# MODEL–DRIVEN SYSTEM TESTING OF SERVICE ORIENTED SYSTEMS
## *A Standard-aligned Approach based on Independent System and Test Models*

Michael Felderer, Joanna Chimiak-Opoka and Ruth Breu
*Institute of Computer Science, University of Innsbruck, Technikerstr. 21a, Innsbruck, Austria*

Keywords: Model–Driven Testing, Requirements Engineering, Service Oriented Architecture, SoaML.

Abstract: This paper presents a novel standard–aligned approach for model–driven system testing of service oriented systems based on tightly integrated but separated platform–independent system and test models. Our testing methodology is capable for test–driven development and guarantees high quality system and test models by checking consistency resp. coverage. Our test models are executable and can be considered as part of the system definition. We show that our approach is suited to handle important system testing aspects of service oriented systems such as the integration of various service technologies or testing of service level agreements. We also provide full traceability between functional resp. non–functional requirements, the system model, the test model, and the executable services of the system which is crucial for efficient test evaluation. The system model and test model are aligned with existing specifications SoaML and the UML Testing Profile via a mapping of metamodel elements. The concepts are presented on an industrial case study.

## 1 INTRODUCTION

The number and complexity of service oriented systems for implementing flexible inter–organizational IT–based business processes is steadily increasing. Basically a *service oriented system* consists of a set of independent peers offering services that provide and require operations (Engels et al., 2008). Orchestration and choreography technologies allow the flexible composition of services to workflows (OASIS Standard, 2007; W3C, 2005). Arising application scenarios have demonstrated the power of service oriented systems. These range from the exchange of health related data among stakeholders in health care, over new business models like SAAS (Software as a Service) to the cross-linking of traffic participants. Elaborated standards, technologies and frameworks for realizing service oriented systems have been developed, but system testing aspects have been neglected so far.

*System testing methods* for service oriented systems, i.e. methods for evaluating the system's compliance with its specified requirements, have to consider specific issues that limit their testability including the integration of various component and communication technologies, the dynamic adaptation and integration of services, the lack of service control, the lack of observability of service code and structure, the cost of testing, and the importance of service level agreements (SLA) (Canfora and Di Penta, 2008).

Model–driven testing approaches, i.e. the derivation of executable test code from test models by analogy to MDA, are particularly suitable for system testing of service oriented systems because they can be adapted easily to changing requirements, they support the optimization of test suites without a running system, they provide an abstract technology and implementation independent view on tests, and they allow the modeling and testing of service level agreements. The latter allows for defining test models in a very early phase of system development even before or simultaneous with system modeling. This raises the agile practice of test–driven development to the model level.

In this paper we define a novel approach to model–driven system testing of service oriented systems that is based on a separated system and test model. Beside the advantages of model–driven testing mentioned above, our approach supports test–driven development on the model level, the definition and execution of tests in a tabular form as in the FIT framework (Mugridge and Cunningham, 2005), guarantees traceability between all types of modeling and system artifacts, and is suitable for testing SLA which we consider as non–functional properties. The test

428 Felderer M., Chimiak-Opoka J. and Breu R. (2010).
MODEL–DRIVEN SYSTEM TESTING OF SERVICE ORIENTED SYSTEMS - A Standard-aligned Approach based on Independent System and Test Models.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Information Systems Analysis and Specification*, pages 428-435
DOI: 10.5220/0002873304280435
Copyright © SciTePress

models are executable and can be considered as partial system definition. We align our approach with SoaML (OMG, 2008), a standard for UML–based system modeling of service oriented systems, and the UML Testing Profile (UTP) (OMG, 2005), a standard for UML–based test modeling, by defining a mapping to these specifications. We also show how our testing approach supports traceability between requirements, system and test models, and the system under test.

The paper is structured as follows. In the next section we describe the system and testing artifacts, the corresponding metamodel, and its mapping to SoaML and UTP. In section 3 we provide an industrial case study that we have conducted based on our approach. In section 4 we provide related work and in section 5 we draw conclusions and discuss future work.

## 2 SYSTEM AND TEST MODELING METHODOLOGY

In this section we define our generic approach for test–driven modeling of service oriented systems and align it with standards for system and test modeling of service oriented systems. We first give an overview of all artifacts and then describe the metamodel and its mapping in more detail.

### 2.1 System and Testing Artifacts

Figure 1 gives an overview of all used artifacts. Informal artifacts are depicted by clouds, formal models by graphs, code by white blocks and running systems by filled blocks.

The **Requirements Model** contains the requirements for system development and testing. Its structured part consists of a requirements hierarchy. The requirements are based on informal requirements depicted as cloud.

The **System Model** describes the system structure and system behavior in a platform independent way. Its static structure is based on the notion of services which are assigned to requirements and provide resp. require interfaces. Each service corresponds to an executable service in the running system to guarantee traceability. Therefore the requirements, the services with their operations and the executable services are traceable.

The **Test Model** defines the test requirements, the test behavior, the test data and test runs. The tests are defined in a hierarchical way representing single test cases, parametrized test cases or a complete test

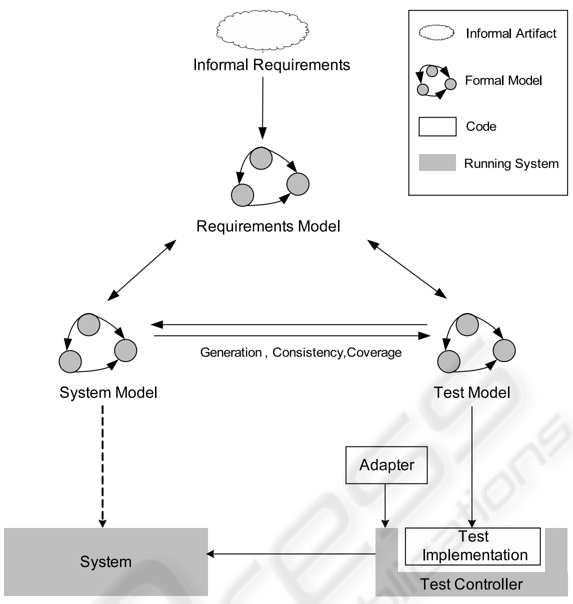


Figure 1: Artifacts overview.

suite. Tests contain calls of service operations and assertions for computing test oracles.

The system and test model are independent and typically not generated from each other. Therefore our approach can be classified as optimal in model–based testing according to (Pretschner and Philipps, 2004). Our approach has the advantage that models can already be checked for correctness, consistency and coverage before they are actually implemented. The overhead of defining two models is compensated by higher efficiency and flexibility needed for service oriented systems resulting in a higher system quality.

The **Test Implementation** is generated by a model–to–text transformation from the test model as explained in (Felderer et al., 2009b). The generated test code is processed by a test execution engine.

**Adapters** are needed to access service operations provided and required by the system under test. For a service implemented as web service, an adapter can be generated from its WSDL description. Adapters for each service guarantee traceability.

### 2.2 Metamodel

In this section we describe the formal models mentioned before, namely the Requirements, System and Test Model in more detail. Figure 2 contains three corresponding packages, `Requirements`, `System` and `Test`. Model elements that enable traceability have a stronger color and traceability links are bold (see section 3.4).

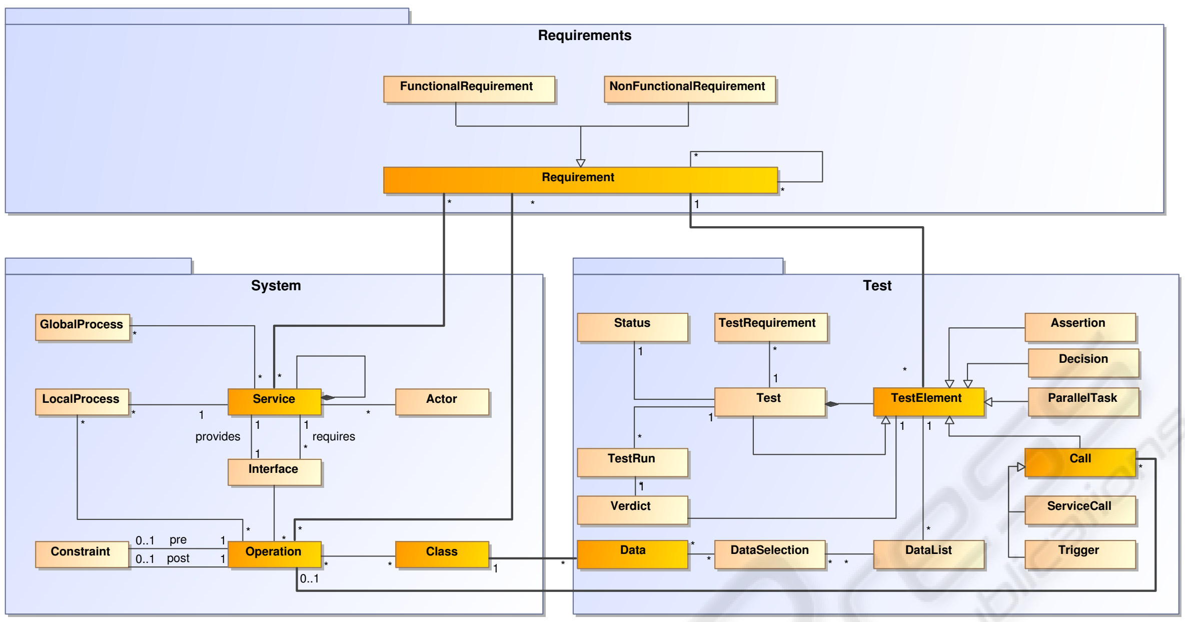


Figure 2: Metamodel for `Requirements`, `System` and `Test`.

The package `Requirements` defines `FunctionalRequirements` and `NonFunctionalRequirements` such as security or performance requirements in a hierarchical way. The requirements themselves can be of an arbitrary level of granularity ranging from abstract goals to concrete performance requirements. Requirements are traceable to tests and therefore test verdicts can be assigned to them.

The package `System` defines `Service` elements which provide and require `Interface` elements and are composed of basic services. Each interface consists of `Operation` elements which refer to classes and may have a `pre` and `post` constraint. Each service has a reference to `Actor` elements. Therefore services somehow play the role of use cases. Services may have `LocalProcess` elements that have a central control implemented by a workflow management system defining its internal behavior. Different services may be integrated into a `GlobalProcess` without central control. Orchestrations can be modeled as local processes, and choreographies as global processes. In the context of this paper, we abstract from service deployment because we are mainly interested in the relationship to requirements and tests. Our concise system view of a service oriented system has been inspired by SECTET (Hafner and Breu, 2008) a framework for security-critical, inter-organizational workflows based on services.

The package `Test` defines all elements needed for system testing of service oriented systems. A `Test` has a `Status` such as new, retestable or obsolete, some `TestRequirement` elements that the test must satisfy or cover, and it consists of `TestElement` artifacts which may be `Assertion` elements, `Call` elements, i.e. `ServiceCall` or `Trigger` elements, `ParallelTask` elements, `Decision` elements, or `Test` elements itself. A test element also has a `DataList` containing `Data` elements that may be generated by a `DataSelection` function. A test has some `TestRun` elements assigning `Verdict` values to assertions.

The system model and the test model are created manually or are partially generated from each other. If the system model and the test model are created manually, we ensure that they are consistent with each other and that the test model fulfills some coverage criteria with respect to the system model by OCL constraints (see (Felderer et al., 2009a) for consistency and coverage checks with OCL). Alternatively, if the system model is complete then test scenarios, test data and oracles can be generated, or otherwise if the test model is complete, behavioral fragments of the system model can be generated.

Each test is linked to a requirement and can therefore be considered as executable requirement by analogy to FIT tests (Mugridge and Cunningham, 2005). Tests in our sense can also be used as testing facets (Canfora and Di Penta, 2008) defining built-in test cases for a service that allows potential actors to evaluate the service. Together with appropriate requirements the tests may serve as executable SLAs.

Our approach emphasizes test–driven development because from test models and adapters it is possible to derive executable tests before the implementation has been finished. It is even possible to apply test–driven modeling because the test model can be defined before the behavioral artifacts of the system model whose design can then be supported by checking consistency and coverage between the system and the test model. In (Felderer et al., 2009a) we therefore have introduced the term *test story* for our way how to define tests by analogy to the agile term user story.

## 2.3 Mapping to SoaML and UTP

A mapping of the system model to SoaML (OMG, 2008) and of the test model to UTP (OMG, 2005) is useful for several reasons. The mapping defines a semantics for our approach by assigning the metamodel concepts to other well-defined concepts, it allows us to reuse tools that have been developed based on these specifications, and it is one instantiation of our abstract metamodel.

**System Model Mapping.** SoaML extends UML2 in four main areas: Participants, ServiceInterfaces, ServiceContracts, and ServiceData. *Participants* define the service providers and consumers in a system. *ServiceInterfaces* enable the explicit modeling of the provided and required operations. Each operation has a precondition, a postcondition, input and output data and describes a *ServiceCapability*. *ServiceContracts* are used to describe interaction patterns between participants. *ServiceData* represent service messages and attachments. Finally, the metamodel provides elements to model service messages explicitly. In Table 1 a mapping from our metamodel elements to SoaML is defined.

Table 1: Mapping of model elements from the package `System` to SoaML.

| System | SoaML |
|---|---|
| Actor | Actor |
| Class | MessageType |
| Constraint | precondition or postcondition on an operation |
| GlobalProcess | Behavior attached to a Service Architecture |
| Interface | Interface |
| LocalProcess | Behavior attached to a Participant |
| Operation | operation in ServiceInterface |
| Service | Participant, Agent, ServiceInterface |
| System | System Architecture |
| provides Interface | Service |
| requires Interface | Request |

**Test Model Mapping.** The UTP (OMG, 2005) contains four concept groups, namely *Test Architecture* defining concepts like Test context and Test component, *Test Behavior* defining concepts like Test case and Verdicts, *Test Data* defining concepts like Data pool and Data selection, and *Time* defining concepts like Timer and Time zone. More details on the informal semantics of the concepts can be found in (OMG, 2005). In (Baker et al., 2007) an operational semantics is defined via a mapping from the UTP to the test execution languages JUnit and TTCN–3 (Willcock et al., 2005). Table 2 defines a mapping from our test metamodel to the UML Testing Profile.

Table 2: Mapping of model elements from the package `Test` to UTP.

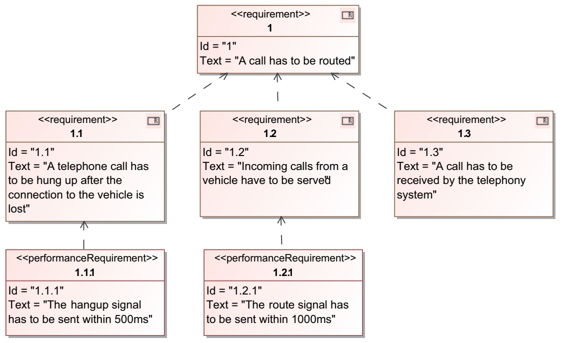
| Test | UTP |
|---|---|
| Assertion | ValidationAction, Arbiter |
| Data | Data |
| DataList | Class |
| DataSelection | DataSelection |
| Decision | decision node |
| ParallelTask | Coordination |
| Service | TestComponent |
| ServiceCall | Stimulus |
| Status | attribute of TestContext |
| System | SUT |
| Test | TestContext, TestControl, Scheduler, TestCase, TestConfiguration |
| TestRequirement | Scheduler, TestObjective |
| TestRun | TestLog, LogAction |
| Trigger | Observation |
| Verdicts | Verdicts |

## 3 TELEPHONY CONNECTOR CASE STUDY

In this section we present an industrial case study that we have designed and tested with our testing methodology. The Telephony Connector is a service oriented telematics system for the automotive industry consisting of a set of independent services. We restrict the case study description for space and understandability reasons to one specific scenario where an emergency call from a car has to be routed to a call center by a telephony connector. In this context car specific information has to be transmitted to the call center answering the phone call including the GPS data of the current car position or the number of fired airbags. After the data transmission the call center agent has the ability to speak with the people inside the crashed car. We first present the requirements, and then the system and test model. We use UML with stereotyped elements to represent the requirements, the system model and the test model.

## 3.1 Requirements

The requirements model provides a hierarchical representation of the functional and non–functional requirements to the system. In Figure 3 the requirements for routing a call are depicted.



<<requirement>>
1
Id = "1"
Text = "A call has to be routed"
<<requirement>>
1.1
Id = "1.1"
Text = "A telephone call has to be hung up after the connection to the vehicle is lost"
<<requirement>>
1.2
Id = "1.2"
Text = "Incoming calls from a vehicle have to be served"
<<requirement>>
1.3
Id = "1.3"
Text = "A call has to be received by the telephony system"
<<performanceRequirement>>
1.1.1
Id = "1.1.1"
Text = "The hangup signal has to be sent within 500ms"
<<performanceRequirement>>
1.2.1
Id = "1.2.1"
Text = "The route signal has to be sent within 1000ms"

Figure 3: Requirements for routing a call.

The functional requirements (1, 1.1, 1.2, 1.3) have the stereotype requirement and the non–functional performance requirements (1.1.1, 1.2.1) have the stereotype performanceRequirement which is a subtype of the meta model element NonFunctionalRequirement.

## 3.2 System

In this section we define a system model for our telephony connector example. The services have concrete interfaces. These interfaces use Java standard types (primitive types, type String), maps and the user defined types ErrorTypes and ByteArray depicted in Figure 4.

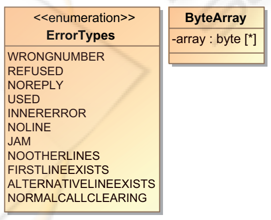


Figure 4: User defined types.

The system consists of four services providing and requiring interfaces. The services are depicted in Figure 5 and the interfaces in Figure 6.

The VehicleService represents the car that has an integrated telephone to initialize a call or to hang it

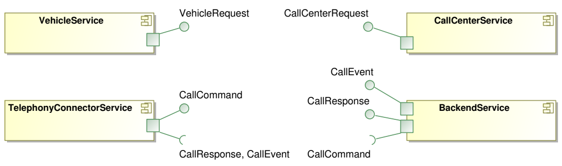
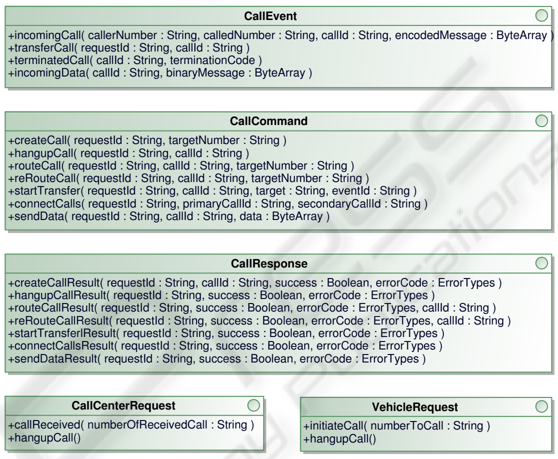


Figure 5: Services with their provided and required interfaces.



CallEvent
+incomingCall( callerNumber : String, calledNumber : String, callId : String, encodedMessage : ByteArray )
+transferCall( requestId : String, callId : String )
+terminatedCall( callId : String, terminationCode )
+incomingData( callId : String, binaryMessage : ByteArray )
CallCommand
+createCall( requestId : String, targetNumber : String )
+hangupCall( requestId : String, callId : String )
+routeCall( requestId : String, callId : String, targetNumber : String )
+reRouteCall( requestId : String, callId : String, targetNumber : String )
+startTransfer( requestId : String, callId : String, target : String, eventId : String )
+connectCalls( requestId : String, primaryCallId : String, secondaryCallId : String )
+sendData( requestId : String, callId : String, data : ByteArray )
CallResponse
+createCallResult( requestId : String, callId : String, success : Boolean, errorCode : ErrorTypes )
+hangupCallResult( requestId : String, success : Boolean, errorCode : ErrorTypes )
+routeCallResult( requestId : String, success : Boolean, errorCode : ErrorTypes, callId : String )
+reRouteCallResult( requestId : String, success : Boolean, errorCode : ErrorTypes, callId : String )
+startTransferlResult( requestId : String, success : Boolean, errorCode : ErrorTypes )
+connectCallsResult( requestId : String, success : Boolean, errorCode : ErrorTypes )
+sendDataResult( requestId : String, success : Boolean, errorCode : ErrorTypes )
CallCenterRequest
+callReceived( numberOfReceivedCall : String )
+hangupCall()
VehicleRequest
+initiateCall( numberToCall : String )
+hangupCall()

Figure 6: Interfaces of all services.

up. The CallCenterService represents a call center that receives routed calls from the vehicle via a telephony system. The TelephonyConnectorService is the service under test. Technically the Telephony-Connector is a standalone server application bridging the actual telephone system on the car vendor's side based on private branch exchange. The BackendService models the remaining infrastructure of the car vendor (Backend) like database systems holding car specific data. Hereby the Telephony Connector provides several operations to the Backend. This includes routing the call from the car to another destination, like the locally responsible police station (routeCall) or the termination of a call from a car (hangupCall), e.g. if the car accidentally calls the call center and there is no emergency or if all necessary actions were initiated. The Telephony-Connector application bridges the TelephonySystem with the remaining Backend of the car vendor, including the actual call center in order to fully control the handling of the call. The Backend is the only service without an underlying telephony infrastructure.

Each operation may have a precondition and a postcondition denoted in OCL. The operation initiateCall for instance has the precondition callerNumber.size>=0 and calledNumber.size>=0 and callId.size>=0 and callerNumber <> calledNumber.

In this example the operations are asynchronous calls without return values. Therefore there are no postconditions defining a relationship between input and output parameters. In general from failing preconditions one can guess an irregularity in the environment and from failing postconditions one can guess irregularities in the operation under test (Meyer et al., 2009). We do not need the full capabilities of pre- and postconditions in our example but we just automate the application of oracles by evaluating assertions.

A local process for the `VehicleService` is represented as state machine in Figure 7.

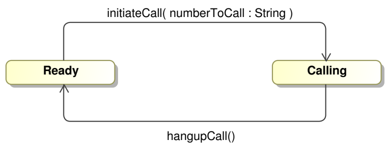


Figure 7: Vehicle states.

A global process for routing a call can be represented by a UML activity diagram. We do not consider a global workflow here but in other scenarios global workflows can be the basis for the generation of tests or for checking the consistency of tests.

## 3.3 Test

In this section we define a test model for our telephony connector example referring to the requirements and system model defined before. The tests are represented as UML activity diagrams and the corresponding data in tables. Figure 8(a) depicts one parametrized test `RouteCall` that is referred in the test `TestSuite` depicted in Figure 8(b) and fed with data which is represented in Table 3.

In the test `RouteCall`, a call is initiated (`initiateCall`), and a trigger in the backend service receives the call (`incomingCall`). Then the telephone call is routed via the service call `routeCall` and the trigger `routeCallResult`, and finally the telephone call is terminated via the service call `hangupCall` and the trigger `hangupCallResult`. Assertions check whether the routing and hangup return the expected values and react within time. The test `TestSuite` calls the parameterized tests `RouteCall` and `ConnectCall` with the data tables `RouteCallTestData` and `ConnectCallTestData`. The assertions check the percentage of pass which has to be 100% for `RouteCall`, more than 90% for `ConnectCall`, and an average time for passed test cases below 100 milliseconds. Basically the tests define activity flows of the system. The attached stereotypes define its operational semantics: `Servicecall` elements are invoked by the test execution engine, `Trigger` elements are invoked on the test execution engine and `Assertion` elements define checks for computing verdicts.

Table 3 defines two test cases. It also uses a data selection function `genInt` to generate an integer between 1 and 100 which serves as request identifier.

Table 3: Testdata for Test RouteCall.

| Parameter | Test1 | Test2 |
|---|---|---|
| initiateCall.numberToCall | SIP:1234 | SIP:1234 |
| routeCall.requestId | genInt(1,100) | genInt(1,100) |
| routeCall.callId | incomingCall.callId | incomingCall.callId |
| routeCall.targetNumber | SIP:transfer | SIP:invalid |
| Assertion1.success | true | false |
| hangupCall.requestId | routeCallResult.requestId | |
| hangupCall.callId | routeCallResult.callId | |
| Assertion2.success | true | |

## 3.4 Traceability

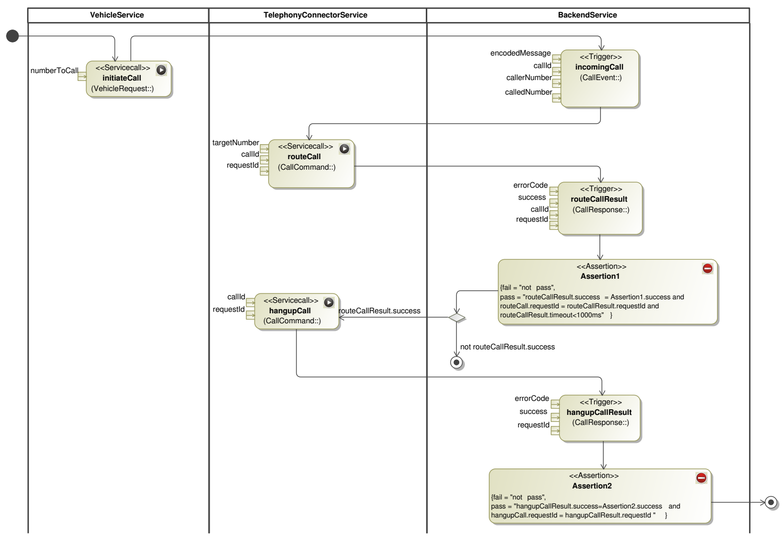
Traceability between requirements and other artifacts generated during the system development and testing process is essential for systems evolution and testing because in both cases all affected artifacts have to be checked or even changed. Our approach comprises traceability between the requirements model, the system model, the test model and the running system. Each test element is assigned to a requirement, and implicitly to its super–artifacts. Tests can therefore be considered as executable requirements because they assign an executable model to requirements. If the execution of a test story fails, then the requirements itself and all its super–artifacts fail. In Table 4 we define the mapping of the test elements to the requirements, i.e. the traces between the packages Test and Requirements in Figure 2.

Table 4: Traceability between test elements and requirements.

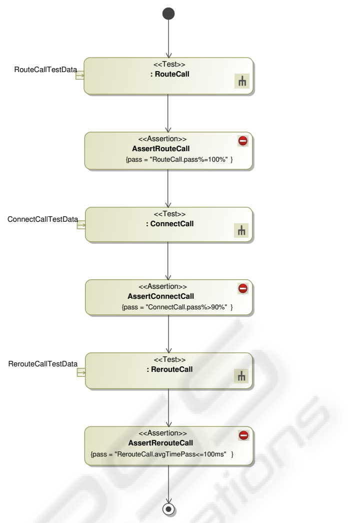
| TestElement | Type | related Requirements |
|---|---|---|
| RouteCall | Test | 1 |
| initiateCall | ServiceCall | 1.3 |
| incomingCall | Trigger | 1.3 |
| routeCall | ServiceCall | 1.2 |
| routeCallResult | Trigger | 1.2, 1.2.1 |
| Assertion1 | Assertion | 1.2, 1.2.1 |
| hangupCall | ServiceCall | 1.1 |
| hangupCallResult | Trigger | 1.1, 1.1.1 |
| Assertion2 | Assertion | 1.1, 1.1.1 |

Due to this mapping, if the execution of a service call via its adapter fails, the failure can be propagated via the test element to specific requirements.

Technically we have implemented traceability between the requirements and the system resp. test model via tagged values storing the referenced test elements resp. requirements.



(a) Test RouteCall



(b) Test TestSuite

Figure 8: Tests RouteCall and TestSuite.

## 3.5 Test Execution and Evaluation

We have implemented a tool for our methodology[1] that has been used for test modeling, test code generation, test execution and test evaluation. Based on the result of the test execution the test data is colored green (if the test passed) or red (if the test failed). Additionally due to the traceability we can trace from test elements to requirements to localize failures. We can also narrow the source of the failure. Compared to the testing process applied before, our framework allowed to detect faults earlier and more frequent, and to fully automate the testing process.

## 4 RELATED WORK

System testing based on SoaML (OMG, 2008) specifications has not been considered so far. We address this problem indirectly by defining a generic system model that can be mapped to SoaML and relating the system model to a test model. This guarantees that our approach can be integrated with other modeling approaches for service oriented systems such as Quasar Enterprise (Engels et al., 2008) or the OASIS SOA Reference Model (OASIS Standard, 2006) which are compatible to SoaML.

(Abbors et al., 2009) defines a requirements model and a system model similar to our approach based on domains models, used and required interfaces, and state machines. But in that approach only the mapping to Qtronic (Conformiq, 2009), a tool for automated test design, and not traceability or the relationship between the system and an independent test model are considered.

A model–driven approach to testing SOA with the UML Testing Profile (OMG, 2005) is defined in (Baker et al., 2007). It focuses on web services technology and uses the whole set of UTP concepts. Our approach can be mapped to that approach but additionally it is designed for arbitrary service technologies, provides a service–centric view on tests, supports the tabular definition of tests and guarantees traceability between all involved artifacts.

In (Zander et al., 2005) model–driven testing is defined and implemented by mapping test models in UTP to test code in TTCN–3. The focus of the paper is on the transformation of test models to executable code whereas we focus on the relationship between requirements, test models and system models.

In (Margaria and Steffen, 2004) a system testing method whose test model is based on test graphs similar to our test stories is presented. Therein consistency and correctness is validated via model checking but the relationship to system models and traceability such as in our approach are not considered.

In (Atkinson et al., 2008) test sheets, an extension of FIT, are used for specifying high quality services. The approach is compatible to our approach but it is based on a pure tabular representation and it does not consider abstract requirements, system and test models as we do.

In (Canfora and Di Penta, 2008) unit testing, inte-

[1]The Telling TestStories tool is available at http://teststories.info/

gration testing, regression testing and non–functional testing of service oriented systems are discussed but in contrast to our approach system testing is not considered explicitly.

## 5 CONCLUSIONS AND FUTURE WORK

We have presented a novel approach to model–driven system testing for service oriented systems based on a separated system and test model. The approach has been aligned with SoaML and the UML Testing Profile. Our methodology integrates system and test modeling for service oriented systems in a concise way and addresses main system testing issues of service oriented systems such as the integration of various service technologies, the evolution of services, or the validation of service level agreements. Additionally, our approach is fully traceable by defining links between the requirements model, the system model, the test model and the executable system. We successfully applied our approach on an industrial case study from the telecommunication domain.

As next step we will consider the evolution of models which is very important in a service–oriented context and which emphasizes model–based regression testing. Additionally we also consider oracle computation based on OCL verification (Cabot et al., 2009) or semantic web technologies to fully elaborate test generation. A mapping from our model–driven approach to the table–driven test sheets approach (Atkinson et al., 2008) provides an additional tabular and textual representation of tests and the capability of usability testing.

## ACKNOWLEDGEMENTS

This work was partially supported by the Telling TestStories project funded by the trans-it and the MATE project funded by the FWF.

## REFERENCES

Abbors, F., Pääjärvi, T., Teittinen, R., Truscan, D., and Lilius, J. (2009). Transformational support for model-based testing – from UML to QML. In *2nd Workshop on Model-based Testing in Practice*.

Atkinson, C., Brenner, D., Falcone, G., and Juhasz, M. (2008). Specifying High-Assurance Services. *Computer*, 41:64–71.

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

Cabot, J., Clarisó, R., and Riera, D. (2009). Verifying UML/OCL Operation Contracts. In *IFM 2009*, pages 40–55, Berlin, Heidelberg. Springer-Verlag.

Canfora, G. and Di Penta, M. (2008). Service-oriented architectures testing: A survey. In Lucia, A. D. and Ferrucci, F., editors, *ISSSE*, volume 5413 of *Lecture Notes in Computer Science*, pages 78–105. Springer.

Conformiq (2009). Qtronic. http://www.conformiq.com/.

Engels, G., Hess, A., Humm, B., Juwig, O., Lohmann, M., Richter, J.-P., Voß, M., and Willkomm, J. (2008). A Method for Engineering a True Service-Oriented Architecture. In *ICEIS 2008*.

Felderer, M., Breu, R., Chimiak-Opoka, J., Breu, M., and Schupp, F. (2009a). Concepts for Model–Based Requirements Testing of Service Oriented Systems. IASTED SE’2009.

Felderer, M., Fiedler, F., Zech, P., and Breu, R. (2009b). Flexible Test Code Generation for Service Oriented Systems. QSIC’2009.

Hafner, M. and Breu, R. (2008). *Security Engineering for Service–Oriented Architectures*. Springer-Verlag, Berlin Heidelberg.

Margaria, T. and Steffen, B. (2004). Lightweight coarse-grained coordination: a scalable system-level approach. *STTT*, 5(2-3).

Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., and Stapf, E. (2009). Programs that test themselves. *Computer*, 42:46–55.

Mugridge, R. and Cunningham, W. (2005). *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall.

OASIS Standard (2006). OASIS SOA Reference Model TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm.

OASIS Standard (2007). Web Services Business Process Execution Language Version 2.0 - OASIS Standard. http://docs.oasis-open.org/wsbpel/2.0/.

OMG (2005). *UML Testing Profile, Version 1.0*. http://www.omg.org/docs/formal/05-07-07.pdf.

OMG (2008). *Service Oriented Architecture Modeling Language (SoaML) - Specificiation for the UML Profile and Metamodel for Services (UPMS)*. Object Modeling Group. http://www.omg.org/docs/ad/08-08-04.pdf.

Pretschner, A. and Philipps, J. (2004). Methodological issues in model-based testing. In *Model-Based Testing of Reactive Systems*, pages 281–291.

W3C (2005). Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/ws-cdl-10/.

Willcock, C., Deiss, T., Tobies, S., Keil, S., Engler, F., and Schulz, S. (2005). *An Introduction to TTCN–3*. John Wiley and Sons.

Zander, J., Dai, Z. R., Schieferdecker, I., and Din, G. (2005). From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In *TestCom*.