Using UML Protocol State Machines in Conformance Testing of Components

Dirk Seifert, Jeanine Souquières

Using UML Protocol State Machines in Conformance Testing of Components

Dirk Seifert and Jeanine Souquières

LORIA – Université Nancy 2
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex
{Dirk.Seifert,Jeanine.Souquieres}@Loria.fr

Abstract

In previous works we designed a comprehensive approach for conformance testing based on UML behavioral state machines. In this paper we propose two extensions to this approach. First, we apply our approach in the context of a component-based development, and address the problem of checking the interoperability of two connected components. Second, we address the problem of selecting relevant input sequences. Therefore we use UML protocol state machines to specify restricted environment models. This means that we restrict the valid protocol at the provided interface of the component under test with respect to a specific test purpose. Based on these models we select relevant input sequences. We implemented both extensions presented here in our TEAGER *tool suite to show their applicability. Both extensions address the behavior at the interfaces of components. We use UML state machines as a unified notation for behavioral and protocol conformance testing as well as for test input selection. This considerably eases the work of test engineers.*

1 Introduction

In a model-based development approach, models of the system which have to be built guide and control the development process [4]. There are various types of models differing in the level of abstractions or in their intended use. For example, the Unified Modeling Language (UML) comprises thirteen diagram types to specify the structure and the behavior of a system or a system component [30]. In the first steps, the models are used to analyze the problem domain and to ease the information exchange among developers. Later on, they form the basis to design and implement the system, and serve as documentation. Nowadays, the models are also used for quality assurance purposes. Before implementing the system, required properties can be verified on the models, or they can be simulated to check the intended behavior. Finally, the models can be used for generating tests to check the implemented system. Hence, models of the system which shall be built allow early starting, continuous and automated quality assurance processes.

Testing means executing a system under test with selected but real data to evaluate its conformance, whereat conformance is evaluated on the basis of the observations made on the system under test. It aims in falsification, that means to show inconsistencies between the specification and the system under test. It benefits from the fact that the real system is brought to execution. Thus, the interaction of the real hardware and the real software can be evaluated. A further important advantage of testing is its applicability at different levels of abstraction and at different stages of the development. In [25], we presented a conformance test approach based on UML state machines, where a state machine model [30] serves as the specification of the system under test. We generate test cases from a state machine specification which include input sequences to stimulate the system under test as well as test oracles to automatically evaluate the test execution. Thus, we are able to automatically generate, execute, and evaluate test cases. The focus of our approach is on the level of unit testing.

In a component-based development approach [29, 15], the problem of building a system out of previously-existing software components from a variety of sources is addressed. Building a system out of components has the potential to reduce the development cost and, at the same time, to enhance its flexibility and maintainability. The components are considered as black-boxes described by interfaces expressing their visible behavior. Components are connected through required and provided interfaces. Interoperability is only guaranteed if the required interfaces correctly implement the provided interfaces of the connected components [5]. In most cases, an adapter (i.e., a piece of glue code, expressing the mapping between a required and a provided interface) has to be introduced [21]. In previous works,

we have used the B method and its refinement and assembling mechanisms to model component interfaces as well as patterns for adapters, allowing the interoperability to be checked with tool support [20]. Our verification and testing techniques complement each other. Verification techniques enable early checks of important properties, whereat testing checks the real implementation. The application of both techniques ensures a high-quality development and a comprehensive quality assurance with reliable results.

The first problem we address in this paper is testing the input-output behavior of a system under test in the context of a component-based development. Now, the system under test becomes a component under test, and we assume it to be connected to other system components. In this setting, we additionally check if the component under test correctly implements the provided interfaces of the connected components. We accomplish this by checking the outputs of the component under test at its required interfaces against the specified protocols of the corresponding provided interfaces of connected components. The protocols are specified by protocol state machines. We do not address the problem of integration testing; we still focus on one component under test. Our extension allows early checks of interoperability on the level of unit testing with insignificant additional effort compared to the primary test approach.

Furthermore, we address the problem of selecting relevant input sequences during test case generation for testing reactive systems. In general, the set of possible input sequences for reactive systems is infinitely large. To generate test cases, we have to select a finite subset. Specifying the behavior at the interfaces of components provides a appropriate basis for input selection. The component under test must work correctly in environments behaving according to the specified behavior at the provided interfaces of the component. Thus, it is worthwhile to select test inputs on the basis of these descriptions. We use protocol state machines to specify restricted environment models and to select relevant input sequences. Moreover, we propose two extensions of protocol state machines. First, we extend them by the ability to specify probabilistic behavior. Second, we enable the use of feedback from the system under test when testing non-deterministic systems.

The contribution of this paper is the integration of protocol state machines into our existing test approach as a uniform notation and their use to address two important problems in testing, namely interoperability and input selection.

The rest of the paper is organized as follows. In Section 2, we briefly introduce both variants of UML state machines and review our test approach for conformance testing. In Section 3, we present our extension for checking the interoperability of two connected components. In Section 4, we present our approach to select relevant input sequences, including the two possible extensions to the notation of protocol state machines. In Section 5, we conclude our work and discuss prospects for future work.

2 Foundations

UML state machines are used to model the discrete reactive behavior of a system or a system component through finite state transition systems [30]. They come in two flavors: behavioral state machines and protocol state machines. *Behavioral state machines* specify the states a system or a system component can take and the actions it can execute during its lifetime in response to external and internal events. They are an object-oriented extension of the classical Harel-Statecharts [13]. The semantics is adapted from the STATEMATE semantics [14] to fit into the object-oriented paradigm. *Protocol state machines* are used to express usage protocols of a system or a system component by expressing legal interaction sequences, which consists of either events or method calls.

2.1 Behavioral State Machines

Behavioral state machines are mathematical models with a graphical representation: the nodes depict simple or composed states of a system and the labeled edges depict transitions between these states (see Figure 1 for an example). Composite states are used to hierarchically and orthogonally structure the model, thus reducing its graphical complexity. Labels express conditions under which transitions can be taken and the actions which will be executed when the transitions are taken. Events are used as triggers to activate transitions and can be parameterized to exchange data. Optionally, every behavioral state machine has a data space which can be read and manipulated by the state machine during execution. More precisely, it is possible to read data values to describe specific conditions when a transition can be taken or to manipulate data values and exchange information within the actions. A transition consists of a *source* state, a *trigger* event, an optional *guard*, an optional *effect* (which comprises a sequence of actions), and a *target* state. We also write a transition as follows:

$$source \xrightarrow{trigger[guard]/effect} target \tag{1}$$

With the optional guard, a fine-grained condition to enable the transition can be described depending on the system's state. Hence, the activation of the source state, the trigger event and the guard condition evaluating to true constitute the condition which must be fulfilled to enable the transition. An action can either be a statement manipulating the data space or the generation of new events. The action sequence and the subsequently active target state constitute

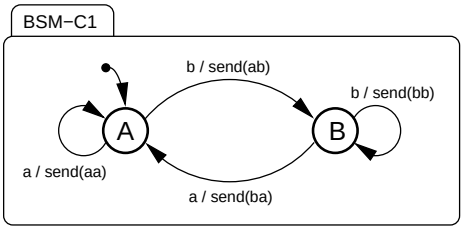

Figure 1. Behavioral state machine for `C1`.

the effect of the transition. In opposite to the classical Statecharts [13], the event processing takes place in a so-called *run-to-completion* step [30]. This asynchronous event processing demands the processing of the previous event to be completely finished before the next event can be processed.

Figure 1 shows a behavioral state machine for a component named `C1` as an example. The top-level composite state `BSM−C1` is refined into two simple states, namely `A` and `B`, whereat state `A` is marked as the default state. The four transitions specify the behavior of `C1`. There are two possible input events, namely `a` and `b`, and four possible output events, namely `aa`, `ab`, `bb` and `ba`. The output events indicate the source state and the target state of a taken transition. For example, an observation `ba` indicates a transition from state `B` to state `A`. For simplification, the state machine neither contains orthogonal regions nor complex guards and actions reading and manipulating data values. Furthermore, the state machine is completely deterministic. For more information on the syntax and semantics of behavioral state machines we refer the interested reader to [25, 30].

The semantic model of behavioral state machines builds on the semantic steps a state machine can execute during its lifetime. Such a step moves the state machine from one semantic state to another semantic state while receiving events from and emitting events to the environment. A semantic state (a *status*) comprises three components: a configuration (a maximal set of active states), an event queue, and all variable assignments. We denote a semantic step as follows:

$$[\![c,q,d]\!] \xrightarrow{in,out} [\![c',q',d']\!] \tag{2}$$

Based on this definition, we describe the execution runs of a state machine as the concatenation of semantic steps and call them *computations*:

$$[\![c_1,q_1,d_1]\!] \xrightarrow{in_1,out_1} \ldots \xrightarrow{in_{n-1},out_{n-1}} [\![c_n,q_n,d_n]\!] \tag{3}$$

2.2 Protocol State Machines

Protocol state machines are attached to interfaces[1] and specify their legal usage protocol. In most applications, protocol state machines are used to specify which operations (call events) can be called in which state, under which condition and what result is expected from their use. In our context of testing reactive systems, we use protocol state machines to specify legal event sequences (signal events) and pre- and postconditions. This involves to specify which events can be processed in which state, under which condition and what result is expected from their processing.

The notation of protocol state machines is very similar to that of behavioral state machines. The keyword $\{protocol\}$ placed close to the name of the state machine differentiates protocol state machine diagrams graphically. The states of a protocol state machine present an external view of the component. The two differences that exist for states in protocol state machines are as follows: first, there exist no entry-, exit- or do-actions and second, invariants can be attached to states in protocol state machines. Protocol transitions specify that the referenced trigger event can be processed in the source state under the precondition, and that at the end of the transition, the target state will be reached under the postcondition. They are labeled with an optional guard (i.e., the precondition), the trigger event, and an optional postcondition. They do not comprise explicit actions:

$$source \xrightarrow{[precondition]\,trigger\,/\,[postcondition]} target \tag{4}$$

Figure 2 shows a composite structure diagram for two components, namely `C1` and `C2`. They are connected via the required interface of `C1` and the provided interface of `C2`. The associated protocol state machines specify the legal behavior at the interfaces. For example, component `C2` expects at its provided interface that when an event `ab` occurs, only the event sequence `bb`·`ba`·`aa` can follow.

If two components A and B are connected, then the protocol state machine of the required interface of A must conform to the protocol state machine of the provided interface of B. In other words, the specification given by a protocol state machine is a requirement to the environment external to that component: it is legal to send events to the component only under the conditions specified by this protocol state machine.

2.3 Conformance Testing

In previous works we designed a comprehensive approach for conformance testing based on UML state machines [26, 25]. In this approach, a UML state machine

[1] In this paper we do not differentiate between interfaces and ports and use the term interface for both meanings.

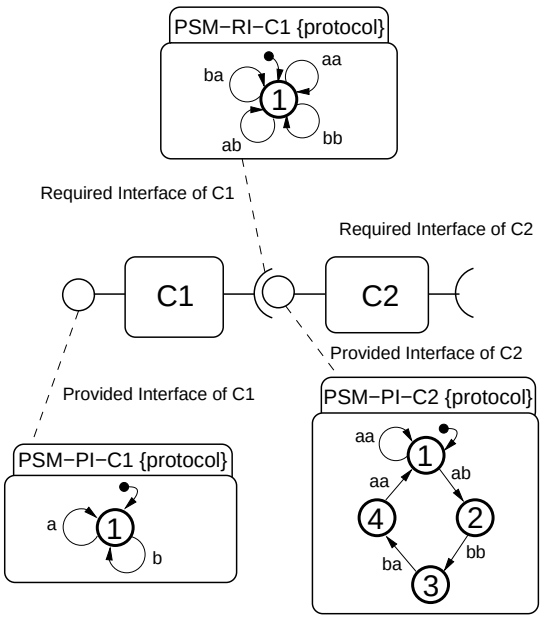

Figure 2. Connected components `C1` and `C2`.

model [30] serves as the formal specification of the system under test. To enable an automated test case generation, we first formalized a substantial subset of UML state machines. This subset includes all relevant aspects to seriously study automated test case generation and evaluation based on state machines. In contrast to other approaches we use a precisely defined semantics for UML state machines including data. We do not restrict state machines to ease test case generation. Instead, we follow the semantics description of the UML standard [30] as much as possible. Only misleading or conflicting statements are clarified. We address all semantic details which arise from the different sources of non-determinism. In particular we address the problem of asynchronous communication which is introduced to the *run-to-completion* semantics.

The precise semantics is a necessary prerequisite for test automation. Furthermore, we need to specify conformance in relation to state machine specifications (to enable automated test evaluation). A system under test conforms to its specification, if the observations for the input sequences on the system under test can be related to the possible observations on the specification. Therefore we compare the observed outputs of the system under test with the precalculated possible correct observations (the test oracle) of the specification:

$$I \leq_{out} S \Leftrightarrow \forall \sigma : \mathrm{seq}\, E_S \bullet out(I,\sigma) \subseteq out(S,\sigma) \qquad (5)$$

An implementation I conforms to its specification S, if and only if, for all input sequences σ over the event set E_S of the specification S, the output sequences of the system under test, $out(I,\sigma)$, are included in the set of all possible output sequences of the specification, $out(S,\sigma)$. According to this notion of conformance we generate test cases on the basis of the computations and the corresponding observations (cf. Definition 3) calculated in a stepwise exploration of the state machine's state space for selected input sequences. Test execution includes stimulating the system under test with those input sequences, observing the outputs of the system under test and comparing them to the pre-calculated possible correct observations.

For example, with respect to the behavioral state machine presented in Figure 1, we could choose to test the system under test with the input sequence `!a!b!b!a!a!b`[2]. For this input sequence we calculate the possible correct observations. Due to the fact that the state machine in Figure 1 specifies only deterministic behavior, we obtain as the test oracle the single observation sequence `?aa?ab?bb?ba?aa?ab`.

In general, the generated test cases include input sequences to stimulate the system under test as well as test oracles to automatically evaluate test execution. A test oracle is a deterministic acyclic *acceptance graph*, accepting all possible correct observation sequences. When a system under test is stimulated with an input sequence of a test case, it must show exactly one complete observation sequence of the test oracle to pass the test. Note that for the sake of simplicity, we did not illustrate all state machine features that make automated test case and test oracle generation a challenge. In particular, the various sources of non-determinism, mainly caused by the asynchronous event processing, by the multiple possible sets of firing transitions and different possible orders of firing transitions, introduce complex behaviors in state machine models and make the computation of test oracles a particular challenge [26]. To evaluate and to show the practicability of our approach we implemented the TEAGER tool suite [27, 24]. TEAGER consists of an environment to automatically generate and execute test cases, and of an environment to execute state machine specifications. The latter we use to analyze the execution behavior and the testability of a state machine, and to measure coverage on a state machine specification to evaluate the quality of generated test suites. The test execution includes both: stimulating the system under test and comparing the observation to the computed possible correct behavior in the acceptance graphs. The communication with the system under test takes place over a socket connection using pre-implemented adapters. This concept offers a flexible way to connect the system under test. It also offers the

[2] In test cases we mark inputs with exclamation marks and outputs with question marks. This may seem complementary to other literature notations. The reason for this annotation is that the outputs of a test case correspond to the inputs of a system under test and the outputs of a system under test correspond to the inputs of a test case.

possibility to use our *State Machine Executor* as a system under test stub.

3 Testing Interface Interoperability

In the previous section we reviewed our approach to test the behavioral conformance of a system under test with respect to a behavioral state machine specification. Now we consider the system under test as a component of a larger system and we address the problem of additionally checking the interoperability of this component with other components (i.e., protocol conformance). Protocol conformance testing is mainly known from testing communication systems [3, 28, 17, 9, 23] and SDL specifications [19, 12]. With our work we focus on interoperability and conformance of classes in an object-oriented programming environment based on UML descriptions.

For example, in Figure 2 the system under test (henceforth the component under test) is component `C1`. For testing `C1` we embed it in a test environment. The test environment is connected to the provided interface of `C1` to allow the sending of inputs to `C1`. The required interface of `C1` is also connected to the test environment to allow the observing of outputs of `C1`. In this test setting, we assume that the required interface of `C1` is intended to be connected to a provided interface of another component — in this example component `C2`. Without loss of generality, we demonstrate the approach with one interface for sending inputs and one interface for observing outputs of the component under test. The approach is also applicable to a larger number of interfaces.

The problem we address here is that component `C2` may require a special protocol at its provided interface. Connecting `C1` to `C2` is only possible if `C1` respects this protocol. The required protocol is specified by a protocol state machine — in Figure 2, namely `PSM-PI-C2`. To test whether `C1` respects this protocol we extended our conformance test approach by checking the observations of `C1` against the protocol state machine of `C2`. In particular, we test for every observation sequence made on `C1`, if the protocol state machine of `C2` can process these observations (i.e., can fire transitions triggered by these observations).

To identify failures in the specified protocol we need to define some semantic variation points of protocol state machines. The interpretation of the reception of an event in an unexpected situation (unexpected current state, violated state invariant or precondition) is a semantic variation point: the event can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error. It corresponds semantically to a precondition violation, for which no predefined behavior is defined in the UML standard. The interpretation of an unexpected resulting behavior, that is an unexpected result of a transition (wrong final state or final state invariant, or postcondition) is also a semantic variation point, that should be interpreted as an error of the implementation of the protocol state machine [30]. We interpret both, event reception in unexpected situations and unexpected behavior as violations of the specified protocol. Thus, we can give a precise definition of *protocol conformance*:

$$I \leq_{protocol} (S_b, S_p) \Leftrightarrow \forall \sigma : \mathrm{seq}\, E_{S_b} \bullet \forall \omega : out(I,\sigma) \bullet S_p \stackrel{\omega}{\Longrightarrow} \quad (6)$$

An implementation I conforms to a protocol specification S_p, if and only if for all input sequences σ over the event set E of a behavioral specification S_b, all output sequences ω in $out(I,\sigma)$ of the implementation, can trigger the protocol specification S_p. The fact that a sequence γ can trigger a state machine *SM* is defined as follows:

$$SM \stackrel{\gamma}{\Longrightarrow} =_{def} \exists [\![c_1,q_1,d_1]\!] \xrightarrow{in_1,out_1} \ldots \xrightarrow{in_{n-1},out_{n-1}} [\![c_n,q_n,d_n]\!] \bullet \; in_1 \frown \cdots \frown in_{n-1} = \gamma \quad (7)$$

Here, we identify *SM* with its initial status and require the existence of a computation of *SM*, such that the sequence of inputs of this computation is equal to the given sequence γ.

As an example, we demonstrate our approach to check protocol conformance by means of two exemplary test cases in the test set up presented in Figure 2 and Figure 1. First, we consider the test case from the previous example. We do not encounter a violation, since for the sequence `?aa?ab?bb?ba?aa?ab`, there exists a valid computation in `PSM-PI-C2`. Second, we choose `!a!b!b!b!a!b` as input sequence to `C1`. With respect to this input sequence we observe the sequence `?aa?ab?bb?bb?aa?ab` at the required interface of `C1`. If we send this sequence as input to `PSM-PI-C2` we encounter a violation. The sequence `?aa?ab?bb` is accepted by `PSM-PI-C2`, changing its state to state (`3`). In state (`3`), the state machine expects the reception of event `ba`. There is no transition that is triggered by the event `bb`. This is a violation of the specified protocol. Consequently, we can state that component `C1` and `C2` are not interoperable in an environment showing behavior (i.e., triggering `C1`) according to `PSM-PI-C1`.

Interpreting Test Results If we encounter a violation of the required protocol during protocol testing of a component A under test against the required protocol of a component B, then the general consequence is that it is impossible to connect component A with component B in the assumed

environment. There could exist several reasons why two components are not interoperable.

The simplest reason could be that the two *interfaces* do not fit. That means that a component A sends events to a component B which are not "understood" by B. In such cases, it is possibly feasible to use adapters to translate, abstract or put events into a concrete form [20]. Thus, from a technical point of view, interoperability could be made possible. More problematic is that a component A can show *behavior* at its required interface which is in general not interoperable with respect to the required protocol of a component B (independently from the way component A is used). Without changing the internal behavior of component A or without using more "intelligent" adapters, A and B cannot be connected. But the reasons could also be that the way component A is used leads to a violation of the required protocol of a component B. In the reverse, that means that for some inputs to A, interoperability with B is possible since the behavior of A for those inputs produces outputs that conform to the required protocol of B.

In particular from the latter reason it follows that the question of protocol conformance must always be seen in conjunction with the assumed environment. A component must not generally conform to the required protocol of another component. Only in the special situation that it should be connected to the other component, and only in the assumed environment. Consequently, the question whether we can restrict the general environment of a component A in such way that A meets the required protocol of a component B becomes immanent in this context. In general, an environment like that is not guaranteed to exist. Usually, domain experts must define which behavior at a provided interface must be or should be allowed, and thereby disallow input sequences that lead to a violation of a required protocol. This could be done by restricting the allowed protocol at provided interfaces (i.e., by restricting the behavior of the associated protocol state machines). Restricting a protocol state machine means restricting the set of valid input sequences.

For example, Figure 3 shows a possible restriction of `PSM-PI-C1` in the protocol state machine `PSM-PI-C1'` (we will explain the remaining picture in the next section). This protocol state machine allows less input sequences that are still valid with respect to the protocol state machine `PSM-PI-C1` (i.e., is a sub-behavior). The input sequence of our second test case (`!a!b!b`**`!b`**`!a!b`) is not a valid input sequence with respect to the protocol state machine `PSM-PI-C1'`. Instead, selecting inputs according to the restricted protocol description and testing a component under test only capable of showing behavior according to behavioral state machine `BSM-C1`, will not encounter a violation of the required protocol at the provided interface of `C2`. Note, `PSM-PI-C1'` does not describe the maximal

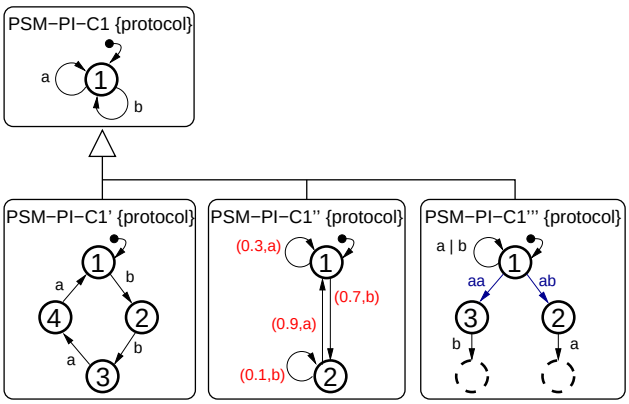

Figure 3. Specialized environments.

set of valid input sequences. In this simple example, we could have used the protocol state machine `PSM-PI-C2` at the provided interface of `C1` to describe the maximal set of valid inputs, whereat we have to replace `aa` and `ba` with `a`, and `bb` and `ab` with `b`.

These considerations lead to two further applications of the results of protocol conformance checks. First, we can use the results of protocol conformance tests in analyzes to specify valid environments for connected components and thus, by explicitly requiring the specification of valid environments, to support the assembling of a system out of pre-fabricated components. Second, it improves the motivation for selecting inputs according to restricted environments (i.e., to exclude disallowed or unwanted input sequences from the test case generation process). We discuss this subject in the following section.

4 Input Selection

Testing consists of executing experiments with the system under test. For these experiments we have to choose the inputs for the stimulation of the system under test. If the domains of the inputs are not finite, or if the number of values in the domains is pretty large, it is impractical to test with all possible values. Even in our case, where the number of events is finite, we have to deal with sequences of inputs which are not restricted in their length. This is due to the fact that most reactive systems are designed as non-terminating systems which continuously process inputs.

Various strategies are studied in the literature for selecting a finite number of test cases [2, 22, 1, 32, 11, 7]. They range from analyzes of the structure or the data-flow of systems under test, via dedicated fault models or explicit test case specifications to the idea of choosing test cases according to statistical data. All have their assets and drawbacks. Automated techniques allow selecting inputs in systematic and efficient way, while domain experts are able to select

"interesting" or "relevant" inputs, but mostly less systematic and with more time needed [2, 1].

We address the problem of selecting "interesting" or "relevant" input sequences within an automated input selection process. We use environment models to describe usage patterns of the system under test. A usage pattern can describe heavily used cases of the system under test, but also sequences of interest to achieve a special test purpose. The motivation for using such test case specifications is that we eventually use the results of testing the system under test to evaluate its quality. Therefore it is essential to execute adequate test cases. Since our environment models usually do not ensure finite behavior, we have to combine their usage with other test case selection strategies as cited at the beginning of this section.

In the previous sections we described that the protocol state machine associated with the provided interface of the component under test specifies the valid behavior of its environment. The component under test is assumed to or must work correctly in an environment behaving like this. Consequently, such protocol state machines specify the most general environments for which this component must work correctly. Hence, we can use it as a basis to select relevant input sequences. Benefits of doing so are that the graphical notation eases the understanding of the described behavior, and that it is possible to describe *sequences* of inputs. Compared to, for example, choosing input sequences only on the basis of the event set, invalid or unlikely input sequences can be avoided. This becomes especially necessary if it cannot be assumed that the system under test is input enabled (i.e., is not blocking for all inputs in all states). To use protocol state machines for input selection we execute them and select the next input according to the fire-able transitions. For example, in Figure 3, `PSM-PI-C1` describes the behavior of the most general environment. In the initial state `(1)` both transitions can be fired. Hence we can choose for the next input either `a` or `b` and thus input sequences containing `a`'s and `b`'s in an arbitrary order.

Further on, we restrict the behavior of an environment to select test cases according to specific system uses (i.e., according to a specific test purpose). For example, `PSM-PI-C1'` restricts the behavior of `PSM-PI-C1` in a way that a correct implementation of `C1` can comply with the specified protocol of component `C2`. It also forms the basis to select input sequences to test for protocol conformance. In the initial state of `PSM-PI-C1'` we can only choose `b` as the next input followed by `b`, then `a`, then `a`, then `b`, and so on. It is not possible to generate input sequences starting with `a`. Thus, in the context of connecting `C1` and `C2`, invalid input sequences for `C1` are avoided. The initial protocol state machine describes the most general environment in which the system under test is assumed to work correctly. So it follows that all restrictions must be a sub-behavior of the initial one.

We extend the protocol state machine notation by two variants to allow a finer description of environments: first, by using probabilities for choosing the next input from the set of possible inputs, and second, by using feedback from the system under test to allow adapting the behavior of the environment according to this feedback. The latter is used when testing non-deterministic systems on-line.

4.1 Input Probabilities

A statistical test case generation usually aims at selecting data values for input variables using a statistical distribution. In model-based testing it is also used to generate input sequences from environment models. For example, Markov chain models are widely used to specify usage profiles [18]. This is especially useful as the system under test moves from one state to another one and thus, the probability of applying an input can change. Whittaker and Thomason [31] proposed an approach for test input selection based on usage profiles described by Markov chains. They use finite states, discrete parameters, time homogeneous Markov chains. We transfer the representation of Markov chains as finite state machines with probabilities attached to the transitions to our protocol state machines describing the behavior of the environment. This allows to express that in some states some inputs are more likely than others. It also allows to use all the theories around Markov chains to perform analyzes of the testing process [8, 18].

In the previous section we selected inputs according to fire-able transitions. There all transitions are equiprobable. We can only express that in some states it is not possible to choose some inputs (i.e., their probability to be chosen is equal to zero). To allow the expressing of varying probability distributions in different states, we extend the label notation of protocol state machine such that it is a tuple (p_i, i), comprising a real value p_i (i.e., the input's probability) and the input i. The value for a p_i must be between zero and one ($0 < p_i \leq 1$) and the sum of all input probabilities in a state must be equal to one ($\sum p_i = 1$). Thus, input sequences can be generated by traversing the protocol state machine, where the random choice of the next transition (i.e., input) is made using the probability distribution of the outgoing transitions.

For example, `PSM-PI-C1''` in Figure 3 uses this extended labeling to specify different probability distributions. In state `(1)`, choosing as the next input `a` has a probability of 0.3. Choosing as the next input `b` has a probability of 0.7. Consequently, if we select input sequences for several test cases, input sequences starting with `b` are more likely than input sequences starting with `a`. Hence, this behavior is tested more intensively than others (which was the intention of using this profile). In particular, the la-

bels of the two transitions in state `(1)` form two intervals, in fact $[0,0.3)$ for `a` and $[0.3,1)$ for `b`. Thus, for implementing this strategy we just need to choose a random number p with $0 \leq p < 1$. Given a random number of 0.56, we would choose for the next input a `b`. If the random number is uniformly distributed the order of the particular intervals does not influence the specified probability distribution. It is only required that the order at a state is fixed.

With this label extension, we are not only able to specify the valid behavior of environments, we are also able to specify which parts are more likely than others. Consequently, we are able to select "interesting" test cases with respect to our intended test purpose (e.g., a specific usage profile).

4.2 Using Feedback

Observing the behavior of environments and users in practice shows that their behavior changes depending on the reactions of the system. A common example for that is the behavior when doing a phone call. If you lift the receiver, the probability that you will dial a number is dependent of hearing the dial tone. Dialing a number is more likely if you hear the dial tone, or, hang up the receiver is more likely if you hear the busy tone. Therefore it would be advantageous to use such information for test input selection.

Note that if the system reaction to all inputs is deterministic, there is no need to analyze the system reactions for input selection. From the previous input it exactly follows in which state the system under test resides (related to the specification). Therefore, we use feedback information only for systems which are non-deterministic in their observable reactions to inputs.

To consider feedback from a system under test, we again slightly change the labels in protocol state machines. We differentiate two disjoint subsets among the label set. The first subset contains all inputs to the system under test, including input events as well as input events extended with probabilities. The second subset contains all reactions of the system under test (i.e., all possible observations at its required interface). When we traverse such extended protocol state machines to generate input sequences we have two options in each state. We can either choose to select the next input as described in the previous section or we can process output of the system under test.

Protocol state machine `PSM-PI-C1'''` in Figure 3 shows the principle of this strategy. In state `(1)` we could choose `a` or `b` as the next input. When we trigger the system under test and observe `ab` as reaction we change to state `(2)`. If the system reaction is `aa` we change to state `(3)`. For the next input we can choose `a` or `b` as the next input depending on the actual state.

The described strategy requires to process system reactions during test input generation. However, a specific system reaction is only available during run-time (i.e., when executing the system under test). Our current off-line test generation approach calculates all possible correct system reactions for a given input sequence and then continues with the next input sequence. Considering feedback in this off-line process would consequently require to consider all possible system reactions to each input separately. The resulting test case would have a tree structure with determined system reactions on each path. The effort needed to calculate such test cases would be enormous.

The idea of using feedback is similar to classical on-line-testing approaches (also known as on-the-fly testing) [10, 6]. In these approaches, test cases are generated at run-time. The exploration of the specification's state space is controlled by the reactions of the system under test. Only these paths are further processed which show the system reactions so far. All the others are discarded. We carry this idea over to our test approach and the input selection with protocol state machines. In classical on-the-fly testing, the feedback is mainly used to avoid state space explosion in the computation of the test oracle, i.e., for test evaluation. In our approach, we do not only facilitate test evaluation, but also use feedback to generate valid input sequences. This is not straight forward for non-deterministic systems, as it may depend on system under test's behavior, which inputs are processable in the next step. With the extension presented here, we can generate more specific and valid input sequences in such a test set-up. Currently, we use feedback information during on-line testing of requirements [16]. In this approach we continuously trigger the system under test and check the system reactions against the explicitly modeled requirements. The inputs to the system under test are selected on the basis of our extended (protocol) state machines.

5 Conclusion

In our approach UML behavioral state machines are used in quality assurance to serve as a formal specification for the desired reactive behavior of the system. It is possible to select relevant and interesting inputs for a test case and to calculate the possible correct observations for given inputs. They allow to automatically evaluate test executions which is in general a difficult and time consuming task. With both extensions to our test approach we still focus on conformance testing at the level of unit testing.

The first extension allows to check the interoperability of the component under test with other connected components based on a precise definition of protocol conformance. We use UML protocol state machines to specify the protocol at the provided interface of a connected component, and check the outputs of the component under test at its required interface against it. If the component under test

respects the specified protocol of the connected component, we call them interoperable. During testing we check the observations of the component under test not only against the pre-calculated test oracle but also against the protocol state machine of a provided interface of a connected component. We only need to check whether there is a legal transition for the observed outputs. If not, a violation related to the interaction between these components can be reported. This is done fully automatically and with relatively small extensions to the current framework.

The second extension allows to use protocol state machines as test input specifications. By restricting the behavior at the provided interface of the component under test, the set of possible input sequences can be restricted and thus, relevant input sequences can be specified. Input sequences are then selected in combination with classical selection strategies. The two extensions of input probabilities and the interpretation of feedback of the system under test allow to describe the desired behavior on a more precise level. Thus, we can set the focus of the test process to a specific test purpose.

We use UML state machines as a unified notation for behavioral and protocol conformance testing as well as for test input selection. This considerably eases the work of test engineers. To show the general applicability we implemented both extensions in our TEAGER tool suite [27, 24] and applied a case study of a sun blind control [16].

Our approach is also applicable for more comprehensive protocol state machines. We do not restrict protocol state machines to a specific subset. But the interpretation of some notations is not straightforward and needs more experience with the presented approach. For example, in Section 2 we described that protocol state machines can also have orthogonal regions, or pre- and postconditions at transitions and state invariants. From orthogonal regions it follows that the protocol state machine is in multiple active states at a time. We interpret this in such a way that for the next step the behavior (i.e., input) enabled in every region can happen. This means for input selection that we can choose the next input depending on the possibilities in every region. But this is not clear in all situations. Further research must address this problem in more detail. Another problem is the interpretation of the mentioned predicates. They are defined on the basis of the state space of the component the protocol state machine is associated with, or on the basis of the data events or parameters can carry along. Since a protocol state machine only specifies these predicates and does not execute any code to define the resulting behavior when taking a transition, these predicates cannot always be evaluated (e.g., if they relate to the connected component). Thus, when these predicates should be taken into account, the manipulation of data (i.e., the evaluation of the associated behavior in the component) must also be taken into account. Therefore, we intent also to execute the component's behavior as far as possible for the future. In future research we also address the problem of automatically identifying the valid behavior of the environment of the component under test, if the component is connected to other components. Further research should also study input selection related to complex data. We want to select test cases including adequate data to cover as much as possible of this relevant behavior.

References

[1] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

[2] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 1999.

[3] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. *Lecture Notes in Computer Science*, pages 187–195, 2001.

[4] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[5] S. Chouali, M. Heisel, and J. Souquières. Proving Component Interoperability with B Refinement. *Electronic Notes in Theoretical Computer Science*, pages 157–172, 2006.

[6] R. G. de Vries and J. Tretmans. On-the-fly Conformance Testing using SPIN. *STTT*, pages 382–393, 2000.

[7] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 1984.

[8] W. Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, 1968.

[9] J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, pages 123–146, 1997.

[10] J.-C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Computer Aided Verification*, pages 348–359. Springer Verlag, 1996.

[11] J. B. Goodenough and S. Gehart. Towards a Theory of Testing: Data Selection Criteria. In *Current Trends in Programming Technology*, pages 44–79. Prentice Hall.

[12] J. Grabowski and D. Hogrefe. SDL- and MSC-based Specification and Automated Test Case Generation for INAP. *Telecommunication Systems*, pages 265–290, 2002.

[13] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 1987.

[14] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, pages 293–333, 1996.

[15] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering*. Addison Wesley, 2001.

[16] M. Heisel, D. Hartebur, T. Santen, and D. Seifert. Using UML Environment Models for Test Case Generation. In *Software Engineering 2008 - Workshopband*, Lecture Notes in Informatics, 2008.

[17] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[18] J. G. Kemeny, J. L. Snell, and G. Thompson. *Finite Markov Chains*. Springer-Verlag, 1974.

[19] A. Kerbrat, T. Jéron, and R. Groz. Automated Test Generation from SDL Specifications. In *SDL Forum*, pages 135–152, 1999.

[20] A. Lanoix and J. Souquières. A Trustworthy Assembly of Components using the B Refinement. *e-Informatica Software Engineering Journal*, 2008.

[21] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation: Specification and Verification. In *Workshop on Component Oriented Programming (ECOOP 2006)*, 2006.

[22] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-based Tests. In *ICECCS*. IEEE Computer Society, 1999.

[23] K. K. Sabnani and A. T. Dahbura. A Protocol Test Generation Procedure. *Computer Networks*, pages 285–297, 1988.

[24] T. Santen and D. Seifert. Teager - Test Automation for UML State Machines. In *Software Engineering 2006*, LNI P-79, pages 73–83. GI, 2006.

[25] D. Seifert. *Automatisiertes Testen asynchroner nichtdeterministischer Systeme mit Daten*. Shaker Verlag, 2007. Also: PhD dissertation, Technische Universität Berlin.

[26] D. Seifert. Conformance Testing based on UML State Machines. Technical Report inria-00268864, DEDALE (LORIA), 2008.

[27] D. Seifert. The TEAGER Tool Suite. Test Execution and Generation Framework for Reactive Systems, 2008. `swt.cs.tu-berlin.de/~seifert/teager.html`.

[28] X. Sun, C. Feng, Y. Shen, and F. Lombardi. *Protocol Conformance Testing Using Unique Input/Output Sequences*. Advanced Series in Electrical and Computer Engineering. World Scientific, 1997.

[29] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[30] UML2. Unified Modeling Language: Infrastructure and Superstructure. Object Management Group, 2007. Version 2.1.1, formal/07-02-03, `www.uml.org/uml`.

[31] J. A. Whittaker and M. G. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Transaction on Software Engineering*, pages 812–824, 1994.

[32] Zhu, Hall, and May. Software Unit Test Coverage and Adequacy. *Computing Surveys*, 1997.