MBT4ES

October 31, 2010

Contents

Chapter 1

Automatic Model-Based Test Generation from UML State Machines

Stephan Weißleder and Holger Schlingloff
Fraunhofer Institute FIRST
Kekuléstraße 7, 12489 Berlin

1.1 Introduction

Model-based testing is an efficient testing technique in which a system under test (SUT) is compared to a formal model that is created from the SUT's requirements. Major benefits of model-based testing compared to conventional testing techniques are the automation of test case design, the early validation of requirements, the traceability of requirements from model elements to test cases, the early detection of failures, and an easy maintenance of test suites for regression testing.

This chapter deals with state machines of the Unified Modeling Language (UML) [91] as a basis for automated generation of tests. The UML is a widespread semi-formal modeling

language for all sorts of computational systems. In particular, UML state machines can be used to model the reactive behavior of embedded systems. We present and compare several approaches for the generation of test suites from UML state machines.

For most computational systems, the set of possible behaviors is infinite. Thus, complete testing of all behaviors in finite time is impossible. Therefore, the fundamental question of every testing methodology is when to stop the testing process. Instead of just testing until the available resources are exhausted, it is better to set certain quality goals for the testing process and to stop testing when these goals have been met. A preferred metrics for the quality of testing is the percentage to which certain aspects of the SUT have been exercised; these aspects could be the requirements, the model elements, the source code or the object code of the SUT. Thus, test generation algorithms often strive to generate test suites satisfying certain coverage criteria. The definition of a coverage criterion, however, does not necessarily entail an algorithm how to generate tests for this criterion.

For model-based testing, coverage is usually measured in terms of covered model elements. The standard literature provides many different coverage criteria, for example, focussing on data flow, control flow, or transition sequences. Most existing coverage criteria had been originally defined for program code and have now been transferred and applied to models. Thus, these criteria can be used to measure the quality of test suites that are generated from models. Test generation algorithms can be designed and optimized with regard to specific coverage criteria. In this chapter, we present several test generation approaches that strive to satisfy different coverage criteria on UML state machines.

This chapter is structured as follows: In the following, we give an introduction to UML state machines and present the basic ideas of testing from UML state machines. Subsequently, we describe abstract path generation and concrete input value generation as two important aspects in automatic test generation from state machines: The former is shown in Section 1.2 by introducing graph traversal techniques. The latter is shown in Section 1.3 by presenting boundary value analysis techniques. In Section 1.4, we describe the relation of these two aspects to other techniques. We go into random testing, evolutionary testing, constraint solving, model checking, and static analysis.

1.1.1 UML State Machines

The Unified Modeling Language (UML) [91] is a widely used modeling language standardized and maintained by the Object Management Group (OMG). In version 2, it comprises models of 13 different diagrams, which can be grouped into two general categories: Structure diagrams are used to represent information about the (spatial) composition of the system. Behavior diagrams are used to describe the (temporal) aspects of the system's actions and reactions. All UML diagram types are defined in a common metamodel, so the same modeling elements may be used in different types of diagrams, and there is no distinct separation between the various diagram types. Amongst the behavior diagrams, state machine diagrams are the most common way to specify the control flow of reactive systems. Intuitively, a UML state machine can be seen as a hierarchical parallel automaton with an extended alphabet of actions. In order to precisely describe test generation algorithms, we give a formal definition of the notion of UML state machines used in this chapter.

A *labeled transition system* is a tuple $M = (\mathcal{A}, S, T, s_0)$, where $\mathcal{A}$ is a finite nonempty alphabet of labels, S and T are finite sets of states and transitions, respectively, $T \subseteq S \times \mathcal{A} \times S$, and s_0 is the *initial* state. In UML, the initial state is a so-called pseudostate (not belonging to the set of states) and marked by a filled circle. Assume a set E of *events*, a set C of *conditions*, and a set A of *actions*. A *simple state machine* is a labeled transition system where $\mathcal{A} = 2^E \times C \times 2^A$, that is, each label consists of a set e of input events, a condition c, and a set a of output actions. The input events of a transition are called its *triggers*, the condition is the *guard*, and the set of actions is the *effect* of the transition. The transition $(s, (e, c, a), s')$ is depicted as $\boxed{\text{s}} \xrightarrow{e[c]/a} \boxed{\text{s'}}$, where sets are just denoted by their elements, and empty triggers, guards, and effects can be omitted. States s and s' are the *source* and *target* of the transition, respectively.

A (finite) *run* of a transition system is any word $w = (s_0, t_0, s_1, t_1, \ldots, t_{n-1}, s_n)$, such that s_0 is the initial state, and $(s_i, t_i, s_{i+1}) \in T$ for all $i < n$. The *trace* of a run is the sequence $(t_0, t_1, \ldots, t_{n-1})$. For a simple state machine, we assume that there is an evaluation relation $\models \subseteq S \times C$ that is established iff a condition $c \in C$ is satisfied in a state $s \in S$. A word w is a run of the state machine if in addition to s_0 being initial, for all $i < n$ and $t_i = (e_i, c_i, a_i)$

it holds that $s_i \models c_i$. Moreover, it must be true that

1. $e_i = \emptyset$ and $(s_i, t_i, s_{i+1}) \in T$, or
2. $e_i = \{e\}$ and $(s_i, (e'_i, c_i, a_i), s_{i+1}) \in T$ for some e'_i containing e, or
3. $e_i = \{e\}$, $(s_i, (e'_i, c'_i, a'_i), s'_{i+1}) \notin T$ for any e'_i containing e, and $s_{i+1} = s_i$

These clauses reflect the semantics of UML state machines, which allows for

1. completion transitions (without trigger),
2. transitions being enabled if any one of its triggers is satisfied, and
3. a trigger being lost if no transition for this trigger exists.

In order to model data dependencies, simple state machines can be extended with a concept of variables. Assume a given set of domains or classes with boolean relations defined between elements. The domains could be integer or real numbers with values 0, 1, $<$, $\leq$, etc. An *extended state machine* is a simple state machine augmented by a number of variables (x, y, ...) on these domains. In an extended state machine, a guard is a boolean expression involving variables. For example, a guard could be $(x > 0 \wedge y \leq 3)$. A transition effect in the state machine may involve the update (assignment) of variables. For example, an effect could be $(x := 0; y := 3)$. The UML standard does not define the syntax of assignments and boolean expressions; it suggests that the Object Constraint Language (OCL) [90] may be used here. For our purposes, we rely on an intuitive understanding of the relevant concepts.

In addition to simple states, UML state machines allow a hierarchical and orthogonal composition of states. Formally, a *UML state machine* consists of a set of *regions*, each of which contains vertices and transitions. A *vertex* can be a state, a pseudostate, or a connection point reference. A *state* can be either simple or composite, where a state is composite if it contains one or more regions. *Pseudostates* can be, for example, initial or fork pseudostates where *connection point references* are used to link certain pseudostates. A *transition* is a connection from a source vertex to a target vertex, and it can contain several triggers, a guard, and an effect. A *trigger* references an event, or example, the reception of a

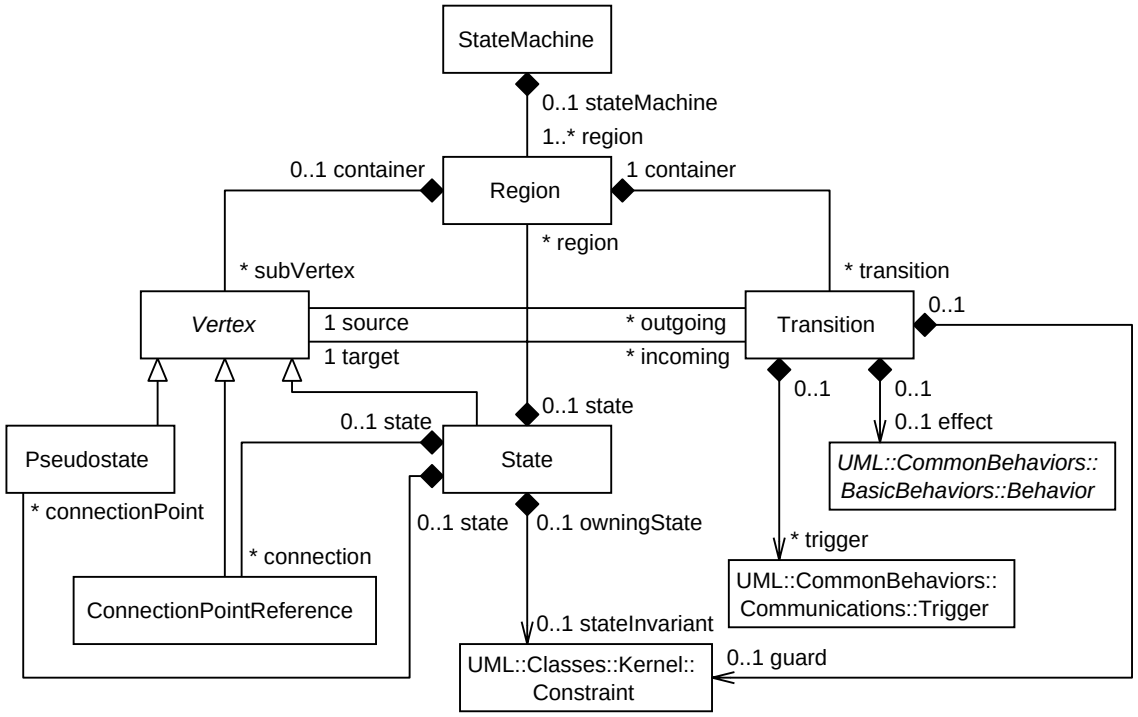

Figure 1.1: Part of the meta model for UML state machines.

message or the execution of an operation. Similar as in extended state machines, a *guard* is a boolean condition on certain variables, for instance, class attributes. Additionally, UML also has a number of further predicates that may be used in guards. Finally, an *effect* can be, for example, the assignment of a value to an attribute, the triggering of an event, or a postcondition defined in OCL. In Figure 1.1, this syntax is graphically described as part of the UML metamodel, a complete description of which can be found in [91].

The UML specification does not give a definite semantics of state machines. However, there is a generally agreed common understanding on the meaning of the above concepts. A state machine describes the behavior of all instances of its context class. The status of each instance is given by the values of all class attributes and the configuration of the state machine, where a *configuration* of the machine is a set of concurrently active vertices. Initially, all those vertices are active that are connected to the outgoing transitions of the initial pseudostates of the state machine's regions. A transition can be traversed if its source vertex is active, one of the triggering events occurs, and the guard evaluates to true. As a consequence, the source vertex becomes inactive, the actions in the effect are executed, and

the target vertex becomes active. In this way, a sequence of configurations and transitions is obtained that forms a run of the state machine. Similar as defined for the labeled transition system, the semantics of a state machine is the set of all these runs.

1.1.2 Example - A Kitchen Toaster

State machines can be used for the high-level specification of the behavior of embedded systems. As an example, we consider a modern kitchen toaster. It has a turning knob to choose a desired browning level, a side lever to push down the bread and start the toasting process, and a stop button to cancel the toasting process. When the user inserts a slice of bread and pushes down the lever, the controller locks the retainer latch and switches on the heating element. In a basic toaster, the heating time depends directly on the selected browning level. In more advanced products, the intensity of heating can be controlled and the heating period is adjusted according to the temperature of the toaster from the previous toasting cycle. When the appropriate time has elapsed or the user pushes the stop button, the the heating is switched off and latch is released. Moreover, we require that the toaster has a "defrost" button that, when activated, causes to heat the slice of bread with low temperature (defrosting) for a designated time before beginning the actual toasting process.

In the following, we present several ways of describing the behavior of this kitchen toaster

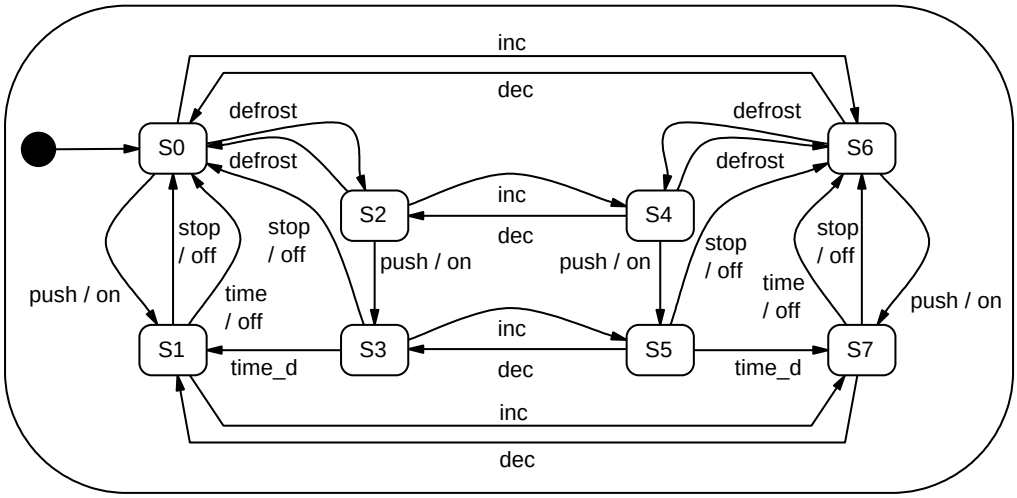

Figure 1.2: Simple state machine model of a kitchen toaster.

with state machines: We give a basic state machine, a semantically equivalent hierarchical machine, and an extended state machine that makes intensive use of variables.

First, the toaster can be modeled by a simple state machine as shown in Figure 1.2. The alphabets are $I = \{push, stop, time, inc, dec, defrost, time\_d\}$ and $O = \{on, off\}$. The toaster can be started by pushing (*push*) down the latch. As a reaction, the heater is turned on (*on*). The toaster stops toasting (*off*) after a certain time (*time*) or after the stop button (*stop*) has been pressed. Furthermore, the toaster has two heating power levels, one of which can be selected by increasing (*inc*) or decreasing (*dec*) the heating temperature. The toaster also has a defrost function (*defrost*) that results in an additional defrosting time (*time_d*) of frozen toast. Note that time is used in our modelling only in a qualitative way, that is, quantitative aspects of timing are not taken into account.

This simple machine consists of two groups of states: $s_0 \ldots s_3$ for regular heating, and $s_4 \ldots s_7$ for heating with increased heating power. From the first group of states, the machine accepts an increase of the heating level, which brings it into the appropriate high-power state; vice versa, from this state, it can be brought back by decreasing the heating level. Thus, in this machine only two heating levels are modeled. It is obvious how the model could be extended for three or more such levels. However, with a growing number of levels the diagram would quickly become illegible.

It is clear that this modeling has other deficits as well. Conceptually, the setting of the heating level and defrosting cycle are independent from the operation of latch and stop button. Thus, they should be modeled separately. Moreover, the decision of whether to start a preheating phase before the actual toasting is "local" to the part dealing with the busy operations of the toaster. Furthermore, the toaster is either inactive or active and so active is a superstate that consists of substates defrosting and toasting.

To cope with these issues, UML offers the possibility of orthogonal regions and hierarchical nesting of states. This allows a compact representation of the behavior. Figure 1.3 shows a hierarchical state machine with orthogonal regions. It has the same behavior as the simple state machine in Figure 1.2. The hierarchical state machine consists of the three regions *side latch*, *set temperature*, and *set defrost*. Each region describes a separate aspect of

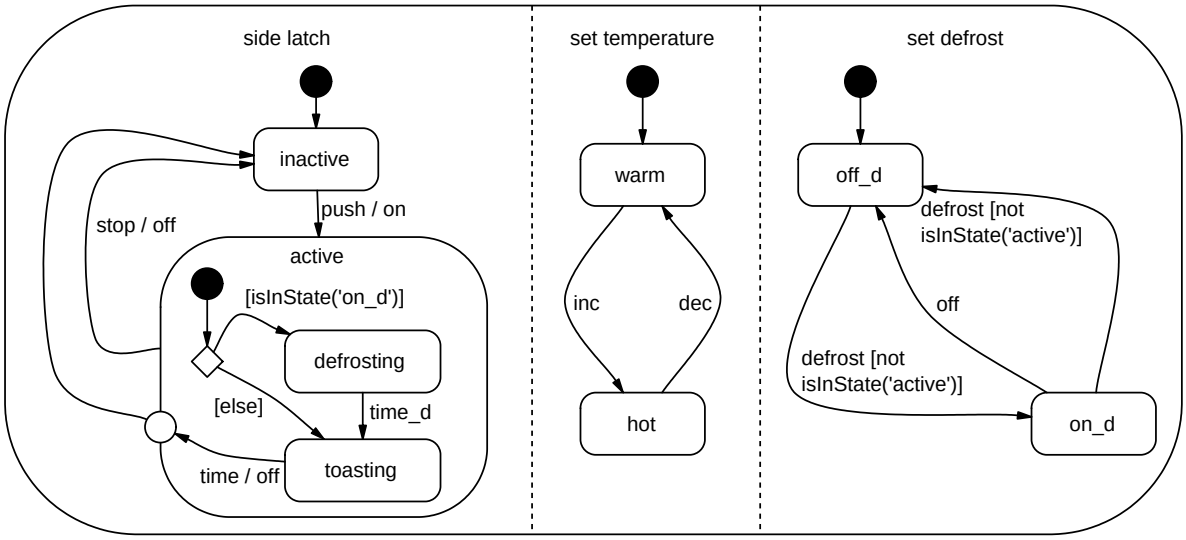

Figure 1.3: A hierarchical state machine model.

the toaster: In region *side latch*, the reactions to moving the side latch, pressing the stop button, and waiting for a certain time *time* are described. The state *active* contains the substates *defrosting* and *toasting*, as well as a choice pseudostate. The region *set temperature* depicts the two heating levels and how to select them. In region *set defrost*, setting up the defrost functionality is described. The defroster can only be (de-)activated if the toaster is not currently in state *active*. Furthermore, the defroster is deactivated after each toasting process.

Both the models in Figure 1.2 and Figure 1.3 are concerned with the control flow only. Additionally, in any computational system the control flow is also influenced by data. In both of the above toaster models, the information about the current toaster setting is encoded in the states of the model. This clutters the information about the control flow and leads to an excessive set of states. Therefore, it is preferable to use a data variable for this purpose. One option to do so is via extended finite state machines, where for instance, the transitions may refer to variables containing numerical data.

Figure 1.4 shows a more detailed model of a toaster that contains several variables. This model also contains the region *residual heat* to describe the remaining internal temperature of the toaster. Since a hot toaster reaches the optimal toasting temperature faster, the internal temperature is used in the computation of the remaining heating time. The state

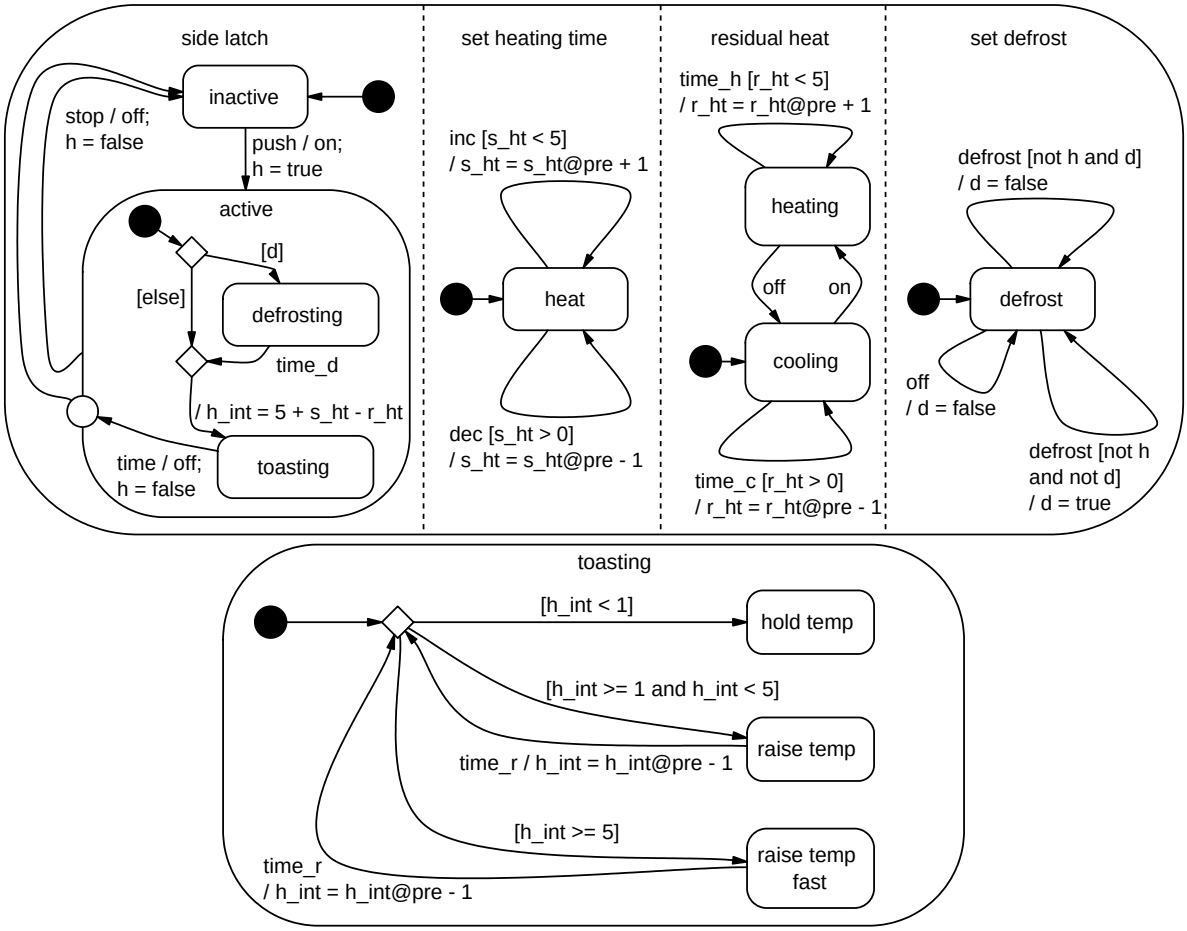

Figure 1.4: A UML state machine with variables.

machine consists of the four regions *side latch*, *set heating time*, *residual heat*, and *set defrost*. The names of the regions describe their responsibilities. The region *side latch* describes the reaction to pressing the side latch: If the side latch is pushed (*push*), the heater is turned on (releasing the event *on* and setting $h = true$). As a result, the toaster is in the state *active*. If the defrost button has been pressed ($d = true$), the toaster will be in the state *defrosting* for a certain time (*time_d*). The heating intensity (*h_int*) for the toasting process will be set depending on the set heat (*s_ht*) for the browning level and the residual heat (*r_ht*). The details of regulating the temperature are described in the composite state *toasting*: Depending on the computed value *h_int*, the toaster decides to raise the temperature (fast) or to hold it at the current level. The toaster performs these actions for the time period *time* and stops toasting, afterwards. As an effect of stopping, it triggers the event *off* and sets $h = false$. The region *set heating time* allows to set the temperature to one of the levels 0 to

6. In the region *residual heat*, the heating-up and the cooling-down of the internal toaster temperature are described. The region *set defrost* allows to (de-)activate the defrost mode. After completing one toasting cycle, the defrost mode will be deactivated.

1.1.3 Testing from UML State Machines

Testing is the process of systematically experimenting with an object in order to detect failures, to measure its quality, or to create confidence in its correctness. One of the most important quality attributes is *functional correctness*, that is, determining whether the SUT satisfies the specified requirements. To this end, the requirements specification is compared to the SUT. In model-based testing, the requirements are represented in a formal model and the SUT is compared to this model. A prominent approach for the latter is to derive test cases from the model and to execute them on the SUT. Following this approach, requirements, test cases, and the SUT can be described by a validation triangle as shown in Figure 1.5.

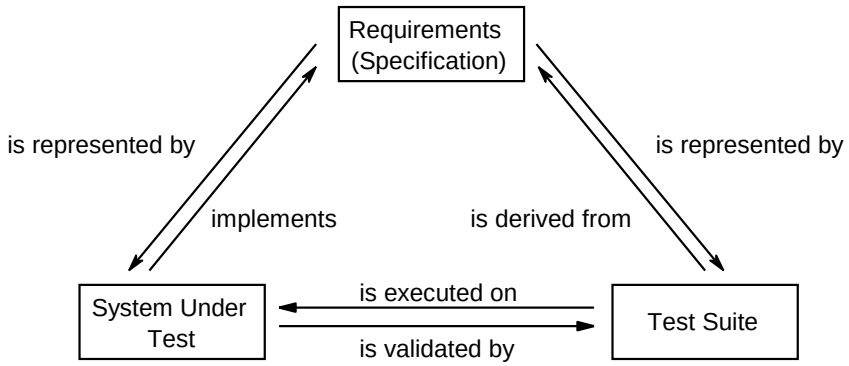

Figure 1.5: Validation triangle.

A *test case* is the description of a (single) test; a *test suite* is a set of test cases. Depending on the aspect of an SUT that is to be considered, test cases can have several forms — see Table 1.1. This table is neither a strict classification, nor is it exhaustive. As a consequence, systems can be in more than one category, and test cases can be formulated in many different ways. Embedded systems usually are modeled as deterministic reactive systems and, thus, test cases are sequences of events. The notion of test execution and test oracle has to be defined for each type of SUT. For example, the execution of reactive system tests consists

of feeding the input events into the SUT and comparing the corresponding output events to the expected ones.

SUT Characteristics	Test Case
functional	pair (input value, output value)
reactive	sequence of events
nondeterministic	decision tree
parallel	partial order
interactive	test script or program
real-time	timed event structure
hybrid	set of real functions

Table 1.1: Different SUT aspects and corresponding test cases.

For our example, the models describe the control of the toaster. They specify (part of) its observable behavior. Therefore, the observable behavior of each run of the state machine can be used as a test case. We can execute such a test case as follows: if the transition is labeled with an input to the SUT (pushing down the lever or pressing a button), we perform the appropriate action whereas if it is labeled with an output of the SUT (locking or releasing the latch, turning heating on or off), we see whether we can observe the appropriate reaction. As shown in Table 1.2, model-based tests can be performed on various interface levels, depending on the development stage of the SUT.

Acronym	Stage	SUT	Testing Interfaces
MiL	Model-in-the-Loop	System Model	Messages and events of the model
SiL	Software-in-the-Loop	Control software (e.g., C or Java code)	Methods, procedures, parameters and variables of the software
PiL	Processor-in-the-Loop	Binary code on a host machine emulating the behavior of the target	Register values and memory contents of the emulator
HiL	Hardware-in-the-Loop	Binary code on the target architecture	I/O pins of the target microcontroller or board
	System-in-the-Loop	Actual physical system	Physical interfaces, buttons, switches, displays, etc.

Table 1.2: Model-based testing levels.

An important fact about model-based testing is that the same logical test cases can be used on all these stages, which can be achieved by defining for each stage a specific test adapter that maps abstract events to the concrete testing interfaces. For example, the user action of pushing the stop button can be mapped to sending the event *stop* to the system model, to the call of Java AWT ActionListener class method `actionPerformed(stop)`, to the writing of a *1* into address `0x0CF3` in a certain emulator running, for example, Java byte code, or to setting the voltage at pin `GPIO5` of a certain processor board to high. System-

in-the-loop tests are notoriously difficult to implement. In our example, we would have to employ a robot that is able to push buttons and observe the browning of a piece of toast.

Coverage Criteria

Complete testing of all possible behaviours of a reactive system is impossible. Therefore, an adequate subset has to be selected which is used in the testing process. Often, coverage criteria are used to control the test generation process or to measure the quality of a test suite. Coverage of a test suite can be defined with respect to different levels of abstraction of the SUT: requirements coverage, model coverage, or code coverage. If a test suite is derived automatically from one of this levels, coverage criteria can be used to measure the extent to which it is represented in the generated test suite.

In the following, we present coverage criteria as a means to measure the quality of a test suite. Experience has shown that there is a direct correlation between the various coverage notions and the fault detection capability of a test suite. The testing effort (another quality aspect) is measured in terms of the size of the test suite. In practice, one has to find a balance between minimal size and maximal coverage of a test suite.

Model coverage criteria can help to estimate to which extent the generated test suite represents the modelled requirements. Usually, a coverage criterion is defined independent from any specific test model, that is, at the meta-model-level. Therefore, it can be applied to any instance of that meta-model. A *model coverage criterion* applied to a certain test model results in a set of *test goals*, which are specific for that test model. A test goal can be any model element (state, transition, event, etc.) or combination of model elements, for example, a sequence describing the potential behavior of model instances. A test case achieves a certain test goal, if it contains the respective model element(s). A test suite *satisfies* (or *is complete for*) a coverage criterion, if for each test goal of the criterion there is a test case in the suite that contains this test goal. The *coverage* of a test suite with respect to a coverage criterion is the percentage of test goals in the criterion that are achieved by the test cases of the test suite. In other words, a test suite is complete for a coverage criterion iff its coverage is 100%. Typical coverage criteria for state machine models are

- All-States: for each state of the machine, there is a test case which contains this state,
- All-Transitions: for each transition of the machine, there is a test case which contains this transition,
- All-Events: the same for each event that is used in any transition,
- Depth-n: for each run $(s_0, a_1, s_1, a_2, \ldots, a_n, s_n)$ of length at most n from the initial state or configuration there is a test case containing this run as a subsequence,
- All-n-Transitions: for each run of length at most n from any state $s \in S$, there is a test case that contains this run as a subsequence (All-2-Transitions is also known as All-Transition-Pairs; All-1-Transitions is the same as All-Transitions, and All-0-Transitions is the same as All-States), and
- All-Paths: all possible transition sequences on the state machine have to be included in the test suite; this coverage criterion is considered infeasible.

In general, satisfying only All-States on the model is considered too weak. The main reason is that only the states are reached but the possible state changes are only partially covered. Accordingly, All-Transitions is regarded a minimal coverage criterion to satisfy. Satisfying the All-Events criterion can also be regarded as an absolute minimal necessity for any systematic black-box testing process. It requires that every input is provided at least once, and every possible output is observed at least once. If there are input events that have never been used, we cannot say that the system has been thoroughly tested. If there are specified output actions that could never be produced during testing, chances are high that the implementation contains a fault. Depth-n and All-n-Transitions can result in test suites with a high probability to detect failures. On the downside, the satisfaction of these criteria also often results in big test suites.

The presented coverage criteria are related. For instance, in a connected state machine, that is, if for any two simple states there is a sequence of transitions connecting them, the satisfaction of All-Transitions implies the satisfaction of All-States. In technical terms, All-Transitions subsumes All-States.

In general, coverage criteria *subsumption* is defined as follows: If any test suite that satisfies coverage criterion A also always satisfies the coverage criterion B, then A is said to *subsume* B. The subsuming coverage criterion is considered *stronger* than the subsumed one. However, this does not mean that a test suite satisfying the coverage criterion A necessarily detects more failures than a test suite satisfying B.

All-Transition-Pairs subsumes All-Transitions. There is no such relation for All-Events and All-Transitions: There may be untriggered transitions that are not executed by a test suite the calls all events; likewise, a transition may be activated by more than one event and a test suite that covers all transitions does not use all of these events. Likewise, Depth-n is unrelated to All-Events and All-Transitions. For practical purposes, besides the All-Transitions criterion often the Depth-n criterion is used, were n is set to the diameter of the model. The criterion All-n-Transitions is more extensive; for $n \geq 3$ this criterion often results in a very large test suite. Clearly, All-n-Transitions subsumes Depth-n, All-$(n+1)$-Transitions subsumes All-n-Transitions for all n, and All-Paths subsumes all of the previously mentioned coverage criteria except All-Events. Figure 1.6 shows the corresponding subsumption hierarchy. The relation between All-n-Transitions and Depth-n is dotted because it only holds if the n for All-n-Transitions has at least the same value as the n of Depth-n.

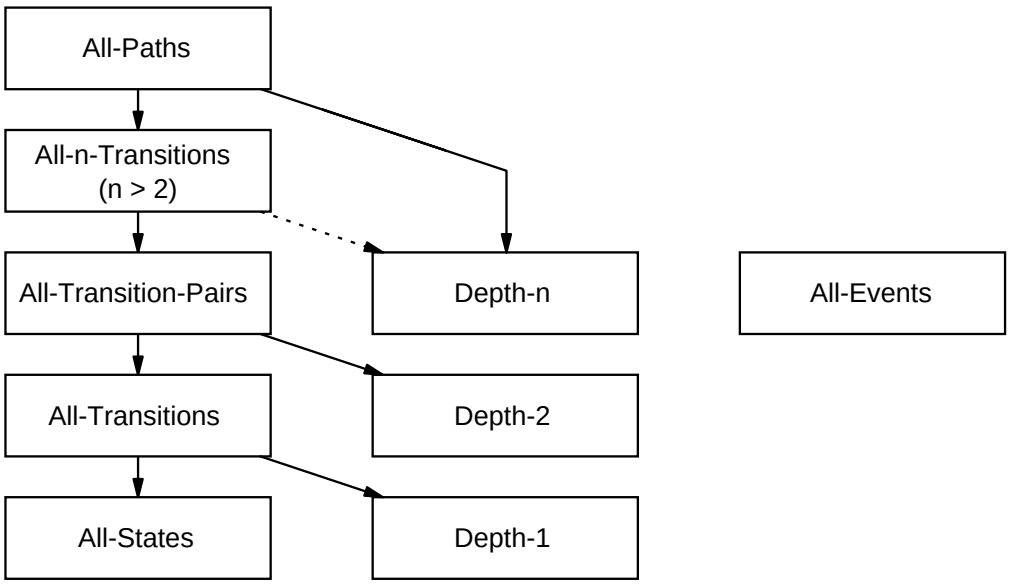

Figure 1.6: Subsumption hierarchy of structural coverage criteria.

Beyond simple states, UML state machines can contain orthogonal regions, pseudostates, and composite states. Accordingly, the All-States criterion can be modified to entail

- all reachable configurations,
- all pseudostates, or
- all composite states.

Likewise, other criteria like the All-Transitions criterion can be modified such that all triggering events of all transitions or all pairs of configurations and outgoing transitions are covered [69]. Since there are potentially exponentially more configurations than simple states, constructing a complete test suite for all reachable configurations is often infeasible.

Conditions in UML state machine transitions are usually formed from atomic conditions with boolean operators {and, or, not} and so the following control-flow-based coverage criteria focussed on transition conditions have been defined [115]:

- Decision Coverage, which requires that for every transition guard c from any state s there is one test case where s is reached and c is true, and one test case where s is reached and c is false;
- Condition Coverage, which requires the same as Decision Coverage for each atomic condition of every guard;
- Condition / Decision Coverage, which requires that the test suite satisfies both Condition Coverage and Decision Coverage;
- Modified Condition / Decision Coverage (MC/DC)[32, 31], which additionally requires to show that each atomic condition has an isolated impact on the evaluation of the guard; and
- Multiple Condition Coverage, which requires test cases for all combinations of atomic conditions in each guard.

Multiple Condition Coverage is the strongest control-flow-based coverage criterion. However, if a transition condition is composed of n atomic conditions, a minimal test suite that

satisfies Multiple Condition Coverage may require up to 2^n test cases. MC/DC [32] is still considered very strong, it is part of DO178-B [107], and requires only linear test effort. The subsumption hierarchy of control-flow-based coverage criteria is shown in Figure 1.7.

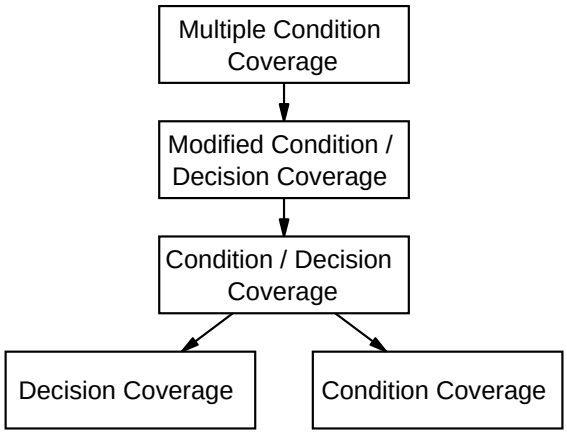

Figure 1.7: Subsumption hierarchy of condition-based coverage criteria.

There are further coverage criteria that are focussed on the data flow in a state machine, for example, on the definition and use of variables.

Size of Test Suites

The existence of a unique, minimal, and complete test suite, for each of the coverage criteria mentioned above, cannot be guaranteed. For the actual execution of a test suite, its size is an important figure. The size of a test suite can be measured in several ways or combinations of these:

- the number of all events, that is, the lengths of all test cases,
- the cardinality, that is, the number of test cases in the test suite, or
- the number of input events.

At first glance, the complexity of the execution of a test suite is determined by the number of all events that occur in it. At a closer look, resetting the SUT after one test in order to

run the next test turns out to be a very costly operation. Hence, it may be advisable to minimize the number of test cases in the test suite. Likewise, for manual test execution, the performance of a (manual) input action can be much more expensive than the observation of the (automatic) output reactions. Hence, in such a case the number of inputs must be minimized. These observations show that there is no universal notion of minimality for test suites; for each testing environment different complexity metrics may be defined. A good test generation algorithm takes these different parameters into account. Usually, the coverage increases with the size of the test suite, however, this relation is often nonlinear.

1.2 Abstract Test Case Generation

In this section, we present the first challenge of automatic test generation from UML state machines: creating paths on the model level to cover test goals of coverage criteria. State machines are extended graphs and graph traversal algorithms can be used to find paths in state machines [3, 62, 80, 82, 84, 88]. These paths can be used as *abstract test cases*, that are test cases that are missing the details about input parameters. In Section 1.3, we present approaches to generate the missing input parameters.

Graph traversal has been thoroughly investigated and is widely used for test generation in practice. For instance, Chow [33] creates tests from a finite state machine by deriving a testing tree using a graph search algorithm. Offutt and Abdurazik [92] identify elements in a UML state machine and apply a graph search algorithm to cover them. Other algorithms also include data flow information [23] to search paths. Harman et al. [67] consider reducing the input space for search-based test generation. Gupta et al. [61] find paths and propose a relaxation method to define suitable input parameters for these paths. We apply graph traversal algorithms that additionally compute the input parameter partitions[126, 127].

Graph traversing consists of starting at a certain start node n_{start} in the graph and traversing edges until a certain stopping condition is satisfied. Such stopping conditions are, for example, that all edges have been traversed (see the Chinese postman problem in [98]) or a certain node has been visited (see structural coverage criteria [115]). There are many

different approaches to graph traversal. One choice is whether to apply *forward* or *backward* searching. In forward searching, transitions are traversed forward from the start state to the target state until the stopping condition is satisfied or it is assumed that the criterion cannot be satisfied. This can be done in several ways like, for instance, breadth-first, depth-first, or weighted breadth-first like in Dijkstra's shortest path algorithm. In backward searching, the stopping condition is to reach the start state. Typical nodes to start this backwards search from are, for example, the states of the state machine in order to satisfy the coverage criterion All-States.

Automated test generation algorithms strive to produce test suites that satisfy a certain coverage criterion, which means reaching 100% of the test goals according to the criterion. The choice of the coverage criterion has significant impact on the particular algorithm and the resulting test suite. However, none of the above described coverage criteria uniquely determines the resulting test suite; for each criterion there may be many different test suites achieving 100% coverage. For certain special cases of models, it is possible to construct test suites that satisfy a certain coverage criterion while consisting of just one test case. The model is strongly connected, if for any two states s and s' there exists a run starting from s and ending in s'. If the model is strongly connected, then for every n there exists a one-element test suite that satisfies All-n-Transitions: From the initial state, for all states s and sequence of length n from s, the designated run traverses this sequence and returns to the initial state.

An Eulerian path is a run that contains each transition exactly once, and a Hamilton path is a run that contains each state exactly once. An Eulerian or Hamiltonian cycle is an Eulerian or Hamiltonian path that ends in the initial state, respectively. Trivially, each test suite containing an Eulerian or Hamiltonian path is complete for All-Transitions or All-States, respectively. There are special algorithms to determine whether such cycles exist in a graph, and to construct them if so.

In the following, we present different kinds of search algorithms: Dijkstra's shortest path, depth-first, and breadth-first. The criteria of when to apply which algorithm depend on many aspects. Several test generation tools implement different search algorithms. For instance,

the Conformiq Test Designer [38] applies forward breadth-first search whereas ParTeG [122] applies backward depth-first search.

1.2.1 Shortest Paths

```
01 void Dijkstra(StateMachine sm, Node source) {
02   for each node n in sm {
03     dist[n] = infinity;              // distance function from source to n
04     previous[n] = undefined;         // Previous nodes determine optimal path
05   }
06   dist[source] = 0:                  // initial distance for source
07   set Q = all nodes in sm;
08   while Q is not empty {
09     u = node in Q with smallest value dist[u];
10     if (dist[u] = infinity)
11       break;                          // all remaining nodes cannot be reached
12     remove u from Q;
13     for each neighbor v of u {
14       alt = dist[u] + dist_between(u, v);
15       if alt < dist[v] {
16         dist[v] = alt;
17         previous[v] = u;
18 } } } }
```

Figure 1.8: Computing shortest distance for all nodes in the graph by Dijkstra.

Complete coverage for All-States in simple state machines can be achieved with Dijkstra's single-source shortest path algorithm [108]. Dijkstra's algorithm computes for each node the minimal distance to the initial node via a greedy search. For computing shortest paths, it can be extended such that it also determines each node's predecessor on this path. The algorithm is depicted in the Figures 1.8 and 1.9: Figure 1.8 shows the algorithm to compute shortest path information for all nodes of the graph. With the algorithm in Figure 1.9, a shortest path is returned for a given node of the graph. The generated test suite consists of all maximal paths that are constructed by the algorithm, that is the shortest paths for all

```
01 Sequence shortestPath(Node target) {
02   S = new Sequence();
03   Node u = target;
04   while previous[u] is defined {
05     insert u at the beginning of S;
06     u = previous[u];
07 } }
```

Figure 1.9: Shortest path selection by Dijkstra.

nodes that are not covered by other shortest paths. For our toaster example in Figure 1.2 on page 6, this algorithm can generate the test cases depicted in Figure 1.10.

```
TC1: (s0, (push, , on), s1, (dec, , ), s7)
TC2: (s0, (defrost, , ), s2, (push, , on), s3, (inc, , ), s5)
TC3: (s0, (inc, , ), s6, (defrost, , ), s4)
```

Figure 1.10: Test cases generated by the shortest path algorithm by Dijkstra.

The same algorithm can be used for covering All-Transitions by inserting a pseudostate in every transition as described in [124]. Furthermore, the generated path is extended by the outgoing transition of the just inserted pseudostate. In the generated test suite, only those sequences must be included that are not prefixes (initial parts) of some other path. This set can be constructed in two ways:

- in decreasing length, where common prefixes are eliminated, or
- in increasing length, where new test cases are only added if their length is maximal.

The presented shortest path generation algorithm is just one of several alternatives. In the following, we will introduce further approaches.

1.2.2 Depth-First and Breadth-First Search

In this section, we describe depth-first and breadth-first graph traversal strategies. We defined several state machines that describe the behavior of a toaster. Here, we use the

flat state machine of Figure 1.2 to illustrate the applicability of depth-first and breadth-first. The algorithm to find a path from the initial pseudostate of a state machine to certain state s via depth-first search is shown in Figure 1.11. The returned path is a sequence of transitions. The initial call is $depthFirstSearch(initialNode, s)$.

```
01 Sequence depthFirstSearch(Node n, Node s) {
02   if(n is equal to s) { // found state s?
03     return new Sequence();
04   }
05   for all outgoing transitions t of n { // search forward
06     Node target = t.target; // target state of t
07     Sequence seq = depthFirstSearch(target, s);
08     if(seq is not null) { // state s has been found before
09       seq.addToFront(t); // add the used transitions
10       return seq;
11   } }
12   if(n has no outgoing transitions) // abort depth-search
13     return null;
14 }
```

Figure 1.11: Depth-first search algorithm.

For the example in Figure 1.2 on page 6, ParTeG generates exactly one test case to satisfy All-States. Figure 1.12 shows this test case in the presented notation.

```
TC: (s0, (push, , on), s1, (inc, , ), s7, (time, , off), s6, (defrost, , ),
     s4, (dec, , ), s2, (push, , on), s3, (inc, , ), s5, (stop, , off), s6)
```

Figure 1.12: Test case generated by ParTeG for All-States.

Figure 1.13 shows an algorithm for breadth-first search. Internally, it uses a tree structure to keep track of all paths. Just like a state machine, a tree is a directed graph with nodes and edges. Each node has incoming and outgoing edges. The nodes and edges of the tree reference nodes and edges of the state machine, respectively. It is initiated with the call $breadthFirstSearch(initialNode, s)$.

```
01 Sequence breadthFirstSearch(Node n, Node s) {
02   TreeStructure tree = new TreeStructure();
03   tree.addNode(n);
04   while(true) { // run forever (until sequence is returned with this loop)
05     NodeSet ls = tree.getAllLeaves(); // get all nodes without outgoing transitions
06     for all nodes/leaves l in ls {
07       if(l references s) { // compare to searched state
08         Sequence seq = new Sequence();
09         while (l.incoming is not empty) { // there are incoming transitions
10           seq.addToFront(l.incoming.get(0)); // add incoming transition
11           l = l.incoming.get(0).source; } // l is set to l's predecessor
12         return seq;
13       } // else
14       for all outgoing transitions t of l { // search forward - build tree
15         Node target = t.target; // target state of t
16         new_l = tree.addNode(target); // get tree node that references target
17         tree.addTransitionFromTo(t, l, new_l); // add an edge from node l
18           // to node new_l; this new edge references transition t
19 } } } }
```

Figure 1.13: Breadth-first search algorithm.

Both algorithms start at the initial pseudostate of the state machine depicted in Figure 1.2 on page 6. They traverse all outgoing transitions and keep on traversing until s has been visited. Here, we present the generated testing tree for breadth-first search in the toaster example. We assume that the goal is to visit state $S5$. The testing tree is shown in Figure 1.14. It contains only edges and nodes; events are not presented here.

Because of loops in transition sequences, the result may be in general an infinite tree. The tree, however, is only built and maintained until the desired condition is satisfied, that is, the identified state is reached. In this example, the right-most path reaches the state $S5$.

A finite representation of this possibly infinite tree is a reachability tree, where each state is visited only once. Figure 1.15 shows such a reachability tree for the toaster example. Again, the figure depicts only edges and nodes, but no event or effect information.

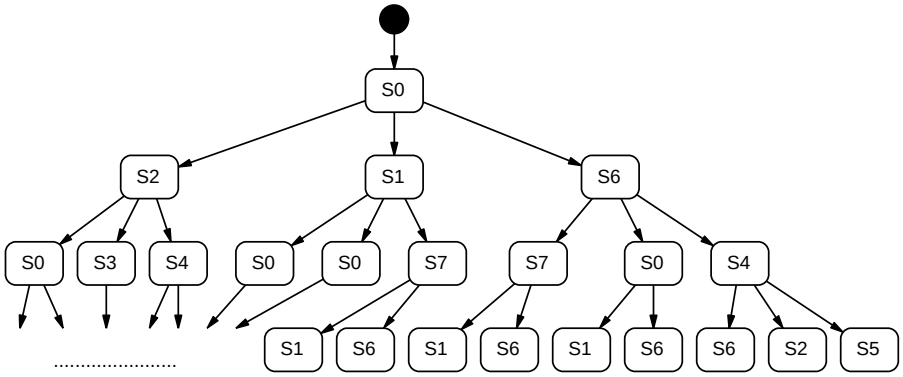

Figure 1.14: Testing tree that shows the paths for breadth-first search.

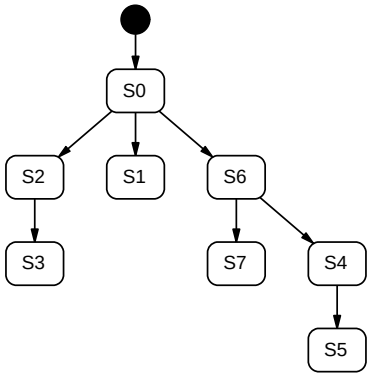

Figure 1.15: Reachability tree that shows only the paths to reach all states.

Graph traversal approaches can also be applied to hierarchical state machines such as presented in Figures 1.3. For each hierarchical state machine there exists an equivalent simple state machine; for instance, the models in Figure 1.3 and Figure 1.2 have exactly the same behaviour. Basically, each state in the flat state machine corresponds to a state configuration, that is, a set of concurrently active states, in the parallel state machine.

Extended state machine such as the one presented in Figure 1.4 can contain variables on infinite domains, and transitions can have arithmetic guard conditions and effects of arbitrary complexity. The problem of reaching a certain state or transition in an extended state machine is therefore non-trivial, and, in the general case, undecidable. Therefore, for such models, the state set is partitioned into equivalence classes, and representatives from the equivalence classes are selected. These methods will be described in the next section.

1.3 Input Value Generation

In this section, we present the second challenge for automatic test generation: selecting concrete input values for testing.

All previously presented test generation techniques are focussed on the satisfaction of coverage criteria that are applied to state machines. The corresponding test cases contain only the necessary information to traverse a certain path. Such test cases are called *abstract* – information about input parameters is given only partly as a partition of the possible input value space. Boundary value analysis is a technique that is focussed on identifying representatives of partitions that are as close as possible to the partition boundaries. In the following, we present partition testing as well as static and dynamic boundary value analysis.

1.3.1 Partition Testing

Partition testing is a technique that consists of defining input value partitions and selecting representatives of them [64, 128, 89] [24, page 302]. There are several variants of partition testing. For instance, the category partition method [96] is a test generation method that is focussed on generating partitions of the test input space. An example for category partitioning is the classification tree method (CTM) [60, 46], which enables testers to manually define partitions and to select representatives. The application of CTM to testing embedded systems is demonstrated in [83]. Basanieri and Bertolino use the category classification approach to derive integration tests with use case diagrams, class diagrams, and sequence diagrams [13]. Alekseev et al. [5] show how to reuse classification tree models. The Cost-Weighted Test Strategy (*CoWTeSt*) [14, 15] is focussed on prioritizing test cases to restrict their absolute number. CoWTeSt and the corresponding tool CowSuite have been developed by the PISATEL laboratory [103]. Another means to select test cases by partitioning and prioritization is the risk-driven approach presented by Kolb [79].

For test selection, a category partition table could list the categories as columns and test cases as rows. In each row, the categories that are tested are marked with an X. For the

toaster, such a category partition table could look like depicted in Table 1.3. There are two test cases *TC1* and *TC2* that cover all of the defined categories.

Test Cases	Defrost	No Defrost	High Browning Level	Low Browning Level
TC1		X		X
TC2	X		X	

Table 1.3: Category partition table.

Most of the presented partition testing approaches are focussed on functional black-box testing that are solely based on system input information. For testing with UML state machines, the structure of the state machine and also the traversed paths have to be included in the computation of reasonable input partitions. Furthermore, the selection of representatives from partitions is an important issue. Boundary value analysis (*BVA*) consists of selecting representatives close to the boundaries of a partition, that is, values whose distances to representatives from other partitions are below a certain threshold. Consider the example in Figure 1.4 on page 9. For the guard condition $s\_ht > 0$, 1 is a meaningful boundary value for s_ht to satisfy the condition and 0 is a meaningful value to violate the condition. The task is to derive these boundary values automatically.

Here, we present two approaches of integrating boundary value analysis and automatic test generation with UML state machines: static and dynamic boundary value analysis [125].

1.3.2 Static Boundary Value Analysis

In *static boundary value analysis*, BVA is included by static changes of the test model. For model-based test generation, this corresponds to transforming the test model. Model transformations for including BVA in test generation from state machines have been presented in [26]. The idea is to, for example, split a guard condition of the test model into several ones. For instance, a guard $[x >= y]$ is split into the three guards $[x = y]$, $[x = y + 1]$, and $[x > y + 1]$. Figure 1.16 presents this transformation applied to a simple state machine. The essence of this transformation is to define guard conditions that represent boundary values of the original guard's variables. As a consequence, the satisfaction of the transformed guards forces the test generator to also select boundary values for the guard variables. This

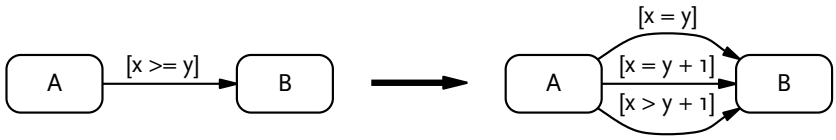

Figure 1.16: Semantic-preserving test model transformation for static BVA.

helps to achieve the satisfaction of, for example, All-Transitions [115, page 117], requires the satisfaction of each guard and, thus, the inclusion of static BVA. There are such approaches for model checkers or constraint solvers that include the transformation or mutation of the test model. As one example, the Conformiq Test Designer [38] implements the approach of static BVA.

The advantages of this approach are the easy implementation and the linear test effort. However, this approach has also several shortfalls regarding the resulting test quality. In [125], we present further details.

1.3.3 Dynamic Boundary Value Analysis

In *dynamic boundary value analysis*, the boundary values are defined dynamically during the test generation process and separately for each abstract test case. Thus, in contrast to static BVA, the generated boundary values of dynamic BVA are specific for each abstract test case. There are several approaches to implement dynamic BVA. In this section, we present a short list of such approaches.

In general, for dynamic boundary value analysis no test model transformations are necessary. For instance, an evolutionary approach can be used to create tests that cover certain parts of the model. In this case, a fitness function that returns good fitness values for parameters that are close to partition boundaries results in test cases with such input parameters that are close to these boundaries. Furthermore, any standard test generation approach can be combined with a constraint solver that is able to include linear optimization, for example, lp_solve [19] or Choco [112], for generating input parameter values. There are many constraint solvers [58, 53, 11, 117, 48] which could be used for this task. Besides the presented approaches to dynamic BVA, there are industrial approaches to support dynamic BVA for

automatic test generation with UML or B/Z [81, 110]. All these approaches to dynamic BVA that are based on searching forward.

Another approach of searching backward instead of forward is called *abstract backward analysis*. It is based on the weakest precondition calculus [49, 129, 30] and on searching backward. During the generation of abstract test cases, all guards to enable the abstract test case are collected and transformed into constraints of input parameters. As a result, the generated abstract test case also contains constraints about the enabling input parameters. These constraints define partitions and can, thus, be used for BVA. This approach has been implemented in the model-based test generation prototype ParTeG [122, 126, 123]. In this implementation, the test generation algorithm starts at certain model elements that are specified by the applied structural coverage criterion and iterates backward to the initial node. As a result, the corresponding structural [115] and boundary-based [81] coverage criteria can be combined.

1.4 Relation to Other Techniques

The previous two sections dealt with the basic issues of generating paths in the state machine and selecting meaningful input data, respectively. In this section, we show several other techniques that may be used to support the two basic issues. In the following, we present random testing in Section 1.4.1, evolutionary testing in Section 1.4.2, constraint solving in Section 1.4.3, model checking in Section 1.4.4, and static analysis in Section 1.4.5.

1.4.1 Random Testing

Many test generation approaches put a lot of effort in generating test cases from test models in a "clever" way, for instance, finding a shortest path to the model element to cover. It has been questioned whether this effort is always justified [104]. Any sort of black-box testing abstracts from internal details of the implementation, which are not in the realm of the test generation process. Nevertheless, these internals could cause the SUT to fail.

Statistical approaches to testing such as *random testing* have proven to be successful in many application areas [21, 85, 97, 34, 116, 35, 36]. Therefore, it has been suggested to apply random selection also to model-based test generation. In random testing, model coverage is not the main concern. The model abstracts from the SUT, but it is assumed that faults are randomly distributed across the entire SUT. Thus, random testing has often advantages over any kind of guided test generation. The model is used to create a large number of test cases without spending much effort on the selection of single tests. Therefore, random algorithms quickly produce results, which can help to exhibit design flaws early in the development process, while the model and SUT are still under development.

There are several publications on the comparison of random test generation techniques and guided test generation techniques. In [8], Andrews et al. use a case study to show that random tests can perform considerably worse than coverage-guided test suites in terms of fault detection and cost effectiveness. However, the effort of applying coverage criteria cannot be easily measured, and it is still unclear which approach results in higher costs. Mayer and Schneckenburger [86] present a systematic comparison of adaptive random testing techniques. Just like Gutjahr [63], Weyuker and Jeng [128], they also focus their work on the comparison of random testing to partition testing. Major reasons for the success of random testing techniques are that other techniques are immature to a certain extent or that the used requirements specifications are partly faulty. Finally, developers as well as testers make errors (see Beizer [17] for the prejudice *Angelic Testers*). For instance, testers can forget some cases or simply do not know about them.

Random test generation can also be applied to model-based testing with UML state machines. For instance, this approach can be combined with the graph traversal approach of the previous section so as the next transition to traverse is selected randomly. Figure 1.17 shows one possible random test generation algorithm. First, it defines the desired length of the test case (line 03). Afterwards, it selects and traverses one of the current node's outgoing transitions (line 06). This step is repeated until the current node has no outgoing transitions (line 07) or the desired test length has been reached (line 05). The resulting sequence is returned in line 13.

```
01 Sequence randomSearch(Node source) {
02   Sequence seq = new Sequence();
03   int length = random();
04   Node currentNode = source;
05   for(int i = 0; i < length; ++i) {
06     transitions = currentNode.getOutgoing();
07     if (transitions.isEmpty()) { break; }
08     traverse = randomly select a representative of transitions;
09     seq.add(traverse);
10     // set current node to target node of traverse
11     currentNode = traverse.getTarget();
12   }
13   return seq;
14 }
```

Figure 1.17: Random search algorithm.

Figure 1.18 shows several randomly generated test cases for our toaster example in Figure 1.2 on page 6.

```
TC1: (s0, (push, , on), s1)
TC2: (s0, (inc, , ), s6, (dec, , ), s0, (push, , ), s1, (stop, , ), s0)
TC3: (s0, (inc, , ), s6, (push, , ), s7, (dec, , ), s1)
```

Figure 1.18: Randomly generated test cases.

1.4.2 Evolutionary Testing

Evolutionary test generation consists of adapting an existing test suite until its quality, for example, measured with a fitness function, reaches a certain threshold. The initial test suite can be created using any of the above approaches. Based on this initial test suite, evolutionary testing consists of four steps: measuring the fitness of the test suite, selecting only the fittest test cases, recombining these test cases, and mutating them. In evolutionary testing, the set of test cases is also called population. Figure 1.19 depicts the process of evolutionary test generation. The dotted lines describe the start and the end of the test

generation process, that is, the initial population and – given that the measured fitness is high enough – the final population.

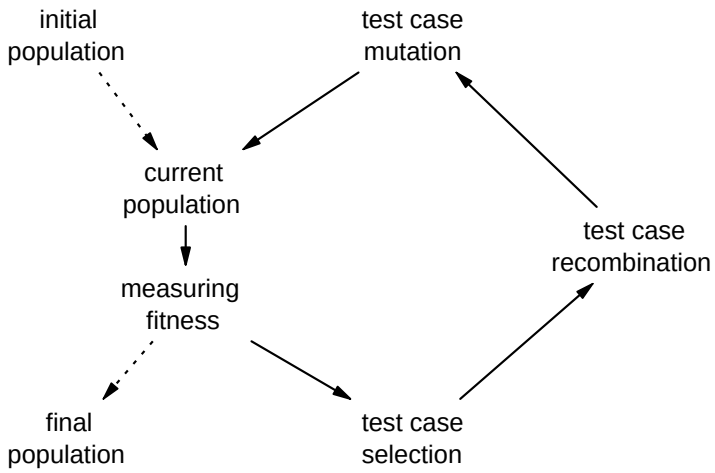

Figure 1.19: Evolutionary testing process.

There are several approaches to steer test generation or execution with *evolutionary approaches* [87, 99, 78, 68, 119]. An initial (e.g., randomly created or arbitrarily defined) set of test input data is refined using mutation and fitness functions to evaluate the quality of the current test suite. For instance, Wegener et al. [120] show application fields of evolutionary testing. A major application area is the area of embedded systems [111]. Wappler and Lammermann apply these algorithms for unit testing in object oriented programs [118]. Bühler and Wegener present a case study about testing an autonomous parking system with evolutionary methods [25].

Baudry et al. [16] present *bacteriological algorithms* as a variation of mutation testing and as an improvement of genetic algorithms. The variation from the genetic approach consists of the insertion of a new memory function and the suppression of the crossover operator. They use examples in Eiffel and a .NET component to test their approach and show its benefits over the genetic approach for test generation.

1.4.3 Constraint Solving

The constraint satisfaction problem is defined as a set of objects that must satisfy a set of contraints. The process of finding these object states is known as *constraint solving*. There are several approaches to constraint solving depending on the size of the application domain. We distinguish large but finite and small domains. For domains over many-valued variables, such as scheduling or timetabling, *Constraint Programming* (CP) [106], *Integer Programming* (IP) [105], or *Satisfiability Modulo Theories* (SMT) [12] with an appropriate theory is used. For extensionally representable domains, using solvers for *Satisfiability* (SAT-Solver) [20] and *Answer Set Programming* (ASP) [10, 57] are state of the art. SAT is often used for hardware verification [50].

There are many tools (*solvers*) to support constraint solving techniques. Examples for constraint programming tools are the Choco Solver [112], MINION [58], and Emma [53]. Integer programming tools are OpenOpt [94] and CVXOPT [45]. An example for SMT solvers is OpenSMT [109]. There are several competitions for solvers [11, 117, 48]. Constraint solving is also used for testing. Gupta et al. [61] use a constraint solver to find input parameter values that enable a generated abstract test case. Aichernig and Salas [4] use constraint solvers and mutation of OCL expressions for model-based test generation. Calame et al. [27] use constraint solving for conformance testing.

1.4.4 Model Checking

Model checking determines whether a model (e.g., a state machine) satisfies a certain property (e.g., a temporal logic formula). The model checking algorithm traverses the state space of the model and formula to deduce whether the model meets the property for certain (e.g., the initial or all) states. Typical properties are deadlock- or livelock-freedom, absence of race-conditions, etc.

If a model checker deduces that a given property does not hold, then it returns a path in the model as a counter-example. This feature can be used for automatic test generation [7,

56, 55]. For that, each test goal is expressed as a temporal logic formula, which is negated and given to the model checker. For example, if the test goal is to reach "state_6", then the formula expresses "state_6 is unreachable". The model checker deduces that the test model does not meet this formula and returns a counter-example. In the example, the counter-example is a path witnessing that state_6 is indeed reachable. This path can be used to create a test case. In this way, test cases for all goals of the coverage criterion can be generated, such that the resulting test suite satisfies the coverage criterion.

For our toaster example, the hierarchical state machine model depicted in Figure 1.3 can be coded in the input language of the NuSMV model checker as shown in Figure 1.20. The property states the states "toasting" and "on_d" are not reachable simultaneously. NuSMV finds that this is not true and delivers the path (test case) shown in Figure 1.21.

Model checking and test generation have been combined in different ways. Our example above is based on the work described in Hong et al. [72], which discuss the application of model checking for automatic test generation with control-flow-based and data-flow-based coverage criteria. They define state machines as Kripke structures [37] and translate them to inputs of the model checker SMV [73]. The applied coverage criteria are defined and negated as properties in the temporal logic CTL [37]. Callahan et al. [28] apply user-specified temporal formulas to generate test cases with a model checker. Gargantini and Heitmeyer [56] also consider control-flow-based coverage criteria. Abdurazik et al. [1] present an evaluation of specification-based coverage criteria and discuss their strengths and weaknesses when used with a model checker. In contrast, Ammann et al. [7] apply mutation analysis to measure the quality of the generated test suites. Ammann and Black [6] present a set of important questions regarding the feasibility of model checking for test generation. Especially, the satisfaction of more complex coverage criteria such as MC/DC [32, 31] is difficult because their satisfaction often requires pairs of test cases. Okun and Black [93] also present a set of issues about software testing with model checkers. They describe, for example, the higher abstraction level of formal specifications, the derivation of logic constraints, and the visibility of faults in test cases. Engler and Musuvathi [51] compare model checking to static analysis. They present three case studies that show that model checking often results in much more effort than static analysis although static analysis detects more errors than

```
MODULE main
VAR state_sidelatch : {inactive, active_defrosting, active_toasting};
    state_settemp : {warm, hot};
    state_setdefrost : {off_d, on_d};
    action : {push, stop, inc, dec, defrost, on, off, time, time_d};
ASSIGN
 init(state_sidelatch) := inactive;
 init(state_settemp) := warm;
 init(state_setdefrost) := off_d;
 next(state_sidelatch) := case
  state_sidelatch=inactive & action=push & state_setdefrost=on_d : active_defrosting;
  state_sidelatch=inactive & action=push : active_toasting;
  state_sidelatch=active_defrosting & action=time_d : active_toasting;
  state_sidelatch=active_toasting & action=time : inactive;
  state_sidelatch=active_defrosting & action=stop : inactive;
  state_sidelatch=active_toasting & action=stop : inactive;
  1 : state_sidelatch; esac;
 next(state_settemp) := case
  state_settemp=warm & action=inc : hot;
  state_settemp=hot & action=dec : warm;
  1 : state_settemp; esac;
 next(state_setdefrost) := case
  state_setdefrost=off_d & action=defrost & state_sidelatch=inactive : on_d;
  state_setdefrost=on_d & action=off : off_d;
  state_setdefrost=on_d & action=defrost & state_sidelatch=inactive : off_d;
  1 : state_setdefrost; esac;
 next(action) := case
  state_sidelatch=inactive & action=push : on;
  state_sidelatch=active_toasting & action=time : off;
  state_sidelatch=active_defrosting & action=stop : off;
  state_sidelatch=active_toasting & action=stop : off;
  1 : {push, stop, inc, dec, defrost, on, off, time, time_d}; esac;
 SPEC AG ! (state_sidelatch=active_toasting & state_setdefrost=on_d)
```

Figure 1.20: SMV code for the hierarchical state machine toaster model

```
>NuSMV.exe toaster-hierarch.smv
*** This is NuSMV 2.5.0 zchaff (compiled on Mon May 17 14:43:17 UTC 2010)
-- specification AG !(state_sidelatch = active_toasting & state_setdefrost = on_d)
-- is false as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state_sidelatch = inactive
  state_settemp = warm
  state_setdefrost = off_d
  action = push
-> State: 1.2 <-
  state_sidelatch = active_toasting
  action = on
-> State: 1.3 <-
  action = time
-> State: 1.4 <-
  state_sidelatch = inactive
  action = off
-> State: 1.5 <-
  action = defrost
-> State: 1.6 <-
  state_setdefrost = on_d
  action = push
-> State: 1.7 <-
  state_sidelatch = active_defrosting
  action = on
-> State: 1.8 <-
  action = time_d
-> State: 1.9 <-
  state_sidelatch = active_toasting
```

Figure 1.21: Result of NuSMV for the above example property

model checking. In [76], a tool is demonstrated that combines model checking and test generation. Further popular model checkers are the SPIN model checker [18], NuSMV [74], and the Java Pathfinder [70].

1.4.5 Static Analysis

Static analysis is a technique for collecting information about the system without executing it. For that, a verification tool is executed on integral parts of the system (e.g., source code) to detect faults (e.g., unwanted or forbidden properties of system attributes). There are several approaches and tools to support static analysis that vary in their strength from analysing only single statements to including the entire source code of a program. Static analysis is known as a formal method. Popular static analysis tools are the PC-Lint tool [59] for C and C++ or the IntelliJ IDEA tool [77] for Java. There are also approaches to apply static analysis on test models for automatic test generation [22, 95, 44, 100, 101]. Abdurazik and Offutt [2] use static analysis on UML collaboration diagrams to generate test cases. In contrast to state-machine-based approaches that are often focussed on describing the behavior of one object, this approach is focussed on the interaction of several objects. Static and dynamic analysis are compared in [9]. Ernst [52] argues for focussing on the similarities of both techniques.

Abstract Interpretation.

Abstract interpretation was initially developed by Patrick Cousot. It is a technique that is focussed on approximating the semantics of systems [40, 42] by deducing information without executing the system and without keeping all information of the system. An abstraction of the real system is created by using an *abstraction function*. Concrete values can be represented as abstract domains that describe the boundaries for the concrete values. Several properties of the SUT can be deduced based on this abstraction. For mapping these properties back to the real system, a *concretization function* is used. The abstractions can be defined, for example, using Galois connections, that is, a widening and a narrowing opera-

tor [41]. Abstract interpretation is often used for static analysis. Commercial tools are, for example, Polyspace ® [113] for Java and C++ or ASTRE [43]. Abstract interpretation is also used for testing [39, 102].

Slicing.

Slicing is a technique to slice parts of a program or a model by removing unnecessary parts and simplify, for example, test generation. The idea is that slices are easier to understand and to generate tests from than from the entire program or model [65]. Program slicing was introduced in the Ph.D. thesis of Weiser [121]. De Lucia [47] discusses several slicing methods (dynamic, static, backward, forward, etc.) that are based on statement deletion for program engineering. Fox et al. [54] present backward conditioning as an alternative to conditioned slicing that consists of slicing backward instead of forward. Whereas conditioned slicing provides answers to the question for the reaction of a program to a certain initial configuration and inputs, backward slicing finds answers to the question of what program parts can possibly lead to reaching a certain part or state of the program. Jalote et al. [75] present a framework for program slicing.

Slicing techniques can be used to support partition testing. For instance, Hierons et al. [71] use the conditioned slicing [29] tool ConSIT for partition testing and to test given input partitions. Harman et al. [66] investigate the influence of variable dependence analysis on slicing and present the corresponding prototype VADA. Dai et al. [46] apply partition testing and rely on the user to provide input partitions. Tip et al. [114] present an approach to apply slicing techniques to class hierarchies in C++. In contrast to the previous approaches, this one is focussed on slicing structural artifacts instead of behavioral ones.

1.5 Conclusion

Model-based test generation from state-based models is a topic that has already been dealt with for many years. Several books about different modeling languages, application scenar-

ios, and test generation approaches have been published. In this chapter, we presented an introduction to UML state machines as one kind of state-based models and showed several approaches to apply model-based test generation. For the interested reader, we provided several references for further studies.

In summary, we presented an introduction to automatic model-based test generation from UML state machines. For that, we gave a short introduction to UML state machines, presented a running example, and described how to generate tests from UML state machines. Afterwards, we sketched different approaches to derive abstract test cases, that is, paths on graphs such as state machines, and described several approaches to the generation of concrete input parameter values. Finally, we presented the core ideas of related techniques such as constraint solving and model checking and how to apply them to model-based test generation.

References

[1] Aynur Abdurazik, Paul Ammann, Wei Ding, and Jeff Offutt. Evaluation of Three Specification-Based Testing Criteria. *IEEE International Conference on Engineering of Complex Computer Systems*, page 179, 2000.

[2] Aynur Abdurazik and Jeff Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pages 383–395. Springer, 2000.

[3] Wasif Afzal, Richard Torkar, and Robert Feldt. A Systematic Review of Search-based Testing for Non-functional System Properties. *Information and Software Technology*, 51(6):957–976, 2009.

[4] Bernhard K. Aichernig and Percy Antonio Pari Salas. Test Case Generation by OCL Mutation and Constraint Solving. *International Conference on Quality Software*, pages 64–71, 2005.

[5] Sergej Alekseev, P. Tollkühn, P. Palaga, Zhen R. Dai, A. Hoffmann, Axel Rennoch, and Ina Schieferdecker. Reuse of Classification Tree Models for Complex Software Projects. In *Conference on Quality Engineering in Software Technology (CONQUEST)*, 2007.

[6] Paul Ammann and Paul E. Black. Test Generation and Recognition with Formal Methods. citeseer.ist.psu.edu/ammann00test.html, 2000.

[7] Paul E. Ammann, Paul E. Black, and William Majurski. Using Model Checking to Generate Tests from Specifications. In *ICFEM'98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, page 46, Washington, DC, USA, 1998. IEEE Computer Society.

[8] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, 32:608–624, 2006.

[9] Cyrille Artho and Armin Biere. Combined Static and Dynamic Analysis. In *AIOOL'05: Proceedings of the 1st International Workshop on Abstract Interpretation of Object-Oriented Languages*, ENTCS, Paris, France, 2005. Elsevier Science.

[10] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[11] Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 3rd annual satisfiability modulo theories competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.

[12] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Biere et al. [20], pages 825–885.

[13] Francesca Basanieri and Antonia Bertolino. A Practical Approach to UML-based Derivation of Integration Tests. In *4th International Software Quality Week Europe*, November 2000.

[14] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. CoWTeSt: A Cost Weighted Test Strategy. In *In Escom-Scope 2001*, pages 387–396, 2001.

[15] Francesca Basanieri, Antonia Bertolino, Eda Marchetti, Alberto Ribolini, Gaetano Lombardi, and Giovanni Nucera. An Automated Test Strategy Based on UML Diagrams. *Proceeding of the Ericsson Rational User Conference*, October 2001.

[16] Benoit Baudry, Franck Fleurey, Jean-Marc Jezequel, and Yves Le Traon. Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to .NET Components. In *Proceedings of ASE'02: Automated Software Engineering, Edinburgh*, 2002.

[17] Boris Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[18] Bell Labs. SPIN Model Checker. http://www.spinroot.com/, 1991.
[19] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.1. http://lpsolve.sourceforge.net/5.5/, 2004.
[20] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
[21] David L Bird and Carlos Urias Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.
[22] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. Using Static Analysis to Improve Automatic Test Generation. In *TACAS'00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 235–250, London, UK, 2000. Springer-Verlag.
[23] Lionel C. Briand, Yvan Labiche, and Qing Lin. Improving Statechart Testing Criteria Using Data Flow Information. In *ISSRE'05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 95–104, Washington, DC, USA, 2005. IEEE Computer Society.
[24] Manfred Broy, Bengt Jonsson, and Joost P. Katoen. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer, August 2005.
[25] Oliver Bühler and Joachim Wegener. Automatic Testing of an Autonomous Parking System using Evolutionary Computation, 2004.
[26] Simon Burton. *Automated Generation of High Integrity Tests from Graphical Specifications*. PhD thesis, University of York, 2001.
[27] Jens R. Calame, Natalia Ioustinova, Jaco van de Pol, and Natalia Sidorova. Data Abstraction and Constraint Solving for Conformance Testing. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 541–548, Washington, DC, USA, 2005. IEEE Computer Society.
[28] John Callahan, Francis Schneider, and Steve Easterbrook. Automated Software Testing Using Model-Checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.
[29] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned Program Slicing. *Information & Software Technology*, 40(11-12):595–607, 1998.
[30] Ana Cavalcanti and David A. Naumann. A Weakest Precondition Semantics for Refinement of Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, 2000.
[31] John Joseph Chilenski. MCDC Forms (Unique-Cause, Masking) versus Error Sensitivity. In *white paper submitted to NASA Langley Research Center under contract NAS1-20341*, January 2001.
[32] John Joseph Chilenski and Steven P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. In *Software Engineering Journal, Issue*, volume 9, pages 193–200, September 1994.
[33] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *Conformance testing methodologies and architectures for OSI protocols*, pages 391–400, 1995.
[34] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object Distance and Its Application to Adaptive Random Testing of Object-Oriented Programs. In *RT'06: Proceedings of the 1st International Workshop on Random Testing*, pages 55–63, New York, NY, USA, 2006. ACM Press.
[35] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental Assessment of Random Testing for Object-Oriented Software. In *ISSTA'07: Proceedings of the International Symposium on Software Testing and Analysis 2007*, pages 84–94, 2007.
[36] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand

Meyer. On the Predictability of Random Tests for Object-Oriented Software. In *ICST'08: Proceedings of the First International Conference on Software Testing, Verification and Validation*, April 2008.

[37] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[38] Conformiq. Qtronic. http://www.conformiq.com/.

[39] Patrick Cousot. Abstract Interpretation Based Program Testing. In *In Proc. SSGRR 2000 Computer & eBusiness International Conference, Compact disk paper 248 and electronic proceedings http://www.ssgrr.it/en/ ssgrr2000/proceedings.htm, 2000. Scuola Superiore G. Reiss Romoli*, 2000.

[40] Patrick Cousot. Automatic Verification by Abstract Interpretation. In *VMCAI'03: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 20–24, London, UK, 2003. Springer-Verlag.

[41] Patrick Cousot and Radhia Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming (PLILP '92)*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[42] Patrick Cousot and Radhia Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[43] Patrick Cousot, Radhia Cousot, Jrme Feret, Laurent Mauborgne, Antoine Min, and Xavier Rival. ASTRE Static Analyzer. http://www.astree.ens.fr/, 2003.

[44] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, New York, NY, USA, 2005. ACM.

[45] Joachim Dahl and Lieven Vandenberghe. CVXOPT 1.1.1. http://abel.ee.ucla.edu/cvxopt/, 2009.

[46] Zhen Ru Dai, Peter H. Deussen, Maik Busch, Laurette Pianta Lacmene, Titus Ngwangwen, Jens Herrmann, and Michael Schmidt. Automatic Test Data Generation for TTCN-3 using CTE. In *International Conference Software and Systems Engineering and their Applications (ICSSEA)*, December 2005.

[47] Andrea de Lucia. Program Slicing: Methods and Applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, November 2001.

[48] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczyński. The Second Answer Set Programming Competition. In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 637–654. Springer-Verlag, 2009.

[49] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[50] Rolf Drechsler, Stephan Eggersglüß, Görschwin Fey, and Daniel Tille. *Test Pattern Generation using Boolean Proof Engines*. Springer, 2009.

[51] Dawson Engler and Madanlal Musuvathi. Static Analysis Versus Software Model Checking for Bug Finding. citeseer.ist.psu.edu/engler04static.html, 2004.

[52] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA'03: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, Oregon, USA, May 9, 2003.

[53] Eve Software Utilities. Emma 1.0. http://www.eveutilities.com/products/emma, 2009.

[54] Chris Fox, Mark Harman, Rob Hierons, Ub Ph, and Sebastian Danicic. Backward Conditioning: A new Program Specialisation Technique and its Application to Program Comprehension. citeseer.ist.psu.edu/fox01backward.html, 2001.

[55] Gordon Fraser and Franz Wotawa. Using Model-Checkers to Generate and Analyze Property Relevant Test-Cases. *Software Quality Journal, 16 (2)*, pages 161–183, 2008.
[56] Angelo Gargantini and Constance Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. *ACM SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999.
[57] Michael Gelfond. Answer Sets. In Vladimir Lifschitz, Frank van Hermelen, and Bruce Porter, editors, *Handbook of Knowledge Representation*, chapter 7. Elsevier, 2008.
[58] Ian Gent, Chris Jefferson, Lars Kotthoff, Ian Miguel, Neil Moore, Peter Nightingale, Karen Petrie, and Andreas Rendl. MINION 0.9. http://minion.sourceforge.net/, 2009.
[59] Gimpel Software. PC-Lint for C/C++. http://www.gimpel.com/, 1985.
[60] Matthias Grochtmann and Klaus Grimm. Classification Trees for Partition Testing. *STVR: Software Testing, Verification and Reliability*, 3(2):63–82, 1993.
[61] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated Test Data Generation Using an Iterative Relaxation Method. In *SIGSOFT'98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 231–244, New York, NY, USA, 1998. ACM.
[62] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. UNA Based Iterative Test Data Generation and its Evaluation. In *ASE'99: Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, page 224, Washington, DC, USA, 1999. IEEE Computer Society.
[63] Walter J. Gutjahr. Partition Testing vs. Random Testing: The Influence of Uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999.
[64] Dick Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence (Program Testing). *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.
[65] Mark Harman and Sebastian Danicic. Using Program Slicing to Simplify Testing. *Software Testing, Verification & Reliability*, 5(3):143–162, 1995.
[66] Mark Harman, Chris Fox, Rob Hierons, Lin Hu, Sebastian Danicic, and Joachim Wegener. VADA: A Transformation-Based System for Variable Dependence Analysis. *IEEE International Workshop on Source Code Analysis and Manipulation*, page 55, 2002.
[67] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The Impact of Input Domain Reduction on Search-Based Test Data Generation. In *ESEC-FSE'07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*, pages 155–164, New York, NY, USA, 2007. ACM.
[68] Mark Harman and Phil McMinn. A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In *ISSTA'07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 73–83, New York, NY, USA, 2007. ACM.
[69] Siamak Haschemi. Model transformations to satisfy all-configuration-transitions on statecharts. In *6th Workshop on Model-Based Design, Verification and Validation (MoDeVVa 2009)*, October 2009.
[70] Klaus Havelund, Willem Visser, Flavio Lerda, Corina Pasareanu, John Penix, Masoud Mansouri-Samani, Owen O'Malley, Dimitra Giannakopoulou, Peter Mehlitz, and Peter Dillinger. Java PathFinder. http://javapathfinder.sourceforge.net/, 1999.
[71] Robert M. Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned Slicing Supports Partition Testing. In *Software Testing, Verification and Reliability*, 2002.
[72] Hyoung Hong, Insup Lee, Oleg Sokolsky, and Sung Cha. Automatic Test Generation from Statecharts Using Model Checking. In *In Proceedings of FATES'01 Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series*, 2001.

[73] ITC-IRST and Carnegie Mellon University and University of Genoa and University of Trento. SMV. http://www.cs.cmu.edu/ modelcheck/smv.html, 1998.
[74] ITC-IRST and Carnegie Mellon University and University of Genoa and University of Trento. NuSMV. http://nusmv.fbk.eu/, 1999.
[75] Pankaj Jalote, Vipindeep Vangala, Taranbir Singh, and Prateek Jain. Program Partitioning: A Framework for Combining Static and Dynamic Analysis. In *WODA'06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pages 11–16, New York, NY, USA, 2006. ACM Press.
[76] Thierry Jéron and Pierre Morel. Test Generation Derived from Model-Checking. In N. Halbwachs and D. Peled, editors, *CAV'99: Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, pages 108–122, London, UK, July 1999. Springer-Verlag.
[77] JetBrains. IntelliJ IDEA. http://www.jetbrains.com/, 2000.
[78] Susan Khor and Peter Grogono. Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically. In *ASE'04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 346–349, Washington, DC, USA, 2004. IEEE Computer Society.
[79] Ronny Kolb. A Risk-Driven Approach for Efficiently Testing Software Product Lines. citeseer.ist.psu.edu/630355.html, 2003.
[80] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
[81] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting. Boundary Coverage Criteria for Test Generation from Formal Models. In *ISSRE'04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 139–150, Washington, DC, USA, 2004. IEEE Computer Society.
[82] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling Dynamic Data Structures in Search Based Testing. In *GECCO'08: Proceedings of the 10th annual conference on Genetic and Evolutionary Computation*, pages 1759–1766, New York, NY, USA, 2008. ACM.
[83] K. Lamberg, M. Beine, M. Eschmann, R. Otterbach, M. Conrad, and I. Fey. Model-based Testing of Embedded Automotive Software using MTest, July 2005.
[84] Nashat Mansour and Miran Salame. Data Generation for Path Testing. *Software Quality Control*, 12(2):121–136, 2004.
[85] Johannes Mayer. On Testing Image Processing Applications with Statistical Methods. In *Software Engineering*, pages 69–78, 2005.
[86] Johannes Mayer and Christoph Schneckenburger. An Empirical Analysis and Comparison of Random Testing Techniques. In *ISESE'06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 105–114, New York, NY, USA, 2006. ACM Press.
[87] Gary Mcgraw, Christoph Michael, and Michael Schatz. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, 27:1085–1110, 1997.
[88] Phil McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *STVR: Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
[89] Simeon C. Ntafos. On Comparisons of Random, Partition, and Proportional Partition Testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, 2001.
[90] Object Management Group. Object Constraint Language (OCL), version 2.0. http://www.uml.org, 2005.
[91] Object Management Group. Unified Modeling Language (UML), version 2.2. http://www.uml.org, 2009.
[92] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA,*

October 28-30. 1999, Proceedings, volume 1723, pages 416–429. Springer, 1999.

[93] Vadim Okun and Paul E. Black. Issues in Software Testing with Model Checkers. citeseer.ist.psu.edu/okun03issues.html, 2003.

[94] Optimization Department of Cybernetic Institute. OpenOpt. http://openopt.org/, 2007.

[95] Thomas Ostrand, Elaine J. Weyuker, and Robert Bell. Using Static Analysis to Determine Where to Focus Dynamic Testing Effort. In *WODA'04: Workshop on Dynamic Analysis*, May 2004.

[96] Thomas J. Ostrand and Marc J. Balcer. The Category-Partition Method for Specifying and Generating Fuctional Tests. *Communications of the ACM*, 31(6):676–686, 1988.

[97] David Owen, Dejan Desovski, and Bojan Cukic. Random Testing of Formal Software Models and Induced Coverage. In *Random Testing*, pages 20–27, 2006.

[98] Christos H. Papadimitriou. On the complexity of edge traversing. *J. ACM*, 23(3):544–554, 1976.

[99] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[100] Jan Peleska, Helge Löding, and Tatiana Kotas. Test Automation Meets Static Analysis. In Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, and Marc Ronthaler, editors, *GI Jahrestagung (2)*, volume 110 of *Lecture Notes in Informatics*, pages 280–290. GI, 2007.

[101] Jan Peleska and Cornelia Zahlten. Integrated Automated Test Case Generation and Static Analysis. In *QA+Test 2007: International Conference on QA+Testing Embedded Systems*, 2007.

[102] Alessandra Di Pierro and Herbert Wiklicky. Probabilistic Abstract Interpretation and Statistical Testing. In *PAPM-PROBMIV '02: Proceedings of the Second Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pages 211–212, London, UK, 2002. Springer-Verlag.

[103] PISATEL LAB. http://www1.isti.cnr.it/ERI/special.htm, 2002.

[104] Alexander Pretschner. Zur Kosteneffektivität des modellbasierten Testens. In *MBEES'06: Modellbasierte Entwicklung eingebetteter Systeme*, pages 85–94, 2006.

[105] A. Ravi Ravindran, editor. *Operations Research and Management Science Handbook*. CRC Press, 2008.

[106] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[107] RTCA Inc. RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, December 1992.

[108] Shane Saunders. A comparison of data structures for dijkstra's single source shortest path algorithm, 1999.

[109] Natasha Sharygina, Robertoe Bruttomesso, Aliaksei Tsitovich, Simone Rollini, Stefano Tonetta, Chiara Braghin, and Katerina Barone-Adesi. OpenSMT. http://verify.inf.unisi.ch/opensmt, 2009.

[110] Smartesting. Test Designer. http://www.smartesting.com.

[111] Harmen Sthamer, Andr Baresel, and Joachim Wegener. Evolutionary Testing of Embedded Systems. In *QW'01: Proceedings of the 14th International Internet & Software Quality Week*, pages 1–34, 2001.

[112] The Choco Team. Choco Solver 2.1.0. http://choco.emn.fr/, 2009.

[113] The Mathworks Inc. Polyspace Embedded Software Verification. http://www.mathworks.com/products/polyspace /index.html, 1994.

[114] Frank Tip, Jong-Deok Choi, John Field, and G. Ramalingam. Slicing Class Hierarchies in C++. In *OOPSLA'96: Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Orogramming, Systems, Languages, and Applications*, pages 179–197, New York, NY, USA, 1996. ACM Press.

[115] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
[116] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-Based Testing. Technical Report 04/2006, Department of Computer Science, The Universiy of Waikato (New Zealand), 2006.
[117] Hans van Maaren and John Franco. The international sat competitions web page. http://www.satcompetition.org/, 2009.
[118] Stefan Wappler and Frank Lammermann. Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software. In *GECCO'05: Proceedings of the conference on Genetic and Evolutionary Computation*, pages 1053–1060, New York, NY, USA, 2005. ACM Press.
[119] Stefan Wappler and Ina Schieferdecker. Improving Evolutionary Class Testing in the Presence of Non-Public Methods. In *ASE'07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 381–384, New York, NY, USA, 2007. ACM.
[120] Joachim Wegener, Harmen Sthamer, and André Baresel. Application Fields for Evolutionary Testing. *Eurostar: Proceedings of the 9th European International Conference on Software Testing Analysis & Review*, 2001.
[121] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1979.
[122] Stephan Weißleder. ParTeG (Partition Test Generator). http://parteg.sourceforge.net.
[123] Stephan Weißleder. Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In *Models'09: 12th International Conference on Model Driven Engineering Languages and Systems*, October 2009.
[124] Stephan Weißleder. Simulated Satisfaction of Coverage Criteria on UML State Machines. In *International Conference on Software Testing, Verification, and Validation (ICST)*, 2010.
[125] Stephan Weißleder. Static and Dynamic Boundary Value Analysis. 2010.
[126] Stephan Weißleder and Holger Schlingloff. Deriving Input Partitions from UML Models for Automatic Test Generation. In Holger Giese, editor, *MoDELS Workshops*, volume 5002 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2007.
[127] Stephan Weißleder and Holger Schlingloff. Quality of Automatically Generated Test Cases based on OCL Expressions. In *ICST'08: International Conference on Software Testing, Verification, and Validation*, pages 517–520, 2008.
[128] Elaine J. Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.
[129] Robin W. Whitty. An exercise in weakest preconditions. *Software Testing, Verification & Reliability*, 1(1):39–43, 1991.