Model-based Testing of Service Infrastructure Components*

László Gönczy[1], Reiko Heckel[2] and Dániel Varró[1]

[1] Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Magyar tudósok krt. 2. H-1117, Budapest- Hungary
{gonczy,varro}@mit.bme.hu
[2] Department of Computer Science
University of Leicester
University Road, LE1 7RH, Leicester - United Kingdom
reiko@mcs.le.ac.uk

Abstract. We present a methodology for testing service infrastructure components described in a high-level (UML-like) language. The technique of graph transformation is used to precisely capture the dynamic aspect of the protocols which is the basis of state space generation.
Then we use model checking techniques to find adequate test sequences for a given requirement. To illustrate our approach, we present the case study of a fault tolerant service broker which implements a well-known dependability pattern at the level of services. Finally, a compact Petri Net representation is derived by workflow mining techniques to generate faithful test cases in a non-deterministic, distributed environment.
Note that our methodology is applicable at the architectural level rather than for testing individual service instances only.
Keywords: Model-based testing, Graph Transformation, Model Checking, Fault-Tolerant Services

1 Introduction

Beyond the usual characteristics of distributed systems, like asynchrony and communication over potentially lossy channels, service-oriented systems are characterised by a high degree of dynamic reconfiguration. Middleware protocols for such systems, therefore, have to specify not only the communication behaviour exhibited by the components involved, but also the potential creation and deletion of their connections, or indeed the components themselves.

This additional focus on structural changes requires an approach to protocol specification which allows to describe (i) the class of configurations the components involved can assume, (ii) their interaction, and (iii) changes to the configuration through actions of either the components under consideration or

* This work was partially supported by European Research Training Network *SegraVis* (on *Syntactic and Semantic Integration of Visual Modelling Techniques*) and the *SENSORIA* European FP6 project (IST-3-016004).

the environment. In general, such a model (however it is specified) will have an infinite state space, which makes full automatic verification more problematic. What is more, to verify the implementation of such protocols, the components implementing them will have to be tested against their specifications.

In this paper we address the testing of service infrastructure components against their specifications. By service infrastructure components we refer to services that are not part a specific application, but play a dedicated role in the service middleware. A typical example are services acting as proxies for implementing fault-tolerance mechanisms, e.g., by managing redundancy, forwarding requests to one of a number of available services chosen based on their performance. In our approach, service infrastructure reconfiguration protocols are specified by *dynamic metamodelling* [9], a combination of static metamodelling for describing structures as instances of class diagrams, with graph transformation rules for modelling changes to these structures. Using a UML-inspired notation for rules, specifications can be understood at an intuitive level while, at the same time, a formal semantics, theory, and tools are available to support verification.

We make use of that background through the state space generation tool Groove [25] for deriving (a bounded subset of) the transition system described by the metamodel and graph transformation rules. This transition system is employed to generate test sequences (i.e. desirable actions and their ordering) by model checking for a specific test requirement expressed as a reachability property. Unfortunately, the direct adaptation of derived test sequences as test cases in a SOA environment is problematic due to (i) internal (non-observable) steps in a test sequence, (ii) the distributed test environment where the interleaving of independent actions is possible, and (iii) not all steps of test execution are controllable.

For this purpose, we propose a technique to synthesize a compact Petri net representation for the possible interactions between the service under test and the test environment by using workflow mining techniques [1]. Concrete test cases can be defined by a sequence of controllable actions in this Petri net, while the test oracle accepts an observable action if the corresponding step can be executed in the Petri net.

The paper is organised as follows. Section 2 presents the structural metamodel for our case study and its extension by graph transformation rules. Section 3 discusses the specification of requirements for test cases and the generation of test sequences using model checking. Section 4 presents the architecture of the test environment and the derivation of test cases. Section 5 describes related work while Section 6 concludes the paper and discussed future research.

2 Modelling a Solution for Fault-Tolerant Service Infrastructure

As running example, we first introduce the dynamic metamodel of a service broker implementing a pattern for fault-tolerant services. The broker acts as a

proxy for service clients, maintaining a list of available services and forwarding clients' requests to individual service *variants*. The replies of these variants will be validated by a separate *checker* before being forwarded to the caller.

During the broker's lifetime services may be created or disappear, or may be temporarily unavailable due to loss of connection. Traditionally in the field of fault-tolerant systems, reconfiguration is captured by semi-formal design patterns. To provide a foundation for test generation, we propose a formal model which generalises these patterns while retaining the intuitive nature of semi-formal descriptions.

2.1 Structural Model

Our approach is based on metamodelling. A *metamodel* describes the ontology of a domain in the form of UML class diagrams. Domain concepts are denoted by *classes* with *attributes* defining their properties. *Inheritance* specifies subclass relations while *associations* define binary relations between classes. Multiplicities of association ends (at-most-one by default, or arbitrary denoted by an asterisk) restrict the connectivity along -instances of- an association. The formal interpretation of such a metamodel is a *type graph* [7], i.e., a graph whose nodes and edges represent types. Instances of the metamodel are formalised as corresponding instance graphs.

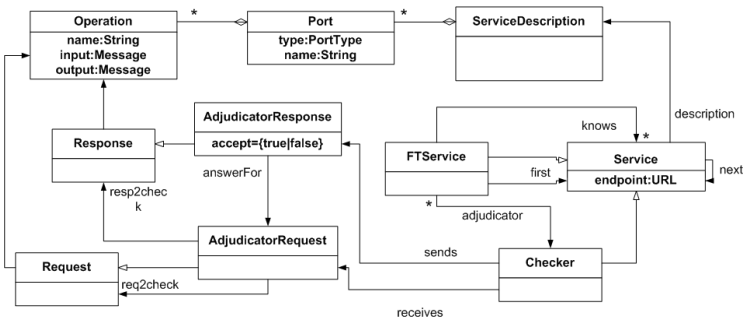

Fig. 1. Metamodel for Fault-Tolerant Services

The metamodel for fault-tolerant services is shown in Fig. 1. The FTService (also known as *broker* or *dispatcher*) is realised as a service, too. It is responsible for forwarding incoming requests to service components with the required functionality, designated by the knows association. The number of service components receiving the same request is determined by the fault-tolerance strategy applied. Responses to a particular request are sent back to the FTService, which sends them to a Checker service (also known as *adjudicator*).

The Checker service can be provided by a third-party component or by a local service running on the same machine. An AdjudicatorRequest sent to this

service is composed of the original request of the client and the response of the variant service. The Checker evaluates the incoming request and decides about its acceptance. As this step is highly application- or domain-dependent, we do not intend to give a general description here. Usually a simple comparison between the expected result, an approximate value, often determined offline, and the response of the variant service is sufficient. If there are multiple answers, another possibility is to compare the different values. The answer of the checker is wrapped in an AdjudicatorResponse and sent back to the FTService. If the answer is acceptable, it is forwarded to the client. In case of an erroneous answer, the next action is chosen according to the applied fault-tolerance algorithm and the number of available variants.

The metamodel presents an overview of the structure of the fault-tolerance pattern, but it does not specify the protocol executed by the component. This is described in the following section by graph transformation rules. In particular, we will model the *Recovery Block* pattern [24], where requests are sent to one variant at a time: the "best" one available according to some metrics gathered over time. This requires to maintain a list of components in the order of preference. More sophisticated strategies (such as load balancing between components by permuting the available components, etc.) are also possible. To mention other FT modeling solutions, OMG introduced an UML profile for QoS and FT [27], however, our solution uses a metamodel specially tailored to needs of SOA and patterns are modeled in more details (which is obviuosly needed for test generation).

2.2 Behavioural Rules

This section describes how the dynamic behaviour of service infrastructure component is specified in a formally verifiable way by graph transformation rules. The theory of graph transformation is described in detail e.g. in [7]. Here we only summarise the background.

A graph transformation rule consists of a *Left Hand Side (LHS)*, a *Right Hand Side (RHS)* and (optionally) a *Negative Application Condition (NAC)*, defined as instances of the type graph representing the metamodel. The *LHS* is a graph pattern consisting of the *mandatory* elements which prescribes a precondition for the application of the rule. The *RHS* is a graph pattern containing all elements which should be present after the application. Elements in $RHS \cap LHS$ are left unchanged by the execution of the transformation, elements in $LHS \setminus RHS$ are deleted while elements in $RHS \setminus LHS$ are newly created by the rule. The negative condition prevents the rule from being applied in a situation where undesirable elements are present in the graph. Formally, we follow the Single Pushout (SPO) Approach with negative application conditions.

A *graph grammar (GG)* consists of a start graph and a set of graph transformation rules. A *graph transition system* is a labelled transition system whose states are all the graphs reachable from the start graph by the application of rules, and whose transitions are given by rule applications labelled rule names.

We use the tool Groove [25] for creating graph transformation systems and generating their transition systems [25]. The "traditional" representation of rules separates LHS from RHS and shows the negative condition as part of the left-hand side (crossed out). The compact representation of Groove, on the other hand, uses a single graph only, with tags on the nodes and arcs to distinguish *new*ly created elements *del*eted elements and elements that must *not* be present, forming part of a negative application condition.

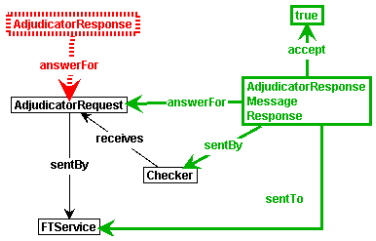

Fig. 2. Compact representation

Fig. 2 shows the compact representation of a sample transformation rule. This rule expresses the behaviour of the proxy when a decision has arrived from the adjudicator, reporting that the response of a particular service variant has passed the acceptance check. In this case, the proxy will send the response of the variant back to the client. In this simplified model, we abstract from changes to the original messages, however, in a real life system, technical changes can be performed on the reply (e.g., the sender of the message can be substituted).

Altogether we have identified four classes of transformation rules according to the nature of the behaviour they describe:

Reconfiguration rules determine the behaviour of the service under test. In our case, these are the rules which identify the actions of the FT proxy (forwarding requests to variant, register a new variant, etc.).

Environmental rules describe the dynamic behaviour of the other components in the infrastructure, still at a high level of abstraction (ignoring implementation-dependent steps). In our case, rules for service variant and checker components will belong to this set. The main difference between these and the reconfiguration rules is that these are possible "intervention points" where the concrete test case may affect the system since they relate to the behaviour of the tester component(s). However, if the *System Under Test* (SUT) changes (e.g., the checker component is the subject of testing), then the classification of rules may change.

Platform-dependent implement low-level operations, such as sending a SOAP message. They are needed to create a connection between the behaviour of different infrastructure components, e.g. to model that a message can be received by the target component only after the source has sent it. They also provide flexibility as other middleware-related aspects can easily be integrated. For instance, if one would like to extend the model by logging or reliable messaging features (e.g. creating acknowledgements for each messages), these extensions can be separated from the main component's high level logic. An example for such an extension was described in [13].

Test-related rules express actions which influence the tests but happen outside the system, including fault injection rules. In the case study, rules describing actions of the client are considered to be clearly test-related. In our fault model we consider *external service faults* representing an incorrect response which will

fail the acceptance check. The checker component is considered to be always correct, but the model is extendable to deal with a possibly unavailable/wrong checker.

The rules of the example are listed in Fig. 3 with their classification and a brief description. The rule classification has an impact on the test case generation, as rules of the above classes will affect test cases in different ways, as described in Sect. 3.

Rule name	Description
Reconfiguration rules	
callAdjudicator	The proxy calls the adjudicator when it receives an answer from a service variant.
callFirstVariant	The proxy forwards the request of the client to the first service variant in the list.
callNextVariant	The proxy forwards the request of the client to the next available service variant.
createFailureMessage	Since there are no further variants, and no acceptable response was given to the client request, the proxy indicates a failure to the client.
createProxyResponse	As the response of the variant was correct, the proxy forwards it to the client.
createNoServiceFailureMessage	As there are no available service variant for a particular request type, the proxy returns with a special failure.
registerVariant	The proxy registers a variant to the service list.
registerFirstVariant	The proxy registers the first variant to the service list.
Environmental rules	
createResponse	A service variant creates a response.
newSubscription	A service variant send a subscription request to the proxy.
makeNegativeDecision	The checker component rejects a variant response.
makePositiveDecision	The checker component accepts a variant response.
Platform-dependent rules	
sendMessage	A message is sent by the middleware.
receiveMessage	A message is received by the middleware.
Test-related rules	
newRequest	The client creates a request.
receiveAnswer	The client receives the answer of the proxy.

Fig. 3. Rules of the fault-tolerant proxy case study

3 Generation of Execution Sequences by Model Checking

This section describes the use of state space generation and model checking to derive executable test cases for the service broker. An architectural overview of our approach is presented in Fig. 4. Conceptually, we follow the principles of [4] to generate test cases as counterexamples for a given property using model checking. The properties are derived from test requirements specifying sequences of transformation steps to be used as test cases.

Given the counterexamples in form of rule sequences, we build a structure representing the possible combination of test sequences. This way nondeterminism introduced by distributed computing is included in our model, and the test oracle will be able to treat all possible branches (with the restriction that we will manage only test cases given as a result of the model checking).

For representing test cases we use the formalism of Petri Nets. We use *critical pair analysis* of the GT rules to find non-determinism in the system. Once we have the rule dependencies and test cases, the α-algorithm of [1] is performed to synthesize a complex Petri Net. Finally we reduce this Petri Net by filtering the rules which are neither observable nor controllable and therefore are not needed for the test oracle. Rules corresponding to middleware behavior and internal operations of the SUT are typically erased from the net.

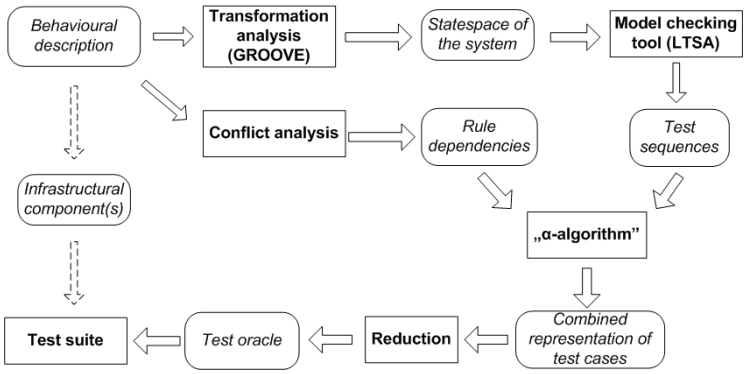

Fig. 4. High-level architectural view of the testing framework

3.1 Test Requirements

The test requirements we express can prescribe a desired action, following a specified sequence as a precondition. More formally in EBNF, our simple property language is defined as follows.

```
<requirement> ::= <sequence> => <rule>
<sequence> ::= <rule> | <rule>.<sequence>
```

Here, arrow (=>) means implication, dot (.) concatenation, while | and ::= are the usual EBNF (meta) operators. Note that although the conclusion of a requirement consists of the application of a single rule, a choice between multiple actions can be modelled by describing requirements for several test cases.

To illustrate our approach, we describe test case generation for a sample requirement: *If a variant response passed the acceptance test, the proxy will forward it to the client.* In terms of graph transformation, this translates to the following rule sequence, using the rule names of Fig. 3.

```
callAdjudicator.makePositiveDecision => createProxyResponse
```

Typical requirements correspond to forbidden behavior (such as, *"If there is a variant which has not been asked, no failure message can be sent to the client"*)

and required actions, e.g., *"If a checker accepts a result, the proxy must forward it to the client"*.

The rules used in requirements will typically belong to the classes of reconfiguration, environmental, or test-related rules, expressing high-level functionality observable at the application level. Platform-dependent steps are normally neglected, which results in reusable requirement patterns. For example, message-based communication could be replaced by remote procedure calls without affecting these requirements.

3.2 State Space Generation and Model Checking

Given the transformation rules described in Sect. 2.2 and an initial configuration (e.g., a proxy, a number of service variants not registered with the proxy and a client) one can generate the entire state space of the graph transition system using the GROOVE tool [25]. Groove performs a bounded state space generation by applying graph transformation rules in all possible ways to the start graph, up to a given search depth. Unfortunately, the implementation of model checking of temporal logic formulae is still in an early stage for GROOVE, therefore we use a separate model checking tool.

We transfer both the graph transition system generated by Groove and the requirements into the Labelled Transition System Analyzer (LTSA) tool [18]. For model checking, LTSA composes an automaton from the LTS of the original system and the property automaton of the requirement. The analysis will find a violation trace if the property automaton reaches an error state.

Thus, a requirement has to be translated into a property automaton with the obvious modification that the "required action" is considered as an error state. Moreover, a separate automaton is derived from the state space of the graph transformation system generated by Groove. In the process, all information about the internal structure of states and transformation steps is lost. Therefore, we have to restrict our execution path retrieved by the LTSA analysis to handle *one request at a time*. However, this does not prevent to apply our test generation technique of Sec. 4 to be used for concurrent messages.

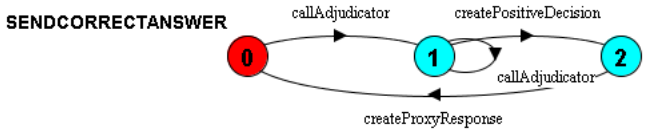

Fig. 5. Requirement expressing the behavior of the proxy

Given such an input, the LTSA tool is able to find low-level rule sequences in the state space of the system which "violate" the property automaton derived from the original requirement by negating the required action. These sequences will serve as the basis for deriving the actual test cases.

As an example, we examine the rule set (of Fig. 3) for a sample configuration consisting of one client, one proxy, one checker and three service variants. The formulation of our sample requirement as a property automaton is given in Fig. 5. The sequence which is found as a "counterexample" for this property is shown in the following example. (We modified the output format of LTSA to make the sequence more readable.)

```
newSubscription => sendMessage => receiveMessage =>
registerFirstVariant => newRequest => sendMessage =>
receiveMessage => callFirstVariant => sendMessage =>
receiveMessage => createResponse => sendMessage =>
receiveMessage => callAdjudicator => sendMessage =>
receiveMessage => createPositiveDecision => sendMessage
=> receiveMessage => createProxyResponse
```

This corresponds to a sequence describing the desired functionality of the system, and it will serve as the basis for a test cases for this requirement. This sequence is one of the shortest possible rule sequences since the execution of some of the steps can be carried out in parallel (e.g. the subscription of a variant and the creation of a client request). That means, although the test case could contain concurrent steps, the model checker will return only a sequence.

Note that although we focus on the generation of test sequences, the same technique can also be used to verify the dynamic behaviour of the model as described in [13]. In this case the output of the model checker represents a decision about whether the system meets a particular requirement. If not, a sequence of events is provided that violates the requirement.

In our case, if the analysis results in a positive decision about the negated requirement, the original requirement is not satisfied by the rules of the model. This provides, almost as a side effect, with a verification of the model (e.g. as described in [13]) possibly leading to a re-design of the rules according to the results of the test case generation.

4 Derivation of Test Cases

At this point, execution sequences derived by the LTSA model checker are available. However, their direct adaptation for test cases in a SOA environment is not at all straightforward as (i) certain steps in the execution sequence are internal to the proxy thus they are not observable, (ii) the tests need to be executed in a distributed environment where the interleaving of independent actions is possible, and (iii) we cannot deterministically control all steps of test execution (non-deterministic testing [22]).

For the first problem, many existing approaches [22, 5] typically use an abstract representation of the test case which only includes controllable and observable actions. For the second problem, one may ask the model checker to derive all possible execution paths which satisfies a given requirement [14]. However, this results in a huge set of test sequences, i.e. a different sequence for each

interleaving, which can be infeasible in practice. For the third problem, a test oracle needs to be derived which identifies if one of the correct execution paths were executed by the FT proxy (i.e. service under test, SUT) for a given input.

After discussing the architecture of the test environment, we propose a technique to synthesize a compact Petri net representation for the possible interaction between the FT proxy and the test environment by using workflow mining techniques [1]. Concrete test cases can be defined by a sequence of controllable actions in this Petri net, while the test oracle accepts an observable action if the corresponding step can be executed in the Petri net.

4.1 A distributed test architecture

The component architecture of the test framework is shown in Fig. 6 with the interfaces of messages arriving to each component and potential interactions between the components.

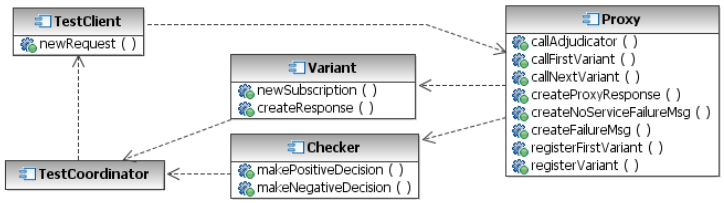

Fig. 6. Architecture of test components

Since in a service-oriented architecture the server will be unaware of the client implementation and communicate only via messages, thus the TestClient does not have to implement any method of the FT proxy (SUT).

Methods of Variant and Checker will be forwarded to the TestCoordinator, thus the test coordinator implements the interfaces of all the other components in the test environment. Operations on these interfaces will be interaction points or controllable and observable actions (see later in Sec. 4.2). This way, no modifications are made to the SUT (the FT proxy) for testing purposes as services implementing other infrastructural elements can replace the interfaces of the coordinator.

The execution of a test case requires to make certain decisions available as test configuration parameters, which are application-dependent during normal operation. For instance, decisions like the result of an acceptance test or the availability of a variant will be influenced by pre-defined test parameters for each decision in the test. For instance, if multiple variants will be asked by the proxy, each of them will ask the coordinator whether to answer the request.

In the paper, we assume that tests components are stand-alone services in a distributed SOA environment, but no further implementation details are provided to better concentrate on presenting the test generation approach itself.

4.2 Creating the test suite

In the field of model based testing, generating executable test sets from abstract test sequences is a well-known problem. Actions in a sequence can be *controllable*, *observable* or *hidden*. These categories respectively correspond to decision points to set up a certain test case (controllable), automatically executed actions within the test framework (observable) and actions inside the SUT (hidden).

In our case, a rule sequence produced by the model checker may contain many rules to be executed automatically, without any test-case specific intervention. *Reconfiguration* rules (like registerVariant) are obviously part of the SUT, i.e., the FT proxy. *Platform-dependent rules* can be observable or hidden, depending on whether they are executed in the SUT or in the tester. However, as the middleware rules are not directly affected during testing, we consider them hidden. Hence, only controllable *test* and *environmental* rules will included in a test case.

Our goal is to build a combined representation of multiple test cases and test oracles in the form of Petri nets. Petri Nets (PN) are a special class of bipartite graphs used to formally model and analyze concurrent systems with a wide range of available tool support. The reader is referred, for instance, to [23] for the theory and application of PN.

For this purpose, we combine *critical pair analysis* of graph transformation rules with the α-algorithm used for *workflow mining* in [1]. The former technique aims at statically detecting conflicts and causal dependency between graph transformation rules, while the latter method is used for building instances of a special class of Petri Nets (called Workflow Nets) from workflow logs.

Step 1: Partial ordering of an execution path. First, we build a Petri Net of each individual test case which makes concurrent behaviour explicit by de-sequencing (totally ordered) actions in the execution path derived by the model checker into a partially ordered transitions in the PN.

In order to detect concurrent actions, we use basic concepts of graph transformation theory. Two rules are in *conflict* with each other, if the execution of a rule does disables the execution of the other rule (otherwise, they are *parallel independent*). A rule is causally dependent on another rule, if the first rule enables the other (e.g. by creating new matchings for it), otherwise, they are called *sequentially independent*.

A partial ordering between actions can be derived by performing *critical pair analysis*[19] of our rules, which is a well-known static analysis technique in the field of graph transformation to detect potential conflicts and causalities of graph transformation rules. Critical pair analysis is able to show minimal conflicting (or causal dependent) situations between two graph transformation rules. These suspicious situations can be inspected by the user to decide if they arise in a certain application or not.

In our case, a critical pair analysis will detect some trivial conflicts due to the semantics of Groove which always explicitly requires a NAC to prevent the system from getting to an infinite loop. After eliminating such trivial dependencies, the result of the analysis for an execution sequence will reveal those parts of the sequence which can be executed concurrently. These will correspond to

the behavior of distributed components and the middleware, the order of which cannot be determined. Fig. 7 shows a partially ordered version of rule sequence in Sect. 3.2 as a PN where controllable and observable actions are highlighted.

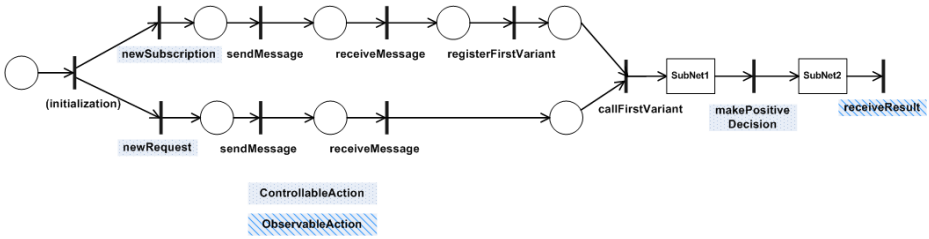

Fig. 7. Petri Net representing a test case

Step 2: Constructing workflow nets from partially ordered paths. Using the method of [1] the system model is reconstructed by workflow mining techniques from individual *observations (cases)*. Our problem is very similar: we have to create an abstract model of observable and controllable actions, which explicitly contains concurrent behavior and potential non-determinism.

This workflow mining technique groups relations between pairs of actions into the following categories: *potential parallelism*, *direct causality*, *precedence* and *concurrency*. These relations can be derived from the critical pair analysis in the previous step. We also have the restriction that the net will be of class Free Choice Net. The only difference is that we do not expect the final model to be a valid WF-Net. The result of the algorithm is shown in Fig. 8.

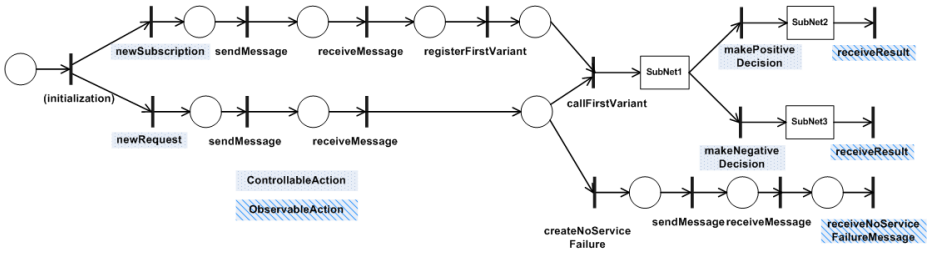

Fig. 8. The Petri Net created after the combination of test sequences

Step 3: Reduction of an observable Petri net. The net is then reduced using standard PN reduction rules described for instance in [23]. The main principle of the reduction is that we erase all sequences and parallel constructs which do not contain any controllable or observable actions, since these correspond to internal behavior of SUT (this case, the proxy) and therefore will not affect the tester. The result of the reduction is shown in Fig. 9. The resulting abstract PN

can be used as a combined test suite and test oracle for a set of given requirements.

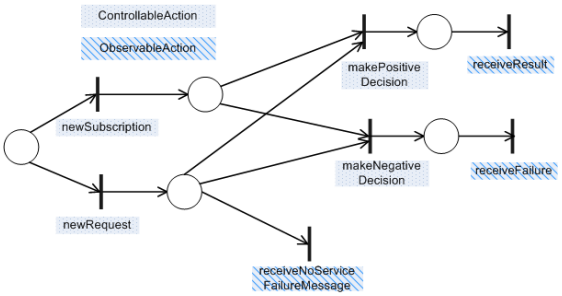

Fig. 9. The Abstract Petri Net created after reduction

4.3 Discussion

Our approach relies on a combination of various formal techniques. Now we discuss the role of each individual technique in the overall approach.

Why to combine two model checkers? At this point, the official Groove release does not yet support model checking facilities, only state space generation. However, this feature of the tool is strong enough because dynamically changing models are supported. Therefore, we project the state space generated by Groove into the LTSA tool to derive actual execution sequences for a given test criterion (this is practically renaming an LTS structure).

Why to use a Petri Net representation for a test case? The final Petri net representation offers two advantages: (i) a compact representation of test case with explicit concurrency and without interleavings (which is not the case of the original GTS state space) (ii) mining techniques are available to derive the PN.

Direct bridging of graph transformation and Petri nets. There are existing approaches to generate a Petri net representation of a graph transformation system on various levels of abstraction [2, 28]. In the future, we plan to investigate more on their applicability in a testing environment. However, we believe that the Petri net representation of a test case is more simple compared to them.

5 Related Work

The modelling technique in our paper is conceptually derived from [3] where SOA-specific reconfigurations where first defined by a combination of graph transformation and metamodelling techniques.

Graph transformation is used as a specification technique for dynamic architectural reconfigurations in [10] using the algebraic framework CommUnity.

Executable visual contracts derived from graph transformation rules are facilitated in [20] where JML code is generated for the run-time checking of (manual) service implementations. In [6] the same contracts are used for specification matching. Graph transformation rules guided the model-based discovery of web services in [15].

The specification and analysis of fault behaviors have been carried out in [8] using graph grammars. While this approach is not directly related to SOA, it uses similar techniques for modeling the behaviour of the system, and also applies model checking techniques for verifying the behavioural specification. However, the behaviour of SOA components typically induces an infinite state space, such a full verification is problematic.

In [16], one of the authors applies graph transformation-based modelling for conformance testing. The novel contribution of the current paper is that (i) we use model checking to generate test sequences, which leads to a higher level of automation (ii) our models focus on changes at the architectural level rather than on the data state transformation with a single service.

The work presented in [17] aims at test generation for processes described in OWL-S. Our work is different as our test cases are derived from high-level formal specification of the dynamic behaviour, rather than being abstracted from its implementation. The same applies to [12] where the SPIN model checker is used to generate test cases for BPEL workflows. Authors of [21] also use SPIN to create test sequences to meet coverage criteria. Categories of actions and formalisms for describing test cases are defined among others in [22] and [5]. However, synthesis of test cases is still an open issue. We already discussed the work described in [22] and [5].

LTSA [18] has already been applied successfully in a SOA context for the formal analysis of business processes given in the form of BPEL specifications in [11]. However, the direct adaptation of this approach is problematic, since the inherent dynamism in the reconfiguration behaviour of the service infrastructure is difficult to be captured in BPEL.

A de-facto industrial standard in the telecommunications domain for a highly available service middleware is the Application Interface Specification (AIS) of SA Forum [26]. Our future work includes the application of our approach to testing of components of the AIS infrastructure.

6 Conclusions and Future Work

We proposed a model-based approach for generating test cases for service infrastructure components exemplified by testing a fault-tolerant proxy. The reconfiguration behaviour of the service infrastructure was captured by a combination of static metamodels and graph transformation rules. The (bounded) state space of the service infrastructure was derived by the Groove tool [25], and post-processed by the LTSA model checker to derive an execution sequence for a given requirement. In order to generated faithful test cases to be executed in a distributed service environment, a compact Petri net representation was derived by workflow

mining techniques. At the final step, this Petri net was reduced by abstracting from internal actions.

The scalability of our method is at the moment mainly limited by the state-space generation feature of Groove which is the range of 100 thousand states; however, these states represent a dynamically changing structure (vs. a BDD with predefined state variables). The quality of our generated test cases strongly corresponds to the requirements which are under investigation (as usual in requirement-based testing).

In the paper, we limited our tests to configurations with one proxy and one service type only. That means, all variants implement the same service. This, however, is only a limitation for illustration purposes, since the rules can easily be extended to model a proxy maintaining multiple variant lists, one for each type of service. Multiple requests can also be tested by starting a corresponding Petri net for observing each request. On the other hand, in case of more sophisticated requirements information about the structure of the graphs needs to be expressed. This is currently is not supported by the state space generation and model-checking tools we use. Future developments in model checking for graph transformation systems are likely to ameliorate this problem. Our long-term purpose is to develop a methodology for testing automatically generated components, modelled by a visual notation that enables (semi-) automatic code generation.

References

1. van der Aalst, W.,Weijters, T. and Maruster, L.: Workflow mining: discovering process models from event logs. In IEEE Trans. on Knowledge and Data Engineering, Vol.16. No.9., 2004.
2. Baldan, P., B. Knig and I. Strmer: Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems. In Proc. of ICGT 2004, pp. 194-209, 2004.
3. Baresi, L., R. Heckel, S. Thöne and D. Varró: Style-Based Modeling and Refinement of Service-Oriented Architectures. Journal of Software and Systems Modelling, Vol. 5(2), pp. 187–207, June 2006.
4. D. Beyer, A. J. Chlipala, and R. Majumadr: Generating Tests from Counterexamples. In Proc. 26th Intern. Conf. on Software Engineering, 2004, pp. 326-335.
5. Campbell, C., W.Grieskamp and L. Nachmanson:Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, 2005.
6. A. Cherchago and R. Heckel:Specification Matching of Web Services Using Conditional Graph Transformation Rules. In Proc. of Intern. Conference on Graph Transformations, 2004, LNCS Vol.**3256**, Springer, pp. 304-318.
7. A. Corradini, Montanari, H., Rossi, F.:Graph Processes. Special Issue of Fundamenta Informaticae, Vol. 26(3-4), pages 241–266. 1996.
8. Dotti,F.L., L.Ribeiro, and O.M. dos Santos: Specification and analysis of fault behaviours using graph grammars. In AGTIVE 2003, Charlottesville, VA, USA, Vol. 3062 of LNCS, pp. 120–133. Springer, 2003.

9. Engels, G., J. Hausmann, R. Heckel and S. Sauer: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Proc. UML 2000, York, UK, LNCS 1939 (2000), pp. 323-337.
10. M. Wermelinger and J. L. Fiadeiro:A graph transformation approach to software architecture reconfiguration. Science of Comp. Progr., 44(2):133–155, 2002.
11. Foster,H., S.Uchitel, J.Magee, and J.Kramer: Model-based verification of web service compositions. In 18th IEEE Intern. Conf. on Automated Software Engineering (ASE 2003), Montreal, Canada, pp. 152–163. IEEE, 2003.
12. Garca-Fanjul, J., J. Tuya, C. de la Riva: Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In Proc. Intern. Workshop on Web Service Modeling and Testing (WS-MATE 2006). pp. 83-85..
13. L. Gönczy, M. Kovács and D. Varró: Modeling and verification of reliable messaging by graph transformation systems. In Proc. of the Workshop on Graph Transformation for Verification and Concurrency (GT-VC2006). Elsevier, 2006.
14. Hamon,G., L. de Moura and J. Rushby:Generating Efficient Test Sets with a Model Checker. in Proc. of SEFM 04, Beijing, China, September 2004
15. Hausmann, J.H., Heckel, R., Lohmann, M.:Model-based Discovery of Web Services. In IEEE Intern. Conf. on Web Services (ICWS), June 6-9, 2004, USA,
16. Heckel, R. and L. Mariani: Automated Conformance Testing of Web Services. In Proc. 8th Intern. Conf. on Fundamental Approaches to Sofware Engineering (FASE 2005), vol. 3442 of LNCS, Springer, pp. 34-48.
17. Huang., H. ,W-T Tsai, R. Paul and Y. Chen: Automated Model Checking and Testing for Composite Web Services. In Proc. of 8th IEEE Intern. Symp. on Object-Oriented Real-Time System Computing (ISORC'05), 2005, pp 300-307.
18. Labelled Transition System Analyser (Version 2.2) `http://www-dse.doc.ic.ac.uk/concurrency/ltsa-v2/index.html`
19. Lambers,L., H. Ehrig and F. Orejas: Conflict Detection for Graph Transformation with Negative Application Conditions. In Proc. of ICGT2006, pp. 61-76, 2006.
20. Lohmann, M., S. Sauer, G. Engels:Executable Visual Contracts. In Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 05), pp. 63-70, 2005.
21. Micskei, Z. and I. Majzik: Model-based Automatic Test Generation for Event-Driven Embedded Systems using Model Checkers. In Proc. ofInt'l Conf. on Dependability of Computer Systems (DEPCOS-RELCOMEX'06),pp. 191-198,2006.
22. Muccini, H.: Software Architecture for Testing, Coordination and Views Model Checking. PhD Thesis, 2002.
23. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proc. of IEEE, Vol.77. No.4. 1989.
24. Randell, B. and J. Xu: The Evolution of the Recovery Block Concept, in Software Fault Tolerance (M. Lyu, Ed.), Trends in Software, pp.1-22, J. Wiley, 1994.
25. Rensink, A.: The GROOVE simulator: A tool for state space generation. In Proc. of Application of Graph Transformations with Industrial Relevance (AGTIVE'03), LNCS Vol. 3062, Springer, pp. 479-485., 2003.
26. SA Forum: Application Interface Specification. `http://www.saforum.org`.
27. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and mechanisms.
28. Varró, D., Sz. Varr-Gyapay, H. Ehrig, U. Prange and G. Taentzer: Termination Analysis of Model Transformations by Petri Nets. In Proc. of ICGT2006, pp. 260-274, 2006.