Model Validation in a Tool–Based Methodology for System Testing of Service–Oriented Systems

Michael Felderer*, Joanna Chimiak–Opoka*, Philipp Zech*, Cornelia Haisjackl*, Frank Fiedler†, Ruth Breu*
*Institute of Computer Science
University of Innsbruck
Innsbruck, Austria
Email: {michael.felderer, joanna.opoka, philipp.zech, cornelia.haisjackl, ruth.breu}@uibk.ac.at
†Softmethod GmbH
Munich, Germany
Email: frank.fiedler@softmethod.de

***Abstract*—In this article we present a novel model–driven system testing methodology for service–centric systems called Telling TestStories, its tool implementation and the underlying model validation mechanism. Telling TestStories is based on tightly integrated but separated platform–independent requirements, system and test models. The test models integrate test data tables and encourage domain experts to design tests. This process is supported by consistency, completeness, and coverage checks in and between the requirements, system and test models which guarantees a high quality of the models. Telling TestStories is capable of test–driven development on the model level and provides full traceability between all system and testing artifacts. The testing process of the Telling TestStories methodology comprises model development, model validation and system validation. The model development and the system validation are managed by the Telling TestStories tool and the model validation is managed by the SQUAM tool. All process steps, the underlying artifacts and the tools for implementing the process steps are presented by an industrial case study.**

***Keywords*-Model–Driven Testing; System Testing; Model Validation; Testing Methodology; Testing Tools; Service–Oriented Architecture;**

I. 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 [1]. Orchestration and choreography technologies allow the flexible composition of services to workflows [2], [3]. 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 tools and methodologies have been neglected so far.

System testing of service–oriented systems, i.e., validating the system's compliance with the specified requirements, has to consider specific issues that limit the testability of such systems 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) [4].

Model–driven testing approaches [5], i.e., the derivation of executable test code from test models by analogy to Model Driven Architecture (MDA) [6], are particularly suitable for system testing of service–oriented systems because they can be adapted easily to changing requirements, they support static model validation to improve the quality of the tests, 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 supporting test–driven development on the model level.

In this article, we present a tool–based methodology to model–driven system testing of service–oriented systems called *Telling TestStories* (TTS) and its integrated model validation mechanism. The methodology is explained by an industrial case study from the telecommunication domain.

TTS is based on separated but interrelated requirements, system, and test models. All requirements in the requirements model are traceable to system and test model artifacts. The test model invokes operations provided by system services.

The quality of manually designed models and therefore the quality of the overall test results in our methodology can significantly be improved by model validation which is therefore a core component of TTS.

Model validation is an activity where the model is statically analyzed against a set of consistency and completeness criteria and metrics. *Consistency* criteria assure that a model is non–contradictory, and *completeness* criteria assure that a model element contains all essential information. We define intra–

model validation checks, e.g., that each test contains at least one assertion, and inter–model validation checks, e.g., that each requirement is traceable to at least one test. Additionally, we consider *coverage* checks, i.e., inter–model completeness checks where one model is the test model, e.g., that each system service is invoked in at least one test. Although our validation rules check the conformance of models and model elements, we do not use the term verification in our respect to avoid confusion with formal verification based on a proof system.

Besides the advantages of model–driven testing and model validation, our approach supports test–driven development on the model level, the definition and execution of tests in a tabular form as in the Framework for Integrated Testing (FIT) [7], guarantees traceability between all types of modeling and system artifacts, and is suitable for testing SLA which we consider as non–functional properties. We also show how our testing approach supports traceability between requirements, system and test models, and the system under test (SUT).

This article substantially extends the tool–based methodology for model–driven system testing of service–oriented systems presented in [8] where the model development and the system validation in TTS have been considered. Herein, we complete the work of [8] by also considering the model validation in TTS. We define consistency and completeness/coverage metrics and criteria in and between the requirements, system and test model. We also explain how the model validation has been implemented in the framework for Systematic Quality Assessment of Models (SQUAM).

The article is structured as follows. We first present the basic concepts of the TTS methodology by defining the underlying artifacts, the testing process and the metamodel of the model artifacts (see Section II). We then explain the model development, the model validation, and the system validation by a case study and its tool integration (see Section III) and describe the architecture of the TTS and SQUAM tool–implementations (see Section IV). Afterwards we provide related work (see Section V), and finally we draw conclusions and discuss future work (see Section VI).

II. Basic concepts of the methodology

A *testing methodology* defines a testing process and the underlying artifacts such as the defined models, the generated code, and the running systems.

In this section, we provide an overview of the TTS artifacts, the testing process and the metamodel of the model artifacts which is the basis for model validation.

A. Artifacts of TTS

Fig. 1 shows the artifacts and dependencies within the TTS framework. In the TTS framework we can distinguish three formalization levels with informal artifacts (at the top), model artifacts (in the middle), and implementation artifacts (at the bottom). Informal artifacts are depicted by clouds, formal models by graphs, code by transparent blocks and running systems by filled blocks. Due to formalization at the two lower levels we can conduct automatic transformations and validations. Formalized dependencies between artifacts are depicted by solid lines, whereas informal dependencies are depicted by dashed lines. In the following paragraphs, we explain the artifacts and dependencies of the TTS framework.

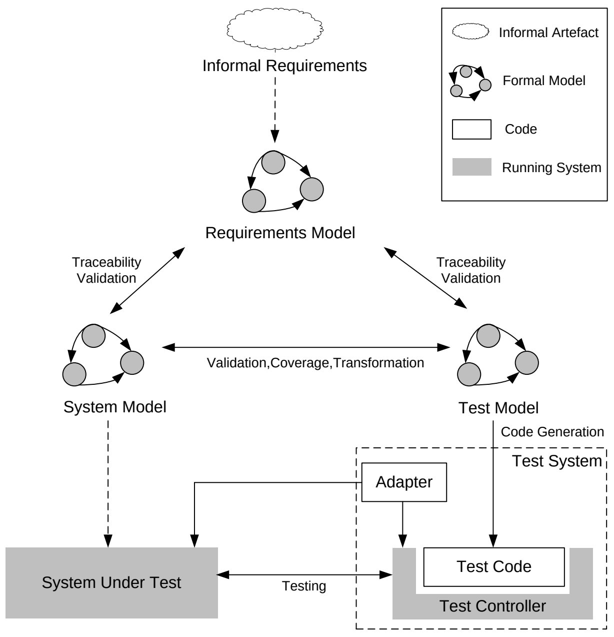


Fig. 1. Overview of the TTS Artifacts

1) Informal level: There is only one type of artifact on this level, namely the **Informal Requirements**, i.e., written or non–written capabilities and properties of the system. This artifact is not discussed in detail because it is not in the main focus of our testing methodology and as an informal one can not be automatically validated.

2) Model level: There are three models on this level: requirements, system and test model. The models are formally related through different dependencies related to: traceability, validation, coverage, and transformation.

Requirements Model. 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 requirements model provides a way to integrate textual descriptions of requirements which are needed for communication with non–technicians into a modeling tool.

System Model. The system model describes the system structure and system behavior in a platform independent way. Its static structure is based on the notions of services, components and types. Each service operation call is assigned to use cases, actors correspond to components providing and requiring services, and domain types correspond to types. We assume that each service in the system model has a one–to–one correspondence to an executable service in the running system to guarantee traceability. Therefore the requirements, the service operations and the executable services are traceable.

Test Model. The test model defines the test data and the test scenarios as so called test stories. *Test stories* are controlled sequences of service operation invocations exemplifying the interaction of components. Test stories may be generic in the sense that they do not contain concrete objects but variables which refer to test objects provided in tables. Test stories can also contain setup procedures, tear down procedures and contain assertions for test result evaluation. The notion of a test story is principally independent of its representation. We have used UML sequence diagrams in previous case studies [9] and use activity diagrams in this article. Test stories include references to a table of test data including values for all free parameters of the test story. Each line in this table defines test data for one test case. We use the terms ’test story’ and ’test’ interchangeably in this article depending on the context. If we address the more abstract view, we use the term ’test’. If we address the more application–oriented and process–oriented view, we use the term ’test story’.

Traceability. For model maintenance, transformations and validations traceability between different model elements is required. In the TTS framework traceability between elements on the model level is guaranteed by links between model elements, and between the model level and the implementation level by adapters for each service. The adapters link service calls in the model to executable services. Therefore every service invocation is traceable to a requirement.

Transformation. In the TTS framework we consider model–to–model transformations to obtain a (partial) test model from the system model. In such a system–driven development approach, test behavior and test data can be extracted from the graph of a global or local workflow.

Validation. Models designed manually require tool supported validation. Our approach is suitable for test–driven modeling because the test model is used to validate the system. In the context of TTS, we consider two properties: consistency and completeness/coverage.

Consistency checks assure that there is no conflicting information in models. Consistency of a model enables error–free transformation from the model to another model or to the source code. For manually designed (parts of) models consistency within and between them should be automatically checked. In TTS we have implemented consistency criteria for all three models and between pairs of them.

Completeness checks assure that one artifact is complete, i.e., contains all essential information. Similarly like for consistency, we can consider completeness within one model (for elements and their properties) and between models. Completeness of the system model is crucial for the TTS framework and determines whether transformations from the system model to the test model can be applied. If the system model is complete, then behavioral parts of the test model can be generated by model transformations.

Coverage can be considered as a variant of inter–model completeness where one model is the test model. This aspect is very important in context of testing and is used to check to what extend the test model covers the requirements and system model and implicitly the system. We adopted a series of coverage criteria from testing [10] and model–driven testing [11] to fit into the TTS framework.

3) Implementation level: At this level the test code generated from the test model is executed by the test controller against the system under test. The executable services of the system under test are invoked by adapters.

Code Generation. The test code is generated automatically by a model–to–text transformation from the test model as explained in [12]. For each test in the test model, a test code file is generated.

Test Code. The test code language is Java. Adapters which bind abstract service calls in the test code to running services of the system under test make the test code executable.

Adapters. The adapters are needed to access service operations provided and required by components of the system under test. For a service implemented as web service, an adapter can be generated from its WSDL description. Adapters for each service are the link for traceability between the executable system, the test model and the requirements. Adapters make it possible to derive executable tests even before the system implementation has been finished which supports test–driven development.

Test Controller. The test controller executes the test code and accesses the system services via adapters. Our implementation of the test controller executes test code in Java but other JVM–based programming or scripting languages are also executable without much implementation effort.

Test System. The test controller, the adapter and the test code constitute the test system.

Testing. The evaluation of the service–centric system by observing its execution [10] is called testing. Services are invoked in the test code executed by the test controller via adapters.

System Under Test. The system under test is a service–oriented system, i.e., offering services that provide and require interfaces. It may contain special interfaces for testing purposes.

The informal requirements can be considered as external input to the TTS framework, and the system under test is the target of the application of TTS. This is shown by two dashed arrows in Fig. 1. The first dashed arrow goes from the informal requirements to the TTS requirements model, and the second dashed arrow goes from the test controller and adapters to the system under test. Both, the informal requirements and the system under test, are out of the scope of this article.

B. Testing Process

The process consists of a *design*, *validation*, *execution*, and *evaluation* phase and is processed in an iterative way. Initially, the process is triggered by requirements for which services and tests have to be defined. The process is depicted in Fig. 2.

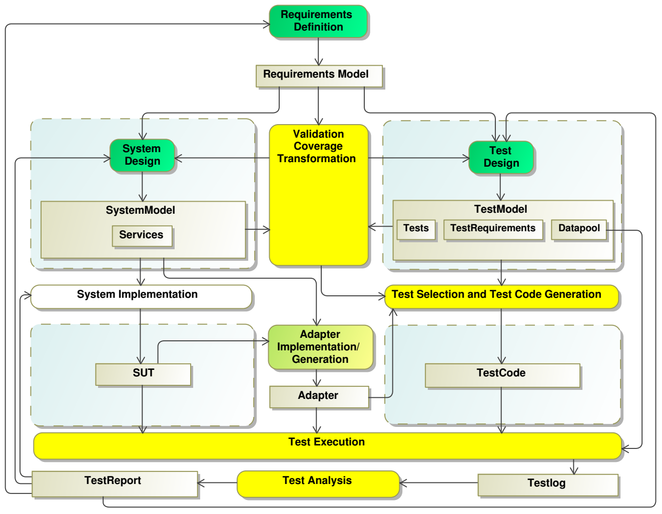


Fig. 2. Model–driven Testing Process

The first step is the definition of requirements. Based on the requirements, the system model containing services and the test model containing tests are designed. The test design additionally includes the data pool definition and the definition of test requirements. The system model and the test model, including the tests, the data pool and the test requirements, can be validated for consistency and completeness and checked for coverage. In a system–driven approach tests can be generated from the system model by model–to–model transformations, and in a test–driven approach tests can be integrated in the system model. This validity checks allow for an iterative improvement of the system and test quality. In principle, the testing process can also be considered as test model–driven development process. Our methodology does not consider the system development itself but is based on traceable services offered by the system under test. As soon as adapters which may be – depending on the technology – generated (semi–)automatically or implemented manually are available for the system services, the process of test selection and test code generation, i.e., model–to–text transformation can take place. In the tool implementation, adapters for web services can be generated automatically based on a WSDL description, adapters for RMI access can only be generated semi–automatically. The generated test code is then automatically compiled and executed by a test controller which logs all occurring events into a test log. The test evaluation is done offline by a test analysis tool which generates test reports and annotations to those elements of the system and test model influencing the test result. Test reports and test logs are implementation artifacts that are not important for the overall process but for the practical evaluation. Therefore test reports and test logs have not been considered in the previous section.

In [13] we have introduced the term *test story* for our way how to define tests by analogy to the agile term user story defining a manageable requirement together with acceptance tests.

The separation of the test behavior and the test data has been influenced by the column fixture of the Framework for Integrated Test (FIT) [7] which allows the test designer to focus directly on the domain because tests are expressed in a very easy–to–write and easy–to–understand tabular form also suited for data–driven testing.

The manual activities in the TTS testing process and the test execution are conducted by specific roles. In Fig. 3 these roles and their activities are shown.

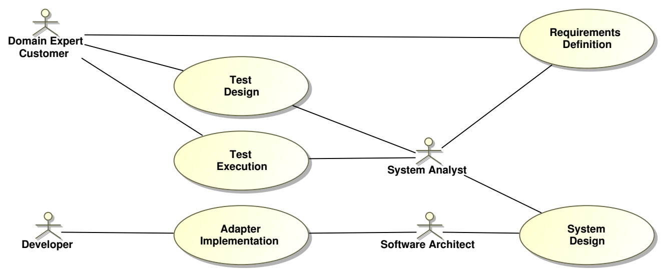


Fig. 3. Roles in the TTS Process

A *Domain Expert* or *Customer*, which are represented by one role in Fig. 3, are responsible for the test design and the requirements definition. Additionally, domain experts or customers may initiate the test execution and all related automatic activities (validation, test code generation, test analysis). Domain experts and customers are responsible for the same activities but have different views on testing, i.e., domain experts represent the internal view conducting system tests and customers represent the external view conducting acceptance tests.

A *System Analyst* is partially responsible for the test design, the requirements definition, and the system design. Additionally, the system analyst may also initiate the test execution and all related activities. The system analyst is especially responsible for the definition of non–functional requirements, e.g., for security or performance and corresponding tests.

A *Software Architect* is partially responsible for the system design and the adapter implementation. The software architect defines interfaces and component technologies in coordination with the system analyst and the developer.

A *Developer* implements adapters if they have to be developed manually.

C. Metamodel

In this section we define a metamodel for the requirements model, the system model, and the test model of TTS. The consistency, completeness, and coverage checks defined in Section III-B are based on this metamodel. The metamodel is shown in Fig. 4.

The package `Requirements Model` defines the element `Requirement` which is supertype of the types `FunctionalRequirement` and `NonFunctionalRequirement`. The type `NonFunctionalRequirement` may have further subtypes for specific types of non–functional requirements, e.g., security or performance. The requirements itself can be of an arbitrary level of granularity ranging from abstract goals to concrete performance requirements. Requirements are modeled explicitly on the metamodel level to define traceability between requirements and other model elements such as tests. The requirements model is very generic but it can easily be extended for specific purposes.

The package `System Model` defines `Service` elements which provide and require `Interface` elements and are composed of basic services. Each interface consists of `Operation` elements which refer to `Type` elements for input and output parameters. Types may be primitive types, enumeration types or reference types. Operations may also have a precondition (`pre` constraint) and a postcondition (`post` constraint). Each service has a reference to `Actor` elements. It is also possible to identify services with a service operation (if there is only one) and to identify them with service calls. Services can be therefore be considered as executable 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.

The package `Test Model` defines all elements needed for system testing of service–centric systems. A `TestSequence` consists of `SequenceElement` blocks, that integrate a `Teststory`, its `DataList`, and an `Arbitration`. A `Teststory` consists of the following elements:

- `Assertion` elements for defining expressions for computing verdicts,
- `Call` elements, i.e. `Servicecall` elements for invoking operations on services, or `Trigger` elements for operations that are called by a service,
- `ParallelTask` elements for the parallel execution of behavior, or
- `Decision` elements for defining alternatives.

`Testsstory` elements are completely recursive and, in principle, there is no limit to the number of levels to which tests can be nested. However, in practice nesting depths greater than three are not applied and it is even not clear what use nesting depths greater than two or three would be.

A `DataList` contains `Data` elements that may be generated by a `DataSelection` function. A `Testsequence` has several `TestRun` elements assigning `Verdict` values to assertions. The verdict can have the values *pass*, *fail*, *inconclusive*, or *error*. Pass indicates that the SUT behaves correctly for the specific test case. Fail indicates that the test case has violated. Inconclusive is used where the test neither passes nor fails. An error verdict indicates exceptions within the test system itself. In the model itself only a pass or a fail can be specified. Inconclusive or error are assigned automatically. This definition of verdicts originates from the OSI Conformance Testing Methodology and Framework [14].

Assertions are boolean expressions that define criteria for computing pass, fail or inconclusive verdicts. Assertions can access all variables in the actual evaluation context.

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, their quality is validated by consistency, completeness and coverage rules. Alternatively, if the system model is complete then test scenarios, test data and oracles can be generated. If the test model is complete, then behavioral fragments of the system model can be generated.

An `Arbitration` element defines a criterion on the set of the verdicts of a test run to determine whether a sequence of tests assigned to a `SequenceElement` has been executed successfully or not.

In the case of a test, a data list defines a test table, i.e., a list of lists. Data selection functions for example randomly generate integers within a specific range. This function is then denoted by `genInt(a,b)` and generates a random integer between the integers *a* and *b*.

The TTS metamodel has been implemented as a UML profile. For all metamodel elements despite data–specific elements that are implemented in tables, stereotypes of the same name have been introduced and assigned to UML metaclasses, e.g., a `Service` in our metamodel is assigned to the UML metaclass `Class`, a `Requirement` is assigned to `Class`, a `Testsequence` is assigned to an `Activity`, a `SequenceElement` is assigned to an `Action`, a `Teststory` is assigned to an `Activity`, and a `Servicecall` is assigned to an `Action`.

III. Tool–Based Case Study

In this section we present a tool–based case study for our testing methodology. We consider the the model development

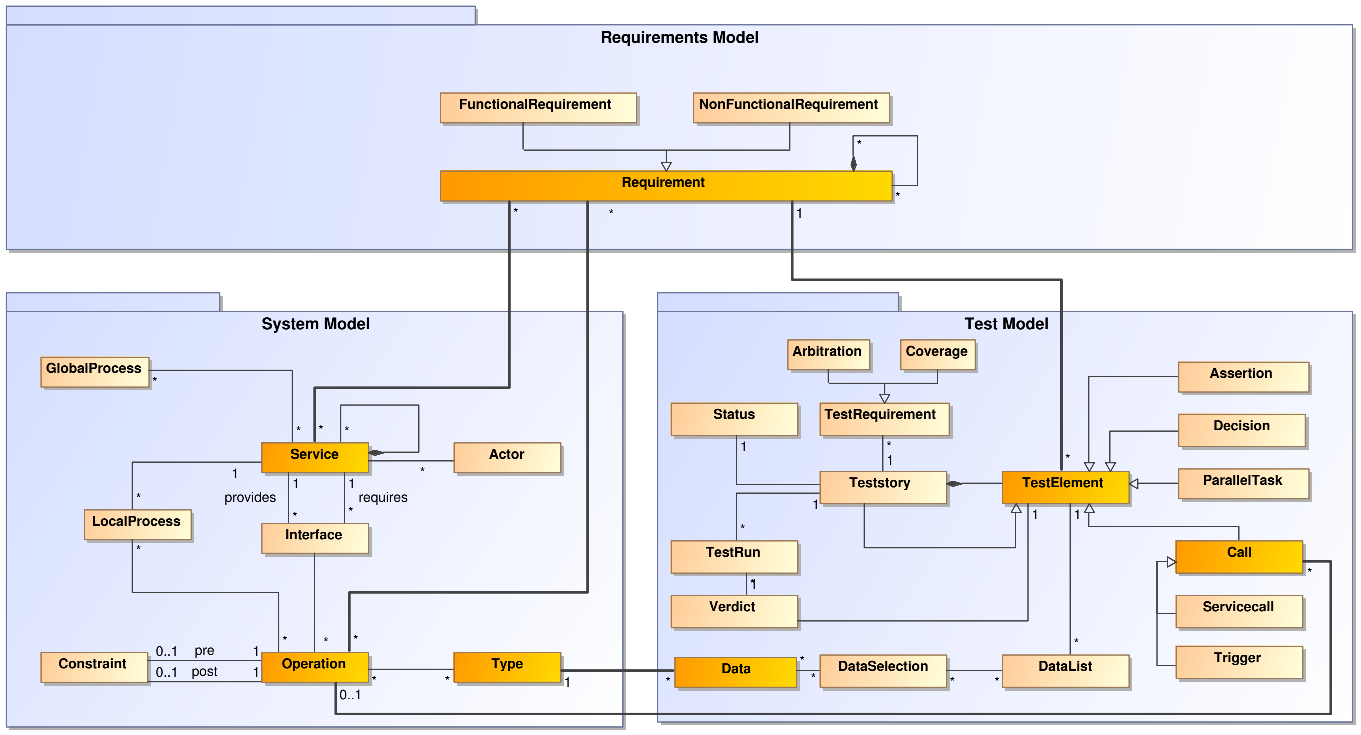


Fig. 4. Requirements, System and Test Metamodel of TTS

phase (see Section III-A), the model validation phase (see Section III-B), and finally the system validation phase (see Section III-C) of the TTS testing process.

A. Model Development

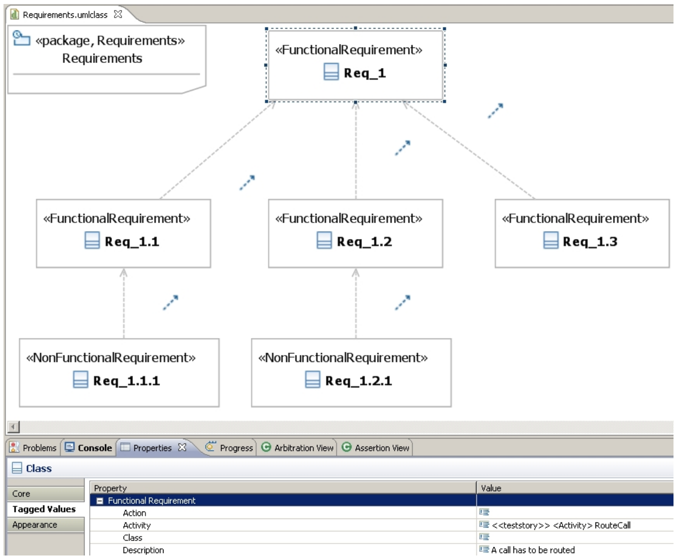
We have applied our testing tool on several case studies, including an industrial one. In this section, we explain our methodology and its tool implementation by a *Telephony Connector* case study.

The Telephony Connector is an application in the area of Computer Telephony Integration (CTI) and parts of it have already been tested with unit tests. But the whole application can currently only be tested by manual tests. TTS provides model–driven testing support for the telephony connector which is more efficient concerning the testing time and the error detection rate.

In this section, we explain how we have tested the telephony connector case study with our framework and which conclusions can be drawn. As first step, we have developed the requirements, the static parts of the system model and the test model with our tool.

The requirements are modeled as class diagram where high level requirements are aggregated by low level functional and non–functional requirements. This representation is analogous to requirements diagrams of SysML [15] and guarantees that requirements are integrated into the model which simplifies the implementation of traceability.

The requirements for routing a call are depicted in Fig. 5.



Property	Value
Functional Requirement	
Action	
Activity	<<teststory>> <Activity> RouteCall
Class	
Description	A call has to be routed

Fig. 5. Requirements to the Callmanager Application

We have modeled a requirement for routing a call (`Req_1`) and its parts, including non–functional performance requirements to hangup a call within 1000ms (`Req_1.1.1`) and to send the route signal within 1000ms (`Req_1.2.1`).

In the system model types are represented as class diagrams, and services as classes with their providing and requiring interfaces. In Fig. 6 the interfaces provided by the service

VehicleService and *TelephonyConnectorService* are depicted. Required contracts for the operations depicted in the interfaces, are modeled in terms of pre– and postconditions. Yet these contracts may only be defined over the scope of input parameters to service invocations, as currently SUT specific program variables reside in another runtime and hence are outside the accessibility of TTS.

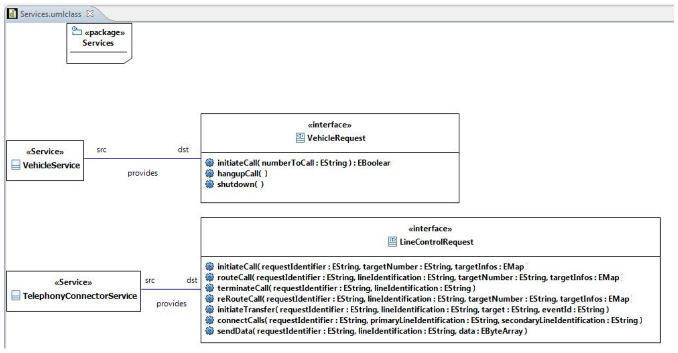
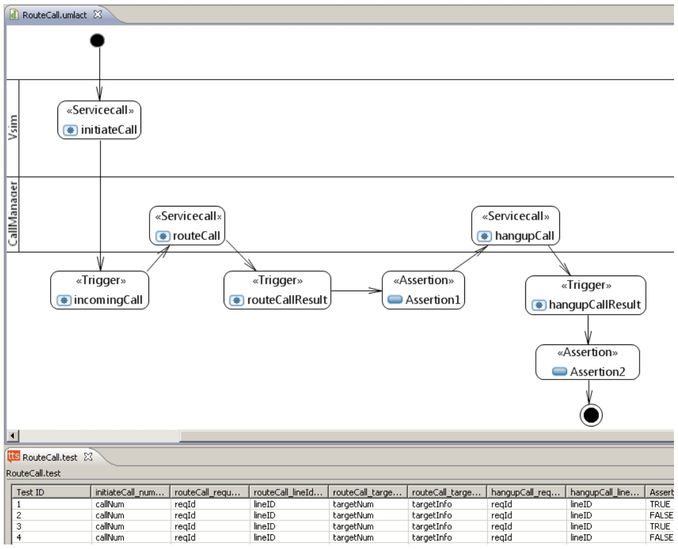

Services.umlclass
«package» Services
«Service» VehicleService
src
dst
provides
«interface» VehicleRequest
initiateCall(numberToCall : EString) : EBoolean
hangupCall( )
shutdown( )
«Service» TelephonyConnectorService
src
dst
provides
«interface» LineControlRequest
initiateCall(requestIdentifier : EString, targetNumber : EString, targetInfos : EMap
routeCall(requestIdentifier : EString, lineIdentification : EString, targetNumber : EString, targetInfos : EMap
terminateCall(requestIdentifier : EString, lineIdentification : EString)
reRouteCall(requestIdentifier : EString, lineIdentification : EString, targetNumber : EString, targetInfos : EMap
initiateTransfer(requestIdentifier : EString, lineIdentification : EString, target : EString, eventId : EString)
connectCalls(requestIdentifier : EString, primaryLineIdentification : EString, secondaryLineIdentification : EString)
sendData(requestIdentifier : EString, lineIdentification : EString, data : EByteArray)

Fig. 6. Extract of the Services of the Callmanager Application

Local and global processes are modeled as behavior diagrams, i.e., state machines, activity diagrams or sequence diagrams. In our experiments we have mainly modeled global workflows by activity diagrams, and local workflows by activity diagrams and state machines.

Tests are modeled as activity diagrams or sequence diagrams. Test sequences or high level test suites are modeled as activity diagrams. In Fig. 7 a test is depicted.



RouteCall.test

Test ID	initiateCall_num...	routeCall_requ...	routeCall_lineId...	routeCall_targe...	routeCall_targe...	hangupCall_req...	hangupCall_line...	Assert
1	callNum	reqId	lineID	targetNum	targetInfo	reqId	lineID	TRUE
2	callNum	reqId	lineID	targetNum	targetInfo	reqId	lineID	FALSE
3	callNum	reqId	lineID	targetNum	targetInfo	reqId	lineID	TRUE
4	callNum	reqId	lineID	targetNum	targetInfo	reqId	lineID	FALSE

Fig. 7. Test story for the Callmanager Application

In the upper part of the test depicted in Fig. 7, the test *RouteCall* for routing a call is depicted. After the car service (*Vsim*) initiated a call, the Callmanager routes the call and terminates the call. The results of these calls are triggered on the test controller. Intermediate assertions check whether the result provided by the trigger equals to the expected one. Each test may have test data which is defined for the test story *RouteCall* in the table *RouteCall.test* depicted in the lower part of Fig. 7. The test stories and their corresponding data files are executed as test sequence elements depicted in Fig. 10 which defines an additional arbitration to define a global verdict specifying when all tests of a story have passed.

B. Model Validation

As mentioned in Section II-A model validation comprises checks for consistency and completeness or coverage. The validation is static, i.e., it is based on rules defined on metamodel elements without program execution. Due to the dynamic evaluation of test models provided by early test execution, static validation provides the most efficient and effective validation result: dynamic validation is provided by early test execution, and the modeling effort and computational complexity for static analysis is not as high as for verification techniques like model checking and constraint solving. The focus on static analysis is also supported by empirical research because in [16] it is shown that incompleteness and inconsistencies in UML models are already detected with OCL–based static analysis and that formal methods such as model checking or constraint solving are not required.

First each model is validated as a separate artifact (intra–model aspects) and then in relation to other models (inter–model aspects). In total we have 3 single models (requirements, system and test model) and 3 pairs of models (Table I). The inter–model relationship between the requirements, system and test model is implemented by tagged values or associations. For instance, the relationship between a requirement (in the requirements model) and a test (in the test model) is implemented by a tagged value of the requirement and the test model element referencing each other. In Fig. 5 the requirement `Req_1` is associated to the test `RouteCall`.

As mentioned before in TTS we investigate three types of validation rules: consistency, completeness, and coverage rules.

Consistency checks assure that there is no conflicting information in models. Consistency of a model enables error–free transformation from the model to another model or to the source code.

Completeness checks assure that one artifact is complete, i.e., contains all essential information. Completeness of the system model is crucial for the TTS framework and determines whether transformations from the system model to the test model can be applied.

Coverage can be considered as a variant of inter–model completeness where one model is the test model. This aspect is very important in context of testing and is used to check to what extend the test model covers the system model and implicitly the system. We adopted a series of coverage criteria from testing [10] and model–driven testing [11] to fit into TTS.

Additionally, we distinguish between criteria and metrics. *Criteria* provide only a boolean result: true if a model fulfills a given criterion, false otherwise. They provide a warning mechanism implemented, i.e., the severity levels information, warning, and error, to inform modelers about an incorrectness in a model. *Metrics* have a numeric result. Typically they provide information to what extend a corresponding criterion is fulfilled, thus they are fractions ranging from `0` to `100%`. They can be used for the evaluation of models in a similar way as for the evaluation of source code in [17]. Metrics are well suited for summarizing particular aspects of models and for detecting outliers in large models. They scale up and aggregate many details of models.

To address particular quality criteria we start from informal descriptions to obtain metrics at the final stage. First we define a criteria in natural language. We adapt it to the context of the TTS metamodel. Next, we express it either as a constraint over a model or as a boolean query. Finally, we construct a numeric metric for it. As a formalization language we apply the Object Constraint Language (OCL) [18]. Table I provides and overview of specified criteria and metrics as a proof of concept. Before we give more details about example criteria and metrics, we will describe their development process.

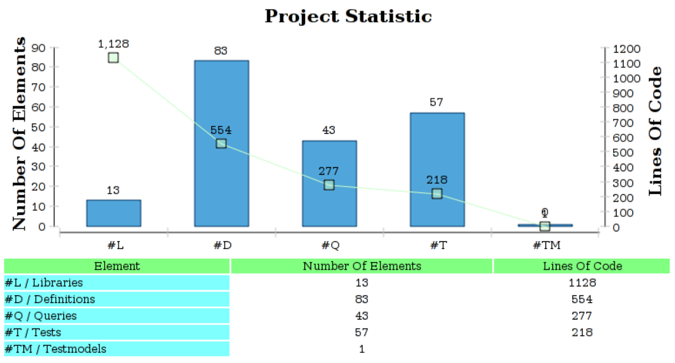
TABLE I
OVERVIEW OF VALIDATION ASPECTS

		consistency		completeness/coverage		
type	models	metrics	criteria	metrics	criteria	
intra	Requirements	×	√	√	√	complet.
	System	×	√	√	√	
	Test	×	√	√	√	
inter	Req–Syst	×	√	√	√	cov.
	Req–Test	×	√	√	√	
	Syst–Test	×	√	√	√	

To specify criteria and metrics we follow the model analysis and OCL library development process (Fig. 8). The upper swimlane corresponds to the manual model analysis, the lower swimlane to the library development process. First, a common requirement for model analysis and library development is specified. A quality aspect is selected, e.g., a completeness criterion defining that each requirement should have a unique name. For this aspect OCL definitions and queries are specified in the development step. The next step is quality assessment, where the results of the manual and automatic analysis are cross–checked. For the selected aspect, manual inspection is used to determine the result of this aspect for the model. Simultaneously, appropriate queries are evaluated on the model. If the results of the model inspection and the query evaluation differ, the reason has to be determined and either the OCL definition specification or manual inspection needs to be repeated. The manual inspection of the model is conducted as long as correctness of a query achieves a defined confidence level. Afterwards the query can be used for automatic model analysis.

If the results are equal, the last step, i.e., quality assurance, can be executed. The aim of this step is to assure semantic correctness of OCL expressions in the future development of the library. For this purpose OCL unit tests [19] are specified and evaluated regularly. In the test evaluation step, OCL unit tests with corresponding OCL test models are required. The OCL test model is an instance of the requirements, system and test metamodel. Note that the OCL test model is not the same as a test model in TTS but a model instance of the considered metamodel for the OCL expression under test. This instance is used as test data for OCL unit tests to assess the desired semantics of definitions. OCL unit tests are similar to JUnit [20] tests applied to assess the semantic correctness of source code.

In Fig. 9 we show the size of the OCL project developed for the TTS approach. We specified `83` definitions used in `43` queries and split into `13` libraries. The number of OCL expressions is higher than the total number of the criteria (26) and metrics (10) as it was necessary to define helper methods. The helper expressions can be used in the specification of further criteria/metrics and make their development less time consuming. To assure correctness of OCL expressions we wrote `57` OCL unit tests [19] and evaluated them over one test model.



Element	Number Of Elements	Lines Of Code
#L / Libraries	13	1128
#D / Definitions	83	554
#Q / Queries	43	277
#T / Tests	57	218
#TM / Testmodels	1	

Fig. 9. OCL Project Statistics

In the next paragraphs we show the OCL formalization of selected consistency, completeness and coverage rules.

a) Consistency: We have defined several consistency criteria to assure that there is no conflicting information in the models.

The criterion `isServiceUnique` in Listing 1 guarantees the uniqueness of service definitions in a system model.

The criterion checks for a specific service whether its name identifier is unique. An additional query `allServicesUnique` checks whether all services in the system model are unique.

The criterion `isAssertionConsistent` in Listing 2 guarantees the consistency of an assertion.

If the definition of pass equals the definition of fail then the definition is inconsistent because it is not possible to compute a meaningful verdict. The definition

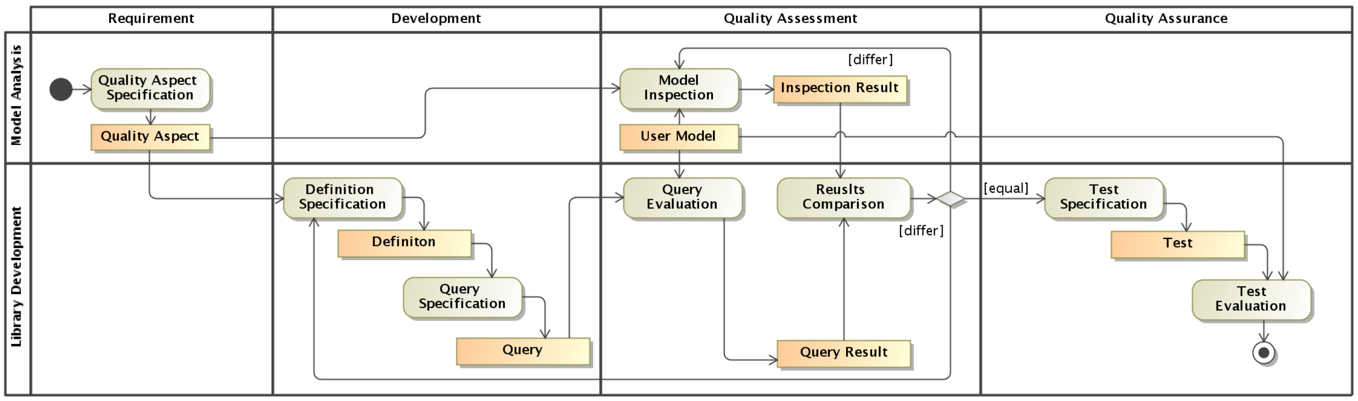


Fig. 8. The model analysis and library development process (from [21]).

```
context TTS::Service
  def isServiceUnique:
    isServiceUnique() : Boolean =
      TTS::Service.allInstances()
        ->select(r|r.base_Class.name=base_Class.name)
        ->size() = 1

context Model
  def allServicesUnique:
    allServicesUnique() : Boolean =
      TTS::Service.allInstances().
        isServiceUnique()->forAll(s | s = true)
```

Listing 1. OCL definition for unique service

```
context TTS::Assertion
  def isAssertionConsistent:
    isAssertionConsistent() : Boolean =
      not (pass = fail)

context Model
  def allAssertionsConsistent:
    allAssertionsConsistent() : Boolean =
      TTS::Assertion.allInstances().
        isAssertionConsistent()->forAll(a | a = true)
```

Listing 2. OCL definition for consistent assertion

`allAssertionsConsistent` checks the criterion `isAssertionConsistent` for all assertions in a specific model.

b) Completeness: We have defined several completeness criteria which guarantee that artifacts contain all essential information. In our respect the completeness of test models is very important because otherwise no meaningful test code can be generated. In Listing 3 a criterion to check the completeness of a test story is defined.

A test story is complete if it has at least one assertion to compute a verdict and a service call to interact with the system. The definition `hasAssertion` checks whether a test story has at least one assertion and the definition

```
context TTS::Teststory
  def hasAssertion:
   hasAssertion() : Boolean =
     base_Activity.allOwnedElements()->select(o |
       o.profileIsTypeOf('Assertion'))->size() > 0

context TTS::Teststory
  def hasServicecall:
    hasServicecall() : Boolean =
      base_Activity.allOwnedElements()
        ->select(o|o.profileIsTypeOf('Servicecall'))
        ->size() > 0

context TTS::Teststory
  def isTeststoryComplete:
    isTeststoryComplete() : Boolean =
          hasAssertion() and hasServicecall()
```

Listing 3. OCL for completeness of test stories

`hasServicecall` checks whether a test story has at least one service call. Finally, the definition `isTestComplete` checks whether a test story has at least one assertion and one service call by invoking `hasAssertion` and `hasServicecall`.

c) Coverage: We have developed several coverage criteria and metrics based on the coverage criteria from testing [10] and model–driven testing [11]. In the next paragraphs we demonstrate how coverage criteria and metrics are implemented in our approach.

As first example we consider the *all requirements coverage* (ARC) defined in [11]. In our context this represents the coverage between the requirement and the test model. In [11] ARC says only that *all requirements are covered*. It is refined in the context of the TTS metamodel to *each requirement has a test story*. “Requirement” in this regard is a model element that applies the `TTS::Requirement` stereotype and “has a test story” means that it has either an action, or an activity defined, i.e., at least one tagged value referring to a test story is set. This informal definition results in the OCL definitions

```
context TTS::Requirement
  def hasTestStory:
    hasTestStory() : Boolean =
      (not self.action.oclIsUndefined()) or
      (not self.activity.oclIsUndefined()) or
      (not self.class.oclIsUndefined())

context Model
  def allRequirementsCoverage:
    allRequirementsCoverage() : Boolean =
      TTS::Requirement.allInstances()
        .hasTestStory()->forAll(e | e = true)
```

Listing 4. OCL definitions for ARC

```
public queries
context Model
  query qRequirementCoverageMetric:
    severity 2
    let result : Integer =
      requirementsWithTestStories()->size()
    message result +' from '
      + TTS::Requirement.allInstances()->size()
      + ' requirements have a test story.( '
      + (result/(TTS::Requirement.allInstances()
      ->size().max(1))*100).round()
      +'%)'
    endmessage
endqueries
```

Listing 5. OCL Metrics for ARC

presented in Listing 4.

In the listing both definitions return a value of the type `Boolean` which provides decision support but is not very informative. To gain more information from the model we have defined informative queries that compute a metric and are based on definitions. The query `qRequirementCoverageMetric` in Listing 5 extracts the total number of requirements and the number of all requirements which have a test story assigned. The number of all tested requirements is computed by the query `requirementsWithTests` which is based on the query `hasTestStory` defined in Listing 4. The metrics then computes the ratio between the total number of requirements and the number of tested requirements. It informs to which degree the coverage criterion is satisfied. From the metric we obtain a value between `0` and `1`. Alternatively, the ratio can be expressed as percentage.

For the callmanager example model (see Fig. 5) we obtained the following result: `6 from 6 requirements have a test story.(100%)`. A coverage metrics assigns a number to a coverage criterion measuring the degree of coverage. The coverage metrics `qRequirementCoverageMetric` in Listing 5 measures the requirements coverage by the ratio of requirements with an assigned test story to the overall number of requirements.

Another important coverage criterion is the all services coverage criterion (ASC) which means that *from every service at least one operation is invoked in at least one test story*. This coverage criterion is defined between the system and the test model. This informal definition results in the OCL definitions shown in Listing 6.

```
  def isServiceInvokedByCall:
    isServiceInvokedByCall() : Boolean =
      self.provides -> exists(i
        | i.getAllOperations()
          -> exists(o
          | TTS::Servicecall.allInstances()
            -> collect (s | s.operation)
            -> includes(o)
          )
      )

   def isServiceInvokedByTrigger:
         /* the same as isServiceInvokedByCall
            but with TTS::Trigger.allInstances() */

   def isServiceInvoked:
     isServiceInvoked() : Boolean =
       self.isServiceInvokedByCall() or
       self.isServiceInvokedByTrigger()

context Model
  def allServicesCoverage:
    allServicesCoverage() : Boolean =
      TTS::Service.allInstances()
      .isServiceInvoked()->forAll(e | e = true)
```

Listing 6. OCL Definitions for ASC

For additional information we have defined a metrics based on the definition `allServicesCoverage` which computes the number of services covered by a set of tests. The metrics prints the number and the ratio of all covered services. For the callmanager example model (see Fig. 6) we obtained the following result: `3 from 3 services have a test story regarding All Services Coverage.(100%)`.

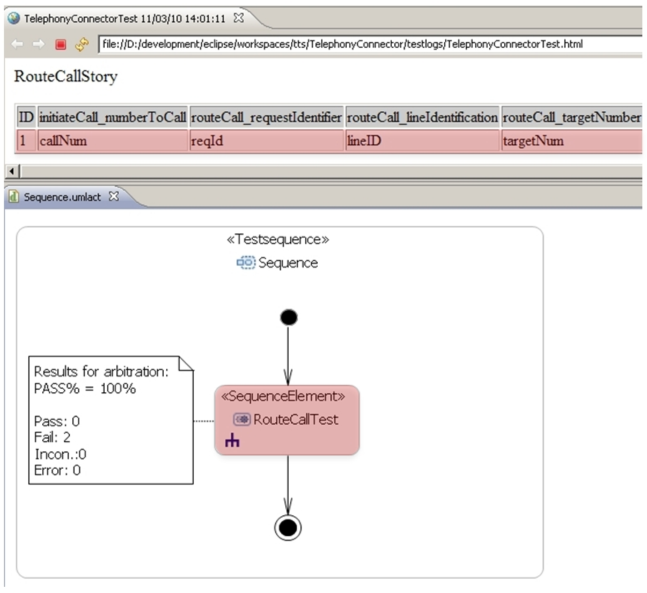
C. System Validation

After the test model quality has been validated, the test execution phase starts. Based on RMI adapters, which have been provided by the system developer, test code in Java has been generated from the test model and afterward executed by the test controller. The evaluation of test run be assigning verdicts is based on the explicitly defined assertions, the implicitly defined preconditions and postconditions of the service calls, and errors originated in the test environment. Details on these components are explained in the next section.

Finally, the test results are evaluated. In Fig. 10 the evaluation result is depicted by coloring test cases in the test data table, by coloring test sequence elements, and by annotating test sequence elements.

The evaluation is based on arbitrations which define criteria for a sequence of test results in an OCL–like language defined in [22].

TTS allowed us to perform system wide tests on the application. In a first step, the system model was developed,



TelephonyConnectorTest 11/03/10 14:01:11

file://D:/development/eclipse/workspaces/tts/TelephonyConnector/testlogs/TelephonyConnectorTest.html

RouteCallStory

ID	initiateCall_numberToCall	routeCall_requestIdentifier	routeCall_lineIdentification	routeCall_targetNumber
1	callNum	reqId	lineID	targetNum

Sequence.umlact

«Testsequence»
Sequence

Results for arbitration:
PASS% = 100%

Pass: 0
Fail: 2
Incon.:0
Error: 0

«SequenceElement»
RouteCallTest

Fig. 10. Test Result of a Test Run

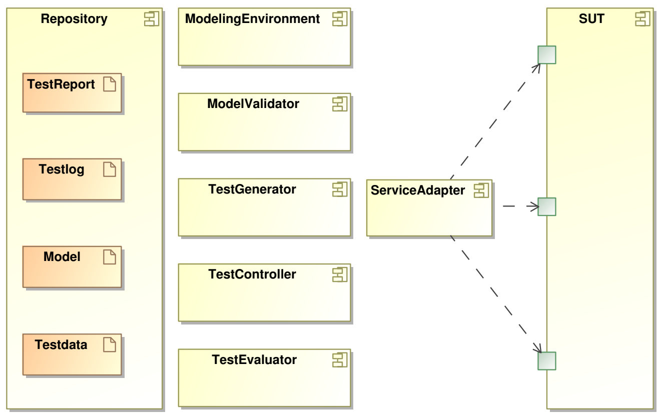


Fig. 11. Components of the TTS Tool

needed for the modeling of the various test cases in a next step. After generating the test code and preparing the test data, finally the tests had been executed against the SUT.

Among the lessons learned during this case study, the three most important are:

1) providing complex test data in a proper way to the testing framework,
2) the communication with asynchronous message exchange patterns, and
3) developing an assertion language capable of iterating complex object structures for test evaluation.

IV. Tool Implementation

Our methodology is tool–driven and based on Telling TestStories providing the modeling and testing environment (presented in Section IV-A) and SQUAM providing the validation environment (presented in Section IV-B).

A. *TTS*

In this section, we describe the TTS tool implementation that has been applied on the case study in the previous section and developed in an industrial cooperation within the Telling TestStories project [23]

Designed as a set of Eclipse [24] plug-ins, the tool consists of various components setting up the whole environment, to keep a high level of modularity. The architecture of the TTS tool is shown in Fig. 11.

The main components correspond to the activities of our testing methodology depicted in Fig. 2 and are as follows:

The **Modeling Environment** is used for designing the requirements model, the system model and the test model. It processes the workflow activities `Requirements Definition`, `System Model Design`, `Test Model Design`, and `Data Pool Definition`.

The **Model Evaluator** is based on the SQUAM framework (Section IV-B) and uses OCL as constraint language. It processes the workflow activity `Validation, Coverage, Transformation`.

The **Test Code Generator** generates executable Java code from the test model. It processes the workflow activity `Test Code Generation`.

The **Service Adapters** are used by the test controller to invoke services on the system under test (SUT). They can be created manually or generated automatically depending on the service technology. Adapters correspond to the workflow activity `Adapter Implementation/Generation`.

The **Test Controller** executes the tests against the SUT. It processes the workflow activity `Test Execution`.

The **Test Evaluator** generates test reports and visualizes test results within the models. It corresponds to the workflow activity `Test Analysis`.

In its modularity, the tool is only bound to the Eclipse framework. The various components can be exchanged by more custom triggered extensions as the tool follows established practices (test data modeling in XML, test case modeling in XMI). In the following, the components will be outlined in more detail.

1) Modeling Environment: The requirements, system and test models are denoted as UML models and stored in the XMI format. Our language is defined via an UML profile implementing the metamodel of Section II-C. Therefore stereotypes are used to label model elements and tagged values are used to define additional attributes. The modeling of requirements and the system model is then straightforward with our customized editor.

The modeling of tests is more complicated because test data has to be considered. In a first step, test stories are described using activity diagrams containing control flow elements, service invocations and assertions. Not only test stories, but also test sequences are described using a UML activity diagram, yet contained in another package of the test model. Valid test models which can be checked via the model

```
public interface IAdapter {
 public Object
   invoke(String servicename, Object... arguments);
}
```

Listing 7. Service Adapter Interface

evaluator are the input for the test code generator.

After describing a test, the tool allows for generating test data tables for each test case as a second step of the test modeling process. These test tables follow the approach of the FIT framework [7], which allows for assigning concrete test data (in our case concrete object IDs) to every input parameter and free variable of assertions occurring in a test story. The various test data objects referred in the test tables are provided from a data context, holding instances of the concrete data objects needed for system testing. By making use of the *Inversion of Control* (IoC) container of the Spring Framework [25], TTS allows for modeling complex data objects and not only primitive types.

Hence, these test data tables allow for describing and modeling the execution flow of test cases in a very fine grained and deterministic way, by manually manipulating the object contents. Additionally, it is possible to trace down the erroneous execution of test cases to the test data level by making use of the IoC's user-defined object IDs.

2) Model Evaluator: The model evaluator is based on the SQUAM framework, which is presented in Section IV-B.

3) Service Adapters: The communication of the test controller with the SUT is encapsulated inside generated or manually implemented adapters, tailored to the concrete SUT. By encapsulating service invocations and data mapping, the modeling of test stories and the generation of test code is eased (see sections IV-A1 and IV-A4) because it can be abstracted from technical details.

Currently the tool supports the automatic generation of adapters for services providing a proper WSDL description of their operation interfaces. The WSDL description itself can be generated from a proper system model.

In Listing 7 the root interface for every adapter is denoted. A tailored adapter only has to implement one single method `invoke` to be ready–for–use in the tool environment during the testing process. It support the manual or automatic development of an adapter for different technologies such as web services, RMI or CORBA as much as possible.

4) Test Code Generation: The test code generation constitutes one of the core components of the tool environment. The code generator is implemented based on oAW [26] which is a framework for domain modeling and model driven development, allowing to realize model–to–text transformations as needed in our case.

The test code generator enables to generate executable Java code out of the modeled test stories (activity diagrams). In its implementation the generator visits the contained model elements of each distinct activity diagram in the order of the modeled execution flow and produces equivalent source code (in our case Java). The generated test code is composed of predefined code templates, called and evaluated during the visitation of the various model elements. For pre– and postconditions aspects in the notation of AspectJ [27] are generated to be evaluated as aspects on service calls during the test execution.

The above mentioned evaluation of the code templates focuses on the processing of the applied stereotypes (see section IV-A1) defined as a part of the tool environment. As already mentioned earlier, those stereotypes define element specific tagged values, containing the required information for proper test case generation. To assure that the test model meets the requirements posed by the test code generator, prior to test code generation, the test model is checked for consistency as explained in section IV-A2. The consistency rules assure that on the one hand side, the test model only calls services defined by the SUT and uses data types processable by the SUT. On the other side, those OCL rules are used to check, that the test model is valid, in a sense, that the tagged values of the test specific stereotypes are set.

5) Test Controller: The test controller processes the test code which is executed with concrete test data and logged afterwards. Additionally, the engine also provides a communication interface to the SUT to realize asynchronous service communication, i.e., the execution of a workflow in the SUT whose completion is indicated by a callback method onto the invoking client.

Considered from an architectural point of view, the test controller itself again consists out of various components:
Data Management Provides the test data objects referenced in the various test data tables (see section IV-A1). Again, the IoC container of Spring is used to provide the test data objects to the test engine. Inside this container the objects are retrieved by their unique object ID used in the test data tables.

Event Handling This component is used by the whole tool environment to process events thrown during test execution, i.e., a *VerdictEvent* to indicate the evaluation of an assertion during test execution. Additionally, this component generates tables as in the FIT framework [7], illustrating the successful or erroneous execution of a set of test data.

Assertion Evaluation Inferring the outcome of a test run is provided by this component. Yet, as in contradiction to JUnit and the like, Telling TestStories allows complex test data to be used, and hence, also this component allows for iterating through complex object structures for test evaluation.

Timing Dealing with asynchronous services requires ensuring that timeouts are met for service responses. This is encapsulated inside this component by ensuring that responses to specific service invocations receive the test controller within a pre–specified duration. Responses are assigned to corresponding service invocations by their method signatures.

```
context Model query checkIsValidTestModel:
let result:Boolean =
 if Package.allInstances()->
  any(o|o.profileIsTypeOf('Test')).oclIsUndefined()
 then false
 else
  Package.allInstances()->
  any(o|o.profileIsTypeOf('Test')).isValidTestModel()
 endif
 message result endmessage
```

Listing 8. OCL Query for Testmodel Validity

In the execution phase, the test controller passes through three states. In an initial state, the test workflow is parsed and according to its contents, *test tasks* are generated encapsulating the modeled test cases. After completion of creating the tasks the engine enters its second, main state in which the tests are executed against the SUT. In a third and final state, the engine generates the above mentioned result table containing the test outcomes, after all events have been processed.

6) Test Evaluator: The test evaluator is responsible for evaluating the results of a test run. As in the FIT framework, our test evaluator colors test case lines in test tables green, red or yellow depending on the test result. If a failure can be assigned to a specific model element, our tool is able to color it in the activity diagram. We have also integrated high–level test reporting based on BIRT [28] which generates different types of graphical test summary reports.

B. SQUAM

In this section, we describe the SQUAM tool implementation that has been used for the model evaluation within the TTS framework. SQUAM (Systematic QUality Assessment of Models) is an integrated framework for UML/OCL–based model development and OCL–based quality analysis.

It provides support for consistency and coverage checks in OCL, and supports the definition and generation of high–quality system and test models. Coverage checks guarantee that the test models are complete with respect to the system model and the requirements model. Additionally, coverage checks are useful exit criteria for test generation. Consistency checks guarantee model validity which is a prerequisite for test code generation. Therefore the test code generator uses the model evaluator to check test model validity prior to test code generation. In Listing 8 a sample top–level query for assuring test model validity is denoted.

The SQUAM tool has a plug–in architecture incorporating in–house developed and existing open source solutions. There are two editions of SQUAM, a community and a professional edition. The community edition is integrated into the TTS framework, whereas the professional one provides features that can be used to obtain an integrated and user–friendly OCL development process for criteria and metrics within the TTS framework. Below we describe the part of the SQUAM framework integrated into the TTS framework.

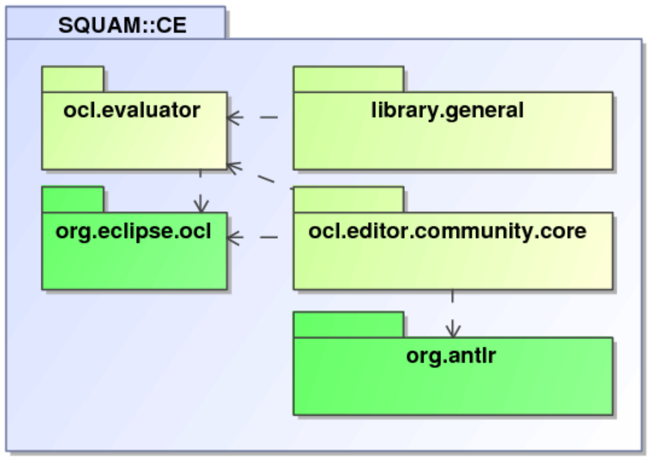


Fig. 12. Architecture of the SQUAM Community Edition Tool

The community edition is the core of the framework. It provides the basic features for writing and editing OCL expressions supporting definitions, (running) queries, (running) unit tests and their documentation (OCLDoc). It consists of several plug–ins (see Fig. 12): 3 in–house plug–ins: core, general library, and OCL evaluator and 2 third plug–ins: Eclipse OCL evaluator and Antlr.

The **core** (*ocl.editor.community.core*) forms the backbone of the SQUAM application. It contains the basic functionality like two–step parsing of libraries: using **Antrl** (*org.antlr*) for pre–parsing our OCL extensions and the **OCL evaluator** (*ocl.evaluator*) with **Eclipse OCL evaluator** (*org.eclipse.ocl*) for parsing standard OCL. Moreover, the core provides basic features for editing OCL expressions (like syntax highlighting, code completion or code formatting), managing UML/ECORE/XML models (and meta–models) and UML profiles (loading, removing and creation of qualified names).

The **general library** (*library.general*) defines an abstract OCL library as an Eclipse extension point to access OCL definitions and queries. The basic principle of the general library is to give third party plug–ins the ability to retrieve all definitions and queries for further processing.

There are two integration points of the SQUAM framework into the TTS framework, one for the evaluation of interactive checks and one for the evaluation of automatic checks. For the interactive checks, the validation view from the core is used, and for the automatic checks, the OCL evaluator is used.

V. Related Work

In this section we present tool–based testing frameworks, model validation techniques, and coverage criteria related to TTS.

A. Tool–based Testing Frameworks

Model–based testing approaches always have a methodological and a tool aspect [11]. There are already some industrial tools available [29]. TDE/UML [30], a tool suite for test generation from UML behavioral models, is related to our approach but focuses on GUI–based systems whereas TTS focuses on service–centric systems.

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.

Although test sheets [31] define a fully tabular approach, support for model–driven testing is missing there.

In [32] a model–driven system testing approach and a tool implementation that enables test engineers to graphically design complex test cases based on METAFrame Technologies' Application Building Center [33] has been defined. The approach is similar to TTS but not based on UML and its generic profiling mechanism.

The PLASTIC framework [34] provides a collection of tools for online and offline testing both functional and non–functional properties of service–oriented applications. Some of the tools are model–based but the tools are not integrated and do not follow a model–driven testing approach as in the TTS tool implementation.

B. Model Validation

Validating consistency of UML models has received greater attention by researchers in recent years [35], but completeness has also been addressed [16]. Even the interplay between consistency and completeness has been investigated [36].

Only a small number of approaches such as [37] consider consistency and completeness of test models, i.e., behavioral descriptions of operations. But consistency and completeness between requirements, system and test models as in TTS is not considered.

C. Coverage Criteria

Comprehensive collections of coverage criteria are defined in [11] and [10]. In [10] a coverage–driven approach to software testing is discussed. There are four different types of testing and corresponding coverage criteria distinguished, i.e. graph coverage, logical expression coverage, input space partitioning, and syntax–based coverage. In our integrated approach, we use artifacts of our metamodel as sources for test coverage. Behaviors provide graphs, decisions provide logical expressions and types or services provide partitions of the input space. We do not have explicit grammar definitions and therefore syntax–based coverage is not relevant in our approach.

A collection of structural UML–based coverage criteria for class diagrams, sequence diagrams, communication diagrams, state machines, activity diagrams and use case diagrams is provided in [38]. It includes coverage criteria for state machines and activity diagrams from [39] where test generation from UML specifications is discussed. Our approach provides specific model–based coverage criteria for service–centric systems supporting the manual or automatic test definition.

VI. Conclusions and Future Work

In this article, we have outlined the model–driven and tabular system testing methodology Telling TestStories and its tool implementation. TTS is based on traceable requirements, system, and test models which are validated even before test code is generated and executed. The TTS tool used for model development and system validation, plus the SQUAM tool used for model validation are explained by a case study from the telecommunication domain. The TTS tool and the SQUAM tool consist of a set of Eclipse plug–ins [24] and are integrated to implement the TTS methodology.

Prior to the use of TTS the system test design has been done ad–hoc in an unsystematic way mainly by testers themselves. TTS allows for designing tests in an effective way because its intuitive graphical and tabular notation supports the design of tests that can be validated by consistency, completeness and coverage checks. The TTS methodology naturally integrates domain experts and customers into the process of formal and executable test design. The integration of domain experts and customers may be helpful to reveal specific test scenarios that would not have been detected otherwise. TTS is also efficient because the tests can be defined on an abstract visual level with tool support. After the initial effort of system model design the advantages of TTS can be applied. Our implementation of the methodology has shown that the checks provide additional support for the validation of the requirements, system, and test model, but also raises the failure detection rate due to the higher test quality. In the model of the callmanager case study we found inconsistencies and incompletenesses that have been detected with our criteria and removed afterwards. With our validation checks the effectiveness and efficiency of the approach has been improved because the quality of test models is higher and failures are detected earlier. The coverage criteria and metrics provide useful information to all stakeholders whether additional tests have to be defined manually or not. Our validation checks are defined statically on the metamodel and do not support the dynamic simulation of a model. But due to our experience this is replaced by early test execution in iterative software testing and therefore not a severe restriction compared to dynamic approaches like model–checking which need additional modeling effort and more knowledge in formal modeling. Additionally, the information provided in this process supports the system engineers who are not experts in test design when defining tests.

Based on research results and user feedback we have planned further extensions to our methodology, the tool, and its application.

In the case study at hand functional and performance requirements are considered. But TTS will also be applied to test other non–functional requirements, e.g., security requirements.

We have already tested positive security requirements with TTS [40], but testing negative security requirements with TTS has not been considered yet. So far coverage criteria in TTS only check the quality of the test model as adequacy criteria but are not applied for the generation of test cases as selection criteria. Our OCL–based coverage criteria can be applied to integrate selective test generation into the TTS methodology.

The TTS tool is already quite mature and on its way to a practically usable open–source tool for model–driven system testing of service–centric systems [41]. For this step the usability of the tool and the interoperability to integrate TTS with other testing tools has to be improved. For instance, the usability of modeling tests can be improved by an additional textual representation of test models which is synchronized with the graphical representation, to support fast text–based editing of test models. TTS is suitable for arbitrary service–centric systems and therefore many application domains are arising. Applications to an industrial service–centric system from the health care domain and to the upcoming paradigm of cloud computing which can also be considered as a specific service–centric system are planned.

Acknowledgements

This work was largely performed as part of the project MATE under support of the FWF (project number P17380), as part of the project QE LaB – Living Models for Open Systems under support of the FFG (project number FFG 822740), and as part of the SecureChange project under support of the EU (project number ICT-FET-231101).

References

[1] G. Engels, A. Hess, B. Humm, O. Juwig, M. Lohmann, J.-P. Richter, M. Voß, and J. Willkomm, "A Method for Engineering a True Service-Oriented Architecture," 2008, ICEIS 2008.

[2] OASIS Standard, "Web Services Business Process Execution Language Version 2.0 - OASIS Standard," 2007, http://docs.oasis-open.org/wsbpel/2.0/.

[3] W3C, "Web Services Choreography Description Language Version 1.0," 2005, http://www.w3.org/TR/ws-cdl-10/.

[4] G. Canfora and M. D. Penta, "Service-oriented architectures testing: A survey," in *ISSSE*, ser. Lecture Notes in Computer Science, A. D. Lucia and F. Ferrucci, Eds., vol. 5413. Springer, 2008.

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

[6] OMG, *MDA Guide Version 1.0.1*, www.omg.org/docs/omg/03-06-01.pdf [accessed: June 30, 2011].

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

[8] M. Felderer, P. Zech, F. Fiedler, and R. Breu, "A Tool-based methodology for System Testing of Service-oriented systems," in *The Second International Conference on Advances in System Testing and Validation Lifecycle (VALID 2010)*, 2010, pp. 108–113.

[9] M. Felderer, B. Agreiter, R. Breu, and A. Armenteros, "Security Testing By Telling TestStories," 2010, Modellierung 2010.

[10] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2008.

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

[12] M. Felderer, F. Fiedler, P. Zech, and R. Breu, "Flexible Test Code Generation for Service Oriented Systems," 2009, QSIC'2009.

[13] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp, "Concepts for Model–Based Requirements Testing of Service Oriented Systems," 2009, IASTED SE'2009.

[14] ISO/IEC, *Information technology – open systems interconnection – conformance testing methodology and framework*, 1994, international ISO/IEC multi–part standard No. 9646.

[15] OMG, *OMG Systems Modeling Language*, 2007, http://www.omg.org/docs/formal/2008-11-01.pdf.

[16] Lange, C. F. J. and Chaudron, M. R. V. , "An empirical assessment of completeness in UML designs," in *In Proc. of the 8th International Conference on Empirical Assessment in Software Engineering (EASE'04)*, 2004.

[17] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[18] OMG, *Object Constraint Language Version 2.0*, 2006, http://www.omg.org/docs/formal/06-05-01.pdf [accessed: June 30, 2011].

[19] J. Chimiak-Opoka, "OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language," in *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, Colorado, USA, October 4-9, 2009, Proceedings. LNCS 5795*, A. Schuerr and B. Selic, Eds. Springer, 2009, pp. 665–669.

[20] Object Mentor, "Junit," 09 2009, http://www.junit.org/.

[21] J. Chimiak-Opoka, B. Agreiter, and R. Breu, "Bringing Models into Practice: Design and Usage of UML Profiles and OCL Queries in a showcase," in *Proc. of the 16th Int. Conf. on Information and Software Technologies,IT'2010*, 2010, pp. 265–273.

[22] J. Chimiak-Opoka, S. Loew, M. Felderer, R. Breu, F. Fiedler, F. Schupp, and M. Breu, "Generic Arbitrations for Test Reporting," 2009, IASTED SE'2009.

[23] "Telling TestStories," http://teststories.info [accessed: June 30, 2011].

[24] "Eclipse," http://www.eclipse.org/ [accessed: June 30, 2011].

[25] "Spring," http://www.springsource.org/ [accessed: June 30, 2011].

[26] "oAW," http://www.openarchitectureware.org/ [accessed: June 30, 2011].

[27] "AspectJ," http://www.eclipse.org/aspectj/ [accessed: June 30, 2011].

[28] "BIRT," http://www.eclipse.org/birt/ [accessed: June 30, 2011].

[29] H. Götz, M. Nickolaus, T. Roßner, and K. Salomon, *iX Studie Modellbasiertes Testen*. Heise Zeitschriften Verlag, 2009.

[30] J. Hartmann, M. Vieira, H. Foster, and A. Ruder, "A UML–based approach to system testing," *ISSE*, vol. 1, no. 1, 2005.

[31] C. Atkinson, D. Brenner, G. Falcone, and M. Juhasz, "Specifying High-Assurance Services," *Computer*, vol. 41, 2008.

[32] T. Margaria and B. Steffen, "Lightweight coarse-grained coordination: a scalable system-level approach," *STTT*, vol. 5, no. 2-3, 2004.

[33] B. Steffen and T. Margaria, "Metaframe in practice: Design of intelligent network services," in *Correct System Design, Recent Insight and Advances*. London, UK: Springer-Verlag, 1999, pp. 390–415.

[34] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "The plastic framework and tools for testing service-oriented applications," in *ISSSE*, ser. Lecture Notes in Computer Science, A. D. Lucia and F. Ferrucci, Eds., vol. 5413. Springer, 2008, pp. 106–139.

[35] M. Elaasar and L. Briand, "An Overview of UML Consistency Management," Department of Systems and Computer Engineering, University of Ottawa, Tech. Rep. SCE-04-18, 2004.

[36] D. Zowghi and V. Gervasi, "On the interplay between consistency, completeness, and correctness in requirements evolution," *Information and Software Technology*, vol. 45, no. 14, pp. 993 – 1009, 2003.

[37] Amit Paradkar and Tim Klinger, "Automated Consistency and Completeness Checking of Testing Models for Interactive Systems," *Computer Software and Applications Conference, Annual International*, vol. 1, pp. 342–348, 2004.

[38] J. McQuillan and J. Power, "A Survey of UML-Based Coverage Criteria for Software Testing," National University of Ireland, Maynooth, Tech. Rep., 2005.

[39] A. J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," in *UML*, R. B. France and B. Rumpe, Eds. Springer, 1999, pp. 416–429.

[40] M. Felderer, B. Agreiter, and R. Breu, "Security Testing by Telling TestStories," in *Modellierung 2010*, 2010.

[41] M. Felderer and P. Zech, "Telling teststories – a tool for tabular and model-driven system testing," *Testing Experience*, vol. 12, 2010.

[42] A. D. Lucia and F. Ferrucci, Eds., *Software Engineering, International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, ser. Lecture Notes in Computer Science, vol. 5413. Springer, 2009.