

Model-driven System Testing of a Telephony Connector with Telling Test Stories

Michael Felderer University of Innsbruck

Frank Fiedler SoftMethod GmbH

Philipp Zech University of Innsbruck

Ruth Breu University of Innsbruck

Abstract

In this paper we report on the applicability of a model-driven system testing approach for cooperative systems, called Telling Test Stories, to an industrial software system.

We explain our testing methodology and testing framework based on a concrete test scenario for our industrial Telephony Connector system. Additionally we explain the handling of service adaptation and asynchronous calls which are of special interest in the context of the Telephony Connector.

1 Introduction

The ever growing complexity and cooperation of IT systems demands new technologies for testing such scenarios. Testing cooperative systems, i.e. distributed systems under participation of different actors, has some specifics such as the integration and coordination of heterogeneous components, the complexity in the definition of test data and test runs, and the unavailability resp. the adaptation of the implementation of some components.

Because these features have not been addressed adequately in existing system development and testing methodologies, Telling Test Stories [Fel09, Fel08] provides a novel methodology and a prototypic tool implementation for *systematic model-driven system testing of complex cooperative systems*.

According to [Zan05] model-driven testing can be defined as testing based Model Driven Architecture (MDA) [OMG03] either by transforming a platform independent test design model directly to test code or to a platform specific test design model. In Telling Test Stories (TTS) the platform independent model is transformed directly to test code.

Compared to other approaches, TTS is based on tightly coupled UML-based system and test models providing support for test-driven development. To make the test models

executable, we have to transform them to executable test code based on a flexible adapter concept that bridges the communication between the System Under Test (SUT) components to the TTS backend (the Test Controller) integrating different target technologies such as RMI, CORBA or Web Services. Additionally the Test Controller has to handle synchronous and asynchronous communication. In this paper we show how we have designed and integrated these concepts within the Telling Test Stories framework based on an industrial application called Telephony Connector.

Many approaches to the generation of executable tests from annotated non-UML models [Mar04] and UML models [Har05] have been developed. But only a few approaches define separate test models based on a UML profile as metamodel such as [Har04] or the most prominent and standardized approach based on the UML Testing Profile (UTP) [OMG05]. According to [Bak07] system level testing is based on the formalization of use cases using interactions and their mapping to UTP test specifications. The aim of our approach is a business oriented view on the test models. Test models can be created together with domain experts in a test-driven way even before the system model has been completed. Therefore the system and test model share concepts, are very abstract -- not considering the test architecture and the underlying technology -- and support the tabular description of data.

The most prominent testing language and framework for cooperative systems is TTCN-3 [Wil05] which has been applied in many industrial case studies [Ttc09]. TTCN-3 has been combined with the UML Testing Profile to define a promising model-driven testing approach [Bak07]. In [Zan05] transformation rules are defined mapping the UML Testing Profile to TTCN-3 which is afterwards compiled to executable test code in Java. In our approach, the test models are directly transformed to test code in Java which is also one of the languages for adapter implementation. Due to traceability of services this emphasizes the feedback cycle to the model elements.

An agile approach to system testing based on tabular definition and execution of test cases via fixture code has been implemented in the FIT framework [Mug05]. Our approach also uses a tabular definition and execution of test code. Additionally we use UML activity diagrams for defining the test behavior.

The paper is structured as follows. First in section 2 the Telling Test Stories methodology and framework is explained. Then in section 3 the Telephony Connector application as a System Under Test is explained. Section 4 presents how the Telephony Connector application is tested with Telling Test Stories, and finally in section 6 we draw conclusions and explain future work.

2 Telling Test Stories

In this section we give an overview of our testing framework called Telling Test Stories. We explain its underlying concepts, its methodology and the system architecture.

2.1 Basic Concepts

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

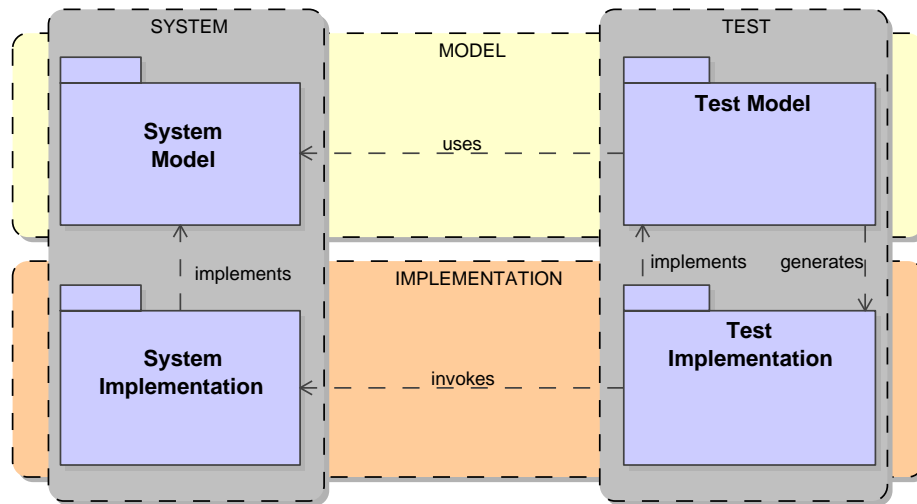


Fig. 1: Telling Test Stories Overview

The system model describes formal system requirements at business level based on a metamodel defined as UML Profile [OMG07]. The system model is specified by the domain experts and system analysts based on the notions of actors, services and classes. A *service* models business or application logic which can be accessed via standardized interfaces, whose operations have input resp. output parameters of arbitrary *class* types and which are callable by *actors*. The semantics of its operations is defined by pre- and postconditions.

An important assumption in TTS is that the system model and the system implementation are traceable. In particular, each business service in the system model can be traced to an executable service in the system implementation.

The system implementation also called the system under test (SUT) provides services callable by the test implementation. This contains services representing business logic and configuration services for test purposes.

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

The test implementation is generated by a compiler which transforms test story files into source code files, so called *test code*, of the execution language. These files are then executed by the *Test Controller*. *Service adapters* make the abstract service calls of the test stories executable by either providing the glue between the service calls and the executable services or by a link to surrogate services like manual input, mock services or external test services. Our prototypical Test Controller implementation is in Java, and the test code is Java source code. The methodology itself is not restricted to Java.

2.2 Methodology

Fig. 2 shows the workflow of our testing methodology.

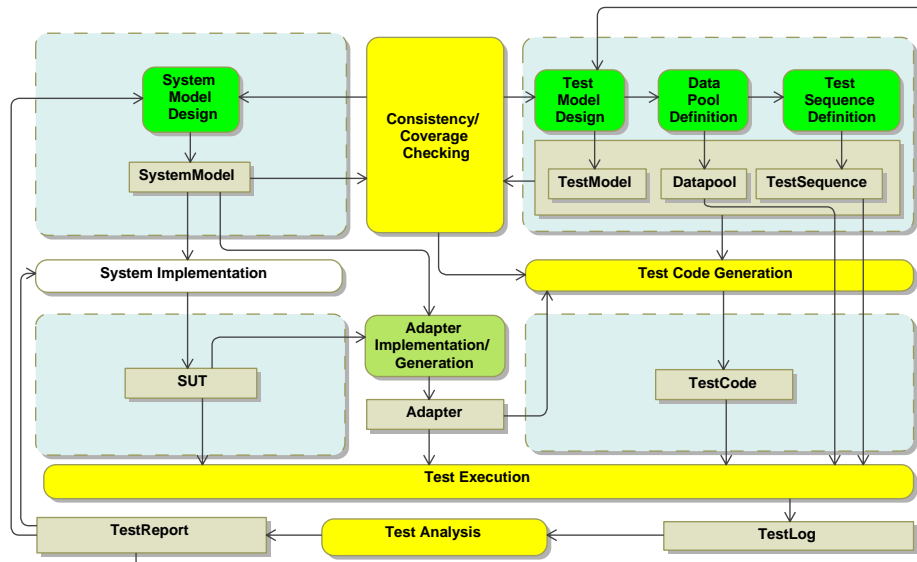


Fig. 2: Methodology of Telling Test Stories

The activities *Consistency/Coverage Checking*, *Test Code Generation*, *Test Execution*, and *Test Analysis* are executed automatically by the TTS framework, and the activities *System Model Design*, *Test Model Design*, *Data Pool Definition* and *Test Sequence Definition* have to be done manually (with tool support). Note that the *Adapter Implementation* can be done manually or automatically based on the system model and the SUT if a stub generator for the underlying service technology is provided. The four rounded boxes group elements and relate them to the four corresponding artifacts, i.e. system model, test model, system implementation and test implementation of Fig. 1.

The methodology of our framework supports test-driven development of systems on the model level. The first step in the development process is the iterative design of a test and a system model. The test design additionally contains a *data pool* definition, i.e. the definition of test data for the test stories, and the *test sequence* definition, which defines the sequence of test stories to be tested and which assigns states and test data to the test stories.

The system model and the test model, including the test stories, the data and the test sequences, can be checked for consistency and completeness. Completeness corresponds to coverage checks in the domain of testing, independently of the system or test implementation itself at any point in time. This allows for an iterative improvement of their quality and supports model-driven system and test development. Our methodology does not consider the system development itself but is based on traceable services offered by a SUT. As soon as adapters which may be generated automatically or implemented manually are available for the system services, the process of test code generation can take place. The generated test code is then automatically compiled and executed by a test execution engine which logs all occurring events into a test log. The test evaluation is done offline by a test analysis tool which generates a test report and annotations to those elements of the system and test model influencing the test result.

2.3 TTS Tool Architecture and Implementation

The architecture of the TTS tool implementation is depicted in Fig. 3.

The system has a *Repository* which stores and versions all object nodes depicted in Fig. 2:

- *SystemModel* holds the system model,
- *TestModel* holds the test model as collection of test stories,
- *Datapool* holds the test tables corresponding to test stories and the initial states for executing test stories,
- *TestSequence* holds sequences of test stories for direct execution,
- *TestLog* holds log files generated by test runs,
- *TestReport* holds test reports.

The *SUT* provides executable services and may additionally provide system services for testing, e.g. for resetting the internal database. The *TestController* executes a sequence of test stories. For every test story an initial state is set up and the stories top-level method is invoked for every line of its corresponding data table. The *TestController* has a *Timing* component supporting timeout monitoring needed for handling asynchronous service calls and a component for *EventHandling* processing the events which occur during the test execution such as errors, timeouts or test verdicts. The *Controller* generates a test log for one test story execution. Test logs are used by the *ReportManager* to produce test reports.

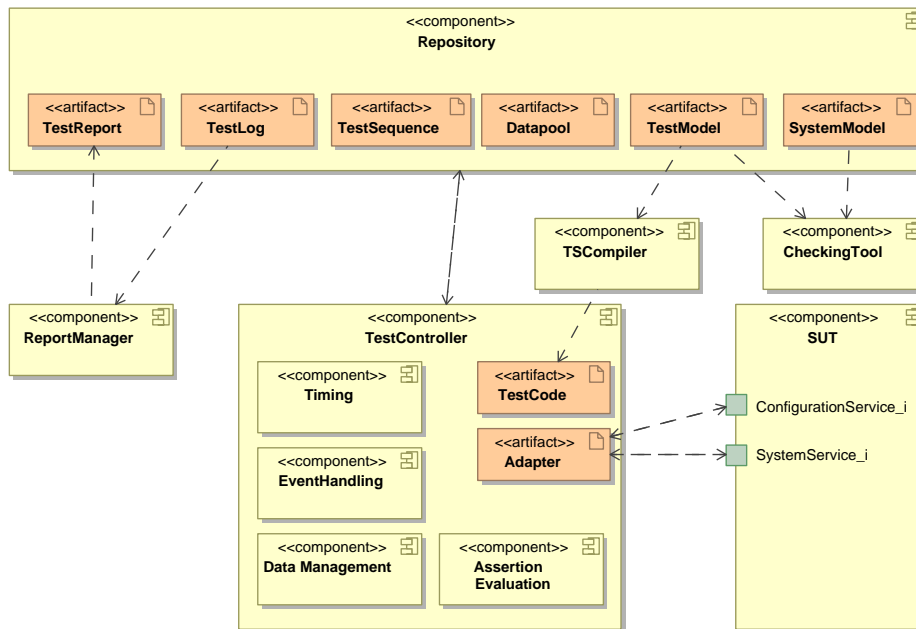


Fig. 3: Architecture of the Telling Test Stories tool

We have implemented our framework based on the Eclipse platform. Therein the repository component corresponds to an versioned Eclipse workspace where all artifacts are stored as files. The *Test Controller* itself has been implemented in Java. Based on that the test code and the adapters are also in Java. The models are in EMF UML2 [Emf09] and for the test code generation Xpand [Xpa09] which is widely accepted and tightly integrated with Eclipse has been used. The reporting has been implemented with business intelligence and reporting tool BIRT [Bir09]. Our implementation is available online via an Eclipse Update Site [Tes09].

3 The Telephony Connector Application

In this section we explain our industrial Telephony Connector application that is tested with TTS.

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. Therefore the combination of TTS with the Telephony Connector provides a good case study but also great effort for our project partner SoftMethod because TTS enables more efficient testing than manual testing.

The Telephony Connector application is a telematics system for the automotive industry developed for a big German car vendor.

The application has been developed according to the European Unions eCall proposal enforcing any newly homologated car to be equipped with a technical device automatically initiating emergency calls when an accident happened. 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.

The Telephony Connector is a standalone server application bridging the actual telephone system on the car vendor's side based on private branch exchange (*Telephony System*) with the remaining infrastructure of the car vendor (*Backend*) like database systems holding car specific data. Hereby the Telephony Connector provides several operations as a service 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 Telephony System with the remaining Backend of the car vendor, including the actual call center in order to fully control the handling of the call. The system components and their interconnection is depicted in Fig. 4.

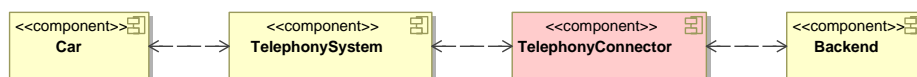


Fig. 4: System Components

In almost all scenarios the car executes the initial call and therefore the Telephony Connector is equipped with an asynchronous notification mechanism to inform the Backend of incoming calls or the termination of the call maybe due to signal loss of the mobile device. Not only the initial notification but all services are executed on asynchronous service calls due to the nature of telephone systems.

As already mentioned the Telephony Connector has a wide variety of possible use cases. In this paper we focus on the following concrete scenario:

We assume a car has an accident, collects all necessary data and initiates a call to the call center. This lets the Telephony Connector come into action and start its internal procedure of collecting the transmitted data, which results in the notification of the Backend of an incoming call. After the initial call was received the scenario will route the call to another destination, which can be the local police station. Afterwards the Backend decides to terminate the call by hanging it up. As earlier mentioned the whole communication is done in an asynchronous way, such that each service call is acknowledged by asynchronous events.

As an additional challenge the Telephony Connector is a standalone server application running on a dedicated server separated from TelephonySystem. Therefore the TTS framework has no possibility to startup the Telephony Connector as a child process. The adapter concept of TTS was used to connect the Telephony Connector SUT to the Test Controller. Technical details on the concrete testing procedure of the Telephony Connector with Telling Test Stories are presented in the next section.

4 Test Design and execution

In this section we explain the system and test modeling, and the test execution of the Telephony Connector application within Telling Test Stories based on the test scenario explained in the last section.

4.1 System and Test modeling

The system model contains services, actors and classes of the Telephony Connector application. Services and their operations may have pre- and postconditions and they may be further refined by sequence diagrams or state machines.

The Telephony Connector provides one service `CallCommand` whose interface is depicted in Fig. 5.

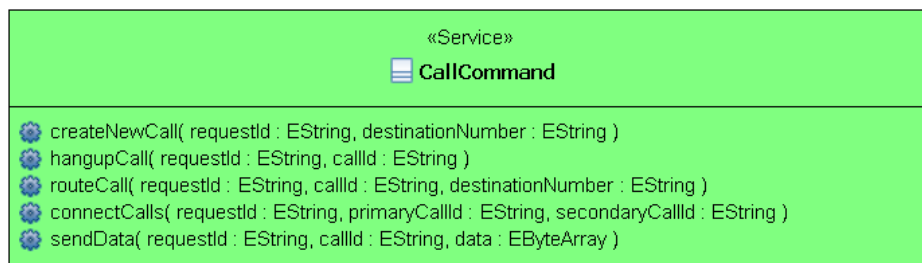


Fig. 5: Service `CallCommand`

The service `CallCommand` contains operations for initializing new calls to a specified destination (`createNewCall`), terminating existing calls (`hangupCall`), redirecting a call to a new destination (`routeCall`), connecting two existing calls (`connectCalls`) and for sending data (`sendData`).

Additionally the system provides two signal services depicted in Fig. 6 and Fig. 7. Operations of these services are invoked by the Telephony Connector actor and sent to the Test Controller. All services operations are executed asynchronously and therefore handled as triggers on the Test Controller side.

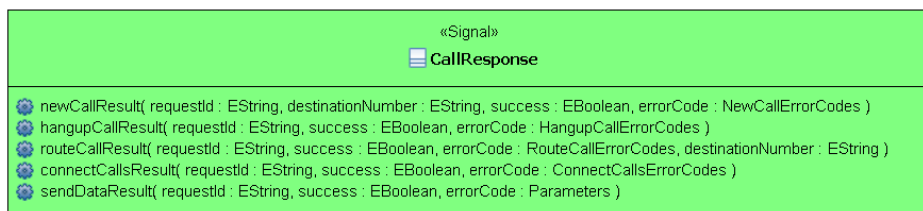


Fig. 6: Signal `CallResponse`

The signal service `CallResponse` contains operations each corresponding to one operation in the service `CallCommand`.

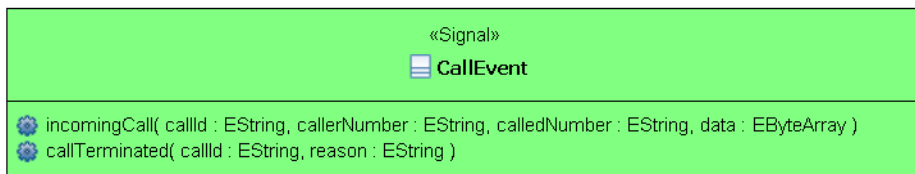


Fig. 7: Signal `CallEvent`

The signal service `CallEvent` contains the operation `incomingCall` and `callTerminated`.

As the Telephony Connector is a standalone server application, there is no need for an additional configuration service to start or stop it. The Telephony Connector is designed to run on a 24/7 bases with no need to restart, it will reset all its internal environment properties when a call has completed its business case.

The system contains some enumeration types as classes. Enumeration types are used by classes as input or output data type, e.g. for the input parameter `errorCode`. Fig. 8 exemplarily depicts the enumeration type `RouteCallErrorCodes`.

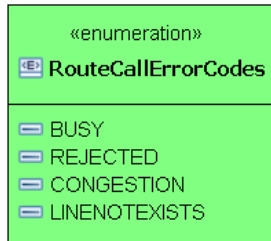


Fig. 8: Enumeration Classes

The system contains three actor types, one for the car (`Car`), one for the Telephony Connector (`TelephonyConnector`) and one for the backend (`Backend`).

Based on the system model, test stories can be defined. As already explained in Section 2.2, tests are modeled as test stories with test data and initial states.

One example test story for routing a received call and its final termination is depicted in Fig. 9. Note that every action in this activity diagram has additional properties, e.g. the referencing operation which is not printed directly in the diagram.

The test story `RouteCall` formalizes the test scenario described in the last section. First the Test Controller waits for an incoming call (`Trigger IncomingCall`). Then the asynchronous calls of the operation `routeCall`, indicated by the Servicecall `RouteCall` and the Trigger `RouteCallResult` as notification for the success of the routing of the call to another destination. Finally the call is asynchronously terminated by the backend via the Servicecall `HangupCall` and its corresponding Trigger `HangupCallResult`.

Every action element in Fig. 9 is of UML metaclass `CallOperationAction` referencing the operation corresponding to the name of the action element, e.g. the action element `RouteCall` references the operation `routeCall` of the service `CallCommand`.

Finally the test story contains an assertion which is used to compute the verdict of every test run. In the test story `RouteCall` the assertion contains for pass the following boolean expression:

```
RouteCallResult.success = Assertion.Result1 and
HangupCallResult.success = Assertion.Result2
```

The corresponding test table contains columns for all input parameters of the operations `routeCall` (`requestId`, `callId`, `destinationNumber`) and `hangupCall` (`requestId`, `callId`), and columns for all free assertion parameters (`Result1` and `Result2` in the example). In the table rows also trigger variables or return variables of synchronous services can be used to set the values, e.g. `IC.callId`.

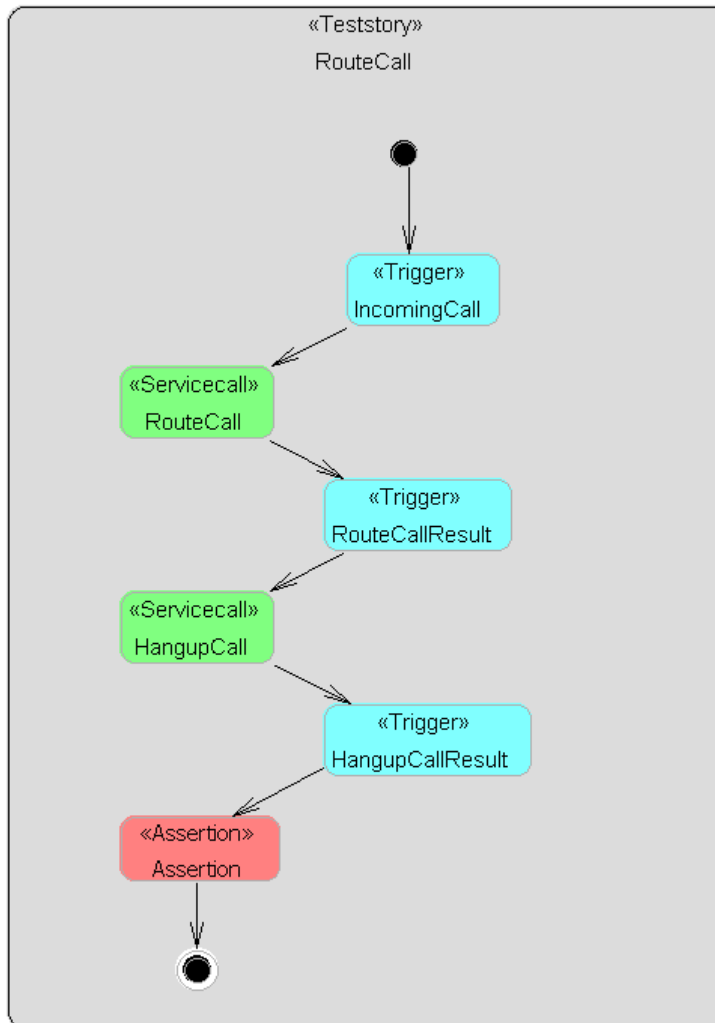


Fig. 9: Test Story RouteCall

The corresponding test table is as follows (due to space limitations the table is separated into two parts):

RC.requestId	RC.lineId	RC.destinationNumber
1	IC.callId	00498912345
3	IC.callId	00435129874

HC.requestId	HC.lineId	A.Result1	A.Result2
2	IC.callId	True	True
4	IC.callId	False	True

Note that RC is an abbreviation for `RouteCall`, HC is an abbreviation for `HangupCall`, and IC is an abbreviation for `IncomingCall`.

The execution order of test stories is defined by a test sequences which is a sequence of story elements each grouping a test story, its test data, a setup procedure, a tear down procedure and a global verdict. The global verdict can be used for aggregating the single test verdict (see [Chi09] for details how this works within TTS). In Fig. 10 an exemplary test sequence `Sequence` calling the test story `RouteCall` is depicted.

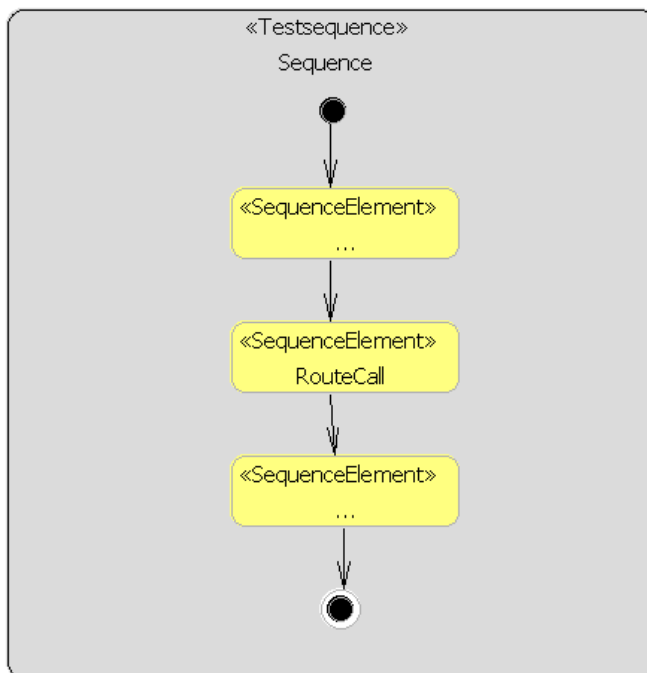


Fig. 10: Test Sequence

The test stories are transformed into Java code which is then executed by the test controller based on the test sequence. Some details relevant for the Telephony Connector are explained in the next section.

4.2 Test execution

As we focus on the testing of the Telephony Connector with the TTS system, the back end is replaced by a RMI adapter and connected to the Test Controller. Due to the simplicity of the TTS Adapter interfaces the concrete implementation of the RMI Adapter on Telephony Connector side was only marginal work.

For executing the triggers and service calls in a test story against the Telephony Connector, the Test Controller first has to provide its RMI controller by starting an own RMI server and register locally. As a next step the SUT side RMI adapter has to register himself at the controller side RMI registry, retrieve the controller's remote object and inform the engine about the successful registration of a SUT adapter, which can further on be used to invoke methods on the SUT. Note that the registration, setup and tear down process is done within the TTS Adapter concept, a concrete SUT does not have to implement the communication procedure. As soon as the Test Controller has been notified about the registration of an adapter, the test execution starts.

The test story `RouteCall` of the last section is then executed as follows: First the story internally suspends its execution to wait for a trigger event. In this special case of the Telephony Connector example, this event is human-driven by performing a call to the actual Telephony Connector, being under test. As soon as this event reaches the engine (via the controller's remote interface, providing a trigger method, and is evaluated successfully against the event being waited for, the execution of the story is resumed. Next, a service call to one of the methods offered by the SUT is performed by calling the `invoke` method, which has to be implemented by default by the corresponding SUT adapter, to encapsulate the actual operation calls. For the remaining activities, the methodology remains the same, until the assertion action, which computes a verdict of the test case. This is done by evaluating a pass, fail or inconclusive expression based on the concrete variable context of the story execution. Every event and verdict computed during the execution of a test story is logged and evaluated offline.

5 Conclusions and Future Work

In this paper we have demonstrated how the Telling Test Stories approach can be applied to an industrial telephony connector application. We have designed a system model and one exemplary test story `RouteCall` modeling a typical test scenario of the system. We have shown how the Telling Test Stories framework can be connected to the system under test via RMI adapters and how asynchronous calls are executed.

Adapters provide flexible ways to access executable services on a system under test. Our approach is the first one that makes requirements for cooperative systems executable by linking them via system models, test models and adapters to executable system services.

At the moment we have just implemented a small number of test stories focusing on functional testing. Due to the importance of performance testing we will also integrate timing constraints and evaluate so called global verdicts, aggregating verdicts of single test cases. Additionally the tests cannot be executed completely automatically at the moment because the incoming call of a car has to be initialized manually. As soon as all test stories can be executed automatically, we start to design regression tests.

6 Acknowledgements

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

References

- [Bak07] P. Baker, P. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, C. E. Williams: Model-Driven Testing – Using the UML Testing Profile, Springer, 2007
- [Bir09] BIRT, available at <http://www.eclipse.org/birt/phenix/> (2009-03-30)
- [Chi09] J. Chimiak-Opoka, S. Loew, M. Felderer, R. Breu, F. Fiedler, F. Schupp, M. Breu: Generic Arbitrations for Test Reporting, IASTED SE, 2009
- [Emf09] EMF UML 2, available at <http://www.eclipse.org/modeling/mdt/?project=uml2> (2009-03-30)
- [Fel08] M. Felderer, J. Chimiak-Opoka, R. Breu: Telling TestStories - Modellbasiertes Akzeptanz-Testen Serviceorientierter Systeme, Softwaretechnik Trends, 2008
- [Fel09] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, F. Schupp: Concepts for Model-Based Requirements Testing of Service oriented systems, IASTED SE, 2009
- [Har04] A. Hartman, K. Nagin: The AGEDIS tools for model based testing, ISSTA, 2004
- [Har05] J. Hartmann, M. Vieira, H. Foster, A. Ruder: A UML-based pproach to system testing, ISSE, 2005
- [Mar04] R. Mugridge, W. Cunningham: Fit for Developing Software: Framework for Integrated Tests, 2005
- [Mug05] R. Mugridge, W. Cunningham: Fit for Developing Software: Framework for Integrated Tests. Prentice Hall, 2005
- [OMG03] OMG: MDA Guide Version 1.0.1, 2003
- [OMG05] OMG: UML Testing Profile, Version 1.0, 2005
- [OMG07] OMG: Unified Modeling Language (OMG UML), Superstructure, V2.1.2, 2007
- [Tes09] Teststories.info, available at <http://www.teststories.info/trs/> (2009-03-30)
- [Ttc09] TTCN-3 Application Areas, available at <http://www.ttcn-3.org/ApplicationAreas.htm> (2009-03-30)
- [Wil05] C. Willcock, T. Deiss, S. Tobies, S. Keil, F. Engler, S. Schulz: An Introduction to TTCN-3. John Wiley and Sons, 2005
- [Xpa09] Xpand, available at <http://www.eclipse.org/modeling/m2t/?project=Xpand> (2009-03-30)
- [Zan05] J. Zander, Z. R. Dai, I. Schieferdecker, G. Din: From U2TP Models to Executable Tests with TTCN-3 – An Approach to Model Driven Testing. TestCom , 2005