2}{\Longrightarrow} X$ like in Figure 4, they are *sequentially independent* if they can be exchanged. This means, as in the parallel case, that their occurrences do only overlap in such elements that are preserved by both transformations, and that application and guard conditions are not affected.

Thus, parallel and sequential independence are defined with respect to given graphs and occurrences, that is, using run-time concepts. To derive test cases

from specifications, however, a static definition of *potential* conflicts and dependencies is required. Therefore, the above notions have to be lifted to the level of rules. For two rules $p_1 : L_1 \to R_1$ and $p_2 : L_2 \to R_2$ we say that

- p_2 *may disable* p_1 if there exist transformation steps $G \overset{p_1}{\Longrightarrow} H_1$ and $G \overset{p_2}{\Longrightarrow} H_2$ like in Figure 4, such that $G \overset{p_1}{\Longrightarrow} H_1$ is *not* independent of $G \overset{p_2}{\Longrightarrow} H_2$,
- p_1 *may cause* p_2 if there exist transformation steps $G \overset{p_1}{\Longrightarrow} H_1 \overset{p_2}{\Longrightarrow} X$ like in Figure 4, such that $H_1 \overset{p_2}{\Longrightarrow} X$ is *not* independent of $G \overset{p_1}{\Longrightarrow} H_1$.

The *may-disable* relation captures possible conflicts among rules and is used to test sequences of actions that should lead to an error when the last request is issued, but that can erroneously produce some side effects on the actual state. The *may-cause* relation captures possible structural and attribute dependencies among rules. The may-disable relation is used to test completeness of rules, while the may-cause relation is used to test the consistency of rules. Formal definitions for dependencies among rules can be found in [15].

The criterion for generating test cases consists of covering the execution of all pairs of rules (p_s, p_t), where p_s *may disable/cause* p_t. However, it may be impossible to immediately execute p_t after p_s. Therefore, a sequence $p_s, p_1, \ldots p_n, p_t$ must be generated, where p_s *may disable/cause* p_t and the overall effect of the sequence $p_1, \ldots, p_n$ does not entirely "invalidates" the *may disable/cause* relation.

The relation between two rules is based on a set of nodes that have been deleted or added, or on a set of attributes that have been modified: the *entities of the relation*. If further rules are executed between the execution of the two related rules (p_s, p_t), the effect of the intermediate rules can overwrite the part of the state where p_s and p_t interact, i.e., the entities of the relation can be modified. In our case, if the effect of $p_1, \ldots, p_n$ modifies all entities of the relation, $p_1, \ldots, p_n$ entirely invalidates the relation. For the purpose of test case generation, it makes sense to cover the execution of only those pairs that effectively interact, i.e., the dependencies have not been overwritten and the conflicts have not been removed by other rules.

Also in this case, a test case specification is composed of three parts: the *precondition*, the *test sequence*, and the *expected result*. The *precondition* for a particular invocation sequence is obtained by anticipating the preconditions of all rules in the sequence [16]. The *test sequence* is given by the sequence of operations that must be executed and the conditions over the parameters that must be used for their invocation. Finally, the *expected result* is obtained by executing the rules over the concrete values. The concrete test cases are obtained by randomly generating concrete values that satisfy the constraints.

Potential conflicts and dependencies between rules are automatically computed by the AGG tool [17]. The additional relations deriving from attribute values can be obtained by simple data-flow analysis over constraints and assignments. If we apply this criterion to the running example and focus on rule p_1 of the service `withdraw`, we derive that the target service is dependent on rule p_1 of `openAccount`, on rule p_1 of `charge`, on rule p_1 of `deposit`, and on

rule p_1 of service `withdraw`. Therefore, test cases for opening the account and withdrawing money, charging payments and withdrawing money, depositing and withdrawing money, and withdrawing money twice are generated. Moreover, rule p_1 of the `withdrawing` service is in conflict with itself and with rule p_1 of the `closeAccount` service. Thus, a test case that closes the account before withdrawing money and a test case that withdraws money twice leading the current deposit to a negative value are generated for testing soundness (attributes and parameters for the latter test case are obtained by randomly generating values that satisfy all rule constraints and the negation of the condition `account.value`$\geq$`v` in p_1). In a similar way, we proceed for the other rules.

The Certified Level of Quality. Once a Web Service has passed the pre-registration testing phase, the client can rely on a high-quality implementation of the discovered services. In particular, the test cases generated for single services validate that all specified scenarios are implemented and that the implementation behaves according to the specification, at least for some inputs. Moreover, test cases validate that any internal decision taken by the Web Service satisfies the specification. Test cases for boundary values and multi-objects also certify that a range of values representing the normal operation of the service is defined and that collections are correctly managed in the cases of 0, 1 and multiple elements. Finally, test cases that violate guard conditions certify that guard conditions are implemented according to the specification and that a proper reaction mechanism is provided for incorrect inputs.

Test cases for sequences of operations validate that the state of the component evolves according to the specification at least for pairs of service invocation. Moreover, test cases check that the Web Service prevents reaching unsound states by multiple invocation of services.

Some interferences among operations, e.g., the definitions and uses of some state variables that cannot be deduced from the specification, cannot be automatically tested. However, clients of high-quality Web Services can rely on both the implementation of all specified behaviors and the existence of guard mechanisms for identifying incorrect inputs and effects.

5 Generation of Invocation Sequences

A test case has a precondition that consists of a set of constraints that must be satisfied by the actual state. Thus, the Web Service must be set to a state that satisfies the precondition of the test case. We assume that a Web Service facilitates this goal by providing a testing interface with three basic additional features:

- the possibility to setting the initial state of the service, possibly choosing from a set of alternatives representing different situations,
- a set of creator/destructor operations that enable the modification of the state of the server (if necessary),
- an implementation of the abstraction function.

A state that enables the execution of a given test case is reached by choosing an initial state from the set of states provided by the Web Service, and searching for a suitable sequence of requests that turns the chosen state into one satisfying the precondition of the test case. Dedicated creator/destructor operations are not required if the "normal" service interface already enables sufficiently free creation and deletion of objects on the server.

A similar problem arises in testing sequences of operations, where a transformation sequence must be generated enabling the execution of pairs of related rules. Both search problems are solved by building a search tree rooted at the selected initial state(s) and then incrementally considering the different rules. Each node is labeled with a path condition, i.e., the constraints that must be satisfied by the state variables to enable the sequence starting from the root and ending with the considered node. The path condition is derived by merging and simplifying both the guards and the assignments of single rules. When a state that satisfies the given properties is identified, the search stops, and the corresponding sequence is used in the test case. This is essentially the strategy employed by [18], the first work on model-based testing with GT rules we are aware of. In that paper, the search tree is in fact the concurrent unfolding of a grammar.

The search problem is realistic because not all rules can be applied at all steps and the overall number of rules is generally small for Web Services. For instance, a meaningful subset of the Amazon Web Service has been specified with 11 GT rules (see Section 6). Moreover, the service provider can further restrict the search space by uploading a specification of the Web Service interaction protocol [19]. Goal directed search strategies can heavily increase performance of the search by considering the structure of the current state, the modifications performed by the GT rules, and the structure of the final state in the search. However, a discussion over effective search strategies is out of the scope of this paper.

Tool support for execution, depth-first search, and bounded state space construction for GT rules is already in place. In particular, PROGRESS allows specifications based on rules with attributes and various application conditions and implements search by means of backtracking [20]. GROOVE [21] can generate bounded fragments of the transition system described by a set of rules in which paths to states with particular properties can be detected. Since the tool does not support attributed graphs, it would have to be complemented with a theorem prover to collect and combine the guard conditions and assignments for the identified sequence.

Concrete test cases are obtained by randomly generating attribute values that satisfy the path condition of the test sequence. Once concrete values have been generated, the expected result can be obtained by executing the rules over the concrete values. When a test case is executed, the final state of the Web Service is retrieved using the abstraction function of the testing interface. If the retrieved final state corresponds to the final state generated by the rules, the test case has been passed. The conformance relation (see Section 3) requires

that the result obtained by applying the rule on the conceptual state coincides with that obtained by the abstraction function on the concrete state reached after the execution of the test case.

The implementation of this function can be simplified by developing the system behind the Web Service with the Model-View-Controller (MVC) design pattern [22]. The design pattern isolates the state of the application (model) from the rest of the system, i.e., the control logic and the presentation layer. This strategy simplifies the access to the actual state and reduces the effort required to the developer for implementing the abstraction function.

Both the specification and the implementation of a service are furnished by the service provider who can, in principle, "cheat" by providing specifications and implementations that do not correspond with the final service, but that can easily pass the testing phase. However, specifications are used by clients for dynamically discovering services. Therefore, if the specification differs from the concrete service, the service cannot be successfully used by clients. In the same way, if the implementation is modified without repeating the testing phase, the registered specification will not match the provided service and interactions with clients will not be possible. Thus, the running version of the service, its specification and the tested implementation must be kept synchronized by the service provider.

6 Early Experience

We performed a number of small experiments in test case generation for real Web Services, initially considering two simple Web Services, the *Weather - Temperature* Web Service available at www.xmethods.com and the *Kayak Paddle Guide* available at `www.terawave.ca/webservices/paddle.html`. The former provides the current temperature in a given U.S. region. The latter computes the recommended length of a paddle, given the height of the person who will use it. Both Web Services provide one single simple operation. We derived the GT specification from the informal description available on Web. Then we generated test cases for single operations by sampling two values from each domain. The *Weather - Temperature* Web Service has been covered with 4 test cases and passed the test. The *Kayak Paddle Guide* has been covered with 6 test cases and failed the test. The technique discovered a fault for values that are expected to represent incorrect heights for a person, e.g., 600cm. In this case, the Web Service returns the longest paddle instead of signaling the incorrect input.

The two Web Services are very simple examples, but their complexity is representative for a large set of Web Services currently available on the Web. However, we decided to move to an example closer to the current state of the art for the Web Service technology. Thus, we considered the Amazon Web Service at `www.amazon.com/gp/aws/landing.html`, which provides a full set of functionalities for browsing and purchasing all items available in the Amazon Web Shop. In our experiment, we considered a comprehensive subset of the provided operations and we derived the GT specification from the online documentation

provided by Amazon. In case of failing test cases, we inspected the fault to understand whether the cause is either a fault or an error in the inferred specification. The operations selected for testing have included the search for DVDs based on the director's name and usual operations for cart management, i.e., item addition, item modification, item deletion, and clearing of the cart. We overcame the necessity of constructor methods for creating DVDs in the catalog by taking advantage of the knowledge of the content of the catalog. In a real scenario, the Web Service should have offered constructor methods for the creation of DVDs that could be then purchased.

The definition of the rules was straightforward, the five operations were specified by 11 GT rules. For testing of single rules, we sampled 2 values for each domain and we obtained 65 test cases. For testing of sequences, 14 dependent pairs and 3 conflicting pairs were identified. All pairs can be directly executed without requiring the generation of an intermediate sequence of operations. For test cases of both single rules and pairs, the initial sequence enabling the execution of the test case was always generated by the initial addition of a proper set of items in the cart. In a real scenario, the sequence would include also the invocation of constructor methods for the creation of DVDs.

Test case execution - which has been performed with a Java client - revealed an incompatibility between the rules and the Web Service. The incompatibility arose from a fault in the specification. In fact, in contrast with our rules, the operation for adding an item did not increase the quantity of items that were already in the cart, but overwrote the quantity instead, e.g., adding a DVD in the cart twice results on a single DVD in the cart. Thus, we modified the rule and generated the test cases again. This time the Web Service passed the test.

Our early experience with the testing of Web Services provided important insights: The technique is useful with respect to the complexity of current Web Services; the inspected Web Services do not require the generation of long initialization sequences, thus the search space that must be inspected is very limited; the number of generated test cases is suitable for a discovery service that automatically performs testing; both test cases for consistency, such as the repeated addition of items in the cart for the Amazon Web Service, and completeness, such as the incorrect input height for the Kayak Web Service, revealed to be useful.

7 Related Work and Conclusions

To our knowledge, there is only one approach to test case generation based on GT rules [18]. We advance research in this area by proposing two novel ideas: (1) the application of existing domain-based testing techniques to the case of graph transformations and (2) the execution of *automatic testing* for validating Web Services. The application of domain-based testing requires the management of the server state as part of the domain, when adapted to graph transformations. Moreover, data-flow testing needs to be reinterpreted in terms of dependencies

and conflicts among rules. Finally, the idea of using agencies which automatically test Web Services before registering them is new.

In contrast with graph-based testing, behavioral descriptions based on UML sequence diagrams and state charts can be used to generate test cases [23, 24]. However, the generated test cases fail to capture the concrete complexity of the exchanged parameters that often are restricted to few simple types, see for instance [24]. Moreover, due to the lack of precise semantics, UML diagrams cannot precisely describe the evolution of the state of the service. Graph transformations instead are suitable to unambiguously correlate the concrete states of the objects involved in an interaction with the behavior of a service. This kind of description enables the automatic generation of test cases that cover complex parameter passing and behaviors that are activated only for given internal states.

The design of GT rules has been demonstrated to be convenient when combined with a development methodology based on UML [5]. Therefore, the additional effort on behalf of the service developer for providing high-quality services is limited to the implementation of the abstraction function and to the eventual definition of additional constructor methods. The result is the publication of the Web Service in discovery services that aim at the dynamic composition of high-quality systems.

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American **284** (2001) 34–43
2. W3C Web Services Architecture Working Group: Web Services architecture requirements. W3C working draft, World Wide Web Consortium (2002)
3. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web Services architecture. W3C working group note, W3C (2004)
4. Hausmann, J.H., Heckel, R., Lohmann, M.: Model-based discovery of Web Services. In: Intl. Conference on Web Services. (2004)
5. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: a software engineering perspective. In: Intl. Conference on Graph Transformation. Volume 1 of LNCS., Springer (2002)
6. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley (1999)
7. Kaplan, S., Loyall, J., Goering, S.: Specifying concurrent languages and systems with δ-grammars. In Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: Proc. 4th Intl. Workshop on Graph Grammars and Their Application to Computer Science. Volume 532 of LNCS., Springer (1991) 475–489
8. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and Their Comparison. Volume 47 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1998)
9. White, L., Cohen, E.J.: A domain strategy for computer program testing. IEEE Transactions on Software Engineering **6** (1980) 247–257
10. Weyuker, E., Jeng, B.: Analyzing partition testing strategies. IEEE Transactions on Software Engineering **17** (1991) 703–711
11. Jeng, B., Weyuker, E.J.: A simplified domain-testing strategy. ACM Transactions on Software Engineering Methodology **3** (1994) 254–270

12. Rapps, S., Wejuker, E.: Data flow analysis techniques for program test data selection. In: 6th Intl. Conference on Software Engineering. (1982) 272–278
13. Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering **14** (1988) 1483–1498
14. Pande, H.D., Landi, W.A., Ryder, B.G.: Interprocedural def-use associations for C systems with single level pointers. IEEE Transactions on Software Engineering **20** (1994) 385–403
15. Hausmann, J., Heckel, R., Taentzer, G.: Detecting conflicting functional requirements in a use case driven approach: a static analysis technique based on graph transformation. In: Intl. Conference on Software Engineering. (2002) 105–155
16. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume Volume 1 - Foundations. World Scientific (1997)
17. Technical University Berlin: The attributed graph grammar system (AGG). http://tfs.cs.tu-berlin.de/agg/ (Visited in 2004)
18. Baldan, P., König, B., Stürmer, I.: Generating test cases for code generators by unfolding graph transformation systems. In: Proc. 2nd Intl. Conference on Graph Transformation, Rome, Italy (2004)
19. Bochmann, G.V., Petrenko, A.: Protocol testing: review of methods and relevance for software testing. In: Proceedings of the 1994 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis, ACM Press (1994) 109–124
20. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: language and environment. In: Handbook of graph grammars and computing by graph transformation: vol.2: applications, languages, and tools, World Scientific (1999) 487–550
21. Rensink, A.: 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., Springer (2004) 479–485
22. Singh, I., Stearns, B., Johnson, M., Enterprise Team: Designing Enterprise Applications with the J2EE Platform. 2nd edn. Addison-Wesley (2002)
23. Fraikin, F., Leonhardt, T.: SeDiTeC - testing based on sequence diagram. In: IEEE Intl. Conference on Automated Software Engineering. (2002)
24. Hartmann, J., Imoberdorf, C., Meisinger, M.: Uml-based integration testing. In: Intl. Symposium on Software Testing and Analysis, ACM Press (2000) 60–70