CONCEPTS FOR MODEL–BASED REQUIREMENTS TESTING OF SERVICE ORIENTED SYSTEMS

Michael Felderer and Ruth Breu
and Joanna Chimiak–Opoka
Institute of Computer Science
University of Innsbruck
Innsbruck, Austria
michael.felderer@uibk.ac.at

Michael Breu
arctis software technology GmbH
Inzing, Austria
michael.breu@arctis.at

Felix Schupp
softmethod GmbH
Munich, Germany
felix.schupp@softmethod.de

ABSTRACT
In this paper we present the core concepts of Telling TestStories, a model–driven framework for test–driven requirements testing of service oriented systems. Telling TestStories provides a new way of eliciting and validating requirements through intertwined specification of requirements and executable test stories. We define a Domain Specific Language (DSL) to formalize the system requirements and the test model. The DSL allows test cases to be specified based on the concepts of the requirements specification (actors, objects, services) and test cases to be separated from test data. To ensure the quality of the designed artifacts we introduce consistency and coverage checks expressed in OCL. We provide a prototypic implementation of the concepts and started an industrial validation of its usability.

KEY WORDS
Model–Based Testing, Quality Assurance, Domain Specific Languages, Service Oriented Systems, Requirements Engineering.

1 Introduction

In the recent years there has been an important shift of paradigm from client–server based applications to service oriented systems. *Service oriented systems* (SOS) are networks of peers offering executable services to each other, i.e. following the architectural style of Service Oriented Architectures (SOA). SOS are an attempt to better link business with technology and are applied in intra–organizational applications to leverage flexible IT landscapes and in inter–organizational scenarios in which peers conduct workflows across different domains and platforms.

1.1 Challenges

With the increasing number of service–oriented system implementations, new challenges concerning the testing of SOS are emerging [1] on all levels: on the unit level, testers only have access to interfaces but lack access to code of services, on the integration level, distributed services based on different technologies and platforms have to be integrated at runtime, and on system level the complete business process has to be considered in a complex environment, the traceability of executable services and business services of the requirements specification has to be guaranteed, the dynamic evolution of requirements has to be handled efficiently and non–functional properties such as security or quality of services have to be tested. Model–based testing seem to be a systematic and promising approach to testing complex SOS but it is still challenging to find the right level of abstraction and appropriate modeling techniques.

1.2 Contribution

In this paper we present a framework and a testing methodology called *Telling TestStories* (TTS) which contributes to the challenges mentioned in Section 1.1. TTS has been designed for *model–based testing* of SOS. It *abstracts from concrete service implementations* by definition of an abstract specification and test model in a domain specific language (Section 3) and from *technologies and platforms* by introduction of a set of adapters as a layer between the model level and implementation level (Section 4). TTS provides a *test–driven approach to requirements specification*, i.e. test models are specified before or at least intertwined with the requirements specification which supports the testing of dynamically evolving services. Therefore we provide *consistency and coverage checks of the test model against the system model* (Section 3.3) which also *support the manual adaptation of the test model to dynamically changing requirements*. In this paper we concentrate on the aspect of functional *requirements testing* or *system testing*. We emphasize the definition of test stories which are a kind of annotated test scenarios on the requirements level that can be executed automatically against the system implementation and support *testmodel–driven system testing*.

1.3 Related Work

There has been a lot of research [2, 3] and tool implementations [4] on model–based testing, i.e. the derivation of tests from a (test) model of the system under test. Our approach focuses on the definition of test models based on the system model but not on the automatic derivation of test cases

as many other approaches do.

There are only a few approaches applying model–based testing to SOS. In [5] model–based testing of SOA applications with the UML Testing Profile has been investigated and in [6] TTCN-3 has been applied for functional and load testing of web services. Our approach can be mapped to the UML Testing Profile and therefore also to TTCN-3. But our metamodel focuses on the core concepts of SOS ideally supporting the development of a test methodology resp. framework for SOS. This allows the investigation of model properties such as consistency and coverage especially for SOS. Coverage criteria for models have systematically been discussed in [4]. But this investigation does not consider special coverage criteria for SOS.

FIT/Fitnesse [7] is the most prominent framework which supports system test–driven development of applications allowing the tabular specification, observation and execution of test cases by system analysts. Our framework is due to the tabular specification of test data based on the ideas of FIT/Fitnesse but integrates it with model–based testing techniques.

1.4 Structure

In the next section we give an overview of our concepts, and in Section 3 we define a metamodel for our system and test model. We then describe a system architecture for our framework (Section 4) and finally in Section 5 we draw conclusions and discuss future work.

2 Overview

In this section we provide a high level overview of the TTS framework and the TTS methodology for system testing of SOS.

2.1 Telling TestStories Framework

Figure 1 shows the basic structure of the TTS artifacts. The artifacts are categorized along two orthogonal classifications: *Model* and *Implementation* on the one side and *System* and *Test* on the other side.

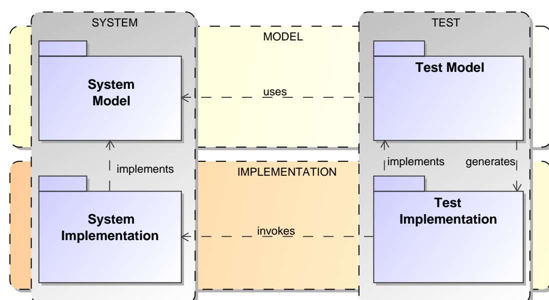

Figure 1. Basic artifacts

The system model describes the system requirements at business level. An important assumption in our framework is that system model and system implementation are traceable. In particular, each business service in the system model can be traced to an executable service in the system implementation. The system model is specified by the domain experts and system analysts based on the notions of *actors*, *services* and *classes*.

The system implementation also called the system under test (SUT) is the set of *executable services* under test.

The test model contains the test case specifications developed in an incremental process. This process starts with the specification of *test stories*. Test stories are structured sequences of service calls at business level exemplifying the interaction of actors with the system. Test stories may be generic in the sense that they do not contain concrete objects but variables which refer to test values provided in tables. For testing purposes, test stories are enhanced by *assertions*, i.e. conditions to be checked within the execution of the test story. For completely specifying tests, each test story has a corresponding initial state and test table. Test stories can be seen as high level descriptions of the test requirements.

The test implementation is generated by a compiler which transforms test story files into source code files of the execution language. These files are then executed by the test controller. Service adapters make the abstract service calls of the test stories executable by either providing the glue between the service calls and the executable services or by a link to surrogate services like manual input, mock services or external test services.

2.2 Telling TestStories Methodology

Figure 2 shows the workflow of the TTS methodology. Actions which are executed automatically by the TTS framework are colored yellow and actions which have to be done manually (with tool support) are colored green. Note that the *Adapter Implementation* action is a both yellow and green because it can be done manually or automatically if a stub generator for the underlying service technology is provided. The four blue boxes group elements and relate them to the four corresponding artifacts, i.e. system model, test model, system implementation and test implementation of Figure 1.

The TTS testing procedure starts with the iterative *definition of a system model and a test model* which is the basis for *test data definition*. *Consistency and coverage checking* may involve iterative adaptation of the system and the test model before test code generation. For every service definition in the system model we need an *adapter implementation* to call the systems corresponding executable service in the system under test. After *generating generic*

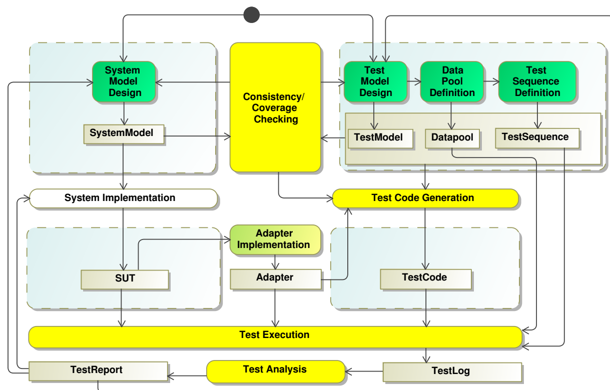

Figure 2. Telling TestStories methodology

test code from the stories, they are executed with test data and then its test log is *analyzed* which may involve changes in the system model, the test model, the test implementation or the system implementation.

3 Telling TestStories Metamodel

In the following subsections we define metamodels for the system model and the test model of our framework, sketch related consistency and coverage issues and demonstrate the concepts on a simple ticket reservation system.

3.1 System Model

Figure 3 depicts the core system metamodel. This metamodel formalizes those elements of the requirements specification that are referred to in the test model. Note that elements colored green in Figure 3 are also depicted in the metamodels of Figure 4 resp. Figure 5 and vice versa, e.g. the concepts `Actor` and `Service` of Figure 3 are also referred to in Figure 5.

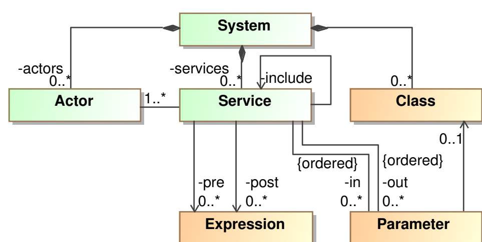

Figure 3. Abstract syntax of the system model

A `Service` describes the basic functionality a system component provides to the outside. Notice that the notion of service is conceived at the business level and is independent of the underlying technology, e.g. web services. Services may be hierarchically structured (`include`). Services have input parameters (`in`) and output parameters (`out`). All parameters are either basic data types or of a `Class` type. Additionally services may have a precondition (`pre`) and a postcondition (`post`) expressed in OCL. Finally, an `Actor` represents a role that interacts with the system.

We have defined a concrete textual syntax for representing the system model which has been used in Listing 1 to define the system model of our ticket reservation system[1].

```
system {
 actordef Customer
 ...
 servicedef ReserveTickets {
  actors (Customer)
  in (cust:Customer event:Event number:Integer)
  out (reservation:Reservation)
  pre event.date>='2000-01-01'
  pre event.date<='2009-12-31'
  pre number>0 and number<100
  post reservation.status = reserved
 } ... }
```

Listing 1. Service description of ReserveTickets

The system model fragment of Listing 1 contains one actor definition for `Customer` and one service definition for `ReserveTickets`. The service definition has one calling actor of type `Actor`, three input parameters of types `Customer`, `Event` and `Integer`, and one return value of type `Reservation`. Additionally the service in Listing 1 has three preconditions which restrict the input parameters `event` and `number`, and one postcondition which states that after the reservation process, `reservation` has status `reserved`. The preconditions check the validity of input values and can be used for generating test data, e.g. by boundary value testing [4].

3.2 Test Model

As mentioned in Section 2.1, the main building blocks of the test model are test stories which are parameterized control sequences of service calls. In order to make a test story executable, we assign to it an initial `State` and a `DataTable` and then call such a triple a `TestSequenceElement`, e.g.

```
State_1 ReserveTickets_Cost Data_1 .
```

A collection of such elements forms a `TestSequence` which is the entry point for test execution. The test sequence file is executed sequentially and therefore defines its execution ordering. Figure 4 depicts the metamodel of the test environment.

In the following paragraphs we describe the elements of a `TestSequenceElement` in more detail.

The initial state `State_1` is defined in the XML file listed in Listing 2. It consists of one service call (`Login`)

[1] Due to shortage of space, the complete system and test model can be found at `http://teststories.info/trs`

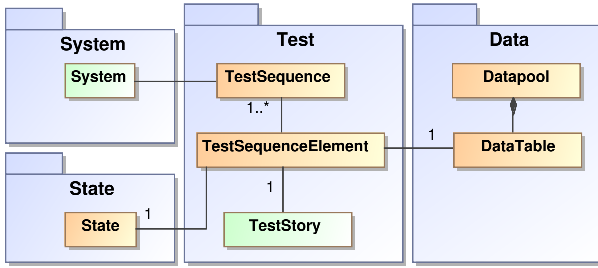

Figure 4. Metamodel of the test environment

which has to be executed before the story itself and its input and output parameters. More general, all initial system states are set by a sequence of service calls which may also invoke special system services without business functionality e.g. for filling a database.

```
<servicecall>
 <servicename>trs:Login</servicename>
 <parameter type="in" name="username"
   datatype="String">Ken</parameter>
 <parameter type="in" name="password"
   datatype="String">xyz</parameter>
 <parameter type="out" name="cId"
   datatype="Integer"/>
</servicecall>
```

Listing 2. Initial state `State_1`

The test story `ReserveTickets_Cost` in Listing 3 is based on the initial state.

```
teststory ReserveTickets_Cost {
 actor Customer c
 sequence {
  service c::ReserveTickets($cId $e $n)::#r
  assertion {
    [pass] #r.cost=$result
    [fail] not #r.cost=$result
} } }
```

Listing 3. Test story for ticket cost

The test story defines an instance `c` of the actor `Customer` and a sequence which calls the service `ReserveTickets` with input parameters `cId` which is the output value of the Login service call in `State_1` holding the identifier of a customer, `e` for an event and `n` for the number of tickets to reserve and the return value `r` for a reservation object. The variable `cost` of that object is then compared with an expected result value `result` in the assertion. Values for variables prefixed with '$' are defined by execution of the initial state (`cId`) or in the corresponding data table `Data_1` (`e`, `n`, `result`) of Table 1.

Each row in the test table has an `id` and defines data for one test case with values for an event (`e`), a number of tickets (`n`) and an expected result (`result`).

To define more complex workflows, test stories may additionally contain alternatives, parallel calls and references. The complete abstract syntax of test stories is given in Figure 5.

id	e	n	result
t_1	Event:e_1	5	135,00
t_2	Event:e_2	4	225,00

Table 1. Data table `Data_1`

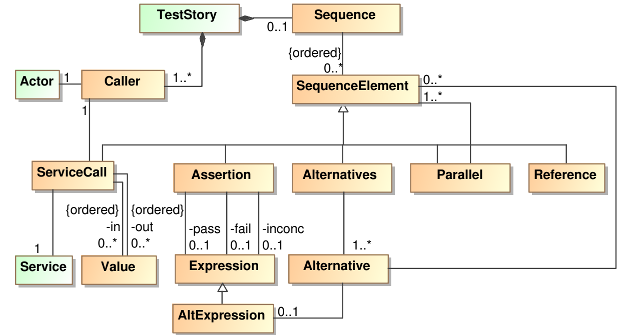

Figure 5. Abstract syntax of test stories

A test story contains one or more `Caller` which refer to an `Actor` and a `Sequence`. The latter may consist of the following elements:

- `ServiceCall` defines the called service, its calling actor, its input and return values.
- `Assertion` defines predicates for computing the test verdict. We define the following three modifiers in the same way as in [5]:
  - `pass` indicates that the test behavior gives evidence for correctness of the system under test (SUT) for the specific test case,
  - `fail` describes that the purpose of the test case has been violated,
  - `inconc` (inconclusive) is used for cases where neither a `pass` nor a `fail` can be given.

  The conditions are executed in order of their appearance, i.e. the first true expression fires. If no condition fires and the execution itself has no error, then the combination of defined modifiers determines the test result, e.g. if just `pass` and `fail` have been defined and no expression evaluates to true, then the test verdict is `inconc`.
- `Alternatives` define a conditional executions of a sequences.
- `Parallel` defines the parallel execution of sequences.
- `Reference` defines a reference to another test story.

3.3 Consistency and Coverage

In this section we give an overview of consistency and coverage which are two important properties that can be

checked at the model level and which support the iterative development of the test and system model. *Consistency rules* ensure that each test case specified in the test model can be mapped to an (abstract) execution in the system model. *Coverage rules* ensure that the set of test cases specified in the test model satisfy some kind of completeness criterion with respect to the set of all executions specified in the system model.

As an easy but representative example for a consistency rule the invariants in Listing 4 check whether a service call has the right number of input resp. output values.

```
context ServiceCall
inv inputpar: service.in->size() = in->size()
inv outputpar: service.out->size() = out->size()
```

Listing 4. OCL consistency example

As an easy example for a coverage rule the constraint in Listing 5 checks whether all system services are called in at least one test story. Therefore it defines a coverage criteria which we call *all-services*.

```
context System
inv all-services:
 services->forAll(s |
  allInstances()->select(oclIsTypeOf(ServiceCall)
  .service.name->includes(s.name))
```

Listing 5. OCL coverage example

At the moment we have implemented 12 consistency criteria and 3 service coverage criteria (*all-actors*, *all-services*, *all-services-all-actors*). The all-services-all-actors criteria which calls every service with all its defined actors has been the most useful according to user feedback. Consistency and coverage checks are an important aspect of the test drivenness of Telling TestStories because the quality of the test model (and because of the automatic mapping of the tests itself) can be investigated before implementing the system. Due to the traceability of services, consistency and coverage criteria are not just model tests but also affect the implementation. The preconditions of services are useful to derive data coverage criteria, e.g. if the minimum and maximum of input values is defined, boundaries coverage [4] can be applied. The performance and scalability of our OCL evaluation technique has already been investigated in [8].

4 Architecture and Implementation

In this section we present the architecture and the implementation of the TTS framework. The architecture is depicted in Figure 6. It has a *Repository* which stores and versions all object nodes depicted in Figure 2 and explained in the previous sections:

- *SystemModel* holds the system model,
- *TestModel* holds the test model as collection of test stories,
- *Datapool* holds the test tables corresponding to test stories and the initial states for executing test stories,
- *TestSequence* holds sequences of test stories that can be executed directly,
- *TestLog* holds the log files generated within test runs,
- *TestReport* holds test reports.

The *CheckingTool* implements model checks within and between the test model and the system model as explained in Section 3.3. The *TSCompiler* translates each test story into *TestCode*, i.e. Java code which contains one top–level story method parameterized with the input parameters during test execution. In the test code *Adapter* objects for accessing the available services are instantiated. The adapters which may be generated automatically or manually encapsulate the communication with the available services which may by implemented e.g. in CORBA or web service technology and are able to handle synchronous and asynchronous service calls. The *SUT* provides executable services and may additionally provide system services for testing, e.g. for resetting the internal database. The *TestController* executes a sequence of test stories. For every test story an initial state is set up and the stories top–level method is invoked for every line of its corresponding data table. The TestController has a *Timing* component supporting timeout monitoring needed for handling asynchronous service calls and a component for *EventHandling* processing the events which occur during the test execution such as errors, timeouts or test verdicts. The Controller generates a test log for one test story execution. Test logs are used by the *ReportManager* to produce test reports.

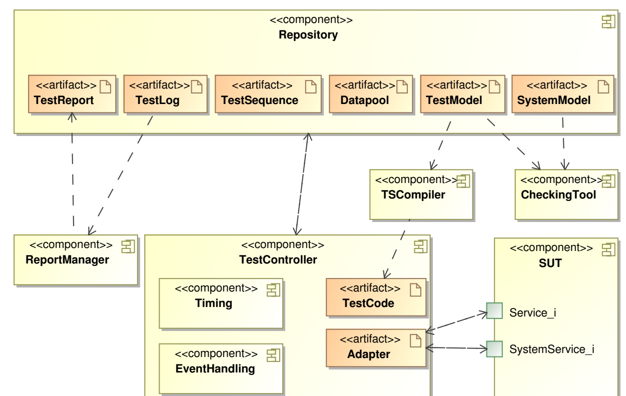

Figure 6. Architecture of TTS

We have implemented our framework based on the Eclipse platform. Therein the repository component corresponds to an versioned Eclipse workspace where all artifacts are stored as files. We have defined ecore metamodels for the test resp. the system model and have implemented editors for them[2]. The test code generator which transforms

[2] In principle every XMI 2.1 compatible tool can be used for external system and test modeling

test models into executable Java code and the consistency resp. the coverage rules have been implemented with openArchitectureWare[3]. The test controller itself has been implemented in Java. Traceability between the model and the implementation level is ensured by the adapters which are assigned to the business services by corresponding names. The report manager has been implemented as Eclipse view. The openArchitectureWare sources of our implementation are available online [4].

According to the classification in [4] our framework is *mixed* because it combines the adaptation approach, i.e. we implement adapters for calling services and the transformation approach, because we transform abstract test stories into Java code. This mixed approach makes transformation easier and generic because it is independent of concrete service calls and it uses adapters for wrapping the bidirectorial communication between SUT and the test framework.

5 Conclusion and Future Work

The concepts of TTS presented in the preceding sections are the first results of a thorough requirements elicitation phase with the goal to support a more efficient model–based system test process for incremental development of SOS. We have conducted first experiments with a service–oriented communication application of our project partner SoftMethod. With TTS it has been possible to model the functional requirements and the tests incrementally and clearly.

From this case study and from experiments with our ticket reservation system we have several technical and conceptual ideas for future improvements. In addition to services, actors and classes, future versions of the system model may be extended by a variety of sub–models and specifications such as business processes, business object life cycles or rules specifying constraints on the order of service execution which constrain the basic execution model. These additional artifacts can then be used for checking consistency resp. coverage of test stories with respect to the extended system model and for generating test cases. We will extend versioning and reporting concepts to support regression testing which is very important to handle the dynamic evolution of requirements. At the moment we restrict ourselves to functional system testing, but we will extend TTS to model and test also non–functional properties such as performance, security and quality of services which are of high importance for SOS.

6 Acknowledgements

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsfrderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT). Moreover part of the research is conducted within Telling TestStories project funded by TransIT (www.transit.ac.at).

[3] `http://www.openarchitectureware.org/.`
[4] `http://www.teststories.info/trs`

References

[1] L. Ribarov, I. Manova, and S. Ilieva. Testing in a service–oriented world. In *InfoTech–2007*, volume 1, 2007.

[2] A. D. Neto, R. Subramanyan, M. Vieira, G. H. Travassos, and F. Shull. Improving Evidence about Software Technologies: A Look at Model–Based Testing. *IEEE Software*, 25(3), 2008.

[3] C. Nebut and F. Fleurey. Automatic test generation: A use case driven approach. *IEEE Trans. Softw. Eng.*, 32(3), 2006.

[4] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2006.

[5] P. Baker, Z. Ru Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. E. Williams. *Model-Driven Testing - Using the UML Testing Profile*. Springer, 2007.

[6] I. Schieferdecker, G. Din, and D. Apostolidis. Distributed functional and load tests for web services. *Int. J. Softw. Tools Technol. Transf.*, 7(4), 2005.

[7] R. Mugridge and W. Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall PTR, 2005.

[8] J. Chimiak-Opoka, M. Felderer, C. Lenz, and C. Lange. Querying UML Models using OCL and Prolog: A Performance Study. In *2008 Model Driven Engineering, Verification, and Validation*, 2008.