Introduction to **Model Based Testing** and the **Axini Modeling Language***

Theo Ruys
Axini B.V.
https://www.axini.com
ruys@axini.com

October 2021

Abstract

This paper introduces the reader to modeling reactive systems using labeled transition systems, a well known state-transition formalism. Labeled transition models have certain characteristics that makes it possible to use them for Model Based Testing.

Model Based Testing (MBT) is a software testing technique where the run time behavior of a system is checked against predictions made by a model. A model is a description of a system's behavior. A considerate part of the paper is devoted to the Axini Modeling Language (AML), the modeling language of the Axini Modeling Platform (AMP), a state-of-the-art MBT tool to automatically test and analyse reactive and control-oriented systems.

1. Introduction

Reactive or control-oriented systems are systems which react on input and evolve: outputs do not only depend on the current input but also on earlier inputs; in control-oriented systems the set of available operations depends on the current state of the system. Typically, such systems are meant to be executed non-stop and will never terminate. Examples of control-oriented systems are communication protocols and multi-threaded systems with several interfaces.

Due to the concurrent and non-terminating nature of reactive systems, such systems are hard to test. In practice, manually constructed sequences of actions or semi-automatic scripts are used to test specific scenario's of the system under test (SUT). The construction of such scenario's is hard. For example, when do you know that you have tested enough? Or, how many actions should a scenario consist of? Furthermore, when the SUT is changed, all test scenario's have to be (manually) updated as well. Moreover, the maintenance of these test scenario's is problematic as well: how to keep track of the different versions of the SUT and the test scenario's?

Model Based Testing (MBT) is a technique where the behavior of the SUT is represented by a formal model. This model is used to automatically and systematically generate scenario's to test the correctness of the SUT. Coverage criteria on the model steer the generation of test scenario's.

*This paper is based on Chapter 13 'State Transitions Models' of the Open University workbook 'Software Testing' [6]. We have tried to remove all references to the original workbook. We apologize for any idiosyncrasies that are still present in the text. Furthermore, Section 2 is heavily based on the work of Jan Tretmans [5].

For the reactive systems in this paper, the SUT is treated as a black-box exhibiting behavior and interacting with its environment, but without knowledge about its internal structure. The only way a tester can control and observe an implementation is via its *interfaces*. The aim of testing is to check the correctness of the behavior of the SUT on its interfaces [5]. These interfaces are not limited to GUIs, but can be any communication interface, e.g., I/O interfaces (standard input/output, file systems, middleware, etc.), API calls, etc.

The fundamental assumption of MBT is that the implementation of the SUT is based on a *specification*, which explicitly describes what the SUT should do. A specification is typically a collection of documents which states in detail how the SUT should behave; such a specification is the blueprint for the development and implementation of the SUT. This specification is also the source of information for the formal model of the system: the model is thus an abstract description of the SUT. Testing then amounts to checking whether the behavior of the SUT is 'equivalent' to the behavior as specified by the abstract model.

The MBT approach is the result of more than three decades of scientific research. The MBT approach has a strong mathematical basis; many theoretical papers and descriptions are available which formally define the test derivation algorithms and prove their correctness. In this paper we introduce MBT by example and only touch upon the formal foundations of MBT, and we are (sometimes very) informal to ease the presentation. For a more formal introduction the interested reader is referred to, e.g., [5].

Overview. In § 2, we will first discuss labeled transition systems, the formal basis for our models. In § 3, we will discuss the main concepts of MBT and its advantages and disadvantages. § 4 describes the basic constructs of the Axini Modeling Language (AML) and shows how these constructs are mapped upon labeled transition systems. After reading this paper the reader is expected to be able to construct AML models for small reactive systems. The paper contains several exercises that can be used to test your understanding of the text.

2. Labeled Transition Systems

A state machine model – or state diagram – is used in computer science and related fields to describe the behavior of reactive systems. State models describe the *states* that a system can have and the actions under which the system changes state: the *transitions*. This is why they are sometimes also called state-transition models. We will use all these terms interchangeably.

This section introduces labeled transition systems, a well-known formalism for state-transition models. Several simple examples of such systems will be presented. We will discuss traces, test cases and what it means for a SUT to conform to a model. Coverage criteria for models will also be presented.

A labeled transition system (LTS) is a graph structure where the vertices represent states and the edges represent labeled transitions. The *states* model the states of the system. The *labeled transitions* can model:

- *outputs*, i.e., the actions that the system can perform, and
- *inputs*, i.e., the actions under which the system changes state.

The labels on the transitions represent the observable actions of the system; they model the system's interactions with the environment.

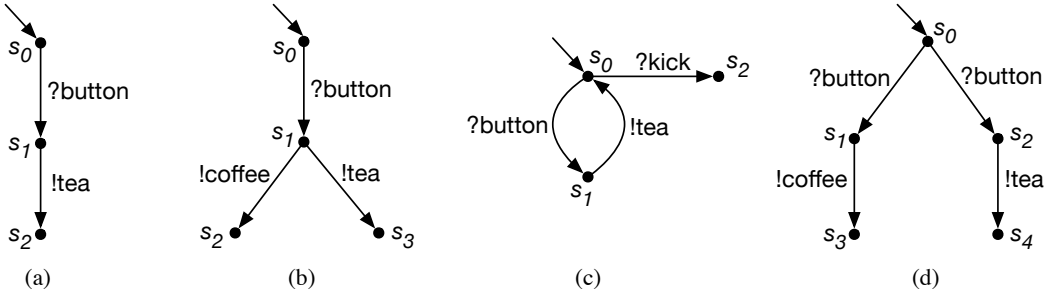


FIGURE 1 Some Labeled Transition Systems.

Some of the advantages of labeled transition systems is that they are easy to draw and it is intuitively clear what the semantics of the system is. Figure 1 shows some examples of LTSs. The bullets represent the states and the arrows between the bullets are the transitions. Each label is either prefixed with a question mark (?) or an exclamation mark (!): a question mark ? represents an input to the system, an exclamation mark ! represents an output from the system. The *start state* of the system is the state with an ingoing arrow which does not have a source state and does not have a label. A state without an outgoing arrow is called a *final state*.

Let us look at the examples of Figure 1 in more detail. Figure 1 (a) describes a system that can deliver a cup of tea. The system has three states. From the start state s_0, a ?button can be pressed and the system goes to state s_1. From state s_1 the system can do a !tea action. After the !tea action, the system goes to the final state s_2, and nothing can happen anymore. Figure 1 (b) describes a system which can deliver coffee or tea, non-deterministically. After receiving the ?button, the system can either deliver !coffee or !tea. Figure 1 (c) is a continuous version of Figure (a): after delivering the tea, the button can be pressed again to deliver tea. If the system is ?kick-ed, however, the system goes out-of-order. Figure 1 (d) is a variant of (b): it also describes a system which can deliver !coffee or !tea, non-deterministically. The difference with (b), though, is that the decision is already made when the ?button is pressed.

Formally, we can define an LTS as follows:[1]

DEFINITION 1
A labeled transition system (LTS) is a tuple $\langle S, L, T, s_0 \rangle$, where
- S is the non-empty set of states,
- L is the set of transitions, with $L = L_I \cup L_O$ and $L_I \cap L_0 = \emptyset$
  - L_I is the set of input transitions
  - L_O is the set of output transitions
- T the transition relation, and
- s_0 is the initial (or start) state.

The labels in L_I and L_O represent the observable actions of a system. A synonym for input is *stimulus*. Likewise, a synonym for output is *response*. The transition relation T defines the structure of the LTS: it connects the states using transitions: $T \subseteq S \times L \times S$.

If we look at Figure 1 (a), the LTS can be formally defined as

$\langle \{s_0, s_1, s_2\}, \{?button\} \cup \{!tea\}, \{(s_0, ?button, s_1), (s_1, !tea, s_2)\}, s_0 \rangle$.

[1] Tretmans [5] takes a more formal approach and distinguishes between labeled transition systems (where the transitions do not have a direction) and input-output transition systems (where the transitions are divided into input- and output transitions).

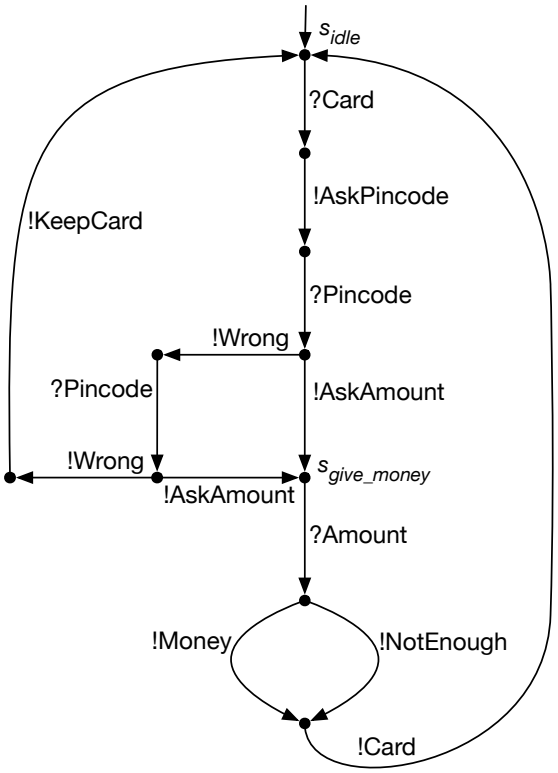


FIGURE 2 LTS of a cash dispenser.

Now let us look at a larger example. Figure 2 shows an LTS of a *cash dispenser*. The user of the cash dispenser can insert a bank card (input ?Card). After inserting the card, the user is asked for the pincode of the card (output !AskPincode). If the entered pincode (input ?Pincode) is correct, then the user is asked how much money is required (output !AskAmount). If the user has enough credit, the money is dispensed (output !Money), otherwise a message is displayed that there is not enough money in the account (output !NotEnough). In both cases, the cash dispenser will return the card (output !Card). If the pincode is wrong, the machine will permit one other attempt to enter the pincode (input ?Pincode). If the pincode is incorrect again, the cash dispenser will issue a message that the entered pincode is wrong (output !Wrong) and the card will be kept (output !KeepCard). Note that we have abstracted away from many details, e.g., the pincode, the checks, the amount of money asked for, the messages on the display of the cash dispenser, etc.

Labeled transition systems constitute a powerful semantic model to reason about the behavior of processes. However, except for the most trivial processes, a representation by means of a state-transition graph is usually not feasible [5]. In Sec. 4 we will introduce AML, a modeling language whose semantics is defined in terms of LTSs. AML allows us to model processes with hundreds of states and transitions.

EXERCISE 1

In this exercise you are asked to define an LTS for a `Stack` machine. The `Stack` can hold a maximum of three elements. The `Stack` accepts two stimuli: push and pop. We abstract from the actual values that are being pushed to and popped from the stack. The `Stack` machine has two responses: value and error. After a pop stimulus, the `Stack` responds with a value label. When the `Stack` machine receives a pop stimulus when there are no more elements, the `Stack` machine outputs an error label. Similarly, when the `Stack` is full and the machine receives a push label, the `Stack` also outputs an error label. Draw an LTS for this `Stack` machine.

2.1 Conformance

Having a formal semantics of a modeling language has an important benefit: we can build a tool which translates a model to its state-transition semantics. This is similar to the way a compiler of a programming language translates a computer program to machine code which can be executed. Subsequently, we can use the generated state-transition system to automatically generate test cases.

A *trace* in an LTS is a sequence of labels representing the transitions that the LTS can do, starting from the start state of the LTS. We denote a trace by the list of labels enclosed in angle brackets ($\langle\rangle$). Given an LTS L, the set **traces**(L) is the set of all possible traces of L.

Consider Figure 1 again.

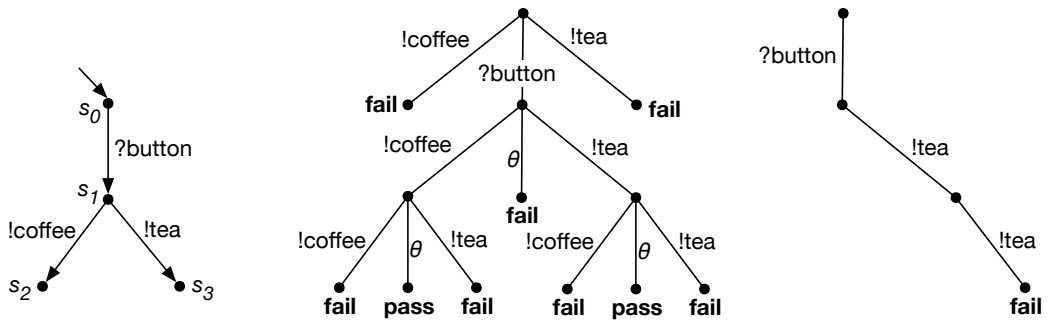
- In (a) there are three possible traces:
  $\langle\rangle$ (i.e., the empty trace),
  $\langle$?button$\rangle$, and
  $\langle$?button, !tea$\rangle$.
  So: **traces**$(L) = \{\langle\rangle, \langle$?button$\rangle, \langle$?button, !tea$\rangle\}$.

- For (b), we have:
  traces$(L) = \{\langle\rangle, \langle$?button$\rangle, \langle$?button, !coffee$\rangle, \langle$?button, !tea$\rangle\}$.

- For (c), the set $traces(L)$ is infinite:
  traces$(L) = \{\langle\rangle, \langle$?kick$\rangle, \langle$?button$\rangle, \langle$?button, !tea$\rangle, \langle$?button, !tea, ?kick$\rangle, \langle$?button, !tea, ?button$\rangle$, $\langle$?button, !tea, ?button, !tea$\rangle, \ldots\}$.

We assume that the SUT can be modeled as an LTS and that the input and output actions of the SUT are the same as specified in L_I and L_O of the LTS. In practice, though, the physical labels of the SUT have to be translated to the logical labels of the model, and vice-versa.

A test of the SUT is an experiment consisting of supplying stimuli to the SUT and observing its responses. The specification of such an experiment, including both the stimuli and the expected responses, is called a *test case*. A set of test cases is called a *test suite*. When we use an LTS model to generate the test cases, this is frequently called *conformance testing*, since it involves assessing whether a SUT conforms to the model. The process of applying test cases to the implementation of the SUT is called *test execution* [5].



(a) LTS (b) test case (c) test execution

FIGURE 3 LTS, a test case and a test execution.

Executing a test case results in a *execution trace* of the SUT, which either corresponds to a trace of the LTS or not. Test execution may be successful, meaning that the observed responses of the SUT correspond to the expected responses of the test case: any output produced by the SUT has been foreseen in the LTS. We say that the test *passes* and the execution trace will correspond to a

trace in the LTS. Test execution may be unsuccessful: when executing the stimuli from the test case, we observe a response from the SUT that is not an expected response from the test case. We say that the test *fails*. In this case the execution trace will not correspond entirely to a trace in the LTS: the last observed response does not correspond to a transition of the LTS.

A test case thus describes a set of correct and incorrect traces and specifies the verdict (pass or fail) at the end of the observed trace. A test case can thus be represented by a tree, where each edge is either a stimulus or a response, and ends with a verdict: pass or fail. For example, recall the LTS in Figure 3 (a). Figure 3 (b) (taken from [1]) is a test case which describes all traces for this LTS. The passing traces all start with a ?button stimulus, followed by either a !coffee or !tea response, and then nothing more. Observing nothing – or the absence of a response – is named *quiescence*. This is represented by θ: it means that we do not observe any response from the SUT. The transition θ will be taken if none of the output responses can be observed. Testing for quiescence is usually implemented as a time-out: if we do not observe any response after a certain amount of time, we conclude that no responses will be generated.

Figure 3 (c) shows a possible execution of the test case at the SUT: ⟨?button, !tea, !tea⟩. After a ?button stimulus, the SUT responds with !tea and then !tea once again. The test execution is a trace in the test case of Figure 3 (b), leading to a **fail** verdict, because the second !tea output is not allowed in the LTS. In other words, the observed execution trace is not a trace of the LTS, and the test fails. Note that if we would execute the test case again, we might observe another test execution, e.g., ⟨?button, !coffee, θ⟩, which represents a passing test case.

Most reactive systems are supposed to work uninterrupted: they are designed to never stop. The set of traces of such systems is clearly infinite and we have already seen that it is impossible to define a test case containing all possible traces. We can still test the machine up to a predefined depth n, though: we limit the test cases to that depth of n labels. If a test execution still conforms to the test case after n steps, we consider the test case passed up to depth n.

For industrial-sized systems the test case will often be too large due to the large number of possible non-deterministic stimuli, even if we limit the test case vertically to a predefined depth of n steps. To limit the test case horizontally we can leave out certain stimuli (in parts of) of the LTS. The choice of stimuli to be included in the test case is driven by the intended coverage of the model.

EXERCISE 2
Consider the LTS of the `Stack` machine of Exercise 1. Because the behavior of `Stack` machine is infinite, we cannot define a complete test case. Draw a test case for the `Stack` machine up to depth 4, i.e., 4 labels deep.

EXERCISE 3
The test case of Exercise 2 includes all traces of depth 4. Consequently, it does not include the interesting trace ⟨?push, ?push, ?push, ?push, !error⟩ of length 5. Draw a test case for the `Stack` machine up to depth 5, where a ?pop stimulus is only allowed after two ?push stimuli.

2.2 Coverage

We need a means to evaluate the quality of a test suite, i.e., a set of test cases. Coverage criteria indicate how much of the specific parts, behavior or characteristics of the SUT have been exercised (i.e., covered) during a test. For example, for an implementation, *statement coverage* defines the percentage of the statements that have been executed by the test. For MBT, we consider the SUT to be a black-box, so we do not have the source code available to evaluate the coverage. Fortunately, the LTS model that describes the SUT can be used to define coverage criteria:

- *State coverage*: how many states of the model are covered by the test suite?

- *Transition coverage*: how many transitions of the model have been covered by the test suite?
- *Trace coverage* : comparing the number of traces executed by the test suite to the complete set of traces is not very informative: the total number of different traces far exceeds the number of traces that can be generated and executed in practice. On the other hand, (sub)paths within the LTS can be of interest. That is, for example the n-switch coverage of a test suite, which tries to include all (sub)paths of length n (from the start state or from each state) in the generated test suite.
- *Data coverage*: in the examples that we have seen so far, the labels of the LTS did not carry any data: they were just abstract names for actions of the system. In practice, though, we will usually add more details to the labels, including data values. In Section 4 on AML we will see examples of this. For such labels, we can use the data coverage criteria like for example input domain modeling with equivalence classes or input domain boundaries [6], to obtain and increase the data coverage.

Coverage is usually specified as a percentage. For example, state coverage of 73% means that 73% of the total number of states of the LTS have been covered by the test. Without the source code of the SUT we do not have any idea how much of the implementation of the SUT has been covered, though.

Consider Figure 1 (c). If we never ?kick this tea machine, and only push the ?button to get tea, both the state coverage and the transition coverage will be 66% as the transition labeled with ?kick and state s_2 will not be covered.

Both state and transition coverage of the LTS model might not say much about the coverage of the complete SUT. The model may be incomplete in the sense that functionality of the SUT has been left out or certain details have been abstracted from. For example, suppose that the SUT is a multi-beverage machine which – apart from tea and coffee – can also deliver soup, hot chocolate, etc. The models that we discussed so far only used tea and coffee beverages. So, using a test suite with 100% state and 100% transition coverage of the model would not test all functionality of this multi-beverage machine.

From a dynamic point of view, coverage of (sub)paths might be much more interesting than the static coverage of states and transitions. Still, even with these reservations, state and transition coverage should be as high as possible.

EXERCISE 4

Consider the LTS of the `Stack` machine of Exercise 1 again. Suppose we would execute the following three tests:

- ⟨?push, ?pop, !value⟩
- ⟨?push, ?push, ?pop, !value, ?push⟩
- ⟨?push, ?push, ?push⟩

What is the state coverage of these three tests? What is the transition coverage of these tests?

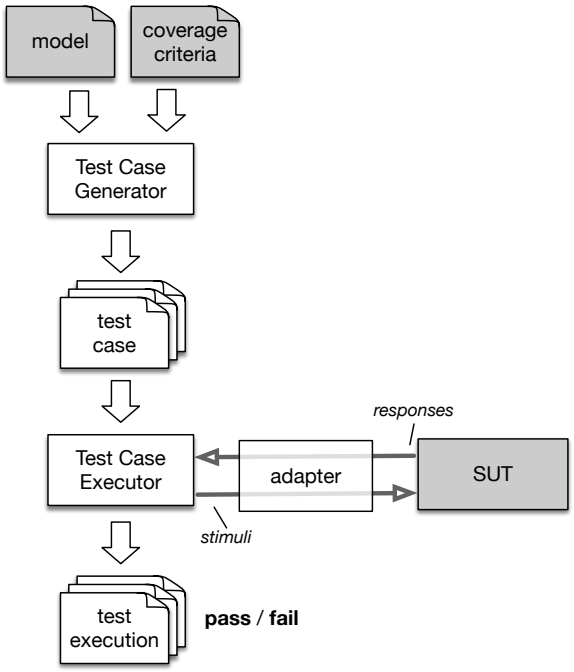


FIGURE 4 Offline Model Based Testing.

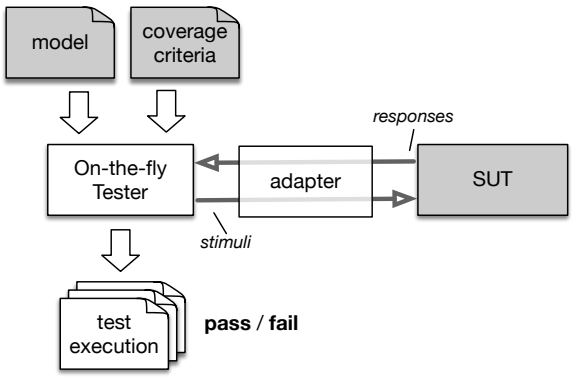


FIGURE 5 Online Model Based Testing.

3. Test Case Generation from LTS

Model Based Testing (MBT)[2] is an automated testing technique where we use a program to generate the test cases. The MBT tool accepts a model in a formal notation with a formal semantics, e.g., labeled transition systems. In the tool one can typically select one or more coverage criteria. The tool then automatically generates test cases given the model and the selected coverage criteria.

Figure 4 gives an overview of an MBT-approach. Given the model, the test generator generates a test suite of test cases. The test executor then attempts to execute each test case on the SUT:

- If the label in the trace is a *stimulus*, the adapter transforms the stimulus to a physical label and offers the physical label to the SUT.
- If the label in the trace is a *response*, the test executor waits until it receives the logical response from the adapter.

If the SUT returns a response which is not expected in the test case, it concludes that the test has failed. If the test executor can execute a complete trace of the test case against the SUT, it concludes that the test has passed. Figure 4 is regarded as an *offline* method: the set of test cases is generated *before* the actual testing takes place at the SUT.

Figure 5 shows an alternative MBT approach: *online* testing. Given the model, the *on-the-fly Tester* computes the LTS of the processes. Test cases are generated on-the-fly and partly, while testing the SUT. The *Tester* walks over the LTS-es of the processes of the model offering stimuli to the adapter and observing the responses of the SUT. The *Tester* decides upon the next test-step after

[2] In [6], all testing activities are regarded as 'model based testing': to test software, one has to make a model of the software, and hence all software testing is model based. In research literature, the testing approach based on state-transition models is usually called *Model-Based Testing* (MBT) or *Conformance Testing*. In [6], the term *Automatic Test Case Generation* (ATCG) is used for the automatic generation of test cases on the basis of state-transition models.

```
1  Algorithm online-test-generation(M, n) : return pass / fail
2     M: model of the SUT as an LTS
3     n: maximum length of the execution trace
4     current_states ← {s0 of M}
5     σ ← ∈
6     while |σ| < n do
7        action ← response_or_stimulus(current_states)
8        case action
9           RESPONSE:   try
10                         observe SUT's response x!
11                         σ ← σx!
12                         return fail if σ ∉ traces(M)
13                         current_states ← advance(current_states, x!)
14                      catch a response has not been observed before timeout
15                         return fail
16                      end
17          STIMULUS:   a? ← choose_stimulus(current_states)
18                      offer stimulus a? at the SUT
19                      σ ← σa?
20                      current_states ← advance(current_states, a?)
21       end
22    end
23    return pass
```

FIGURE 6 Pseudo algorithm for online test generation.

a stimulus or response (as opposed to generating all test-steps before testing). Typically, state and transition information is recorded during a test execution such that – e.g., for a next test execution – the *Tester* can decide what stimulus to choose to increase either the state or transition coverage of the model, or even both.

3.1 Algorithm for online test generation

Figure 6 sketches a pseudo algorithm for online test generation. The input for the algorithm is a model M of the SUT, represented by a LTS and n, the maximum length of the (execution) trace. The algorithm returns either *pass* or *fail*.

The variable *current_states* holds the possible states of M that the model is currently in. Due to the non-deterministic nature of the model, this is not a single state, but a set of states.[3] The variable σ holds the (execution) trace we have witnessed so far. As long as the length of σ is smaller than n we will append either a response or stimulus to σ.

Based on the set of current states, the procedure response_or_stimulus decides whether we have to wait for a response or have to do a stimulus. If the decision is a *response*, the *Tester* will wait for an output from the SUT. When a response $x!$ is observed, this response $x!$ is appended to σ and we check whether the new σ is still a valid trace for M. If not, we conclude that the test has failed. If σ is still a valid trace, we update *current_states* on the basis of $x!$. It is possible, however, that after waiting for some time, no response is observed. If that is the case, the test should fail as well: a response that was expected has not been observed.

[3] For example, in Figure 1 (d), after the stimulus ?button, the LTS can be either be in state s_1 or s_2.

In the case of a *stimulus*, the method choose_stimulus chooses one of the possible stimuli, based on the current set of states. This can be a random choice but could also be based on earlier decisions when this set of states was reached; either in this test run or during a previous test run. The adapter offers the chosen stimulus $a?$ to the SUT. Both σ and *current_states* are updated on the basis of $a?$. Note that we do not have to check that the new σ is a valid trace: as $a?$ is chosen from the current of states of M, the added $a?$ will always be a valid choice.

The pseudo algorithm of Figure 6 is neither correct nor complete. For example, what should happen when the SUT issues an unexpected response just as we want to offer a stimulus? Or what if the model M 'ends' before σ has grown to length n? And how do the strategies to increase the coverage of M fit into the algorithm? And how does the method choose_stimulus chooses a stimulus on the basis of previous test runs? Still, we hope that the algorithm provides some intuition on how online test generation works in practice. In fact, this algorithm forms the basis of the actual test generation algorithm found in AMP. For alternative algorithms for offline and online test generation the reader is referred to [4] and [5].

3.2 Discussion on MBT

The biggest advantage of MBT is the automated nature of it. After developing a model of the SUT and the adapter to connect to the SUT, the actual testing process is fully automated. Given coverage criteria, the MBT tool automatically generates test cases for the SUT. MBT thus allows the automatic production of large and provably sound test suites [5]. The smart coverage strategies (state, transition, n-switch, data) of the MBT-tool ensure that we can stop testing when a certain level of coverage has been reached. Another important benefit of MBT is that it supports the development cycle of the SUT more naturally than manually constructed test suites. In case of (evolutionary) changes to the SUT, retesting the SUT with MBT is relatively easy. In most cases only the model has to be changed in accordance with the changes to the SUT, and new test cases can be generated by the tool. No test scripts have to be manually changed. This makes MBT well-suited for iterative development approaches like Agile and Scrum.

Crucial to the MBT approach is the *adapter* which translates logical labels of the model to physical labels of the SUT and vice versa. If we consider our coffee machine of our previous example, the adapter has to translate the stimulus ?button to an actual press on the button. Furthermore, the adapter has to observe the deliverance of !tea and !coffee (in a cup). Fortunately, even without MBT, the developers of a SUT need to test their system themselves. So usually the manufacturer has provided a testing interface to the system to analyse and diagnose the system. Such an interface can be used to connect the adapter as well. Still, the development of an adapter can be expensive and can be a considerable part of the MBT effort.

MBT in practice. The most important activity of software testing is the development of the models which will be used for testing. This is not different for the MBT approach. Typically, several errors are already found during this modeling phase. The specification on which the model is based (and which is used for the development of the SUT) is often ambigious and/or incomplete. To develop a precise model such specification issues have to be fixed. Furthermore – for the online MBT approach – we have witnessed that almost all bugs in the SUT are found by random testing. That is, simply walk over the model, randomly select the stimuli, and check whether the observed responses conform to the model. When no more bugs are found with random testing, we will switch to a testing strategy in which the various coverage criteria are exploited to steer the test cases in order to improve the test coverage of the model. Futhermore, longer and deeper test cases are generated. Remarkably, in practice, usually only a few more bugs are found in this last phase.

So although coverage criteria are input for the MBT approach, most bugs are being found during modeling and random testing, which do not use these coverage criteria.

4. Axini Modeling Language

Pure state-transition models are tedious to construct and maintain by hand. Therefore, we typically use a modeling language which has a higher abstraction level. The semantics of this high-level modeling language is then defined in terms of a state-transition system. The high-level modeling language has thus a formal semantics: for every construct of the language it is formally defined to what state-transition model this is mapped.

As an example of a state-transition modeling language, this section describes the basics of the Axini Modeling Language (AML), the modeling language of the Axini Modeling Platform (AMP), a state-of-the-art, online MBT tool. The semantics of AML models is defined upon Symbolic Transitions Systems (STS) [3], a data-extension of LTS. But in this text we will mainly use the LTS part of AML.

AML can be used to describe the behavior of reactive systems. In this context, a reactive system is a (part of a) message-oriented system that reacts with observable outputs to inputs received from the environment. A model of the system describes the inputs that it should accept and the outputs it may, or should, send. For each language construct of AML a formal translation to a (part of an) LTS is defined. Each process defined in an AML model is translated to a complete LTS, embodying the behavior of the process.

4.1 Model

A *model* is an abstract, high-level description of the SUT. A model describes the behavior of the system, and specifies *what* the system should do. Conversely, a computer program (i.e., the software) prescribes the system, and dictates *how* the system works. Whe model is a genuine description of the SUT, the test tool is responsible for generating test cases from the model.

An AML model consists of the declaration of the external interfaces of the system and the processes which together define the behavior of the SUT. Each process should be named with a unique string, so they can be distinguished from each other.

```
1 process('main') {
2   # declarations of labels, variables
3   # behaviour of the process
4 }
```

In the example above (lines 2-3) we have used comments as placeholders for the declaration of the labels, and (optionally) variables, and the behavior of the process. AML supports line comments which start with a hash (#): everything after # until the end of the line will be ignored.

The process `'main'` has no behavior defined; the semantics of this process in terms of an LTS is a single state with no outgoing transitions.

4.2 Communication: labels

A process should specify the external behavior: the interactions of the system with its environment. The interactions can be split into two groups:

- *stimuli*: the inputs that the system can process, and
- *responses*: the outputs that the system can or should send.

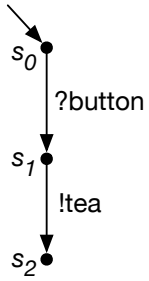
The generic name for a stimulus or a response is *label*. This name reflects the fact that the names of the stimuli and responses are used to label transitions in the underlying transition system that is constructed from the model. A common synonym for label is *action*, because each label represents an action that can be executed on the system or by the system.

A *channel* represents the communication interface via which the action is to be executed. A channel must be declared explicitly before actions can be associated with it. We distinguish *internal* and *external* channels. External channels are used for communication with the SUT. Internal channels are used for communication between processes. We will not use internal channels in this text.

Below is an example of the definiton of an external channel `'extern'` and a process `'button-tea'`:

```
external 'extern'
process('button-tea') {
  # declarations of labels, variables
  timeout 10.0
  stimulus 'button', on: 'extern'
  response 'tea', on: 'extern'

  # behaviour of the process
  receive 'button'
  send 'tea'
}
```



The process `'button-tea'` can receive a stimulus (press on a) `'button'` label through the external channel and subsequently sends a response `'tea'` label. The model is a description of the SUT, so the stimuli are the inputs for the SUT and the responses are the output of the SUT.

Labels like `'button'` and `'tea'` have to be declared before they can be used in the behavioral part of the process. The semantics of this model coincides with the LTS of the Figure 1 (a). As can be seen in this example, a label has to be assigned to a specific *channel*.

Also note the `timeout` declaration at the start of the process. The test tool needs to know how long it has to wait for responses to arrive. This `timeout 10.0` declaration specifies that the default waiting time for responses is `10.0` seconds in process `'button-tea'`. If – when waiting for the response `'tea'` – the response `'tea'` does not arrive in `10.0` seconds, the test tool will exit with the verdict **fail**: quiescence is observed instead of `'tea'`.

4.3 Non-deterministic choice

The `choice` construct allows for the specification of a (non-deterministic) choice between multiple actions. The alternatives of the choice-construct are non-empty sequences of other AML constructs (typically stimuli or responses).

```
choice {
  o { <alternative_1> }
  o { <alternative_2> }
  ...
  o { <alternative_n> }
}
```

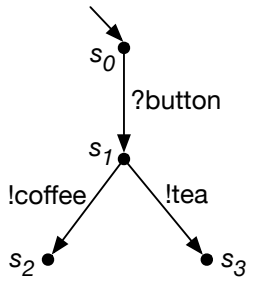
The options of the choice construct are enclosed by curly braces. A single option is specified by lower case `o` (from *option*) followed by a sequence of actions, also enclosed by curly braces. The placeholder *<alternative_i>* stands for a sequence of AML constructs. In general the choice construct is used to model exclusive alternatives. When multiple alternatives are possible, they will be taken together.

Note that a non-deterministic `choice` is a high-level construct that does not have a counterpart in a programming language: a computer program dictates precisely what instructions have to be executed, whereas the `choice` construct specifies that multiple options are possible.

Below is an example for the LTS we have seen before in Figure 1 (repeated below for covenience). The process `'tea-or-coffee'` first waits for `'button'` to be pressed. Then it non-deterministically sends a `'tea'` or a `'coffee'`. The behavior of the process `'tea-or-coffee'` is captured by the LTS.

```
external 'extern'
process('tea-or-coffee') {
  # declarations of labels, variables
  timeout 10.0
  channel('extern') {
    stimulus 'button'
    responses 'tea', 'coffee'
  }

  # behaviour of the process
  receive 'button'
  choice {
    o { send 'tea' }
    o { send 'coffee' }
  }
}
```



We used some shorthand notation in this example. Instead of annotating each label with an explicit channel using the `on:` annotation, we defined the `stimulus` and `responses` using a `channel` declaration. Also observe that two or more responses can be declared with the keyword `responses`. Similarly, two or more stimuli can be declared with the keyword `stimuli`.

Typically, the alternatives of a `choice` are all stimuli or all responses. Mixing stimuli and responses in a `choice` seems natural from a modeling point of view. A system might be able to accept a stimulus or do some output. However, with respect to testing it is not clear what this would mean. When should we do the stimulus? How long should we wait for the response? And although the syntax of AML allows the mixing of stimuli and responses in a `choice`, the tester will always give precedence to the responses: it will wait to observe the responses, and thus ignore the stimuli. The reason for this is subtle: if we would choose the stimulus and later observe the response, we do not know whether the response is caused by the stimulus or whether the response is just late.

4.4 Loop: repeat

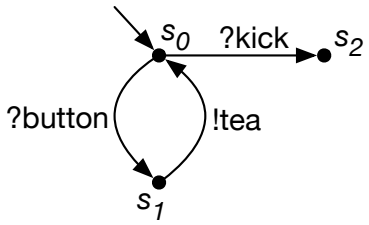
The `repeat` statement can be used to specify that a certain sequence of transitions can be repeated.

```
repeat {
  o  { <alternative_1> }
  o  { <alternative_2> }
  ...
  o  { <alternative_n> }
}
```

The syntax and semantics of the `repeat` construct are similar to those of the `choice` construct. The difference with the `choice` is that after an alternative has been executed, the `repeat` construct is executed again. The `stop_repetition` statement can be used to break out of the loop.

Below, the behavior of the process 'tea-or-kick' is captured by the LTS of Figure 1 (c).

```
external 'extern'
process('tea-or-kick') {
  timeout 10.0
  channel('extern') {
    stimuli 'button', 'kick'
    response 'tea'
  }
  repeat {
    o { receive 'kick'  ; stop_repetition }
    o { receive 'button'; send 'tea'      }
  }
}
```



Again, apart from the repeat construct itself, we introduced some shorthand notation. The actions in the alternatives of the repeat are written on a single line. With this syntax the actions have to be separated by a semi-colon (;).

4.5 States and goto

Within programming languages, the goto statement is often considered harmful [2]. When modeling reactive systems, though, the use of states and goto statements is natural: the behavior of the SUT is often specified as a state-transition system in the documentation of the SUT.

AML supports named states and a goto construct to transfer control to a named state. States are identified by a string. The following AML snippet, for example, defines an infinite loop of 'ping' and 'pong' messages.

```
state 'loop'
receive 'ping'
send 'pong'
goto 'loop'
```

The previous 'tea-or-kick' process can be rewritten using a named state, a choice and a goto statement:

```
external 'extern'
process('tea-or-kick-goto') {
  timeout 10.0
  channel('extern') {
    stimuli 'button', 'kick'
    response 'tea'
  }

state 'loop'
  choice {
    o { receive 'kick' }
    o { receive 'button'; send 'tea'; goto 'loop' }
  }
}
```

After the stimulus 'kick', there is no need to explicitly jump to the end of the process: after the choice, the process is ended anyway.

For the larger example of the ATM, or cash dispenser, from Figure 2 and the given the AML constructs that we have seen so far, we can now make the following AML model:

```
external 'extern'
process('cash-dispenser') {
  timeout 10.0
  channel('extern') {
    stimuli 'Card', 'Pincode', 'Amount'
    responses 'AskPincode', 'AskAmount', 'Wrong', 'Money',
              'NotEnough', 'Card', 'KeepCard'
  }

state 'idle'
  receive 'Card'
  send 'AskPincode'
  receive 'Pincode'

  choice {
    o { send 'AskAmount'; goto 'give money' }
    o {
      send 'Wrong'
      receive 'Pincode'
      choice {
        o { send 'AskAmount'; goto 'give money' }
        o { send 'Wrong'; send 'KeepCard'; goto 'idle' }
      }
    }
  }

state 'give money'
  receive 'Amount'
  choice {
    o { send 'Money' }
    o { send 'NotEnough' }
  }
  send 'Card'; goto 'idle'
}
```

EXERCISE 5
Consider the `Stack` machine of Exercise 1 again. Write an AML model for this `Stack` machine which uses named states and `goto` statements.

4.6 Data: parameters

So far, we have only seen models where the observable labels were abstract names. In practice, however, stimuli and responses usually carry data. AML supports data through parameters on labels. The names of parameters are strings and the types can be simple types (`:integer`, `:string`, `:boolean`, `:decimal`) or structured types (lists, structs, hashes). In this text, we will only use simple types. The names and types of label parameters are specified in a list enclosed by curly braces and have to be specified with the definition of the label.

For example, consider the following AML model where we have to insert coins to get tea.

```
external 'extern'
process('coin-tea-parameters') {
  timeout 10.0
  channel('extern') {
    stimulus 'coin', {'value' => :integer}
```

```
6     response 'tea',  {'volume' => :integer}
7   }
8
9   repeat {
10    o {
11      receive 'coin', constraint: 'value == 50'
12      send 'tea',     constraint: 'volume == 200'
13    }
14    o {
15      receive 'coin', constraint: 'value == 100'
16      send 'tea',     constraint: 'volume == 300'
17    }
18  }
19 }
```

The stimulus `'coin'` (line 5) has a parameter `'value'` of type `:integer`, denoting the worth of the coin. The response `'tea'` (line 6) has a parameter `'volume'` of type `:integer`, expressing the volume of the tea delivered to the user. For a stimulus (like `'coin'`), we now should specify the values of the parameters. In the example above, in lines 11 and 15 we constrain the `'value'` to be exact, e.g., `50` and `100` respectively. For a response (like `'tea'`), we can constrain the value of the parameters that the SUT should return (lines 12 and 16). As a result the generated test-case will check the value of the parameters as offered by the SUT. In the example above, we check that the `'volume'` is `200` (line 12) or `300` (line 16). Besides equality, AML also supports relational and logical operators to constrain the parameters, e.g.,

```
1 receive 'coin', constraint: 'value >= 50 && value < 100'
```

Here we specify that the `'value'` of the `'coin'` has to be at least `50`, but smaller than `100`.

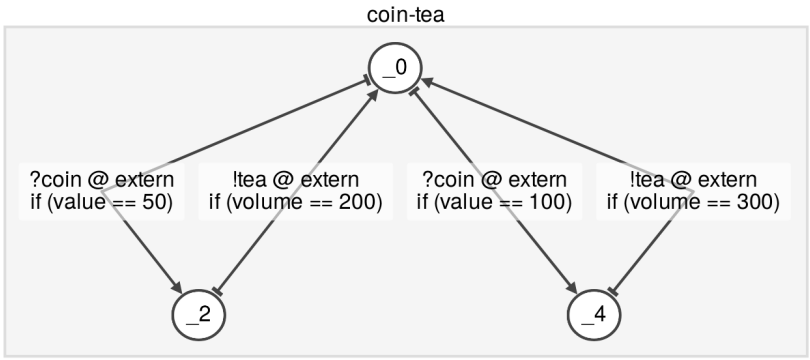


FIGURE 7 LTS of the AML process `'coin-tea-parameters'`.

Figure 7 shows the LTS for the process `'coin-tea-parameters'`, as generated by AMP.

4.7 State variables

We take the previous example one step further. We add a stimulus `'stop'` button to the machine which stops the coin-tea loop. After pressing `'stop'`, the following response will be shown on the `'display'` of the machine: total value of the coins inserted.

To model this, we introduce another feature of AML: *state variables*. State variables can be used to store information. In the following example, we will use the state variable `'total'` of type `:integer` to store the total value of the coins inserted.

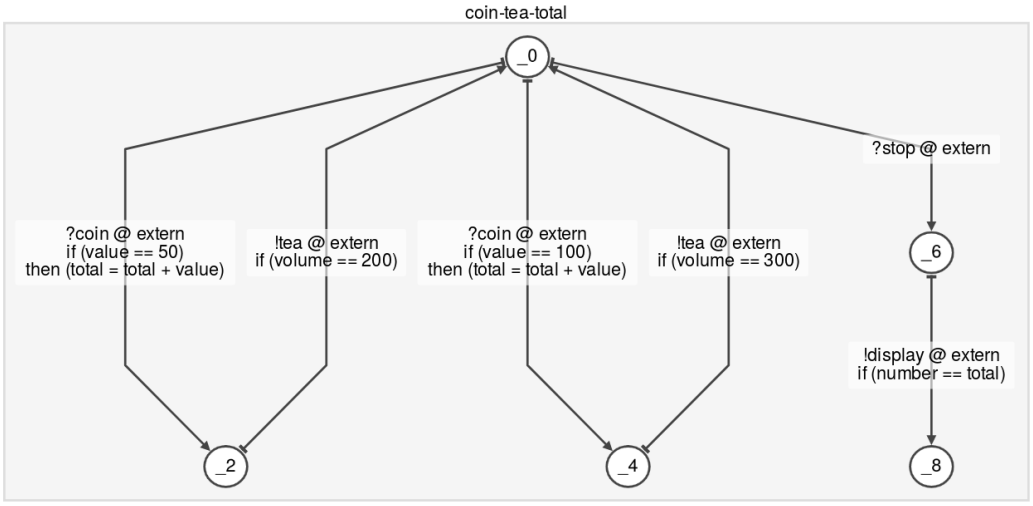
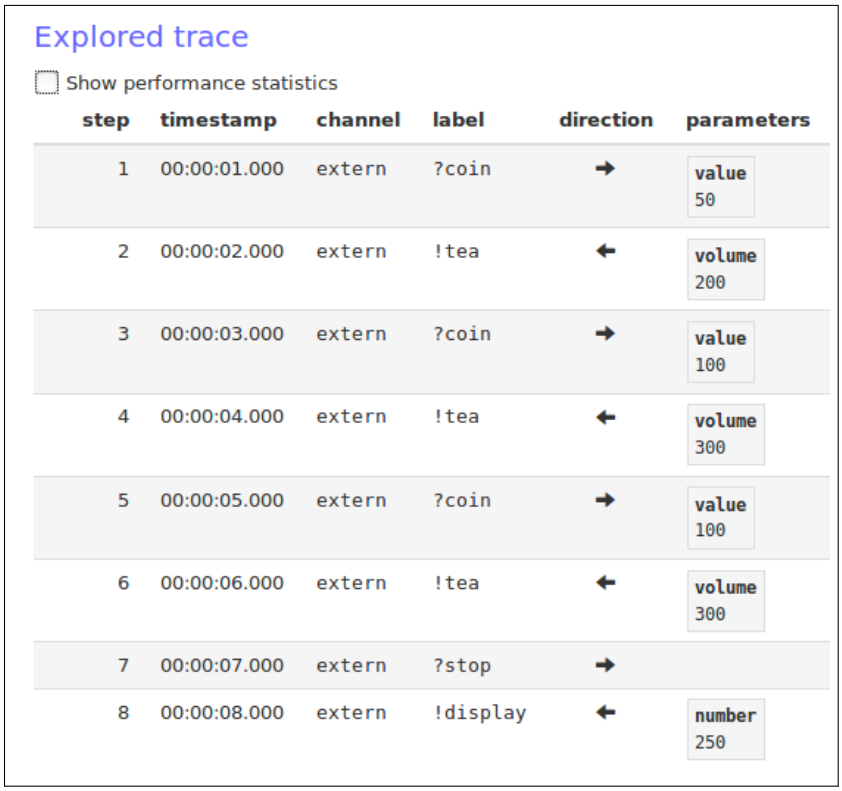


FIGURE 8 LTS of the AML process `'coin-tea-total'`.

```
1  external 'extern'
2  process('coin-tea-total') {
3    timeout 10.0
4    channel('extern') {
5      stimulus 'coin',    {'value' => :integer}
6      stimulus 'stop'
7      response 'tea',     {'volume' => :integer}
8      response 'display', {'number' => :integer}
9    }
10   var 'total', :integer, 0
11
12   repeat {
13     o {
14       receive 'coin', constraint: 'value == 50',
15                          update: 'total = total + value'
16       send 'tea',     constraint: 'volume == 200'
17     }
18     o {
19       receive 'coin', constraint: 'value == 100',
20                          update: 'total = total + value'
21       send 'tea',     constraint: 'volume == 300'
22     }
23     o { receive 'stop'; stop_repetition }
24   }
25   send 'display', constraint: 'number == total'
26 }
```

First, stimulus `'stop'` and response `'display'` are added on lines 6 and 8 respectively. The state variable `'total'` is initialised to `0` on line 10. State variables can be updated in an `update:` clause of a label (lines 15 and 20). Note that for the response `'display'` the parameter `'number'` is checked against the state variable `'total'`, which has been updated during the test run. If the SUT would send a response with an incorrect number, the test case would fail.

Figure 8 shows the LTS for the process `'coin-tea-total'`. Figure 9 shows a sample (passing) trace of the process `'coin-tea-total'` by running AMP's EXPLORER on the model: three coins have been inserted to deliver one small and two medium cups of tea.



Explored trace

Show performance statistics

step	timestamp	channel	label	direction	parameters
1	00:00:01.000	extern	?coin	→	value 50
2	00:00:02.000	extern	!tea	←	volume 200
3	00:00:03.000	extern	?coin	→	value 100
4	00:00:04.000	extern	!tea	←	volume 300
5	00:00:05.000	extern	?coin	→	value 100
6	00:00:06.000	extern	!tea	←	volume 300
7	00:00:07.000	extern	?stop	→	
8	00:00:08.000	extern	!display	←	number 250

FIGURE 9 Explored trace of the AML process `'coin-tea-total'`.

EXERCISE 6

Consider the `Stack` machine of Exercise 1 again. In Exercise 5 you developed an AML model which uses named states and explicit `goto` statements to model the `Stack` machine. Rewrite your AML model such that it does no longer use named states and `goto` statements but instead uses a `repeat` statement and a state variable to count the number of values on the stack.

Note that the underlying LTS of both AML models will be different. And hence the number of states and transitions will be different as well. This also means that given a SUT and the same test execution, the state and transition coverage for both AML models could be different.

4.8 Advanced features

This concludes our introduction to AML. Still, several advanced and useful aspects of AML have not been discussed, including:

- structured data types for parameters and variables: lists, structures, and hashes,
- the inclusion of model parts to structure the model,
- behavior definitions to encapsulate common behavior,
- internal communication between processes to pass information between processes,
- internal transitions to update state variables,
- urgent transitions to enforce internal communication,
- function definitions for complex computations with variables, and
- the use of the Ruby programming language as preprocesser to structure the model.

The interested reader is referred to the AML tutorial which is available from within AMP.

Answers to exercises

1. Figure 10 shows an LTS that defines the behavior of the Stack machine for three values. By com-

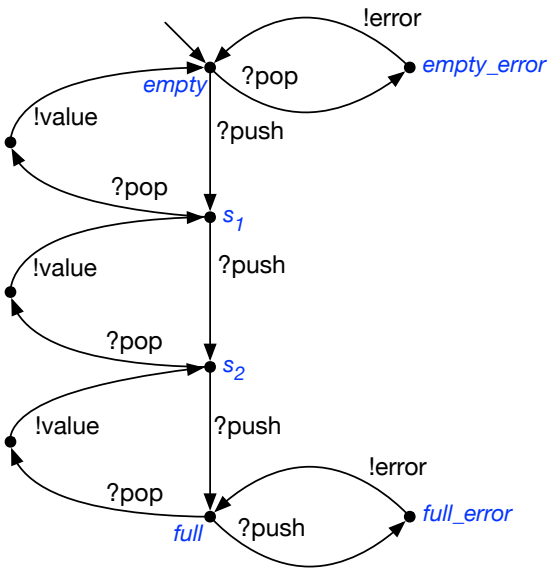


FIGURE 10 LTS of a Stack machine.

parison, Figure 11 shows a Mealy machine that defines the behavior of the Stack machine. Note the difference between the LTS and the Mealy machine: for an LTS each label is either a stimulus or response, never both. For the Mealy machine, a transition contains both an input and an output: the output is determined by the current state and the current input. For stack machines

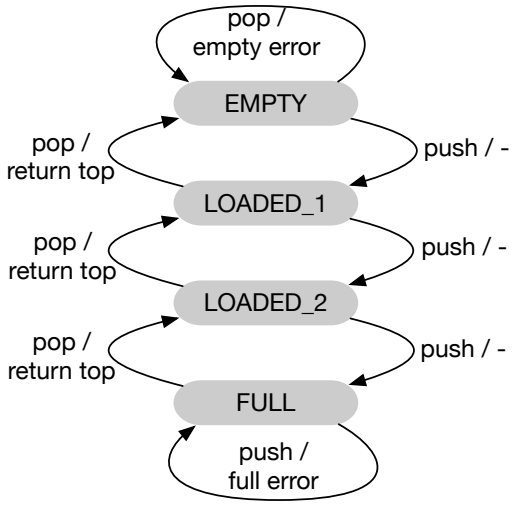


FIGURE 11 Mealy machine of a Stack machine.

with more than 3 elements, both the LTS and Mealy models will get much larger, of course. It is clear that pure state-transition models are tedious to construct. Therefore, we typically use a modeling language which has a higher abstraction level, and, e.g., includes support to store and retrieve data.

2. Figure 12 shows a test case of the LTS of the Stack machine up to depth four. To increase the readability of the test case we have added state names (in grey) to selected locations which correspond to the states of the LTS of Figure 10. Note that although the state *full_error* can be visited by this test case, the resulting !error response will not be checked.

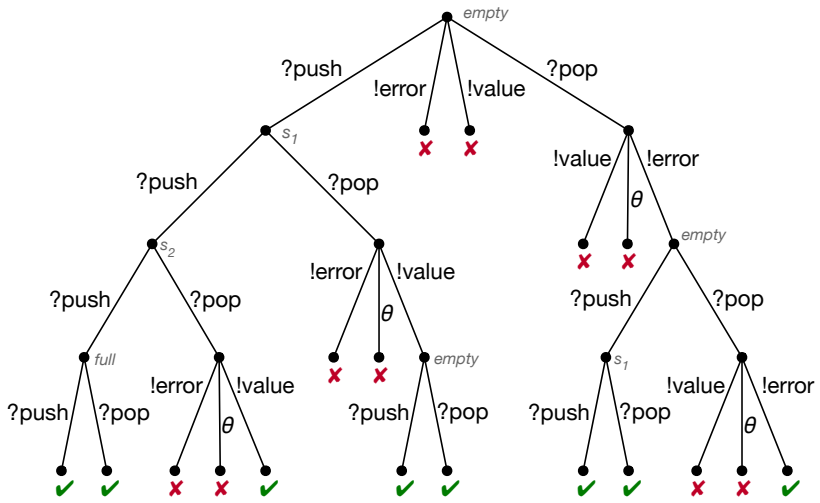


FIGURE 12 Test case for the Stack machine up to depth 4.

3. Figure 13 shows a test case of the LTS of the Stack machine up to depth 5, where ?pop stimuli are only allowed after two ?push stimuli.

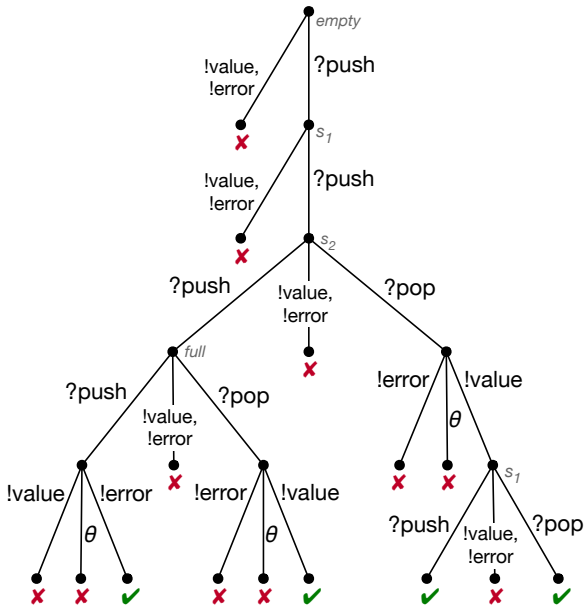


FIGURE 13 Test case for the Stack machine up to depth 5.

4. Figure 10 shows a possible LTS for the Stack machine. This LTS has 9 states and 13 transitions. Given the three tests, the states *empty_error* and *full_error* would not be visited. Furthermore, the anonymous state between *full* and s_2 is not visited. So the state coverage is $(9-3)/9 = 66.7\%$. Similary, $2+2+2 = 6$ transitions are not covered by these tests. Hence, the transition coverage is $(13-6)/13 = 53.8\%$.

5. A possible AML model for the Stack machine is the following. Note that we only use named states and goto statements to keep as close to the LTS of Exercise 1 as possible.

```
external 'extern'
process('stack') {
  timeout 10.0
  channel('extern') {
    stimuli 'push', 'pop'
    responses 'value', 'error'
  }

goto 'empty'

state 'empty_error'
  send 'error'

state 'empty'
  choice {
    o { receive 'push'; goto 's1' }
    o { receive 'pop';  goto 'empty_error' }
  }

state 's1'
  choice {
    o { receive 'push'; goto 's2' }
    o { receive 'pop'; send 'value'; goto 'empty' }
  }

state 's2'
  choice {
    o { receive 'push'; goto 'full' }
    o { receive 'pop'; send 'value'; goto 's1' }
  }

state 'full'
  choice {
    o { receive 'push'; goto 'full_error' }
    o { receive 'pop'; send 'value'; goto 's2' }
  }

state 'full_error'
  send 'error'
  goto 'full'
}
```

6. A possible AML model for the Stack machine is the following with a repeat statement and a state variable n to count the number of values.

```
external 'extern'
process('stack') {
  timeout 10.0
  channel('extern') {
    stimuli 'push', 'pop'
    responses 'value', 'error'
  }

  var 'n', :integer, 0

```

```
  repeat {
    o {
      receive 'push', constraint: 'n == 3'
      send 'error'
    }
    o {
      receive 'push', constraint: 'n < 3',
                           update: 'n = n + 1'
    }
    o {
      receive 'pop',  constraint: 'n == 0'
      send 'error'
    }
    o {
      receive 'pop',  constraint: 'n > 0',
                           update: 'n = n - 1'
      send 'value'
    }
  }
}
```

References

[1] René G. de Vries and Jan Tretmans. On-the-fly Conformance Testing using SPIN. *STTT*, 2(4):382–393, 2000.

[2] Edsger W. Dijkstra. Letters to the editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[3] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test Generation Based on Symbolic Specifications. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[4] Mark Timmer, Ed Brinksma, and Mariëlle Stoelinga. Model-Based Testing. In Manfred Broy, Christian Leuxner, and Tony Hoare, editors, *Software and Systems Safety - Specification and Verification*, volume 30 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 1–32. IOS Press, 2011.

[5] Jan Tretmans. Model Based Testing with Labelled Transition Systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[6] Tanja E.J. Vos and Nikè van Vugt-Hage. *Software Testing – Workbook for O/U course 'Software Testen', cursuscode IB3202*. Open University, Heerlen, The Netherlands, 2019.