From Scenarios to Test Implementations Via Promela

Andreas Ulrich, El-Hachemi Alikacem, Hesham H. Hallal, Sergiy Boroday

From Scenarios to Test Implementations via Promela

Andreas Ulrich[1], El-Hachemi Alikacem[2], Hesham H. Hallal[3], Sergiy Boroday[2]

[1] Siemens AG, Corporate Technology, Munich, Germany
andreas.ulrich@siemens.com
[2] CRIM, Montreal, Canada
{sergiy.boroday, el-hachemi.alikacem}@crim.ca
[3] Lebanese International University, Beirut, Lebanon
hesham.hallal@crim.ca

Abstract. We report on a tool for generating executable concurrent tests from scenarios specified as message sequence charts. The proposed approach features three steps: 1) Deriving a MSC test implementation from a MSC scenario, 2) Mapping the test implementation into a Promela model, 3) Generating executable test scripts in Java. The generation of an intermediate Promela model allows for model-checking to inspect the test implementation for properties like soundness, fault detection power as well as for consistency checking between different test scenarios. Moreover decoupling the executable test scripts from the scenario specification makes it possible to use different backend code generators to support other scripting languages when needed.

Keywords. Scenario-based testing, distributed testing, test consistency, Promela, Message Sequence Charts, UML2 sequence diagrams, tool implementation.

1 Introduction

A recent survey on model-based testing (MBT) approaches [1] analyzed about 400 papers to find evidence about the use of MBT in industrial projects. It could identify 85 papers describing UML-based and non-UML approaches of some degree of uniqueness. However only 11 papers out of them report about an industrial application or experimental case studies that go beyond a proof of concept. It turns out that most MBT approaches of industrial strength target the domains of safety-critical or embedded systems; wider use in general software engineering is still deficient. The paper concludes that "it's […] risky to choose an MBT approach without having a clear view about its complexity, cost, effort, and skill required to create the necessary models".

The trend towards complex systems with an increasing degree of connectivity, configurability, heterogeneity, and distributedness requires improved processes and methods especially for the system integration phase. Testing is still the preferred validation method used in this context. Because of the nature of complex systems, an MBT approach seems to be a good candidate for its ability to abstract away certain system aspects that distract test engineers from the specification of proper tests.

To support system integration testing, a scenario-based testing approach is developed and re-fined to the needs of the domain of embedded systems as they are of interest, for example, at Siemens. Scenario-based testing has been broadly applied in the telecommunication domain already. It is based on the specification of interaction scenarios between components to be integrated and is typically described in terms of message sequence charts (MSCs) or UML2 sequence diagrams [2], [3]. Additional specification features however need to be added to a scenario specification such as the expression of real-time constraints to make it applicable to the considered class of systems. The UML2 profiles SysML and MARTE provide some useful language features that are an initial input to develop the new test specification approach.

This paper presents an overview about the test tool *ScenTest* that currently supports the untimed specification of scenarios for testing distributed systems and the generation of concurrent testers in Java. As a boon the tool supports the verification of test implementations as well as the consistency check between different test scenarios via the intermediately generated Promela model. The possibility to verify test implementations turned out to be particularly helpful when developing and fine-tuning the test generation algorithms outlined in [4]. The consistency checks are of great help for tool users if they incrementally build up a test suite from a number of test cases that all together should not contradict each other.

The paper is organized as follows. Section 2 introduces the scenario-based testing approach and puts it into the context of other MBT approaches. Section 3 describes the different steps of the test generation process in some detail. Afterwards, Section 4 explains the verification feature of the tool before Section 5 concludes the paper.

2 Scenario-Based Testing

In software testing practice, MBT approaches evolved as the latest innovation step as depicted in Fig. 1 [5]. To apply these approaches successfully in today's industrial software development projects, one has to put efforts to automate the execution of tests in the first place. Today, a full range of test automation solutions of varying abstraction levels exists depending on the actual application domain and the project history. However, with an improved applicability of model-driven approaches due to the provision of better tools and supporting development processes, a high impact of MBT approaches on software testing can be expected now and in the near future.

There are various modeling technologies that can be used for MBT. The orientation towards an industrial context imposes however some extra requirements and constraints. First of all, it is not wise to assume that a formal (complete and consistent) model of the software system always exists. What is reasonable to assume instead is an informal model that describes the system (or parts of it) mostly in natural language. Therefore, domain understanding and modeling play the major role for a successful application of MBT in a typical industrial context. Semi-formal models, such as UML, are becoming increasingly common in industrial projects contributing to the reduction in MBT modeling efforts. However, the development of a domain-specific language that is adequate for modeling the requirements of a given application domain remains a crucial factor that decides about the success of MBT.

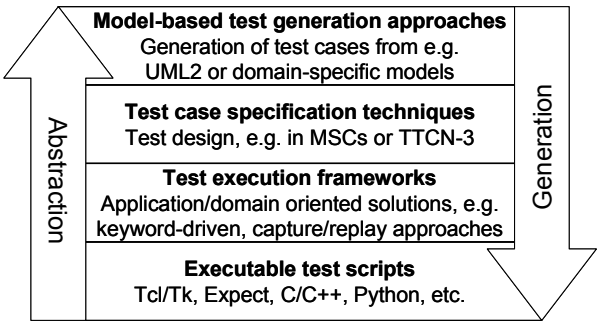

Fig. 1. Testing approaches according to their level of abstraction they offer.

Second, MBT can only be effective in industrial projects when the total efforts for applying an MBT approach are affordable. This means that the testing approach should produce readily executable tests, even if the software system is modeled only partially. The model should also have some level of resilience to modifications (changing requirements, product evolution). Another important aspect is that the level of abstraction in the model must be adequate for describing the intended test purposes concisely. This implies that the different abstraction levels must be bridged in order to obtain executable tests (cf. Fig. 1).

Driven by these experiences, a scenario-based testing approach is developed that differs from other MBT approaches as follows. The basic assumption is that the (sub-) system that is the subject of the integration test only needs to be specified partially in terms of test scenarios that can be observed at the system's boundary and possibly at internal interfaces if they are exposed to a tester.

The test scenario (see next section) is a restricted MSC that represents the system under test (SUT) only as a single lifeline, but considers all of the SUT's interfaces to be covered by concurrent tester components. It turned out in practice that this representation of a test scenario is flexible enough to cover a wide range of system integration tests where the SUT consists of one or many components. Since the scenario-based testing approach is basically a black-box test, a single SUT lifeline is therefore sufficient to specify the interactions of the SUT with its environment (tester) as long as its interfaces are clearly distinguishable.

Because of its capability to work with partial system specifications the entrance level to apply this method is lower than other MBT approaches that require more complete system specifications such as Conformiq Qtronic or Microsoft SpecExplorer to name just a few. Although the latter tools provide an even higher abstraction and are more powerful because test cases are automatically generated, the overall efforts to specify system specifications is still a limiting factor to apply MBT approaches in the industrial practice. Therefore we believe that scenario-based testing with appropriate tool support offers a valuable contribution to many real-world projects.

A scenario-based testing approach is not new. The closest contender of our *ScenTest* tool is Motorola's *ptk* tool [6]. Since it is an in-house tool we need to rely on published data to assess its capabilities. First of all, it clearly addresses the needs in the telecommunication domain and strictly sticks to the full MSC semantics. Outside of this domain however, a strict MSC notation is seldom used. Instead a general ubiq-

uity of UML can be observed. With the UML2 version it became also sufficiently strong enough to be of use also for the purpose of (test) code generation. Therefore we decided to embrace rather a UML2 approach because of its higher acceptance and also easier tool support. Another distinguishing factor of both approaches might be the capability to check the consistency between test scenarios (see Section 4) that allows to incrementally build up a more and more complete system specification (in terms of test scenarios).

3 Test Tool *ScenTest*

3.1 Tool Overview

In this section we describe an approach, employed by our prototype tool, to map test scenarios, created by the test designer from more general use case scenarios, into test scripts. In order to develop the test scenarios, the test designer uses a UML2 editor. Currently, the prototype tool works with Sparx Enterprise Architect, which is one of more affordable commercial UML editors with XMI export support.

Each test scenario is a sequence diagram or MSC with one designated SUT lifeline (instance). All the other lifelines represent test components. The test scenario can depict communications between the SUT and test components, but not among test components. All the necessary communication between test components are automatically implemented by the tool; the test designer is, therefore, relieved from doing this manually. At the moment the prototype tool supports parallel blocks with a limited support of alternative blocks and loops.

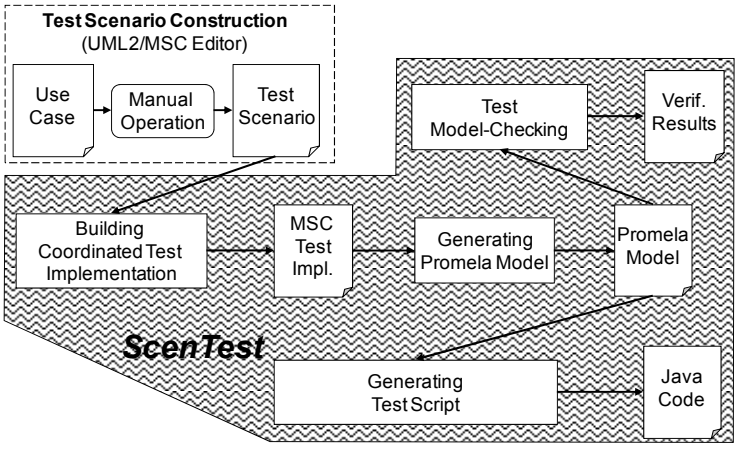

Fig. 2. *ScenTest* – Test implementation and verification framework.

The tool generates test scripts in the platform independent language Promela [7] developed for modeling and automated analysis of distributed systems. While currently the test scripts are mapped further into Java, support of other languages is also foreseen. One of the benefits of using Promela is the possibility of simulation and formal

verification of tests, which can be performed by the test tool developers to debug the tool itself and by the test designers who wish to simulate test scripts or check their properties. Model-checking of test script properties and test scenario consistency [8] is discussed in Section 4.

The tool chain is illustrated in Fig 2. The main advantage of the underlying approach over ones that rely on state machines or labeled transition systems, is that the test designer is not required to provide a complete formal specification of the SUT; it suffices to have a partial model in the form of an MSC scenario.

3.2 Mapping Test Scenario MSC to Test Implementation MSC

Correct implementation of a test scenario requires coordinating messages and delays [4] to ensure test soundness and increase fault detection. While the need for coordinating messages was identified in early research in scenario-based testing already [9], it has been only demonstrated recently that in addition delays have to be introduced to assure soundness [4]. For instance, consider a vending machine example (Fig. 3). The vending machine has two interfaces, one for a service man only and another for consumers. Thus, a tester for the vending machine consists of two test components emulating the behaviors of the serviceman and a consumer. If the *turn on* message is followed by the *coin* message with no delay in a test implementation, the outcome would depend on the race between these two messages, and the tester may produce the fail verdict even though the SUT behavior is correct. Thus, sound testing requires delays.

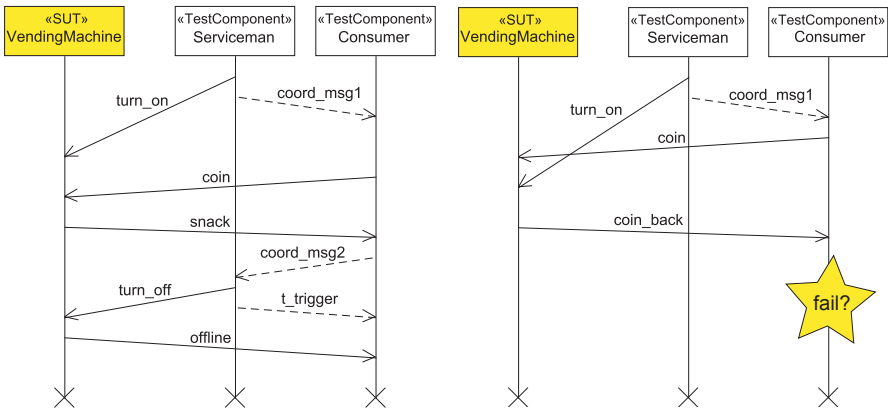

Fig. 3. Sound test implementation (left) and a possible execution of an unsound test implementation (right).

As a consequence, following the approach in [4], coordinating messages and delays are inserted. The duration of the delay is determined according to message latency. However, while approaches suggested in [4] and [9] address all the races [10] among coordinating messages by allowing them to occur concurrently, one can avoid additional concurrency by storing coordinating messages arriving from different test components in different buffers [11]. Thus, since the tool uses different buffers for

different channels the degree of concurrency in the original test scenario is preserved in the test implementation. We also add additional coordinating messages, such as *t_trigger* in Fig. 3, which allow one to detect missing SUT messages using a local timeout. The duration of this timeout needs to suffice for a message to arrive at the SUT, trigger a response message from it, and this latter message to arrive back at the test component that waits for this message.

Simple alternative blocks, where each section starts with messages sent from the SUT to the same test component, are supported by a straightforward extension of the algorithm suggested in [8]. Meanwhile, test scenarios containing alternative blocks of a general nature are treated as ill-formed scenarios at the moment. They could be still processed in future by adding additional coordinating messages, which allow different test components to inform each other of the alternative taken by the SUT.

3.3 Mapping Test Implementation MSC to Promela

Next, we map an MSC representing a test implementation into an intermediate Promela model. Promela is a modeling language for the Spin model checker [7] that is not intended for execution but for simulation and analysis of concurrent systems. Model-checking in our context helps detect errors prior to producing executable test scripts in Java. The advantages of a model checker over usual simulation are obvious since we deal with concurrent processes: model-checking allows the verification of all the possible execution paths. Since test scenarios describe finite behavior, the potential state space explosion can be usually controlled and lies in the limits of Spin.

The Promela model addresses several concerns omitted in the test implementation MSC, namely a mechanism producing the final verdict, the architecture of the test deployment, and the detection of unexpected SUT outputs. The Promela model does not address concerns related to non-deterministic channel delays and resolution of stalling test executions in case of an incorrect SUT since these issues require a notion of time.

The main idea behind the suggested Promela model generation procedure is to represent each test component and in addition each section of a parallel block by separate concurrent Promela processes and to represent alternatives using Promela's selection construct if..fi. In alternative blocks the coordinating messages are inserted similarly to the ones in parallel blocks as explained in [4].

The resulting test architecture, represented in Promela, is defined as follows. Each test component communicates with the SUT through a pair of FIFO channels representing the interfaces of the SUT. Test components are also pair-wise interconnected by a couple of unidirectional channels. Additionally, a master test component *MTC* is defined, which starts the test components and issues a final verdict:

- Pass, when all the test components notify the MTC by sending *pass* messages, and
- Fail, when at the least one test component sends a *fail* message to the MTC.

A test component can have subcomponents introduced to handle parallel blocks, which execute concurrently within the test component and which share the same

channels with the test component in the communications with the SUT and other test components.

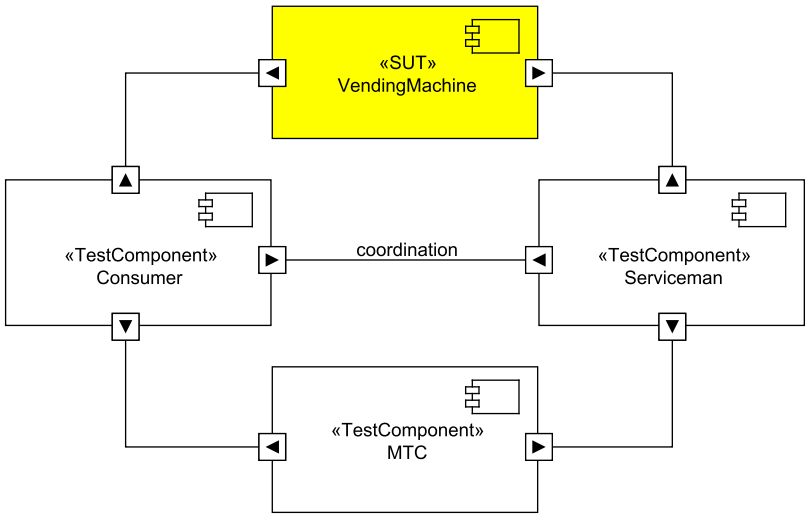

Fig. 4. Test architecture of the example system.

We conclude the section with a sample Promela model for a test component from the test implementation in Fig. 3 that fits to the test architecture shown in Fig. 4. In the presented code, the notation “ServiceConsumer ? coordMsg2” means the reception of message coordMsg2 by process Consumer from process Service over the channel ServiceConsumer. Similarly, the notation “ConsumerService ! coordMsg1” means that coordMsg1 is sent by Consumer to Service over the channel ConsumerService. In order to allow quick detection of unexpected messages we use the case selection construct of Promela if..fi along with the else clause. The use of the else clause to catch unexpected messages is only allowed in polling mode, i.e., checking the head of the input queue without actual consumption. Choice of the else clause due to the emptiness of the buffer is prevented by the nempty guard. The timeout statement after the reception of the first coordinating message is used to model delay.

```
proctype Consumer () {
  ConsumerMTC ! init_confirm;
  ServiceConsumer ? coordMsg2;
  timeout;
  ConsumerSUT ! coin;
  nempty(SUTConsumer);
  if
  :: SUTConsumer ?[snack] -> {SUTConsumer ? snack;}
  :: else -> ConsumerMTC ! fail
  fi;
  ConsumerService ! coordMsg1;
  ServiceConsumer ? timerTrigger1;
  nempty(SUTConsumer);
```

```
    if
    :: SUTConsumer ?[offline] -> {SUTConsumer ? offline;}
    :: else -> ConsumerMTC ! fail
    fi;
    ConsumerMTC ! pass;
  }
```

3.4 Mapping Promela to Java

Apparently, no translator from Promela to Java is available, save few research prototypes, such as HiSpin, a Promela simulator [12], and SpinJ [13], a reimplementation of Spin in Java. Thus, we develop a translator for a subset of Promela corresponding to the language elements that are currently used to represent test implementation MSCs. For each of these constructs, we define a corresponding Java code that preserves the semantics of Promela test models. Table 1 summarizes the Promela language subset and its corresponding Java counterparts. Based on this mapping, the translator generates a multi-threaded Java code, in which each Promela *proctype* is mapped into a Java thread.

Table 1. Promela-to-Java mapping.

Promela Code	**Java Code**
Proctype	Java thread
Mtype	a new class *MTypeSymbols*, in which the *mtype*'s elements are defined as static fields.
buffered channels	Java queue
data types: *byte*, *int* and *bool*	corresponding Java types
local variable	variable in the corresponding thread
operators +, -, ! (not), &&	similar operators in Java
assignment	similar construct in Java
send, receive, polling and *nempty*	invocation of corresponding methods
run	creation of a new thread
if..fi, *do..od*, *goto*, *skip*	corresponding Java patterns

To map Promela channels, we implement a dedicated Java class PromelaChannel, which represents a blocked queue data structure that contains thread-safe methods corresponding to the Promela communication operations send, receive, polling, nempty. Therefore, for each channel present in the Promela model, a PromelaChannel Java object is created. The translation of a Promela communication operation is in fact the generation of its corresponding Java method invocation on the object corresponding to the channel involved in the communication operation.

Along with that, we implement timed and non-timed Java methods to support Promela communication operations. The timed methods must be completed within a given duration, otherwise an exception is raised. Timed methods are used only in the communication of the test components with the SUT. Since the communication be-

tween test components is assumed to be reliable, the non-timed versions are employed in communications between test components. Handling the exception of a timed Java communication method consists of sending a *fail* message to the MTC indicating that an expected message from the SUT was not delivered. The Promela communication operations are translated to either timed or non-timed methods depending on whether the communication is between a test component and the SUT or between test components.

For most of the Promela constructs, the translation to the corresponding Java code is straightforward. However, in few cases specific Promela patterns, rather than individual operations are matched and translated to the corresponding Java code. This approach simplifies the translation of some Promela constructs that have no direct counterparts in Java, e.g., goto, while preserving atomicity of Promela operations.

The Promela's construct *mtype,* which is used to declare symbolic message names and constants, is mapped into a Java class *MtypeSymbols*, in which a static field is defined for each element of *mtype*.

For Promela's selection construct *if..fi*, which allows one to describe several alternative execution sequences, called options, we define a Java pattern, which is based on Java's *switch* construct that implements a similar behavior. The Java pattern for the selection construct of receive statements consists in finding an executable option (i.e., choosing an execution path) among the ones listed within the construct. In the case when no option is executable the thread either executes the *else* clause of the selection construct or, in absence of the *else* clause, re-attempts to select an executable option upon receiving a notification from the corresponding input buffer. Once an executable option is found, it is executed. To detect missing SUT output messages, the described process terminates after a certain timeout by sending a *fail* message to the MTC thread. The repetition construct *do..od*, which in Promela resembles the selection construct, is mapped in Java in the same manner. Unlike the selection construct however, options of the repetition construct can be executed several times until the *break* command is encountered.

Fig. 5 depicts the Java code generation workflow. In this workflow, the input is the generated Promela model that gets parsed to obtain the abstract syntax tree of the model. We use the ANTLR [14] tool to implement the parser for the supported subset of Promela. From an internal representation the Java code generator module produces Java code according to the mapping rules presented above.

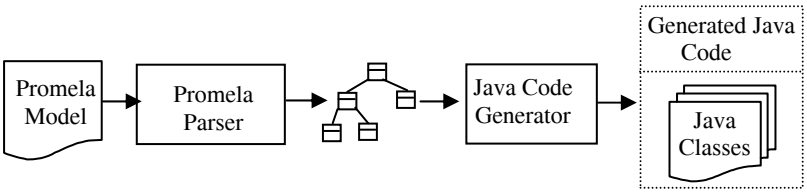

Fig. 5. Java code generation workflow.

3.5 Interfacing Tester with the SUT

In the automatic test execution setup, the generated Java tester is executed against the SUT. Therefore, test adaptors, which are system specific, need to be implemented to interface the SUT with the test components. Test scenarios and implementations use abstract names for messages, which typically convey the meaning or purpose of concrete messages without revealing full details. We implement a keyword driven approach that maps abstract message names to concrete messages ready to be sent out to the SUT. The adaptor implementation accepts an abstract message from the Java queue of a Promela channel implementation and calls the corresponding procedure to pass the input to the SUT using the appropriate technology, e.g. (remote) method invocation or network operations. Another adaptor attempts to identify a received SUT output message and maps it to an abstract message to be forwarded to the waiting test component.

4 Validation of Tests

As a direct side effect of generating an intermediate test implementation in Promela, the model checker Spin can be used to evaluate the validity of tests. In particular, we discuss in this section two test validation methods. The first method aims at verifying soundness of a test implementation by checking its model in Spin against predefined properties, mainly reachability properties to check if, for example, a pass verdict is reachable on all possible paths in the model. The second method, a model based comparison technique, aims at detecting inconsistency between different test scenarios by comparing and analyzing the corresponding SUT models.

4.1 Test Case Verification

The Promela model of a test implementation can be simulated and formally verified in Spin. In particular, model checking a test implementation model together with an SUT model can help ensure soundness of the test implementation, while model checking with a mutant SUT helps check fault detection power. In the case when an independent Promela model of the SUT is unavailable, the *ScenTest* tool is able to a build a partial model from the test scenario's SUT lifeline.

As discussed in Section 3, the tool implements the test generation approach elaborated in [4], which uses delays to guarantee the soundness of test implementations. To illustrate the latter point, we consider an example of the vending machine that deploys the Promela timeout construct to model delays. The Promela code is given in Section 3.3, while the test implementation MSC can be found in Fig. 3. We combine this test scenario with a SUT model of the vending machine. Being turned on, the vending machine generates a snack when a coin is inserted. However, when the machine is not turned on, it simply returns back the coin. Model-checking with Spin against the property "eventually pass on all executions" confirms that the pass verdict is always produced. However, after removal of the delay, model-checking in Spin against the same property produces execution paths that do not result in the pass verdict. In one

of such paths, shown in a Spin counterexample MSC in Fig. 6 where the two test components send turn on and coin messages without a delay between them, the latter message can overtake the former, resulting in the return of the coin and, since the return of the coin is not foreseen by the test scenario, the test subsequently fails.

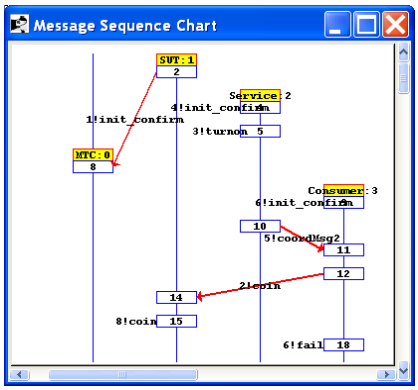

Fig. 6. Model checking demonstrates unsoundness of a test implementation without timer.

4.2 Test Scenario Inconsistency Check

In this section, we discuss how the *ScenTest* tool implements the model-based approach to check a set of test scenarios for mutual inconsistencies. The approach is based on comparing the Promela SUT models using exhaustive simulation to detect differences between them. This approach follows from the fact that in our framework a test scenario is treated as a closed system that includes the test specification (the tester component lifelines) and the SUT (the SUT lifeline). Consequently, the inputs of the SUT are the outputs of the test components and the outputs of the SUT are the inputs of the test components. This duality allows us to state that two test scenarios are inconsistent if their SUT models are inconsistent [8], i.e., the two models contain at least one common trace that leads to states, which have different outputs enabled. In the tool, the comparison approach is implemented following the workflow outlined in Fig. 7. The SUT Model Extractor module builds and combines the models of the SUT lifelines of two test scenarios into a joint Promela model, where one model plays the role of a sender and the other one of a receiver.

The joint model reaches a deadlock whenever the two SUT models that correspond to different test scenarios diverge in their behavior in an improper state. Up to the state where the deadlock occurs, the two models share a common set of traces. The divergence could be either due to different SUT input messages, which does not constitute inconsistency, or due to different SUT output messages. The latter case indicates that some common trace of the two models can be continued with different outputs (Fig. 8), which constitutes an inconsistency. Thus, in order to detect inconsistency we model-check the joint model against a property, which states that each dead-

lock in the simulation of the joint model is due to divergent receive messages. The violation of the property indicates the inconsistency.

Let us consider the example in Fig. 8 with two different test scenarios; the set of outputs of the vending machine after receiving a *coin* is different in the scenarios. In the first scenario, the machine either issues *snack* or returns *coin-back* while in the second one only *snack* can be issued. Fig. 9 shows the first step in building the joint Promela model from the two SUT lifelines extracted from the test scenarios, where one process is transformed into a receiver process and other into a sender process. The SUT outputs turn into inputs and vice versa in order to compose two processes into a joint model (underlined message names in Fig. 9). The verification in Spin reveals that the two test scenarios are inconsistent because of the additional SUT output *coin_back*. Our tool is not able to decide, which of the two scenarios is deficient; it is up to the user to decide whether he should remove a redundant SUT output from the first scenario or add an alternative to the second scenario.

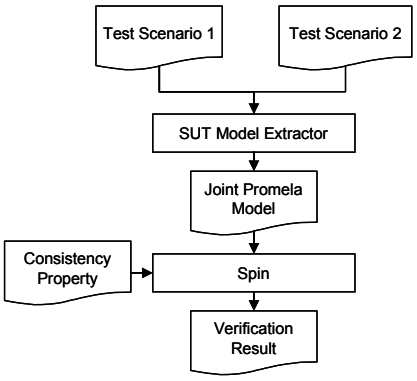

Fig. 7. Workflow for inconsistency checking.

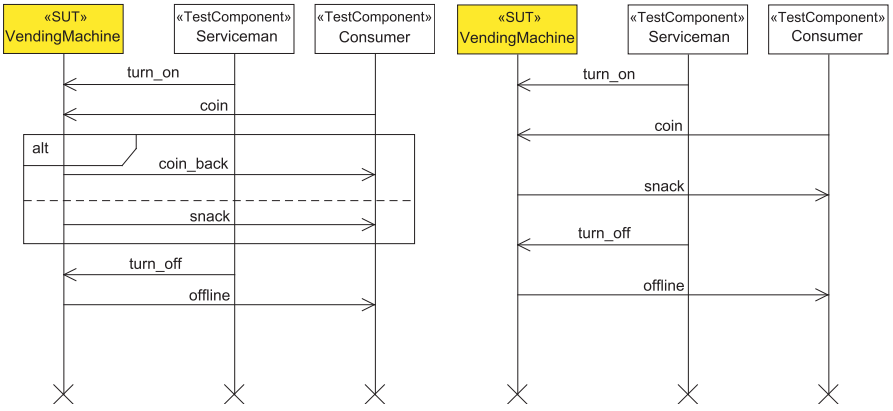

Fig. 8. Two inconsistent test scenarios.

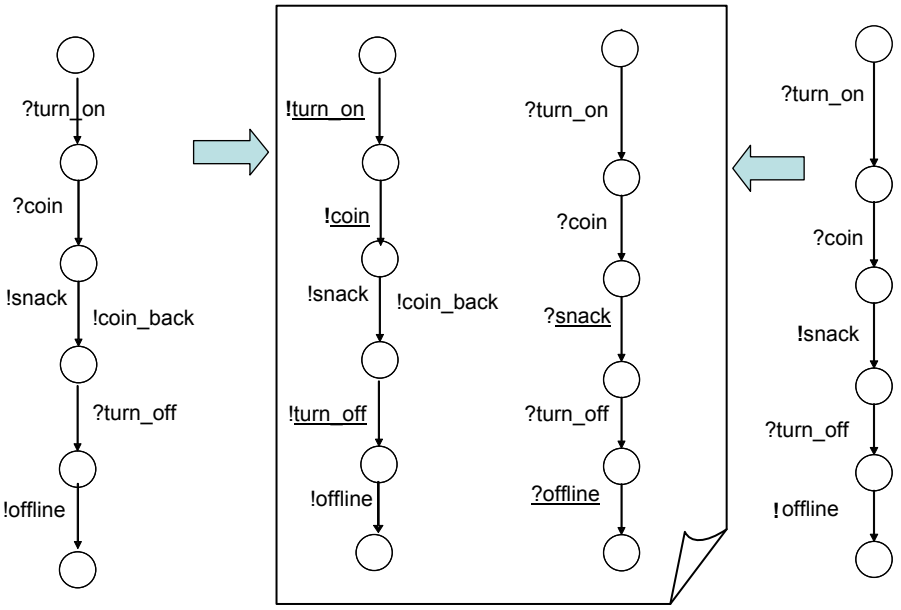

Fig. 9. Detecting a scenario inconsistency in Spin.

5 Conclusions and Outlook

We have presented a scenario-based testing approach and associated tool support. It supports the specification of test scenarios for system integration tests, the generation of executable test scripts, and the test validation and consistency check of test scenarios. Because we rely on Promela as an intermediate representation of the test implementation, it is quite easy to change the generated output language of the test scripts, which is currently Java. Different backend code generators for, for example, TTCN-3 or scripting languages such as Python are possible. The tool is currently completed to be used in first industrial case studies at Siemens.

Although it supports already many MSC/UML2 features to specify test scenarios, the formal specification approach requires further improvements by providing better language facilities to specify data flows within the control flow structure of typical sequence diagrams, e.g. local variables and parameterization of messages and whole test scenarios, variable assignments and local operations on data. Textual extensions to describe such data flow features need to be worked out and integrated into the tool.

Last but not least, support for real-time testing is needed to cover a wider applicability of the tool. For this purpose, an extended test scenario specification approach based on an appropriate representation of timed behavior needs to be worked out. Notwithstanding, we plan to evaluate the efficiency of our MBT approach in the context of industrial projects in the near future.

References

1. Neto, A.D., Subramanyan, R., Vieira, M., Travassos, G.H., Shull, F.: Improving Evidence about Software Technologies – A Look at Model-Based Testing. IEEE Software, vol. 24, pp. 10–13 (2008).
2. Haugen, O.: Comparing UML 2.0 Interactions and MSC-2000. In: SAM 2004: SDL and MSC Fourth International Workshop. LNCS, vol. 3319, pp. 69–84. Springer (2004).
3. OMG UML Specification http://www.omg.org/spec/UML/2.1.2/.
4. Boroday, S., Petrenko, A., Ulrich, A.: Implementing MSC Tests with Quiescence Observation. In: TestCom 2009, LNCS vol. 5826, pp. 235–238. Springer (2009).
5. Ulrich, A.: Introducing Model-Based Testing Techniques in Industrial Projects. GI-Edition. Lecture Notes in Informatics (LNI), Proc. Bd.106, pp. 29–34 (2007).
6. Baker, P., Bristow, P., Jervis, C., King, D., Mitchell, W.: Automatic Generation of Conformance Tests from Message Sequence Charts. In: SAM 2002: SDL and MSC Fourth International Workshop. LNCS, vol. 2599, pp. 170–198. Springer (2003).
7. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley. (2003).
8. Boroday, S., Petrenko, A., Ulrich, A.: Test Suite Consistency Verification. In: 6th IEEE East-West Design & Test Symposium (EWDTS 2008), pp. 235–238 (2008).
9. Grabowski, J., Koch, B., Schmitt, M., Hogrefe, D.: SDL and MSC Based Test Generation for Distributed Test Architectures. In: SDL Forum'99, pp. 389–404. (1999).
10. Mitchell, B.: Lazy Buffer Demantics for Partial Order Scenarios. Automated Software Engineering, Springer. 14, pp. 419–441. Springer (2007).
11. Holzmann, G., Peled, D., Redberg, M.: Design Tools for Requirement Engineering. Bell Labs Technical J, vol. 2, pp. 86–95 (1997).
12. Papesch, M.: Generating Implementations from Formal Specifications: A Translator from Promela to Java. Student thesis no 1826. University of Stuttgart (2002).
13. de Jonge, M.: The SpinJ Model Checker. Master's Thesis. University of Twente
14. —: ANTLR Parser Generator. http://www.antlr.org/.