Model-based generation of test cases for reactive systems

Technical Report 2010-573

Karolina Zurowska and Juergen Dingel

Applied Formal Methods Group
School of Computing
Queen's University
Kingston, Ontario, Canada

July, 2010

Abstract

Testing is one of the most popular methods to ensure the quality of software. However the increasing complexity of software makes the testing more complex. One of the methods that reduce this complexity by automation is the model-based testing. In this approach models of software are used to generate a set of test cases, which are then executed on the system. This paper reviews methods used to generate test cases for reactive systems and focuses on methods, in which models are specified using precise or even formal languages. The presentation starts with techniques that use all specified behaviors to generate test cases. Next, this requirement is released and methods that use parts of models selected using coverage criteria or more general properties are shown. The paper also gives a comparative analysis of tools that implement the model-based test case generation. In this way a more practical view of the model-based testing methods and their requirements is also considered.

Contents

1 Introduction

Testing is one of the most often used methods for evaluating the quality of software products [9]. In the most general sense it is a process of executing a system or a program with intent to find errors [77]. Testing is therefore a very broad concept and can be classified using many different criteria, some of which are presented in Figure 1 [102]. The testing differs depending on what is tested (the levels of details axis), why it is tested (the characteristics axis) or how the tested entity is accessed (the accessibility axis). What is common for all types of testing is its increasing complexity [111], which is the direct consequence of growing sizes of software systems. Additionally these systems penetrate more and more areas and in some of them the demand for the quality is very high, for example in medically-related software [9].

Model-based testing aims to reduce the complexity of software testing and in terms of dimensions from Figure 1 it is functional black box testing at all levels of details. The underlying idea of this approach is presented in Figure 2 (adapted from [103, 107]). The most important components are a model and a system under test (SUT), which are related with a **conforms** relation. The model is an abstract specification of a behavior and in some cases of the structure of the system. The relation **conforms** holds if behaviors of the model and of the SUT are similar, where the exact definition of similarity differs between approaches. Based on the model and the definition of the relation **conforms** a set of test cases, called a test suite, is generated. The test suite is then executed on the SUT. Since models represent the required behavior, generated test cases also specify responses that are necessary to pass them during execution. The execution of the test suite is therefore a basis to infer whether the **conforms** relation holds. The crucial part of the process from Figure 2 is the test case generation, which is the topic of this paper.

The model-based testing approach is very attractive, because it promises the automation of the test case generation process [86]. This is possible because the generation is founded on models, which are usually precise or even formal specifications. Automatic test case generation reduces the costs of testing. Moreover, as opposed to the manual test case generation, it may also make the whole process

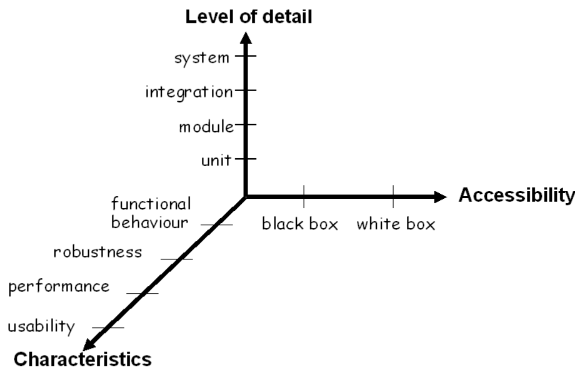


Figure 1: Dimensions and types of testing [102]

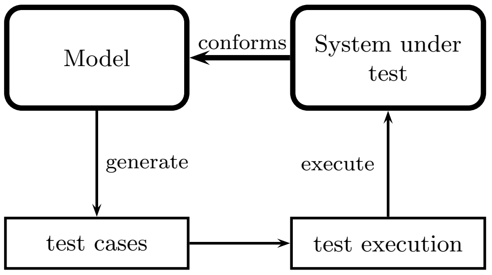


Figure 2: Model-based testing (adapted from [103, 107])

more systematic and more quantifiable. The biggest flaw is the necessity to build and maintain models, which is an additional and sometimes expensive activity [84]. Another problem may arise due to the limited scalability of modeling languages and the test case generation algorithms. Additionally for non-trivial systems the number of generated test cases might be impractical.

In this work the model-based testing approach is considered for reactive systems. Such systems continuously wait for a stimulus from their environment and after receiving it, they perform their computations and respond to the environment [74]. This model of computation is popular in modern software systems, especially in the embedded software that combine software and hardware components, when a reaction is an obvious abstraction [110]. On the other hand reactive systems are also more complex to test, and model-based testing has been proposed to deal with this complexity [17]. The restriction of the domain to reactive systems excludes from this paper works proposing model-based test case generation from specification languages which are not well suited for this domain. For instance approaches based on Z [52, 97] or B [72] languages, which are used rather to describe possible operations of systems, are not presented.

Besides restricting the domain to reactive systems, the emphasis in this paper is put on behavioral aspects of discrete models, which use formal specification languages. Such assumptions restrict the presented works in several ways. Firstly, this paper investigates works that model behavior, which excludes approaches based on models of static aspects of systems. Secondly, we consider discrete systems, so the works that operate on hybrid specification languages [98, 10] or tools like Reactis [90] are not included. Finally, only works that using models with formal or precise semantics are reviewed. This means the paper does not analyze some of the available commercial tools used for model-based testing like IBM Rational Rhapsody [59] or Confirmiq Qtronic [26].

This paper consists of two parts. The first one reviews the techniques and algorithms used to generate test cases from models. The second part presents the comparative analysis of tools, which implement some of the methods shown in the first part..

2 Review of test case generation algorithms

This part of the paper reviews the model-based test case generation methods, which are divided into three groups. In the first group, methods, which assume that the conformance relation (from Figure 2) is based on all specified behaviors, are shown. Because of this completeness, such an approach might be impractical, hence the two other groups take into account only selected behaviors. These two groups gather methods to generate test cases used to verify whether an SUT conforms to a selected part of a specification. The selection which part is of interest is additionally specified using coverage criteria or in properties.

The main criterion to distinguish methods for the test case generation based on coverage and on selected properties is how the additional information is given. In case of coverage, only a type of a required coverage criterion is given. Typically this means that test case generation algorithms for different coverage types are not the same. In works that deal with more general notion of properties, it is assumed that these properties are given as a separate specification. In some cases properties may coincide with coverage criteria if the language used to express properties allows it.

2.1 Conformance based on complete models

The conformance based on complete models assumes that all behaviors given in a specification are examined in order to check for the **conforms** relations. The general approach is shown in Figure 3. It assumes that there are models that represent a specification (from the set $SPEC$), an SUT (from the set $IMPL$) and a test suit (from the set $TEST$). The sets $SPEC$, $IMPL$ and $TEST$ are not necessarily related, but usually they are not disjoint. The goal of algorithms presented in this part can be specified as follows.

Problem 1. Given a specification $s \in SPEC$ and a relation **conforms** $\subseteq SPEC \times IMPL$ generate a test suite $ts \in TEST$. The test suite ts is executed on the $i \in IMPL$ and gives a verdict. A verdict is a function $v : (IMPL \times TEST) \rightarrow \{pass, fail\}$. The verdict $v(i, ts)$ is then the basis to check whether $(s, i) \in$ **conforms**.

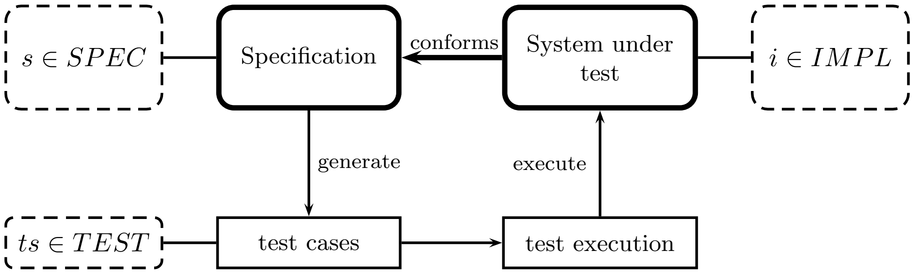


Figure 3: Model-based testing for conformance based on complete models

Based on the above problem definition, test cases provide a means to interact with a model of an implementation $i \in IMPL$ and to check its behavior. There are two properties of the generated test cases: soundness and exhaustiveness specified in Definition 1 [103].

Definition 1. A set of test cases (test suite) ts generated for a specification s is:

- sound iff implementations that failed to pass ts are incorrect, so iff:

$$\forall i \in IMPL \ : \ s \textbf{ conforms } i \Rightarrow v(i, ts) = \mathit{pass}$$

- exhaustive iff all incorrect implementations cannot pass ts, so iff:

$$\forall i \in IMPL \ : \ v(i, ts) = \mathit{pass} \Rightarrow s \textbf{ conforms } i$$

The test suite ts is complete if it is sound and exhaustive, so if correct implementations pass ts and incorrect do not, so iff:

$$\forall i \in IMPL \ : \ s \textbf{ conforms } i \Leftrightarrow v(i, ts) = \mathit{pass}$$

Based on Problem 1 the SUT is assumed to have certain properties, called a test hypothesis. In all methods presented in this section a test hypothesis includes the assumption that an implementation can be treated as a valid model from the set $IMPL$. This assumption requires that an implementation reacts to the stimuli and produces output recognized by the model in the way specified in the model.

The approaches presented in this section are classified according to the modeling language or formalism used to specify models in $SPEC$ and $IMPL$. This choice is important, since it impacts how specifications or implementations behave and influences the exact definition of the **conforms** relation. The presentation of the works starts with the more formal modeling languages, and then moves on to the more succinct representations.

2.1.1 Finite State Machines

Finite State Machines (FSMs) were used as specifications in the most traditional branch of model-based conformance checking, with the first works dating back to the 1950s [71]. During this time many algorithms have been proposed. Currently FSMs are extensively and successfully used to test the communication protocols [71].

The definition of an FSM is as follows [71]:

Definition 2. A Finite State Machine M is a tuple $M = (I, O, S, \delta, \lambda)$, where:

- I, O and S are finite, nonempty sets of input and output symbols and states,

- $\delta : S \times I \rightarrow S$ is a total function that defines transition between states (extended to sequences of input symbols $\delta : S \times I^* \rightarrow S$),

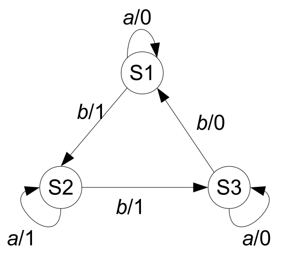


Figure 4: An example of a Finite State Machine [71].

- $\lambda : S \times I \rightarrow O$ is a total function that defines output generated after input symbols (extended to sequences of input and output symbols $\lambda : S \times I^* \rightarrow O^*$).

Example 1. Figure 4 presents an example of an FSM. This FSM is characterized with sets $S = \{S1, S2, S3\}$, $I = \{a, b\}$ and $O = \{1, 0\}$. The labels next to arcs define input and output pairs, so functions δ and λ. For example $\delta(S1, b) = S2$, $\lambda(S1, a) = 0$ and $\lambda(S1, ab) = 01$.

FSMs serve as $SPEC$ and $IMPL$ sets from Figure 3 and the **conforms** relation uses the equivalence between two FSMs. Two FSMs S and M, with common input and output symbols, are equivalent if for every state $s \in S_s$ in the machine S there is a state $q \in S_M$ in the machine M such that for each sequence $x \in I^*$ machines have the same output sequences, so $\lambda_S(s, x) = \lambda_M(q, x)$, and the same holds for all states of M [71]. Hence, the relation **conforms** requires that both a specification $s \in SPEC$ and an implementation $i \in IMPL$ have the same output sequences for all possible input sequences. The test hypothesis consists of the following assumptions:

- a specification S is a strongly connected and reduced (minimal) FSM, the second condition guarantees that all states can be distinguished by some input sequence,
- an implementation is an FSM, which implies finiteness of states of implementation and that in any state all inputs are enabled,
- input and output symbols of an implementation include all input and output symbols of a specification.

One of the most popular algorithms for the test case generation based on FSMs is the W-method [22]. This method assumes that there is a distinguished initial state and that an FSM has a reliable reset signal that moves it to this initial state. The test cases are based on sequences from two sets: a transition cover set (P set) and a characterizing set (W set).

In order to construct a P set, Algorithm 1 is used. It builds a test tree T in the breadth first search manner, by starting from the initial state and labeling nodes of the tree T with states and its edges with input symbols. During the exploration of states, the set *Term* contains terminated states, i.e., states that already have been explored. The outcome of the algorithm is a set of all paths in the tree T including partial ones.

Example 2. For the FSM in Example 1 and with $S1$ as the initial state, the algorithm builds a tree with its root labeled $S1$. On the first level this root node is explored. For all inputs $\{a, b\}$ edges are added and connected to nodes with labels $S1$ and $S2$. On the second level node $S1$ is explored first, but it is already present in the previous level, so it is added to the $Term$ set. The next is the node $S2$, which again has 2 children nodes. After all states $S1, S2, S3$ are in the $Term$ set, the partial paths in the constructed tree are taken. This gives a set $P = \{\varepsilon, a, b, ba, bb, bba, bbb\}$.

To build a characterization set W, Algorithm 2 is used. It partitions a set of states S using the output they produce in response to the same input sequence, until all partitions are singletons. A sequence x in this algorithm is called a separating sequence and it must exists for each pair of states, since an FSM is minimal.

Algorithm 1 Algorithm for constructing a transition cover set (the test tree method) [22]

```
Require: A minimal FSM $M = (I, O, S, \delta, \lambda)$ with the initial node $s1 \in S$
  label the root of the tree $T$ with $s1$
  $k \leftarrow 1$
  $Term \leftarrow \emptyset$
  while $S \neq Term$ do
    for all nodes $n$ at level $k$ of the tree $T$ from left to right do
      if $n$ is equal to another node at level $j \leq k$ then
        $Term \leftarrow Term \cup \{n\}$
      else
        $s \leftarrow$ label of a node $n$
        for all $i \in I$ do
          add a child $n_c$ to $n$
          label an edge with $i$ and $n_c$ with $\delta(s, i) = s'$
        end for
      end if
    end for
    $k \leftarrow k + 1$
  end while
  P$\leftarrow$ labels from edges for all partial paths from $T$
  return P
```

Example 3. For the FSM in Example 1 we have $B_1 = \{S1, S2, S3\}$. Then for $S1$ and $S2$ the input sequence that distinguishes them is a, because $\lambda(S1, a) = 0$ and $\lambda(S2, a) = 1$. The set B_1 is partitioned into $B_2 = \{S1, S3\}$ (the common output on a is 0) and $B_3 = \{S2\}$. At this point $W = \{a\}$. Now the set B_2 is partitioned. The separating sequence for $S1$ and $S3$ is b, because $\lambda(S1, b) = 1$ and $\lambda(S3, b) = 0$. This leads to the following partitions $B_4 = \{S1\}$ and $B_5 = \{S3\}$ and $W = \{a, b\}$. The remaining sets B_3, B_4, B_5 are singletons and the algorithm terminates.

Test cases are the result of concatenation of all sequences from sets P and W along with a reset signal. In this way each transition in the specification is executed and its target state is checked whether it is the expected one. Simple concatenation of sequences from sets P and W is not enough if an implementation contains more states than a specification. In such a case all additional states must be traversed. This is achieved by adding all possible combinations of input symbols, with the length up to the difference in number of states between the specification and the implementation. A set of all test cases is defined as:

Definition 3. A set of test cases in W-method for a specification FSM $M = (I, O, S, \delta, \lambda)$ is:

$$TC = \{reset\} \cdot P \cdot (I^0 \cup I^1 \cup I^2 ... \cup I^{(m-n)}) \cdot W$$

where:

- P is a transition cover set (Algorithm 1),
- I^0 is an empty sequence, I^k is a set of all sequences from I combined k-times,
- m is a number of states in an implementation and n in a specification M,

Algorithm 2 Algorithm for constructing a characterization set [22]

```
Require: A minimal FSM $M = (I, O, S, \delta, \lambda)$
  $W \leftarrow \emptyset$
  $B_1 \leftarrow S$
  while $B_i$ is not singleton for all $i = 0, .., n$ do
    if $s, t \in B_j$ for some $j$ and $s \neq t$ then
      find $x \in I^*$ such that $\lambda(s, x) \neq \lambda(t, x)$
      partition $B_j$ into $B_{j_1}, ..., B_{j_m}$ such that $r, r' \in B_{j_k} \Leftrightarrow \lambda(r, x) = \lambda(r', x)$ for all $k = 1, ..., m$
      $W \leftarrow W \cup \{x\}$
    end if
  end while
  return W
```

- W is a characterization set (Algorithm 2).

Example 4. For the FSM as in Example 1 if the implementation has the same number of states as the specification the generated test cases are:

$$TC = \{reset\} \cdot P \cdot W = \{reset\} \cdot \{\varepsilon, a, b, ba, bb, bba, bbb\} \cdot \{a, b\}$$

So some of the test cases are $\{reset\ a\ a, reset\ a\ b, reset\ b\ b\ a\ a, reset\ b\ b\ a\ b\}$

The optimizations of the W-method, which can produce shorter sequences, are:

- If an FSM has a single sequence that produces different output for each state (called a distinguishing sequence), then this sequence constitutes a W set [95].
- The partial W-method [35] is divided into two phases. In the first phase, a state cover set (Q set) is used, which is a set of input sequences such that for each state $s \in S$ there is a sequence $x \in Q$ that takes machine to this state: $\delta(s_1, x) = s$, where s_i is the initial state. The input sequences from the Q set are concatenated with the sequences from the W set, and such test sequences check whether all states are correctly implemented. In the second phase, all transitions that were not used in the first phase are tested. In order to check whether their target state is correct, only the subset of the W set is used, namely these sequences that can distinguish this particular target state from the others.
- An input sequence $x \in I^*$ is a unique input/output sequence (UIO) for a state s if an output generated for this state is different that generated by all other states (so for all states $q \neq s$: $\lambda(s, x) \neq \lambda(q, x)$). The UIOv method [21] that uses such sequences is similar to the partial W-method and also contains two phases. However in the first phase a set of UIO sequences for all states is used as a W set, and in the second phase only a specific UIO sequence for a target state is used.

If a reliable reset signal is not available in an implementation, then UIO or distinguishing sequences are used [2]. These methods first check every state $s_1, s_2, ...$ of an FSM in the given order, by transferring a machine to each state and then by applying UIO or a distinguishing sequence in this state. In the second phase all transitions between states s_i, s_j are tested by moving a machine to a state $s_{(i-1)}$, checking this state and then moving to s_i state (based on the previous phase - the correctness of s_i has been already verified), applying an input symbol and finally checking s_j state.

FSMs can be extended to include variables and operations on them. The definition of an extended FSMs (EFSM) is as follows [71]:

Definition 4. An Extended Finite State Machine (EFSM) is a tuple $M = (I, O, S, \overrightarrow{x}, T)$, where:

- I, O, S are non-empty, finite sets of input and output symbols and states,
- $\overrightarrow{x}$ are variables,
- T is a set of transitions, each of which is a tuple $t = (s_t, q_t, a_t, o_t, P_t, A_t)$, where s_t is the current state, q_t is the next state, a_t is an input symbol, o_t is an output symbol, $P_t(\overrightarrow{x})$ is a predicate on the variables and $A_t(\overrightarrow{x})$ updates values of variables. Upon receiving an input a_t in a state s_t if current values of variables $\overrightarrow{x}$ ensures that $P_t(\overrightarrow{x}) = TRUE$, then the current state is q_t and values of variables are as in $A_t(\overrightarrow{x})$.

An EFSM is a more succinct representation of an ordinary FSM and can be translated into it. In an FSM translated from an EFSM a state is a configuration: a pair of a state in the EFSM and of an assignment of variables, which record values of variables in this state. Such FSMs can be very large in terms of number of states and they might not be minimal, which is required by the previous algorithms. Algorithms that directly construct minimized versions of EFSMs have been proposed [70]. For the minimized machines the algorithms, like the W-method, can be applied to generate test cases.

2.1.2 Labeled Transition Systems

Labeled Transition Systems (LTSs) are another formalism used to generate test cases. LTSs can model non-determinism: from any state multiple transitions with the same action are possible and states may change in an unobservable manner using special internal actions. This poses challenges on how to test the conformance relation between LTSs. The general approach is to construct a set of observers (tests) and to conclude that two LTSs are equivalent if they pass the same tests. However tests can have different capabilities with respect to the way they interact with an LTS. If a test can only observe actions taken by an LTS, then two systems are equivalent if they have the same sequences of actions (trace equivalence) [28]. But if a test can also compare alternative actions to choose from, then systems are equivalent if at any time they enable the same set of actions (testing equivalence, bisimulation) [29, 28]. Finally if a test can examine sets of refused actions, then systems are equivalent if they always refuse the same sets of actions (refusal testing) [82].

The above equivalence relations assume symmetric interactions between LTSs. But reactive systems require that interactions of a system with its environment are based on inputs and outputs, therefore they have distinct roles. From the perspective of a system, it can always control its output actions and has no control over received input, and for an environment it is the opposite. In order to model this, Labeled Transition Systems with inputs and outputs were introduced [101]:

Definition 5. A Labeled Transition System with inputs and outputs is a tuple (Q, L_I, L_U, T, q_0), where Q is a non-empty set of states, L_I and L_U are disjoint sets of input and output labels, $T \subseteq (Q \times (L_I \cup L_U \cup \{\tau\}) \times Q)$ is a transition relation ($\tau \notin (L_I \cup L_U)$ is an internal action) and q_0 is the initial state. A transition $(q, a, q') \in T$ is denoted also as $q \xrightarrow{a} q'$. An LTS is sometimes identified simply by its initial state q_0.

LTSs with inputs and outputs are used as specifications, so they represent the $SPEC$ set from Figure 3. To model implementations, i.e., the set $IMPL$, Input-Output Transition Systems (IOTS) are defined [101]:

Definition 6. An Input-Output Transition System (IOTS) is an LTS with inputs and outputs $S = (Q, L_I, L_U, T, q_0)$ in which all input actions are enabled in any reachable state (possibly after some internal actions τ):

$$\forall q \in der(q_0), \forall a \in L_I : q \overset{a}{\Longrightarrow}$$

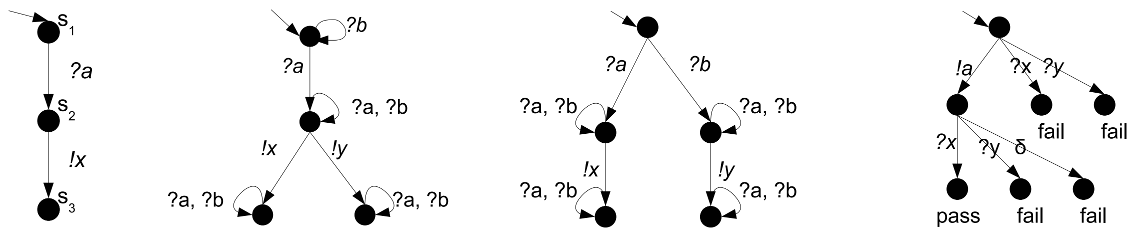
where :

- $q \overset{a}{\Longrightarrow} q'$ iff $\exists q_1, q_2' : q \xrightarrow{\tau^*} q_1 \xrightarrow{a} q_2 \xrightarrow{\tau^*} q'$ (τ^* is a sequence of 0 or more τ actions),
- $q \overset{a}{\Longrightarrow}$ iff $\exists q' : q \overset{a}{\Longrightarrow} q'$,
- $der(q) = \{q' \in Q | \exists \sigma \in (L_U \cup L_I)^* : q \overset{\sigma}{\Longrightarrow} q'\}$ ($(L_U \cup L_I)^*$ is a set of sequences that concatenate zero or more actions from $(L_U \cup L_I)$)

Example 5. An example of an LTS with inputs and outputs is given in Figure 5(a). It accepts an input a and outputs x. In Figures 5(b) and 5(c) there are examples of IOTSs with $L_I = \{a, b\}$. Input enabling is ensured by self loops in states that do not accept all inputs.

If an LTS S interacts with its environment and cannot perform any output action, then we say that S is in a quiescent state. This state is detected with an additional output $\delta \notin (L_U \cup L_I)$. Sequences of input and output actions possible in an LTS are called traces and if they contain δ they become suspension traces [103] as in Definition 7.

Definition 7.
1. A state q of an LTS $S = (Q, L_I, L_U, T, q_0)$ is quiescent, denoted by $\delta(q)$, iff $\forall \mu \in (L_U \cup \tau) : not\ \exists q' \in Q : q \xrightarrow{\mu} q'$,
2. $Straces(S) \subseteq (L_I \cup L_U \cup \delta)^*$ are possible traces of S enhanced with δ whenever a state q, such that $\delta(q)$, is reached.



(a) s_1 (b) i_1 (c) i_2 (d) tc

Figure 5: Examples of LTSs (a) an LTS with input and outputs (b), (c) input output transition systems(IOTS) [103], (d) an IOTS as a test case (a symbol ? precedes input actions and ! output actions, an incoming arrow indicates an initial state q_0).

The implementation relation **conforms** from Figure 3 is called the **ioco** relation (**i**input **o**utput **co**nformance). According to the definition of this relation, an implementation $I \in IOTS$ conforms to a specification $S \in LTS$ if after any observation possible in S, outputs generated by I are foreseen by S. More formally [103]:

Definition 8. The relation **ioco**$\subseteq IOTS \times LTS$ is defined (for $I \in IOTS$ and $S \in LTS$, here coincide with their initial states):

$$I \textbf{ ioco } S \Leftrightarrow_{def} \forall \sigma \in Straces(S) : out(I \ \ after \ \sigma) \subseteq out(S \ \ after \ \sigma)$$

where:

- $(q \ \ after \ \sigma) = \{p' \in Q | q \overset{\sigma}{\Longrightarrow} p'\}$ is a set of states reachable from q after a sequence σ; for a set of states P: $(P \ \ after \ \sigma) = \bigcup_{q \in P}(q \ \ after \ \sigma)$,
- $out(q) = \{x \in L_U | q \xrightarrow{x}\} \cup \{\delta | \delta(q)\}$ is a set of enabled outputs and quiescence in a state q; for a set of states P: $out(P) = \bigcup_{q \in P} out(q)$

Example 6. For LTSs from Example 5, s_1 can serve as a specification (Figure 5(a)). The IOTS i_1 (Figure 5(b)) is not in the **ioco** relation with this specification, because after the sequence $?a$ it may output $!y$, which is not part of the specification ($out(i_1 \ \ after \ ?a) \nsubseteq out(s_1 \ \ after \ ?a)$). The IOTS i_2 (Figure 5(c)) is in the **ioco** relation with s_1, because its outputs are the same as in s_1 for all suspension traces of s_1. Although $out(i_2 \ \ after \ ?b) = \{!y\} \nsubseteq out(i_1 \ \ after \ ?b) = \emptyset$, but we also have $?b \notin Straces(s_1)$. This means that an implementation must not produce any outputs other than specified, but the specification can be partial, so behavior of an implementation on traces not in $Straces(S)$ is unconstrained.

For the **ioco** relation the test case generation algorithm (Algorithm 3) [103] generates another IOTS that represents a test case (so $TEST$ in Figure 3 is a subset of all IOTSs). Such an IOTS has inversed input and output actions and two distinguished types of states **fail** or **pass**, which are always reachable and final, meaning that no inputs can be further sent to an implementation. The algorithm allows a choice from three possibilities. The first one is to mark a current state with **pass** and terminate. The second choice is to send an input and check specified outputs (not quiescence). The last choice is to check outputs and if no output is received from an implementation then to observe quiescence. Test cases generated by this algorithm are sound (according to Defintion 1) and a test suite that contains all possible test cases generated by the algorithm is also exhaustive [101].

Example 7. The test case generation algorithm for a specification s_1 in Example 5 starts with a set $S = (s_1 \ \ after \ \varepsilon) = \{s_1\}$. At this point one of the three choices can be applied. If the second possibility is selected then an input $?a$ is explored, because $(S \ \ after \ ?a) \neq \emptyset$. Since $out(S) = \emptyset$ then all actions in $L_U = \{x, y\}$ lead to states marked **fail**, which is represented with transitions labeled $?x, ?y$. Now S is set to $(S \ \ after \ ?a) = \{s_2\}$. There are no enabled inputs in s_2, so the test case can be terminated

Algorithm 3 An abstract test case generation algorithm for **ioco** relation [103].

```
Require: a specification $S = (Q, L_I, L_U, T, q_0)$
  S ← $s\ after\ \epsilon$
  t is empty IOTS
  loop
    1. change current state to pass
    t ← pass
    return t;
    2. choose input from inputs enabled in states in S
    if ($a \in L_I$) and (($S$ after $a$) $\neq \emptyset$) then
      add $t \xrightarrow{a} t_a$, S ← $S$ after $a$, explore $t_a$
      for all $x_i \in out(S)$ and $x_i \neq \delta$ do
        add transition $t \xrightarrow{x_i} t_{x_i}$, S ← $S$ after $x_i$, explore $t_{x_i}$
      end for
      for all $x_i \notin out(S)$ and $x_i \neq \delta$ do
        add transition $t \xrightarrow{x_i}$fail
      end for
    end if
    3. check outputs possible in states in S
    for all $x_i \in out(S)$ do
      add $t \xrightarrow{x_i} t_{x_i}$, S ← $S$ after $x_i$, explore $t_{x_i}$
    end for
    for all $x_i \notin out(S)$ do
      add $t \xrightarrow{x_i}$fail
    end for
  end loop
```

or outputs can be checked. The latter produces transitions labeled with $?x$,$?y$ and δ, and the last two actions lead to the **fail** state, since $y \notin out(S)$ and $\delta \notin out(S)$. After the transition $?x$ an additional state is added to the test case and S is set to $(S\ after\ !x) = \{s_3\}$. Now the first choice is applied and this state is marked as the **pass** state. The resulting IOTS is shown in Figure 5(d).

There are several extensions of the **ioco** relation and of Algorithm 3. The first one, called **mioco**, assumes that inputs and outputs are arbitrarily partitioned into classes [47]. In this way the assumption of input enabling is relaxed to a given class of inputs, and the algorithm is adjusted to deal only with the specific class. Another extension considers variables and data [31] and defines **ioco** for symbolic specifications. Algorithm 3 is enhanced with constraint solving, before sending inputs and with checking constraints for generated outputs. Finally, a combination of the relation **ioco** with actions refinement has been proposed [108], which allows actions of a specification and of an implementation to be at different levels of abstractions. The test cases are generated from an abstract specification as in Algorithm 3, but then they are then refined based on the given refinement relation between actions.

2.1.3 Real-time specifications

In some reactive systems timing requirements cannot be excluded from models, because they may influence responses to inputs. For example, two consecutive alarm signals might trigger different actions, depending on how much time has elapsed between them. In such a case the behavior of a system changes due to the passage of time and this should be reflected in its specification. Specifications use two models of time: discrete and dense (real-time) [5]. In the first model, time progresses in ticks and events happen after ticks. Because such models are discrete, the previously described methods can be applied to generate test cases. In dense or real-time models time progresses in infinitely small amounts. In turn actions may happen at infinitely many moments, which makes the test case generation much more challenging. The way to overcome this challenge is to represent real-time specifications in a discrete way [48].

The most common formalism to represent timed specifications are timed automata [3]. Their definition is given in Definition 9. For reactive systems it is assumed that a set of actions is partitioned into input and output actions.

Definition 9. 1. Let X be a finite set of real-valued variables called clocks. A set of constraints

$\Phi(X)$ is defined using the following grammar (x is a clock, c is an integer): $\phi ::= x \leq c \mid c \leq x \mid x < c \mid c < x \mid \phi \wedge \phi$.

2. A timed automaton is a tuple $\mathcal{A} = (L, L_0, \Sigma, X, Inv, E)$, where:
   - L is a finite set of locations and L_0 is a set of initial locations,
   - Σ is a finite set of actions,
   - X is a finite set of clocks,
   - $Inv : L \rightarrow \Phi(X)$ is a function assigning constraints to locations (invariants),
   - $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of transitions. A transition $(l, a, g, c, l') \in E$ represent a change from a location l to l' upon receiving a. A guard g is an enabling condition and a set of clocks c is reset after this transition.

The semantics of a timed automaton is described with an LTS [3]. Such an LTS has states that reflect a location and values of clocks. Its transitions are either delays of time or the execution of actions. Definition 10 gives the semantics more formally.

Definition 10. 1. A clock valuation v is a mapping from a set of clocks X into non-negative reals ($\mathbb{R}^{\geq 0}$). A delay $d \in \mathbb{R}^{\geq 0}$ is denoted by $v + d$ and means that each clock $x \in X$ has the value $v(x) + d$. A reset of clocks $Y \subseteq X$ is denoted with $v[Y := 0]$, which assigns 0 to all $y \in Y$ and leaves $v(x)$ unchanged, for all other clocks $x \in (X \backslash Y)$. The set of all possible valuations is denoted by $V(X)$. For $v \in V(X)$ and $\phi \in \Phi(X)$: $v \models \phi$ denotes the fact that the valuation v satisfies the constraints ϕ.

2. Semantics for a timed automaton $\mathcal{TA} = (L, L_0, \Sigma, X, Inv, E)$ is given by an LTS $\mathcal{S}(\mathcal{TA}) = (S, S_0, A, T)$, where:
   - S is a set of states, which are pairs $(l, v) \in L \times V(X)$ and for each pair $v \models Inv(l)$,
   - $S_0 \subset S$ are initial states, and $(l, v) \in S_0$ iff $l \in L_0$ and $v(c) = 0$ for all clocks $c \in X$,
   - $A = \Sigma \cup \mathbb{R}^{\geq 0}$,
   - $T \subseteq S \times S$ is the smallest transition relation that satisfies (for $d, d' \in \mathbb{R}^{\geq 0}$):

$$\frac{\forall d' < d : (v + d') \models Inv(l), v' = v + d}{(l, v) \xrightarrow{d} (l, v')} \qquad \frac{(l, a, g, c, l') \in E, v \models g, v' = v[c := 0], v' \models Inv(l')}{(l, v) \xrightarrow{a} (l', v')}$$

An LTS that represents semantics of a timed automaton is non-deterministic and infinite. So it cannot be used as the basis to generate test cases. To overcome the non-determinism Timed Input Output Automata have been defined, which restrict a timed automaton with input and outputs in the following ways [96]:

Definition 11. A Timed Input Output Automaton (TIOA) is a timed automaton $\mathcal{TA}$ with inputs and outputs, its semantics given by $\mathcal{S}(\mathcal{TA})$, that has the following properties:

- determinism: in each location of $\mathcal{TA}$ only one transition with a given action can be enabled, that is if there are two transitions with the same action their guards cannot be both satisfied for the same valuation of clocks,
- isolated outputs: if in a given state of $\mathcal{S}(\mathcal{TA})$ a transition with an output action is enabled (guards are satisfied), then no other transition (neither an action or a delay) is possible from this state,
- input enabling: for all states of $\mathcal{S}(\mathcal{TA})$ in which an output action is not enabled, transitions with input actions are always possible,
- progressiveness: it is always possible to delay time, so from each state in $\mathcal{S}(\mathcal{TA})$ with a given valuation, a state with the greater valuation of clocks is reachable.

An LTS $\mathcal{S}(\mathcal{TA})$ that represents the semantics of a TIOA $\mathcal{TA}$ is still infinite due to the infinite number of possible clock valuations. One of the solutions to this problem is to use a grid automaton [30, 96], which is a proper subautomaton of the LTS $\mathcal{S}(\mathcal{TA})$. In such an automaton delays are restricted only to values 2^{-n}, where $n \in \mathbb{N}$ is the size of a grid. The choice of the appropriate grid size n is based on regions for timed automata [4]. Regions are equivalence classes on sets of states in the semantics of a timed automaton. The equivalence relation $\cong$ that generates those classes is defined for states of an LTS $\mathcal{S}$ as: $(q, v) \cong (q', v') =_{def} q = q' \wedge v \cong v'$. Valuations v, v' are equivalent if for all clocks they agree on their integral parts and also on the ordering of fractional parts. As a result regions contain states that cannot be distinguished by any clock constraints [4]. The number of regions generated for all clocks in both TIOAs used as a specification and as an implementation is the grid size n [96]. Such a grid size n guarantees that if there is a trace that distinguishes those two TIOAs, then such a trace is going to be included in the grid automaton [96].

Because grid automata are finite and deterministic it is possible to adapt the W-method for FSMs presented in Section 2.1.1. A transition cover set P is defined for all transitions of the grid automaton, including transitions that represent actions and delays: $a \in \Sigma \cup \{2^{-n}\}$. A characterizing set W is based on sequences of output actions. So a test sequence $\sigma \in (\Sigma \cup 2^{-n})^*$ distinguishes states if its output sequences produced by a grid automaton are different. Additionally each TIOA must have a reset signal, which in case of timed systems is combined with a delay necessary for a TIOA to return to its initial state.

The above approach is complete [96] according to Definition 1. However the algorithm may generate a huge number of test cases, because grids may become very small. One way to overcome this problem is to increase the granularity of state space with zones, which are clock constraints [68], rather than integral values of clocks used in regions. This concept is used in the algorithm that generates test cases for Event Recording Automata (ERA), a class of timed automata that are deterministic [79]. In ERA clocks can only record time that elapsed after each action. The algorithm uses equivalence classes for states based on zones and the partitioning of clocks values. Another approach to reduce the number of test cases is to limit the specification. This was proposed for example for communicating timed automata with test views [20].

Timing constraints have also been investigated in the context of the **ioco** theory (Section 2.1.2). Definition 12 presents the **tioco** relation, which includes time delays [15].

Definition 12. 1. A Timed Input-Output Transition System (TIOTS) is an IOTS (Definition 6) $S = (Q, L_I, L_U, T, q_0)$ in which a transition relation is defined as: $T \subseteq (Q \times (L_I \cup L_U \cup \{\tau\} \cup D) \times Q)$, where D is $\{\varepsilon(d) | d \in \mathbb{R}^{\geq 0}\}$. $\varepsilon(d)$ represents an empty action followed by a delay, so simply a delay.

2. Let p be a state of a TIOTS, then $out_{\mathcal{M}}(p) = \{\mu(d) | \mu \in L_U \wedge p \overset{\mu(d)}{\Longrightarrow}\} \cup \{\delta | \delta(p)\}$. The set out is a set of of output actions that hapen before an arbitrarily chosen maximal time $\mathcal{M} \in \mathbb{R}^{\geq 0}$, followed by appropriate delays (it is assumed that two consecutive delays are added if possible).

3. Let s and i be TIOTS, then:

$$i\ \mathbf{tioco}_{\mathcal{M}}\ s \text{ iff } \forall \sigma \in Tstraces_{\mathcal{M}}(s) : out_{\mathcal{M}}(i\ after\ \sigma) \subseteq out_{\mathcal{M}}(s\ after\ \sigma)$$

where $Tstraces_{\mathcal{M}}(s)$ are suspension traces that contain time delays and are bounded by $\mathcal{M}$.

The test generation algorithm is similar to Algorithm 3, but delays are included after inputs, outputs and quiescence. The dense time inputs and outputs are difficult to observe so the above algorithm has been extended to generate discrete test cases [66], which are more practical. Finally there is also an extension of the $\mathbf{tioco}_{\mathcal{M}}$ relation that enables partitioning of inputs and outputs into classes [16], which is based on the relation **mioco** and extends its generation algorithm with delays.

2.1.4 Other specifications

The formalisms such as LTSs or FSMs are not well suited to specify complex systems, because they require the explicit representation of all states. Therefore in this section we present languages that provide a more succinct representation: Statecharts, UML state machines and Abstract State Machines.

Statecharts [44] were introduced as a modeling language that has several modeling features and at the same time a precise semantics [45]. Statecharts are based on EFSMs (Definition 4): they have variables that carry data and transitions that include triggers, guards and updates. The current values of all variables is called memory, inputs and outputs are called signals and updates are called actions. The following features characterize Statecharts [45]:

- signals may carry data values, triggers are boolean combinations of input signals and actions change the memory or produce output signals,
- states may be organized in a hierarchical way as OR and AND compositions. The first type means that a system is in only one of the state's substates and the second means that a system is in all of the state's substates,
- semantics is described using steps, in each step all possible and non-conflicting transitions are fired, based on the currently available set of input signals and the memory.

The test case generation algorithms based on Statecharts use the methods introduced for FSMs such as the W-method (Section 2.1.1). This idea was first proposed in the context of X-machines [61], which use functions as actions that can alter memory and produce an output. In the method for Statecharts [13] a specification and an implementation must meet the following requirements: all triggers and actions are correctly implemented, all transitions can be triggered and the combination of input signals and output uniquely identifies a transition (called output distinguishability). Based on such assumptions the W-method is adapted [13]:

- a set of transitions consists of all input signals an their combinations,
- instead of a transition cover set P a state cover set C is used, which contains sequences of transitions that visit all states and in case of hierarchical states includes also transitions (or their combinations) that visit the internal states,
- a characterization set W contains sequences of transitions to distinguish states at all levels of hierarchy.

The language μSZ combines Statecharts to describe dynamic behavior and the Z language to describe data [53] to enable more comprehensive data specification. During the test case generation process partitioning methods proposed for Z are used [52], which divide the input domain into subdomains. The test hypothesis must encompass the uniformity hypothesis, which states that an SUT behaves in the same way in all subdomains. Partitioning divides states in Statecharts into subdomains and original transitions into subtransitions, based on possible values of input data and memory. In this way an EFSM (Definition 4) is constructed and used to generate test cases [53].

In the object-oriented designs a variant of Statecharts is the basis of UML state machines, which express behavior of elements in UML models [1]. They differ from Statecharts in some semantic aspects, like priorities of enabled transitions or the order of executed actions. But the main challenge in using UML state machines for the test case generation purposes lies in their informal semantics [1].

The lack of formalization of UML state machines is overcome by specifying their semantics using a formalism like LTSs [69]. States in such an LTS are configurations, so sets of states and substates from an original UML state machine. Transitions in the LTSs are labeled with input/output pairs of signals based on actions in the state machine. The transition function relates configurations, so each triggering signal may enable and fire more than one transition in the original UML state machine. For such settings the testing equivalence is defined [69], as well as the **ioco** relation [40]. In the latter case the set *out* from Definition 8 contains input/output pairs. So it is defined as outputs generated after a given trace and for a given input. The test case generation algorithm is similar to Algorithm 3, however only two cases must be considered, since outputs are immediately available.

Abstract State Machines (ASM) are a formal language based on a system $A = (states, \mathbf{init}, F)$. In such a system *states* and the transition function F use the algebraic notions of structures and signatures. Definition 13 presents these aspects more formally [42].

Definition 13. 1. A structure $S = (U, \phi_1, ..., \phi_k)$ consists of a set U (called universe) and finitely many functions $\phi : U^n \rightarrow U$ with an arity n. Constants are functions with $n = 0$.

2. A signature $\Sigma = (f_1, ..., f_l, a_1, ..., a_l)$ gathers function symbols $(f_1, ..., f_l)$ and arities $(a_1, ..., a_l)$. A set of terms T_Σ is inductively defined as: each f_n with an arity zero is a term and if $t_1, ..., t_m \in T_\Sigma$ then $f(t_1, ..., t_m) \in T_\Sigma$ if f has an arity m.

3. A Σ-structure is a structure that can be described by a signature Σ, so the number of functions is the same $(l = k)$ and their arities agree. A set *states* is a set of Σ-structures, which all use U.

4. A transition function F is defined by a set of conditional assignments of the form: if α then r. A condition α is composed of equalities $s = u$, $s, u \in T_\Sigma$ and boolean connectives $\wedge, \neg$. An update r is of the form $f(t_1, ..., t_n) := t$ with $f \in \Sigma$ and $t_1, ..., t_n, t \in T_\Sigma$.

5. A specification in ASM consists of an initial Σ-structure and a set of conditional assignments. During execution of an ASM the terms in the conditional assignments are "evaluated" in the current state, i.e. the current Σ-structure.

A specification described with ASM can be transformed to an FSM and the test case generation algorithms from Section 2.1.1 may be applied [41]. The transformation of ASMs is based on execution of updates that are enabled (their conditions are evaluated to true) in each reachable state [41]. This means that the number of states might be infinite. To make the representation finite goals are used and only updates that are on the path to a goal are executed [41]. In this way an FSM is constructed, which represents a reachable portion of the ASM specification.

2.1.5 Summary

In this section we presented the model-based test case generation techniques, which assumed that the whole model of a specification is considered. This kind of conformance requires that both $SPEC$ and $IMPL$ are sets of models. Therefore several assumptions about them must be made, which constitute a test hypothesis. Table 1 presents the summary of used relations, test hypotheses and some properties of the test case generation algorithms.

The above summary shows that there are two main approaches to test case generation. The first one is based on finite and deterministic FSMs and the W-method and the second one on non-deterministic LTSs and the **ioco** theory. The problem of generating test cases for FSMs is well researched and there are many algorithms available [71]. The main idea behind most of them is similar to the W-method: to test whether all possible transitions generate required output and move the machine to the correct target state. In case of non-deterministic specifications it is much more difficult to give a clear definition of a **conforms** relation, which can also easily be tested. The **ioco** theory provides both: the definition of conformance and the algorithm to generate test cases used to check the conformance [103]. The main idea in this theory is to check whether output produced by an implementation after a given input is as specified. There are several further definitions of this relation and the adaptations of the original algorithm, which deal with classes of inputs [47], timing [15, 16] and symbolic analysis of data [31].

Besides the improvements of the original algorithms there are also proposed extensions to use them for more comprehensive modeling languages like timed automata, UML state machines or Statecharts. There are two ways of achieving this. The first way is to translate more complex specifications into the basic ones, for example methods based on EFSMs [70], timing analysis with grids [30, 96] or μSZ [53]. The second approach is to redefine the key theoretical ideas to deal with more complex structure, for example as in Statecharts [13] or UML state machines [40].

The complexity of algorithms summarized in Table 1 shows that dealing with deterministic specifications is less complex that in case of the non deterministic. This is because in all non-deterministic solutions one needs to consider a subset of all states, to which a single action may lead (or a transition combined with an internal action). This is the source of exponential growth in the complexity of those algorithms. Moreover, algorithms based on the **ioco** relation may produce infinite sequences, because these algorithms do not keep track of which states have been already explored. In modeling languages like UML state machines or Statecharts the additional complexity of the algorithms [13, 53] arise as the result of flattening hierarchical structures.

The methods presented in this section specify a set of requirements that an implementation must meet. These requirements restrict the behavior of an SUT, so that it becomes possible to infer confor-

Method and/or authors	**conforms** relation	Test hypotheses								Algorithm	
		Specification $SPEC$					Implementation $IMPL$			Complexity	Comments
		Model	Determ.	Data	Timed	Other	Model	Determ.	Other		
W-method Chow [22] and Fujiwara et al. [35]	trace equivalence	FSM	Yes	No	No	-minimal -strongly connected	FSM	Yes	-reliable reset	$\mathcal{O}(pn^3)$, where: p is number of transitions, n is number of states	-lengths of sequences differ for specific methods -test suite is complete
D-method Sidhu et al. [95]	trace equivalence	FSM	Yes	No	No	-has a distinguishing sequence	FSM	Yes	-reliable reset		
UIOv-method Chen et al. [21]	trace equivalence	FSM	Yes	No	No	-UIO sequences	FSM	Yes	-reliable reset		
UIO-method Aho et al. [2]	trace equivalence	FSM	Yes	No	No	-UIO sequences	FSM	Yes	-number of states as in specification		
Tretmans [101]	ioco	LTS with I/O	No	No	No	-finite behavior	IOLTS	No	-input enabled	if specification has finite traces: $\mathcal{O}(a2^n)$, where: a is number of actions, n is number of states	-test suite might be infinite, -each test suite is sound -test suites are complete if all possible test cases generated
Heerink et al. [47]	mioco	LTS with I/O	No	No	No	-finite behavior	IOLTS	No	-input enabled for I/O classess		
Frantzen et al. [31]	ioco$_{\mathcal{F}}$	IOSTS	No	Yes	No	-symbolic soecification	IOLTS	No	-uniformity of data values		
Bijl et al. [108]	ioco	LTS with I/O	No	No	No	-finite behavior	IOLTS	No	-given action refinement		
Springintveld et al. [96]	trace equivalence	TIOA	Yes	No	Yes	-separated outputs -input enabled	TIOA	Yes	-as specification	$\mathcal{O}(pr^3)$, where: r is number of combined regions	-based on W-method -test suite complete
Nielsen et al. [79]	trace equivalence	ERA	Yes	No	Yes	-event recording	ERA	Yes	-as specification	$\mathcal{O}(pz^3)$, where: z is number of zones	-based on W-method
Briones et al. [15] and Krichen et al. [66]	tioco$_{\mathcal{M}}$	Timed Automata with I/O	No	No	Yes	- finite behavior	TIOTS	No	-input enabled	if specififcation has finite traces: $\mathcal{O}(ac2^n)$, where: c is number of delays	-based on **ioco**
Briones et al. [16]	mtioco$_{\mathcal{M}}$	Timed Automata with I/O	No	No	Yes	- finite behavior	TIOTS	No	-input enabled for classes of I/O		
Bogdanov et al. [13]	trace equivalence	Statecharts	Yes	Yes	No	- output distinguish.	FSM	Yes	-correct triggers and actions	$\mathcal{O}(pn^3)$ where: p, n in statechart	-based on W-method
Hierons et al. [53]	trace equivalence	μSZ	Yes	Yes	No	-output distinguish.	FSM	Yes	-uniformity of subdomains	$\mathcal{O}(ps(ns)^3)$, where: s is number of subdomains	-based on W-method
Gnesi et al. [40]	ioco$_{\mathcal{F}}$	UML state machines	No	No	No	-only control behavior	IOLTS	Yes	-triggers correct	$\mathcal{O}(t2^n)$, where t is number of triggers	-based on **ioco**
Grieskamp et al. [41]	action equivalence	ASM	No	No	No	-finite with goals	FSM	Yes	-updates correct	goal dependant	-based on FSMs

Table 1: The summary of test hypotheses in the test case generation for conformance based on complete models.

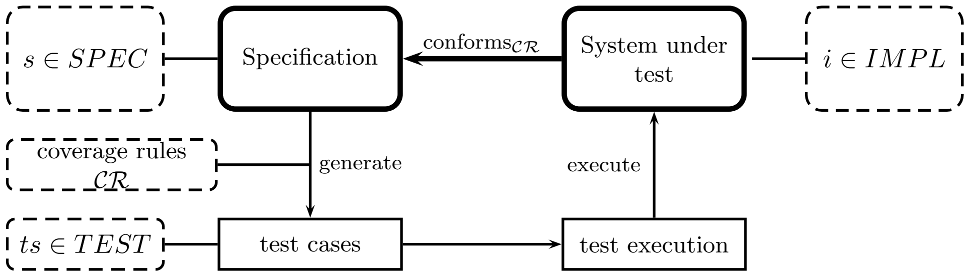


Figure 6: Model-based testing for coverage criteria.

mance after a test suite is generated and executed. The most obvious requirement is that implementations must be treated as models - for example as FSMs or IOLTSs. This leads to basic assumptions about behavior, for example that sending an input changes a state (LTSs) or changes a state and produces an output (FSMs). There are several other limitations like reliable resets, input enabling or precise timing information, which are specific for the test case generation algorithm.

The algorithms shown here are rather expensive in terms of time and other resources. One of the reasons of such complexity lies in the assumption that all behaviors must be considered. The ways to overcome this requirement are presented in the following sections.

2.2 Test case generation based on coverage

This section gives an overview of methods proposed to generate test cases based on coverage. These methods attempt to overcome the complexity of the test case generation by focusing on certain parts of a specification. This means that not all behaviors in a model are considered, but only those that satisfy specified structural coverage criteria. Therefore resulting test suites can only prove or disprove the existence of the **conforms** relation between a specification and an SUT with respect to the given criteria.

Figure 6 depicts the idea of the model-based testing based on coverage criteria. Similarly to techniques proposed for the conformance based on complete models, there are models of a specification, of an implementation and of test suites. Additionally the test case generation depends on the given coverage criteria or coverage rules. The problem considered by works presented in this section is specified as follows.

Problem 2. Given a specification $s \in SPEC$ and coverage rules $\mathcal{CR}$ generate a test suite $ts \in TEST$. The ts should be such that all coverage rules $\mathcal{CR}$ are satisfied. The execution of ts on an implementation $i \in IMPL$ gives a verdict $v \in \{fail, pass\}$, which determines whether $(s, i) \in \mathbf{conforms}_{\mathcal{CR}}$.

There are many types of coverage criteria, which have been introduced for source code and white-box testing [8, 39] and which are adapted to work with models. They can be divided into two main categories: control and data flow coverage criteria [39]. The first group contains criteria that describe control aspects, for example statement coverage. In model-based testing these criteria are defined usually in the context of graph-like specifications and they constitute node or transition coverage. Control flow coverage criteria are also used to analyze relations between branching conditions and boolean expressions they are built from [8]. The data flow coverage criteria are typically concerned with variables and their goal is to cover paths in models that connect definition and uses of specified variables.

Some of the methods presented in this section employ existing model checkers [23]. Model checkers can automatically verify whether a formula in temporal logics such as Linear Temporal Logic (LTL) [83] or Computation Tree Logic (CTL) [24] holds in the given specification of a system. If the formula does not hold then an appropriate counterexample is returned, which is a path to a state in which a violation has been found. In the test case generation methods counterexamples are assumed to be the test cases. So the goal of these methods is to find a formula that expresses a violation of a desired property, run a model checker and then use counterexamples as test cases [34]. Methods presented in this section that use model checking are mostly concerned with generating appropriate formulas in

Algorithm 4 Algorithm to generate test suite for transition and state coverage criteria [51]

```
PASS$\leftarrow \emptyset$; WAIT$\leftarrow \{(s_0, C_0, \varepsilon)\}$; SUITE$\leftarrow \emptyset$; COV$\leftarrow C_0$
while WAIT$\neq \emptyset$ do
  select $(s, C, \omega)$ from WAIT
  PASS $\leftarrow$ PASS $\cup\{(s, C, \omega)\}$
  for all $(s', C', \omega.t)$ such that $(s, C, \omega) \overset{t}{\Rightarrow} (s', C', \omega.t)$ do
    if $C' \nsubseteq$ COV then
      SUITE$\leftarrow$SUITE $\cup$ $\{(\omega.t, C')\}$
      COV$\leftarrow$COV $\cup C'$
    end if
    if $\neg\exists(s'', C'', \omega'') : (s'', C'', \omega'') \in$(PASS $\cup$ WAIT)$\wedge(s'' = s')$ then
      WAIT$\leftarrow$WAIT $\cup\{(s', C', \omega.t)\}$
    end if
  end for
end while
return SUITE
```

order to satisfy coverage rules. How exactly counterexamples are then translated to sequences of inputs and outputs is not further presented and is assumed to be specific to the implementation. Nevertheless such a translation is not always trivial [112] .

In the following we present algorithms for generating test cases divided into groups based on the type of criteria. The models assumed in the presented works use more comprehensive modeling languages (EFSMs, Statecharts, ASMs), which can represent complex behaviors in a more succinct way.

2.2.1 Control flow coverage criteria

Control flow coverage criteria are used if generated test cases should cover parts of the possible control flow of a specification $s \in SPEC$. Therefore a test suite that satisfies the criteria must contain required elements of the given model or must satisfy conditions that will lead to certain parts of the model. Since modeling languages differ, the exact criteria are described in more detail below.

The most basic control flow coverage criteria require that test suites contain all states or transitions of the given specification. For EFMSs (Definition 4) the version of the breadth-first search reachability algorithm is presented in Algorithm 4 [51]. The algorithm identifies: a set COV of globally covered elements, a set SUITE of paths and covered elements, and two sets WAIT and PASS of triples (s, C, ω), where s is a state in an EFSM, C is a set of covered elements in a path and ω is this path. A triple from the set WAIT is explored for each enabled transition t in the EFSM. The set C is updated into C' to contain the newly discovered transition or state, depending on which coverage criterion is used. If this new coverage set C' adds an element to COV then a path $\omega.t$ is added to the set SUITE (along with covered elements C'). Finally if a new state of the EFSM has not been yet explored then the triple is stored for further exploration in the set WAIT.

For Statecharts and UML state machines, besides the transition coverage criterion based on breadth first search [80], the transition pairs coverage criterion has been considered [93, 11]. This criterion aims to cover all possible combinations of incoming and outgoing transitions for each state. The proposed algorithms assume that a Statechart or an UML state machine is flat, so it has that a graph-like structure. From such a structure a transition graph is constructed [93, 11]. A transition graph is a directed graph with nodes representing original transitions and arcs connecting those nodes which are incoming and outgoing transitions for some state. The transition graph is then used to construct transition pairs by traversing it and gathering all nodes. This idea can be extended to consider not only pairs of transitions but also sequences up to a certain constant value [11].

Model checking techniques are another approach to generate test cases that satisfy certain coverage criteria. The underlying idea is to prepare a set of test predicates, which are translated to formulas called trap properties [36]. For a test predicate tp, a trap property is a CTL formula `AG!`tp, which means that tp is never satisfied in the given specification. A model checker returns a counterexample if it finds a state in which a formula does not hold, hence tp is satisfied. A counterexample is a path leading to such a state, so it can be interpreted as a test case that satisfies a test predicate tp.

The above idea was initially proposed for specifications modeled with Software Cost Reduction (SCR) requirements method [36]. SCR represents requirements with a set of monitored and controlled variables, which are inputs and outputs, respectively. States are given by valuations of those variables and transitions are described with two types of tables. Event tables are used to represent transitions triggered by events and condition tables gather conditions necessary to change values of controlled variables. These tables are the basis to generate test predicates, which are combinations of possible events and conditions to change a state. The set of test predicates defined in this way satisfy a branch coverage criterion [36]. The translation of the SCR specification along with encoded trap properties are run in SPIN [54] and SMV [23] model checkers. If counterexamples are generated then they are test cases, if not then the particular test predicate is either not reachable or the state space is too big for a model checker to exhaustively explore it.

The similar approach is also taken for ASM-based specifications [37, 38], however for different coverage criteria. According to Definition 13, an ASM specification consists of a set of guarded updates (rules) of the form: R = `if` *Cond* `then` *Updates*. The following coverage criteria are defined (there are m rules R_i for $i = 1, ..., m$) [37, 38]:

- rule coverage: requires that all rules are fired, so a set of test predicates contains $\{Cond_i, \neg Cond_i\}$ for $i = 1, ..., m$,
- rule update coverage: requires that all updates are executed, so a set of test predicates is $\{Cond_i \wedge f_{i,j}(\overline{t_{i,j}}) = t_{i,j}\}$ for each rule R_i and each update in this rule $f_{i,j}(\overline{t_{i,j}})$ ($\overline{t_{i,j}}$ is a vector of terms),
- parallel rule coverage: requires that each possible set $\{R_i\}$ of n rules fire simultaneously, so a set of test predicates contains conjunctions of guards in each rule in a set,
- modified condition decision coverage (MCDC): requires that there is a test predicate in which each atomic condition c_k (subcondition without connectives) of $Cond_i$ evaluates to true and to false. The valuations of all other atomic conditions are chosen in such a way that the outcome of $Cond_i$ is true in one case and false in the other.

An ASM specification is translated to input language of SPIN [38] or SMV [37]. Trap properties generated from the test predicates specified for the above coverage criteria are formulas verified by the model checkers.

2.2.2 Data flow coverage criteria

Data flow coverage criteria are used to ensure that generated test cases cover states that follow data flow between variables [88]. The main idea is to differentiate between definition and use occurrences of variables and then cover paths between them. A definition occurrence of a variable x is an assignment to this variable and a use occurrence is a statement that uses a value of the variable. The latter type of occurrence can be divided further into a computational or a predicate use occurrence. The most common data flow coverage criterion is the definition-use, which requires that a test suite contains at least one path for each definition and use pair and that this path is free of further definition occurrences. In this way it is possible to trace dependencies between variables and their values.

In order to generate test cases that satisfy the definition-use criterion Algorithm 4 can be adjusted [51] to enable partial coverage. The criterion C is extended to be a pair (F, U), where F is not empty if a definition of a variable x has been encountered. Then if its use is found a value from F and the current state is added to U.

Test case generation for EFSMs (Definition 4) can also be based on flowgraphs as shown in Definition 14 [106].

Definition 14. A flowgraph G for an EFSM $M = (I, O, S, \overrightarrow{x}, T)$ is a directed graph $G = (V, v_s, v_f, E)$, where

- V is a set of nodes, which are marked as s-node, i-node or t-node,
- $v_s, v_f \in V$ are an initial node and a final node, respectively,
- E is a set of edges, which are marked as an si-edge, it-edge or ts-edge.

The mapping between an EFSM S and a flowgraph G is such that, for each $s \in S$, G has:

- one s-node,
- one i-node for each input action possible from S, so for each $i \in I_s$, where $I_s = \{i \mid i \in I \wedge \exists s' : (s, s', i, o, P, A) \in T$ for some $o, P, A\}$,
- one t-node for each transition from s, so for each $t \in T_s$, where $T_s = \{t \in T | \exists s' : (s, s', i, o, P, A) \in T$ for some $i, o, P, A\}$,
- one si-edge from s-node to each i-node, $i \in I_s$,
- one it-edge from each i-node to t-node, $i \in I_s$ and $t \in T_s$ and i is an input action of t,
- one ts-edge from each t-node to s'-node, where s' is a target for the transition t.

For all variables x, t-nodes are identified as the definition occurrence (def(x)) if they assign to x or the computational use occurrence (c-use(x)) if they use x for output or assignment. The it-edges are marked as the predicate use occurrence (p-use(x)), if a guard for a transition t uses a variable x. The flowgraph is then used to identify associations between definition and use of variables:

- def-c-use association for a variable x is a triple (x, i, j) if i is a def(x) node, j is a c-use(x) node and there exists a path $(i, n_1, ..., n_m, j)$ for $m \geq 0$ such that nodes $n_1, ..., n_m$ are not marked def(x),
- def-p-use association for a variable x is a triple $(x, i, (j, k))$ if i is a def(x) node, (j, k) is a p-use(x) edge and there exists a path $(i, n_1, ..., n_m, j, k)$ for $m \geq 0$ such that nodes $n_1, ..., n_m, j$ are not marked def(x).

The definition of flowcharts and associations for variables are proposed for specifications in Estelle [106] and Statcharts [56]. In the first case additionally more complex relations between input and output variables (as a special type of definition and use occurrence) are defined. Statecharts are represented as EFSMs and their hierarchy is flat, so states of an appropriate EFSM are configurations (sets of all states that a Statechart is in). Transitions can be triggered with multiple signals. The generated paths in a flowchart are then translated into sequences of signals, which are test cases.

The above data flow criteria can be also expressed as a model checking problem [57, 55]. A flowchart can be seen as a Kripke structure as in Definition 15.

Definition 15. A Kripke structure is a tuple $K = (V, v_s, L, E)$, where V is a set of states, v_s is an initial state, L is a labeling function and E is a set of edges. The labeling function L assigns to each state a subset of atomic propositions AP, so $L : V \rightarrow 2^{AP}$.

A Kripke structure that represent a flowchart has the function L that labels v_s and v_f with $\{start\}$ and $\{final\}$ and other states with def(x), c-use(x) or p-use(x) [55]. A specified definition-use association can be expressed as a CTL formula, and then a Kripke structure is used to find witnesses of such a formula [25]. A witness is a path leading to state in the Kripke structure that satisfies a given CTL formula and is directly used as a test case. This approach has been also extended to include control flow dependencies for EFSMs [57, 58].

A different view of the data flow coverage criteria is used in combinatorial testing, which aims at covering combinations of possible values of input variables [67, 19]. The rationale is to check how such combinations may influence an implementation. Because for n variables that range over m values each, there are m^n combinations, in practice only subsets of variables are used, called k-way combinations. The factor k is usually no more than 6. All combinations are translated into trap properties and appropriate counterexamples are used as test cases.

2.2.3 Summary

In this section we presented techniques to generate test cases, which are restricted to structural coverage criteria. These criteria are defined using rules, so no specific language is required to express them. All presented approaches are summarized in Table 2.

Authors	Model $SPEC$	Type of coverage criteria		Method	Comments
		Control flow	Data flow		
Offutt et al. [80]	UML state machine	-transitions, -transition pairs		breadth first search	-requires flat structure of state machines
Belli et al. [11]	Statecharts	-transition pairs, -k-transitions, -faulty transitions		transition graphs	-optimization with number of transitions
Santiago et al. [93]	Statecharts	-transition pairs		transition graphs	-requires flat model
Hessel et al. [50]	EFSM	-control states, -transitions	-definition use	reachability algorithm	-EFSM represents timed automata with zones, -optimization with delays
Gargantini [36]	SCR	-branch coverage		model checking	-use SPIN and SMV model checkers
Gargantini et al. [37, 38]	ASM	-rules, -rule updates, -parallel rule, -MCDC		model checking	-use SMV and SPIN model checkers
Ural et al. [106]	Estelle		-definition use, -input output relations	exploration of flowcharts	-assumes normal form of specification (based on EFSM)
Hong et al. [56]	Statecharts		-definition use	exploration of flowcharts	-requires flat model
Hong et al. [57, 55, 58]	EFSM		-definition use -chains of defintion use	model checking	-adapted model checking algorithm
Khun et al. [67], Calvagna et al. [19]			-combinatorial testing	model checking	-use SMV model checker

Table 2: Properties of methods for test case generation based on coverage.

The complexity of presented techniques depends on the underlying methods, so it is not directly given in Table 2. In case of reachability- or graph-based algorithms the complexity may be even exponential if Statecharts or UML state machines must be flattened, because all possible subsets of the original specification are used. For methods based on model checking the complexity depends on the model checker, the size of the specification (in terms of processes and used variables) and the number and size of formulas to check. This in case of larger systems may reduce the practical value of the presented works.

In this section we presented approaches that do not require any special representation of coverage criteria. In the next section approaches that require explicit specification of properties are presented. In some cases this includes also coverage criteria as a special case of a property.

2.3 Test case generation based on properties

Generation of test cases based on properties follows a similar pragmatic approach as in the coverage-based testing. The assumption is therefore to construct test cases that can check whether an SUT behaves in the same way as the given specification for some part of it. The idea of model-based testing for conformance based on properties is depicted in Figure 7.

There are two components of a specification. The first one specifies required behaviors as in the previous sections, the second describes the selected properties $p \in PROP$. The sets $SPEC$ and $PROP$ are in some cases disjoint and in some cases they overlap depending on the method used. The problem considered in this section can be described as follows.

Problem 3. Given a specification $s \in SPEC$ and properties $p \in PROP$ generate a test suite $ts \in TEST$. The test suite ts should be such that its execution on an implementation $i \in IMPL$ respects the properties p. The execution of ts gives a verdict $v \in \{fail, pass\}$, which determines whether $(s, i) \in \mathbf{conforms}_p$.

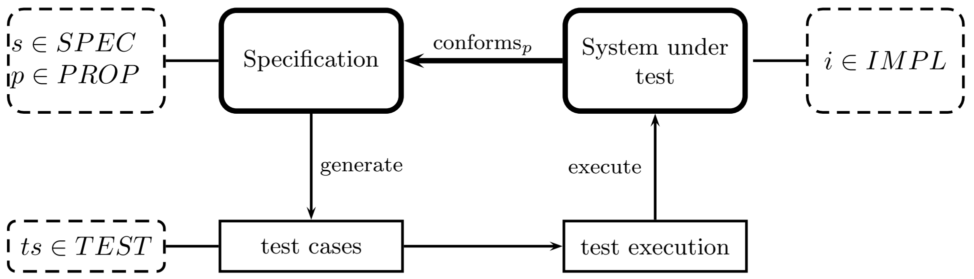


Figure 7: Model-based testing for selected properties.

Properties and how test cases can satisfy them is understood in several different ways. The first approach is to use properties to restrict a specification and to test only parts of it given by properties (i.e., $PROP$ and $SPEC$ coincide). Another approach is to use the properties as a restriction of an environment, in which a reactive system is going to operate. In this way it can be compared whether a specification and an implementation behaves in the same way under the given conditions. Finally formulas in temporal logics (like LTL or CTL) can be properties. The formulas may restrict specifications and environment at the same time. In this case $SPEC$ and $PROP$ are typically disjoint.

2.3.1 Restricting a specification

Properties for which the conformance is checked can represent a certain part of a specification. In this way the specification is restricted and test cases are generated only within such a restriction. There are several ways how to express the properties and how to combine it with the original specification.

For instance, in the context of LTSs and the **ioco** relation (Section 2.1.2), the TGV tool proposes test purposes to restrict a specification [62]. A test purpose is an LTS with inputs and outputs (Definition 5) as in Definition 16.

Definition 16. A test purpose is an LTS $TP = (Q^{TP}, A^{TP}, T^{TP}, q_0^{TP})$, where actions A^{TP} are partitioned into L_I^{TP} and L_U^{TP} and which has the following characteristics:

- it is deterministic, i.e. $\forall q, s, s' \in Q^{TP}\ , \forall l \in A^{TP} : (q \xrightarrow{l} s \wedge q \xrightarrow{l} s') \Rightarrow (s = s')$
- it has trap states $Accept^{TP}$ and $Refuse^{TP}$ with self-loops on each action $l \in A^{TP}$, i.e. $\forall q \in (Accept^{TP} \cup Refuse^{TP})\ , \forall l \in A^{TP} : q \xrightarrow{l} q$,
- it is complete, i.e. in all states all actions are enabled $\forall q \in Q^{TP}\ .\forall l \in A^{TP} q \xrightarrow{l}$.

A specification is an LTS with inputs and outputs $S = (Q^S, A^S, T^S, q_0^S)$ and a test purpose must have the same actions, so $L_I^{TP} = L_I^S$ and $L_U^{TP} = L_U^S$.

The steps to build test cases are outlined below [62]:

1. Construct the synchronous product of S and TP: $SP = S||TP$. SP is an LTS with inputs and outputs such that $SP = (Q^S \times Q^{TP}, A^S, T^{SP}, (q_0^S, q_0^{TP}))$. The transition relation is defined by :

$$(p, q) \xrightarrow{a}_{SP} (p', q') \Leftrightarrow p \xrightarrow{a}_{S} p' \wedge q \xrightarrow{a}_{TP} q'$$

   Accepting states are $Accept_{SP} = Q^{SP} \cap (Q^S \times Accept^{TP})$ and refusing are $Refuse_{SP} = Q^{SP} \cap (Q^S \times Refuse^{TP})$

2. Add quiescent transitions (self transitions in states that do not enable any output actions), determinize and remove internal actions. These steps result in an LTS with the visible behavior $SP_{VIS} = (Q^{VIS}, A^S, T^{VIS}, q_0^{VIS})$.
3. Construct a complete test graph $CTG = (Q^{CTG}, A^S, T^{CTG}, q_0^{CTG})$. Its inputs and outputs are switched with respect to the specification S. States in Q^{CTG} are following:

- states from which accepting states can be reached: $coreachable(Accept^{VIS}) = \{q \in Q^{VIS} \mid \exists \sigma \in (A^S)^* : q \xrightarrow{\sigma} Accept^{VIS}\}$,
- inconclusive states ($Inconc$), which are not in $coreachable(Accept^{VIS})$, but are direct successors of those states in SP^{VIS} by an output $a \in L_U^{VIS}$,
- a new failing state ($Fail$),
- passing states ($Pass$), which are $Accept^{VIS}$ states.

The transition relation T^{CTG} consists of:

- transitions $t \in T^{VIS}$, which are between states in a set $coreachable(Accept^{VIS}) \cup Inconc$,
- transitions $\{(q, a, Fail) \mid q \in coreachable(Accept^{VIS}) \wedge a \in L_I^{CTG} \wedge \neg(q \xrightarrow{a}_{VIS})\}$

This procedure constructs an LTS that represents all possible test cases with respect to the given test purpose TP and can be used to test an implementation. There are three possible verdicts. Either an implementation reaches a desired state of TP ($Pass$ states), or it produces an output that is not part of a test purpose (but is conformant with the specification) and a desired state cannot be reached ($Inconc$ states), or it produces an output which is not conformant ($Fail$ state) [62].

The TGV method is extended to deal with symbolic transition systems (ioSTS) [91, 64]. These systems include variables to represent a state of a system or parameters of inputs. Variables guard transitions and their values update other variables [63, 92]. In such systems reachability of certain states depends not only on actions, but also on received input values. Therefore, searching for states, from which accepting ones can be reached, must also take into account values of input variables and their assignments. Because such analysis cannot be exact [64], coreachable states are overapproximated. This means that test cases may still not produce any verdict, although the accepting state of the TP is not reachable anymore. The above approach is further extended to enable verification of safety and liveness properties [27]. Even if such properties cannot be formally verified, because the state space is too big, it may is still possible to generate test cases that can check whether an implementation satisfies them.

The TGV method is used in the AGEDIS tool, in which models are based on UML state machines [46]. In this tool definitions of test purposes are called testing scenarios and are coverage oriented. The coverage can be defined in terms of functional as well as structural criteria. It is also possible to manually define required test cases by defining an explicit testing scenario.

To restrict specifications modeled with communicating timed automata in UPPAAL the notion of observers has been proposed [49]. Observers express coverage criteria. The underlying idea is to make them to "accept" traces that lead to states in which coverage is satisfied. Formally, observers are timed automata (Definition 9) with a set of accepting states. Their transitions are synchronized with the transitions of a UPPAAL timed automaton, so both are executed in parallel [49].

An approach based on Constraint Logic Programming (CLP) has been introduced to generate test cases from specifications in the modeling language of AutoFocus tool [87]. AutoFocus models are hierarchies of components connected with typed ports. Behavior of a component is described with a state machine similar to an EFSM (Definition 4). The translation from AutoFocus models to CLP iterates through all transitions in all state machines. A transition with its guards and variable assignments, defines a predicate [73]. Predicates are used also to specify constraints imposed by test purposes (expressed with constraint handling rules - CRH). An execution of a CLP program, which is a translated specification and a test purpose, may lead to a solution. Several search strategies are proposed to find the most useful solutions that serve as test cases [85, 86]. These strategies are variations of the depth first search, but they use heuristics to reduce the length of test cases. Because the language used to specify test purposes is based on predicate logic it is expressive enough to allow the specification of complex purposes: functional, coverage-based or stochastic [81].

2.3.2 Model checking and mutation analysis

Model checking is one of the underlying techniques used to generate test cases based on properties. As presented in the context of the coverage-based test case generation (Section 2.2), the idea is to use a counterexample or a witness as a test case.

One of the ways how counterexamples may be produced is based on mutation analysis [6], that is on small syntactical changes applied to some parts of a model. For a given specification S and a property p, which holds in the specification (otherwise the model is incorrect and should not be used to generate test cases) we can mutate either of them. There are two possibilities: a property p is checked against a mutated specification S' or a mutated property p' is checked against the original model S. The interpretation of produced counterexamples differs in both of those cases. In the first one (mutated specification) a negative test case is obtained, to pass it an implementation must produce a failure, in the second case (mutated properties) a positive test case is produced and an implementation must not return a failure [32].

The above idea was used to generate test cases based on a Software Cost Reduction (SCR) requirement specification [6]. An SCR specification describes controllable and observable variables in a system and how their values change. Such a specification is the basis to build a model used by SMV model checker along with a set of formal requirements [6]. After applying mutation operators on a model and on properties test cases are obtained. Applying different mutation operators may lead to different levels of coverage, for example the highest coverage is achieved when replacing variables with the other ones from the specification [12]

Another way of using mutations of specifications is to combine a mutated specification S' and an original one S [7, 14]. S and S' are Kripke structures (Definition 15), in which atomic propositions (AP) are partitioned into input and output. A set of properties used by the model checker is based on comparisons of outputs produced by S and S'. A counterexample that leads to a state in which outputs differ is a test case. To deal with non determinism, possible input sequences are also encoded as Kripke structures and they restrict the combination of mutant and specification further [14].

Mutations of properties enables detection how well a property has been tested [99]. Because there is possibly an infinite number of paths in a specification S that lead to states that satisfy a property p, it is not possible to test all of them. To increase the confidence that at least the most important ones were selected and tested property-based metrics and coverage are used. For a given property p a test suite covers it, if for every subformula ϕ a mutation replacing ϕ with ψ ($p[\phi \leftarrow \psi]$), which is not satisfied in S, is represented with a test case. The selection of ψ can be based on the notion of polarity of a subformula, so ψ is set to true if ϕ is enclosed in odd number of negations and ψ is set to false otherwise [99]. Another approach to deal with a possibly infinite number of paths in a specification is to partition them based on satisfaction of a set of properties. Then a path from each partition is selected as a representative test case [18].

2.3.3 Constraining the environment of a system

This section shows how properties are used to constrain the environment, in which a reactive system operates. To achieve this testing scenarios are specified and they that should be covered by test cases.

The restriction of possible inputs data is extensively studied in the context of the Lustre language. Lustre is a synchronous data flow programming language [43] designed for specifying reactive systems. A synchronous language means that changes in states of a system are propagated at ticks of some virtual clock. A data flow language means that a program is based on the description of sequences of typed data and relations between such sequences. For example a specification $Z = X + Y$ is interpreted as a flow Z equals to $(x_1 + y_1, x_2 + y_2, ...)$ for X equals to $(x_1, x_2, ...)$ and Y equals to $(y_1, y_2, ...)$. Besides usual arithmetic and boolean operators, there are two specific ones: previous `pre(`X`)` is defined as $(nil, x_1, x_2, ...)$ and followed by X `->` Y is defined as $(x_1, y_2, y_3, ...)$. Building blocks in the Lustre program are called nodes and they define input flows and the resulting flow [43].

There are several tools based on the Lustre language that generate test sequences according to required constraints imposed on the environment:

- in order to generate test data Lurette [89] requires an environment and an oracle modeled as Lustre nodes. An environment is described as a set of constraints on data flows, and an oracle is a set of assertions between input and output flows. The generated input values are sent to an SUT, which responds with output values and the next inputs are generated. The test data generation is combined with checking the correctness of output values in the oracle. To generate relevant test sequences, so the ones that respect the set of constraints specified in an environment node, the constraints are abstracted and internally represented as binary decision diagrams (BDDs). By traversing such diagrams and solving constraints, required inputs are produced.

- Lutess [94] uses a very similar approach as Lurette, i.e., it requires constraints on the environment and an oracle node. However, the internal implementation is different and is based on Constraint Logic Programming. A set of constraints is translated to a program and then solved. Lutess allows also assigning conditional probabilities to input values. So if conditions are satisfied a probability of generating an input with a required value should be as specified.

- GATeL [75] requires that test objectives are given in the specification. So a Lustre program is enhanced with assertions, which state conditions that flows must meet, and with requirements that must be reached in the current test sequence. The test sequence generation is based on finding sequences that satisfy the assertions. But, unlike Lurrete and Lutess, the process starts from the last state in which test objectives hold. Then constraints are simplified and translated to a CLP program.

In the object oriented domain SpecExlorer has been developed to allow for the test case generation for reactive systems [109]. The tool uses the languages Spec# or ASML#, which both have their formal semantics defined with ASMs (Definition 13). Conditional assignments, specific for ASMs, are represented as methods with preconditions, called actions, declared in classes. Actions that represent outputs in a system are controllable and the ones for inputs are observable. A program written in Spec# or ASML# is a model program and a result of its execution is a model automaton. SpecExplorer has the following methods that control how the model automaton is constructed [109]:

- restricting values of parameters used in actions,

- strengthening preconditions of actions,

- filtering states based on attributes of a class or on auxiliary variables,

- directed search by assigning probabilities to certain values of parameters and by bounding state exploration,

- grouping states with an equivalence relation into classes, in which an arbitrary number of states is further explored.

The test case generation is realized by traversing a model automaton. The result of such a traversal is a test automaton, which extends the model automaton with additional features like timeouts or test variables. SpecExplorer implements the traversal algorithms as games, in which a test tool tries to win by going into predefined accepting states [78].

2.3.4 Summary

In this section we presented methods for test case generation based on properties. As in the coverage-based conformance, the proposed methods follow rather pragmatic approach. In such an approach the number of generated test cases is reduced by considering only part of a specification. The presented works are summarized in Table 3.

The presented methods for generating test cases typically rely on some other methods. These methods include a parallel composition [62, 64, 63, 91, 92], different search strategies [109], constraint solving [86, 87, 89, 94, 75] and model checking [6, 32, 7, 14, 99, 18]. Such diversity of underlying techniques and algorithms is the result of the increasing complexity of specifications and the increasing number of modeling languages. Therefore test case generation methods must become also diversified.

There is also a lot of variation in the way properties are expressed. In some cases, they are given using languages that are the same or very similar to the specification language. In other cases a completely different languages is used. The properties differ also in what they actually describe: test purposes or goals, test specifications or test scenarios. Testing purposes or goals specify which paths of a specification to consider during generation. In contrast, testing specifications are used to provide a general state that should be reached by a specification. Testing scenarios restrict the environment,

Authors and/or tool	*SPEC*		*PROP*	Method	Comments
	Model	Determ.	Model		
TGV - Jard et al. [62]	LTS with I/O	No	LTS with Accept and Refuse	parallel composition with coreachability	-coreachability based on strongly connected components
Jeron [64], Jeannet et al. [63], Rusu et al. [91, 92] Constant et al. [27]	ioSTS (symbolic LTS)	No	ioSTS with Accept and/or Refuse	parallel composition with coreachability	-overapproximation of coreachable states
AGEDIS - Hartman et al. [46]	UML state machines	No	LTS with Accept and Refuse	based on TGV	-testing scenarios, -properties for coverage criteria
UPPAAL Cover - Hessel et al. [49]	UPPAAL timed automata	No	observers	model checking	-properties for coverage criteria
AutoFocus - Pretschner et al. [85, 86, 87] Phillipps et al. [81]	AutoFocus	Yes	Constraint Handling Rules (CHR)	translation into CLP	-several search strategies based on heuristics
Amman et al. [6]	SCR	Yes	CTL/LTL	model checking	-mutation of specification and properties -SMV model checker
Fraser et al. [32]	Kripke structure with I/O	Yes	CTL/LTL	model checking	-property relevance -positive and negative test cases
Amman et al. [7]	input lang. for SMV	Yes	CTL/LTL	model checking	-using SMV model checker -analysis of dangerous traces
Boroday et al. [14]	Kripke structure with I/O	No	CTL/LTL	module checking	-support for non-determinism
Tan et al. [99]	Kripke structure	Yes	LTL	model checking	-properties based coverage
Callahan et al. [18]	Promela	Yes	LTL	model checking	-partitioning based on properties
Lurette - Raymond et al. [89]	Lustre	Yes	Lustre	constraint solving and BDDs	-generates test data -online checking with oracle
Lutess - Seljimi et al. [94]	Lustre	Yes	Lustre	translation into CLP	-generates test data -online checking with oracle -stochastic specification
GaTEL - Marre et al. [75]	Lustre	Yes	Lustre	translation into CLP	-generates test data based on goal state
SpecExplorer - Veanes et al. [109]	Spec#, ASML#	No	exploration rules	searching graphs, games	-restricts possible actions -stochastic specification

Table 3: Properties of methods for test case generation based on properties.

and test cases are used to check whether behaviors of a given specification and of an SUT are the same for some subset of all possible input data.

As for the coverage-based test case generation, the complexity of the algorithms is not given. The complexity depends on the underlying method such as constraint logic programming or model checking. This also means that the scalability of the presented techniques is not directly specified. The approaches based on model checking or other search strategies may not be scalable, but it also depends on the complexity of properties used.

2.4 Conclusions

This part of the paper reviews the most common model-based test case generation methods. They are divided into three categories according to the definition of the conformance relation between a model of a specification and of an SUT. The three categories are test case generation based on complete models, on coverage criteria and on properties.

Methods proposed for conformance testing based on complete models are influenced by two main approaches: one based on FSMs and one based on non-deterministic LTSs. In the first case, since the conformance relation is straightforward, there are a lot of approaches that propose improvements to the most popular W-method [22]. The popularity of this method is due to its simplicity, but also to its non restrictive assumptions. In the second approach that is based on non-deterministic LTSs the emphasis is put on defining the relation **conforms** (Figure 3) precisely and to use a straightforward algorithm. The definition in this case is important, because it influences how SUTs are tested. The most popular conformance relation is the **ioco** relation [103], which is also easily tested. Other approaches in the conformance based on complete models are usually variations of these two methods.

The test case generation based on coverage or properties differs from the above mainly because the notion of **conforms** relation is not of the most importance. What is important in those approaches are algorithms that efficiently can produce test cases, which additionally satisfy given constraints based on coverage criteria or properties. In these approaches the conformance is usually assumed to be the trace equivalence, with an exception of TGV-based methods, which use the **ioco** relation [27, 62, 63, 64, 91, 92, **?**].

The presented review also showed that there are many formalisms and modeling languages used to represent specifications of reactive systems. The simple formalisms such as FSMs or LTSs are quite common and they allow for model-based testing to be placed on formal foundations. However in practice more comprehensive modeling languages such as Statecharts, UML state machines, ASMs (Spec#) are more popular. The methods for the test case generation proposed for these languages are usually specific for them, but they reuse existing algorithms such as the W-method or the **ioco** theory or other analysis techniques, for instance model checking or CLP.

The additional sources of complexity in the test case generation methods are timing requirements for real-time models (like timed automata [5]). This complexity arises from the often infinite ways how such systems may behave and from intrinsic non-determinism. To overcome these challenges authors of the presented methods assume several restrictions to ensure deterministic executions and to represent the state space in a discrete way. Discrete models are typically achieved using regions [96] or zones [79]. The latter approach results in coarser state spaces, but at the same time the uniform behavior within zones must be assumed to hold in an SUT.

One of the common techniques used to generate test cases based on properties or coverage is model checking. The most important advantage of using model checkers is that they are easily available and they offer optimizations such as symbolic model checking and partial order reduction [23]. But model checkers are not intended to generate test cases, so they are also limited in how they can be used for this purpose [33, 34]. The problems arise, because model checkers are usually satisfied with one counterexample or witness, which might not be the shortest one. Consequently generated test cases might be longer and therefore not optimal. The other problem lies in the exhaustiveness of generated test cases: model checkers usually cannot find all possible ways to violate a given formula, since one path is enough. Finally counterexamples or witnesses are not directly executable, so they must be adjusted to the required format accepted by an implementation.

3 Comparative analysis of tools

This part of the paper analyzes and compares several tools, which implement model-based test case generation. The goal of this analysis is to show how the test case generation is performed in practice and what the requirements of the specific methods are. Due to the high diversity of tools, they are compared using some high-level criteria.

The tools compared in this section are:

- TorX [104]: the implementation of the conformance checking based on the **ioco** theory (Section 2.1.2),

- TGV [60]: also based on the **ioco** relation, but with test purposes (Section 2.3.1),

- UPPAAL Cover [105]: used for test case generation from timed automata in UPPAAL with observers (Section 2.3.1),

- SpecExplorer [76]: introduces the testing scenarios for specifications in Spec#(Section 2.3.3).

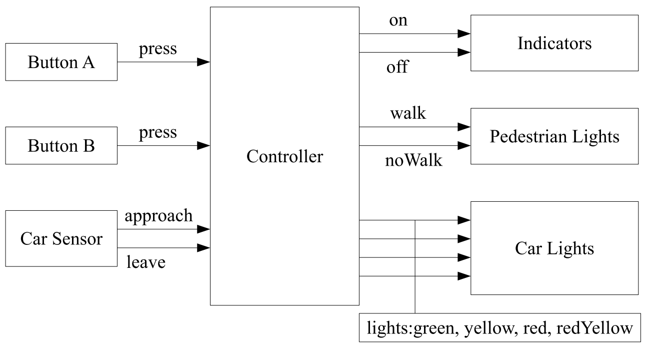


Figure 8: The structure of the traffic lights system.

The main reason these tools were selected is that they represent different types of the model-based testing: conformance testing based on complete models (TorX), testing based on properties (TGV, SpecExplorer, UPPAAL Cover) and support for coverage criteria (UPPAAL Cover). The tools differs also in formalisms and the modeling languages they rely on: labeled transition systems and LOTOS (TorX, TGV), communicating timed automata (UPPAAL Cover) and ASM based Spec# (SpecExplorer). With the selected tools it is possible to compare specific requirements of the various methods presented in the review part of this paper.

The above selection is by no means exhaustive, since there are other model-based test case generation tools developed in academia as well as commercial ones. Unfortunately some of the tools such as AutoFocus [87, 86] or AGEDIS [46] are no longer maintained and could not be used here. The commercial tools like Confirmiq Qtronic [26] or IBM Rational Rhapsody ATG [59] do not provide details of the semantics used in models and of their test case generation algorithms, so it is hard to use them in this comparison. Finally tools for generating data sets (like Lutess [94], Lurette [89], GATeL [75]) or for combinatorial testing (like AETG for Web [100]) are not of the main interest of this work, because they can generate only data sets from a specifications. So they do not contain correct responses of a system, and in turn cannot automatically give verdicts.

3.1 Used procedure

The procedure used during the analysis assumes that the common specification, as described in Section 3.1.1, is given. Based on this specification the following steps were performed for all tools:

1. The specification was implemented in the modeling language supported by a tool.
2. If necessary other required artifacts (test purposes, implementations) were prepared.
3. The test case generation was performed.
4. Outputs produced by a tool were gathered.

All experiments in this section were performed on a PC machine (Pentium 4 2.8 GHz, 1.96GB of RAM).

3.1.1 Specification

The common specification is the control part of a traffic light, which is used to allow pedestrians to cross a street. The abstract structure of this system is given in Figure 8. There are two types of input signals to the system: a car sensor and two buttons at both sides of the street. The sensor detects whether there are cars in the vicinity of the traffic lights and it sends two signals: `approach` if there is one or more cars and `leave` if the crossing is empty. The buttons are used by pedestrians to make a request to cross the street and they both produce a signal `press`. The output signals from the controller manage three devices: indicators, pedestrian and car lights. The indicators are flashing after the request has been made and they receive two signals `on` and `off`. The pedestrian lights receive two signals `walk` and `noWalk`, which allow and disallow crossing the street. The car lights receive one of four signals with the color of the light.

The requirements imposed on the behavior of a traffic light are as follows:

1. After receiving the signal `press` the controller sends `on` and `walk` signals.
2. A `walk` signal cannot be sent if the controller has not sent a sequence of lights `yellow` and `red`.
3. The controller cannot send a `yellow` signal if the signal `approach` has been received and after it the signal `leave` has not been received or the appropriate delay has not passed.
4. The controller sends the signal `noWalk` only after the predefined delay after sending the last `walk` signal.
5. After each `noWalk` signal the controller sends a sequence of signals `yellowRed` and `green`.
6. Before the signal `green` is sent, the controller sends an `off` signal.
7. The controller sends the signal `yellow` only after a delay after the last `green` signal.

From the above description it follows that the traffic light represents a reactive system, which waits for the input (pressing buttons or approaching cars) and then produces the required output.

3.1.2 Criteria

The tools compared in this analysis vary substantially with respect to the required input information or produced outputs. Therefore the comparison criteria are not quantifiable and they are listed below:

1. How is a specification represented: required/allowed formats of models, alternative representations,
2. What are other required artifacts?
3. Is the test case generation performed offline or online (test case generation with execution) or both are supported?
4. What is the outcome of running the tool?
5. What is the estimated scalability of the tool? The scalability is understood in terms of the size of the model of specification, for which each tool can produce test cases. In the presented experiments we asses the scalability by using a composition of the specification described in Section 3.1.1 and for how many compositions we can obtain test cases.
6. Are there interfaces to other tools?
7. How easy is it to use generated test cases directly on an implementation?

3.2 Results of the analysis

3.2.1 TorX

TorX [104] is the tool that implements an online version of Algorithm 3 for checking the **ioco** relation between LTSs. In steps that resemble choices of Algorithm 3 an implementation is executed along with the analysis of a specification. In each step the specification is analyzed for possible outputs and inputs after input-output sequence generated so far. There is a choice between sending an input to the implementation and checking an output. If at any time the implementation returns an output that is not contained in outputs predicted by the specification then the verdict of testing is failed. The choice between sending one of the inputs and checking an output is random or can be guided by a user.

Settings In order to run TorX the following are required: a specification, an implementation and configuration information, like partitioning of signals into inputs and outputs.

A specification can be in several formats: LOTOS (an example in Listing 1), Promela (an input language for SPIN [54]), LTSA and Aldebaran (explicit enumeration of all transitions). In this analysis the LOTOS language has been selected and an excerpt of the specification is given in Listing 1.

Listing 1: Excerpt from the specification of traffic lights in LOTOS

```
1  specification
     LIGHTS[PRESS_A, PRESS_B, APPROACH, LEAVE, WALK, NOWALK, ON,
3                OFF, GREEN, YELLOW, RED,YELLOWRED]: noexit
     behaviour
5    hide PRESS, A, L in(
         Controller[PRESS, A, L, WALK, NOWALK, ON, OFF, GREEN, YELLOW, RED, YELLOWRED]
7            |[PRESS,WALK, NOWALK, A, L, ON, OFF, GREEN, YELLOW, RED, YELLOWRED]|
         (
9            Buttons[PRESS_A,PRESS_B,PRESS]|[PRESS]|CarSensor[APPROACH,LEAVE,A,L,PRESS]|||
             PedestrianLights[WALK,NOWALK]  ||| CarLights[GREEN,YELLOW,RED,YELLOWRED]|||
11           Indicators[ON,OFF]
         ))
13   where
         /*processes Buttons, CarSensor, PedestrianLights, CarLights, Indicators*/
15       process Controller[PRESS, A, L, WALK, NOWALK,
                         ON, OFF, GREEN, YELLOW, RED, YELLOWRED]:noexit :=
17           A; Controller_WAIT[/*channels*/]   []
             PRESS; Controller_WALK[/*channels*/]
19       endproc
         process Controller_WAIT[/*channels*/]:noexit :=
21           L; Controller[/*channels*/]  []
             PRESS; (hide TIMEOUT in
23                   (L;exit  [] TIMEOUT;exit) >> Controller_WALK[/*channels*/])
         endproc
25       process Controller_WALK[/*channels*/]:noexit :=
             (YELLOW; RED; exit  ||| ON;exit)>>
27               WALK; NOWALK; (OFF;exit  ||| YELLOWRED;GREEN;exit) >>
                     PRESS; (hide DELAY in (
29                           (DELAY; Controller[/*channels*/]) []
                             (PRESS; DELAY; Controller_WALK[/*channels*/])))
31       endproc
   endspec
```

The specification identifies several signals (lines 2-3), called channels, which represent environment of the system. The overall behavior is given in lines 5-12 and it consists of several processes (like `Buttons` or `CarSensor`), which run in parallel (notation `|||`) or synchronize on predefined channels (notation `|[...]|` with channel names between square brackets). The keyword `hide` introduces channels that represent internal actions. Only the specification of the `Controller` process is shown, whereas the other ones are straightforward. The process operates in one of three "modes". In the first and the initial one (lines 15-19) the process waits either for a request to walk (`PRESS` channel) or (notation `[]`) a signal indicating approaching cars (`A` channel). In the second mode (lines 20-24) requests are deferred until all cars leave (`L` channel) or there is a request to walk and a timeout. The third mode (lines 25-31) is for dealing with request to cross the street, so the appropriate sequences of car and pedestrian lights are generated and indicators are turned on and off. Then after the predefined delay the process is ready to start the cycle again.

Implementations in TorX can be either in C with additional wrappers or can also be represented in the same way as a specification. In the analysis the second possibility was selected and two faulty models were implemented in LOTOS:

- Fault 1: the model allows for signals `YELLOWRED` and `GREEN` to be sent before `NOWALK`, so line 27 becomes `WALK; (OFF; exit ||| YELLOWRED;GREEN; exit |||NOWALK;exit)` >>, which enables parallel execution of the three sequences,

- Fault 2: the model stops after the first request, so lines 28-30 are changed to a single action `stop`.

To estimated the scallability of TorXOne the above specification was composed in parallel, i.e., if

Specification	Number of transitions in LTS	Number of steps	
		Fault 1	Fault 2
1 traffic lights model	832	6	23
2 traffic lights models	317,960	41	43
3 traffic lights models	91,570,188	77	47

Table 4: The average number of steps to detect non-conformant implementations.

several traffic lights processes operate independently. The same was applied to the faulty implementations: for the model with Fault 1 only one composed process is faulty and for the model with Fault 2 all processes are faulty.

Outcome The result of running the tool is one of the verdicts: pass, fail or inconclusive. For the specification of traffic lights TorX cannot return the pass verdict and cannot prove conformance, because the specification has infinite runs. Therefore for non-trivial systems the tool is useful mostly to detect non-conformance, in which case it also returns a trace to the erroneous state.

During the experiments the traffic lights specifications (for a single, 2 and 3 compositions) were checked against two faulty models of implementations. Table 4 gathers the number of transitions of the explicit labeled transition system for the given LOTOS specification and the average number of steps required to discover the failure in 3 consecutive trial runs. Although the size of a specification increases substantially, the tool could detect non-conformance using small number of steps in all cases. The increase in the number of necessary steps was more considerable for the first type of fault, because this implementation contains only one faulty process, therefore it takes more steps to find a fault. For specifications with more than 3 processes the tool could not proceed after several steps.

3.2.2 TGV

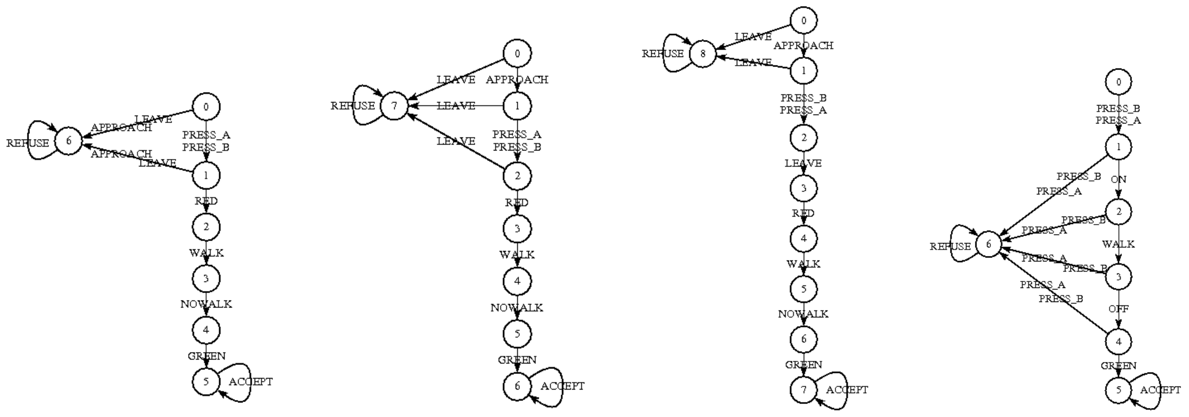
TGV is a tool that generates test cases for the version of the **ioco** relation, but with the given test purpose (Section 2.3.1). As opposed to TorX, test cases are generated offline and only after generation is finished, they can be executed on an implementation. The tool was implemented as the part of the CADP toolset and uses LOTOS as its primary specification language.

Settings Running TGV requires a specification, a test purpose and partition of channels into inputs and outputs.

A specification is implemented in LOTOS and for this analysis it is the same as the one given in Listing 1. The scalability analysis is based on the parallel composition, so there are several processes for traffic lights that run in parallel.

A test purpose (see Definition 16) uses the Aldebaran format, in which transitions are explicitly specified. In the analysis test purposes depicted in Figure 9 are used. All of them have the `ACCEPT` and `REFUSE` transitions as a self loop for a state, which is an accepting or refusing state, respectively. The test purposes describe the following situations:

- *tp*_1 (Figure 9(a)): this test purpose is used to generate test cases for the requests if there is no car approaching, so there are transitions with `APPROACH` and `LEAVE` to the refusing state,
- *tp*_2 (Figure 9(b)): this test purpose is similar to *tp*_1, but cars are approaching and there is no `LEAVE` input, so lights should be changed after a predefined delay,
- *tp*_3 (Figure 9(c)): this test purpose is as *tp*_2, but cars are approaching and then leaving,
- *tp*_4 (Figure 9(d)) - this test purpose generates test cases that check whether indicators are turned on before there is `WALK` signal and turned off before `GREEN`. To prune the state space no `PRESS_A` and `PRESS_B` signals are allowed before the `GREEN` signal.



(a) *tp*_1 (b) *tp*_2 (c) *tp*_3 (d) *tp*_4

Figure 9: Test purposes used in experiments with TGV.

Specification	Test purpose			
	*tp*_1	*tp*_2	*tp*_3	*tp*_4
1 traffic lights model	28	34	34	25
2 traffic lights model	490	620	829	73

Table 5: Number of transitions in LTSs generated by TGV.

Outcome For a given specification and a test purpose, TGV generates an LTS, which is a test case. Such an LTS indicates in each transition whether the signal is an input or output from the test execution perspective. A test case is used to send inputs to an implementation and observe outputs and if they are as expected then the test case passes. For instance, for the test purpose *tp*_1 the generated LTS starts with output signals `PRESS_A` or `PRESS_B` and then a sequence of inputs to the test case is expected. This sequence is a sequence of required signals for car and pedestrian lights.

The experiments with TGV were performed for the specification with a single traffic lights process and for specifications with 2 parallel processes. The sizes of appropriate LTSs for the specifications are the same as given in Table 4, and the sizes of the generated test cases, in terms of the number of transitions in an LTS, are given in Table 5. For larger specifications, generated test cases are also substantially larger, especially for the *tp*_2 and *tp*_3. The growth is not that significant in the other two purposes, because they are more restricted. For specifications with the more than 2 traffic lights the tool was not able to generate test cases for test purposes *tp*_1, *tp*_2 and *tp*_3 in experiments that were running for 12 hours.

3.2.3 Cover

Cover [105] is a tool developed for the UPPAAL, which is a model checker for communicating timed automata. Cover tool therefore supports the same language. The algorithm implemented in Cover is based on parallel execution of a specification and of properties, which are specified as observers. Cover works offline and produces a set of traces that satisfies the properties for the given specification.

Settings Cover requires to run a specification in UPPAAL and properties specified as observers and queries.

A specification in UPPAAL consists of timed automata that communicate through channels. In the traffic light specification these automata are: `Buttons`, `Indicators`, `CarSensor`, `CarLights`, `PedestrianLights`, `Controller` and `Environment`. This last automaton is required to continuously provide input signals such as pressing button, approaching and leaving cars. The controller automaton is shown in Figure 10. From its initial location after a `press` signal (input from either of buttons) an appropriate sequence of car lights is generated and then a `walk` signal. Before lights are changed

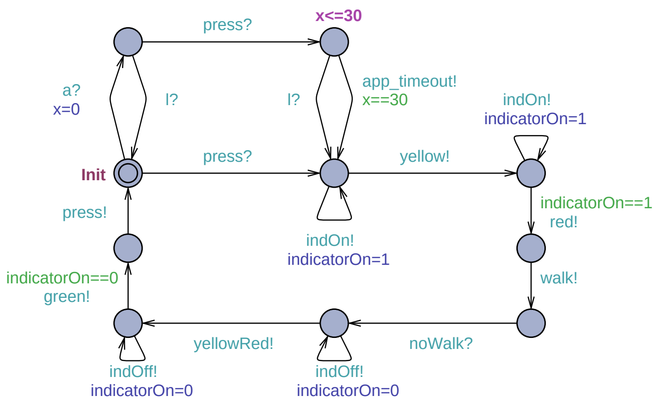


Figure 10: The `Controller` timed automaton for traffic light.

the indicators are turned on and if this happens the variable `indicatorOn` is updated. Similarly after synchronization on the `noWalk` channel with the automaton `PedestrianLights` appropriate sequence of car lights is generated and indicators are turned off. If there is a signal about approaching cars (signal `a`) and a `press` signal then either the controller waits for the leaving signal (signal `l`) or for a timeout. The latter situation is modeled with a clock variable `x`, an invariant `x <= 30` and a guard `x==30` on the timeout transition. The timed automata provide opportunity to specify timing requirements explicitly. In the traffic lights this feature is used in `CarLights` in which delays between changes of lights are included and in `PedestrianLights` to model delays between `walk` and `noWalk`.

As in the case of the previous tools the scalability analysis is based on multiplying the original specification. This is possible because each automaton in UPPAAL is implemented as a template instantiated to define a system. To specify systems that consist of multiple traffic lights, templates are instantiated several times. Unfortunately in case of the presented traffic lights model these models are not independent in the composite system. This results in interactions between automata, which might not be possible in real world applications. Nevertheless, the composite systems are in this analysis just the basis to observe the scalability of the tool.

Besides a specification Cover requires a definition of properties. Such a definition consists of an observer and a query. The first one is a parameterized automaton with rules that serve as its transitions, and which are fired while executing a UPPAAL system. A query uses an observer by providing possible values for its parameters. In the traffic lights model the following observers are implemented:

- `nodeObs`: used to generate test cases that cover all location of an automaton and satisfy the location coverage criterion,
- `edgeObs`: used to generate test cases that satisfy the edge coverage criterion,
- `varObs`: used for the definition-use coverage for the `indicatorOn` variable.

Queries connect the above observers and the specific UPPAAL system of timed automata. Therefore, they were implemented separately for each specification, depending on how many traffic light models it consisted of.

Outcome Given a UPPAAL model, an observer and a query, Cover generates a set of traces which satisfy the coverage given in the observer. For example for a `nodeObs` and a traffic lights model there are 2 traces. The first one starts in the initial state and then synchronizes on the `pressA`, `approach`, `a`, `press` and `leave` channels. The second one goes through locations in the loop without the `approach` channel. This trace also identifies delays if they are required. Unfortunately the tool generates the same traces for location and edge coverage criteria, which is not explained in the documentation. In turn the edge coverage criterion is not satisfied by the generated traces.

The experiments with Cover were performed for observers described above and for specifications that consisted of 1 and 2 traffic lights models. For each of these cases and for each observer the queries

Specification	Observer					
	`nodeObs`		`edgeObs`		`varObs`	
	number of traces	total length	number of traces	total length	number of traces	total length
1 traffic lights	2	21	2	21	3	41
2 traffic lights	2	71	2	71	4	96

Table 6: Number and total length of traces generated by Cover.

were specified. Table 6 shows the number of generated traces as well as their total length in number of transitions in each trace. It can be observed that the number of traces is similar for both specifications, however the size of those traces increases. This is the result of the cyclic behavior of traffic lights, which means that in each test case there are several cycles through the timed automaton to cover all locations or edges. Any experiments with more than 2 traffic lights models finished with an exception stating that there is not enough memory.

3.2.4 SpecExplorer

SpecExplorer [76] is a model-based test generation tool that uses object-oriented specifications in Spec# or ASML#. These languages have their semantics based on ASMs (Defintion 13). A model in Spec# or ASML# is a set of actions with enabling conditions, some of which are distinguished as observable ones for input from the environment. The test case generation in SpecExplorer consists of two phases. The first one is to explore the specification by executing all possible actions enabled in each state. In this way the finite state representation is built. In the second phase the representation is used to construct test cases which are sequences of actions. The tool provides several ways to restrict the exploration and to specify search criteria of the constructed finite representation.

Settings In the analysis the Spec# language was chosen to implement the traffic lights system. This specification consists of several classes that store states of buttons, indicators, a car sensor, car and pedestrian lights. The class Controller defines actions possible in the model. An excerpt of this class is given in Listing 2.

Listing 2: Excerpt from the Specification of the Controller class in Spec#

```
class Controller{
    Indicators indicators;
    PedestrianLights pLights;
    Button buttonA, buttonB;
    CarLights cLights;
    CarSensor cs;
    Delay delayWalk, delayGreen, delayIsCar;

    [Action(Kind=ActionAttributeKind.Observable)]
    void press_A()
    {
        if (cLights.l == Lights.Green && !indicators.on) {
            buttonA.pressed = true;
        }
    }
    [Action]
    SignalWalk walk()
        requires !pLights.walk;
        requires cLights.l == Lights.Red;
        requires indicators.on;
    {
        delayWalk.set();
        pLights.setWalk();
        return SignalWalk.Walk;
  }
    /*other actions : press_B, approach, leave, indOff, indOn, timeout, toRed,
        toYellowred, toGreen*/
}
```

Specification	Numer of transitions	Kind of test suite				
		Transition cov.		Random walk		Shortest path
		Number of segments	Length	Number of segments	Length	Number of segments
1 traffic lights model	421	425	536	123	200	72
2 traffic lights models	59,896	n/a	n/a	9,125	200	7,209

Table 7: Number of segments and lengths of test suites generated by SpecExplorer.

The class has several attributes declared in lines 2-7. Two actions are shown: one is observable and one is controllable. In the example, observable actions like `press_A` (lines 9-15), may be always executed, but they update a state only if the specified conditions are satisfied. For example pressing the button is considered only if lights for cars are green and indicators are turned off (line 12). The second action (lines 16-25) is controllable and has several conditions to restrict its enabling. The method `walk` requires (lines 18-20) that lights for cars are red, indicators are turned on and the lights for pedestrians are not set to walk already. Executing this action starts the timer and changes the state of pedestrian lights. Delays in this specification are also treated as actions that are enabled only if appropriate timer is set. The complete specification contains implementations of other actions in the Controller class, other classes and helper methods.

The model exploration in SpecExplorer can be restricted in different ways. One way is to limit the possible values of parameters used in actions. The other way is to define filters on states so that only states that satisfy the given conditions are explored. States can be also grouped based on a user-defined expression, and only a limited number of representative states is further explored. The exploration can be also bounded. In the traffic lights model only restricting the values of parameters is introduced in an action that is responsible for creating the required number of Controller objects.

The explored specification is represented as a finite machine. The test generation from such a machine can be performed offline and online. In the first mode the test cases can cover all transitions in the machine, they can be generated by a random selection of the next transition up to a certain depth bound or they can cover the shortest path to a specified state. All of these possibilities were used during the experiments. The online testing uses a similar approach but actions are executed directly on the implementation, therefore test cases are generated dynamically. This option was not used in the experiments.

Outcome The model of the traffic lights system is first executed to build a finite machine. The test suite generated from the machine contains test segments, which are sequences of actions and are distinguished to account for different observable actions. The user can optionally provide actions bindings, which indicate how actions are mapped to methods of an implementation provided in reference libraries. With those bindings a test suite can run within SpecExplorer or can be exported as a C# library or an executable program.

The experiments in SpecExplorer were performed for specifications with 1 and 2 traffic lights models. Table 7 presents the sizes of machines (number of transitions) in each case and the sizes of test suites for the transition coverage, random choice bounded to the length 200 and the shortest path to the state in which all car traffic lights have the yellow red color.

The number of states in the finite machine for the model with 2 traffic lights is substantially larger than that of the model with 1 traffic light. For the 2 traffic lights case the tool could not produce test cases for the transition coverage after running it for 12 hours. For other test suites the number of generated test segments was also much larger, even for bounded random walk, which indicates that there are more possibilities based on observable actions. For models with a greater number of traffic lights the tool could not explore them in experiments that lasted 24 hours.

3.3 Summary

In this part of the paper we provided an overview of tools used in the model-based test case generation for reactive systems. Table 8 summarizes those tools based on the questions presented at the beginning of this section.

Aspect	TorX	TGV	Cover	SpecExplorer
1.Representation of specification	LOTOS, Promela, Aldebaran, LTSA	LOTOS, Aldebaran	UPPAAL timed automata	Spec#, ASML#
2.Other required artifacts	Implementation (C code, model as specification), partition of signals	Test purpose (Aldebaran), partition of signals	Observer, query	Exploration settings, kind of test suite, bindings of actions (for exporting and online)
3.Support for offline/online testing	Online only	Offline only	Offline only	Offline and online
4.Outcome of running a tool	Verdict: fail, pass, inconclusive	LTS representation of a test case	Set of traces (with transitions and delays)	Test harness (offline), verdict (online)
5.Scalability of a tool (number of compositions of a traffic lights model that a tool can work with)	up to 3	up to 2	up to 2	up to 2
6.Interfaces to other tools	CADP, SPIN - for specifications	CADP - for specifications	UPPAAL - for specifications	Microsoft Visual Studio - for test execution
7.Support for direct execution on an implementation	must implement wrappers on C code	must manually translate LTSs	must manually translate traces	must define bindings between actions and methods (direct execution supported)

Table 8: Summary of comparative analysis of tools.

The analyzed tools implement several different types of model-based test case generation. The tools vary with respect to the artifacts they require. For instance, TorX implements conformance based on complete models, so it requires only models of specification and an implementation. The other tools implement properties-based testing, therefore they require additional information about the selected properties. TGV is based on the parallel composition of a specification and properties, so properties are modeled in the similar way as specifications. UPPAAL Cover is used to generate test cases based on coverage, which are provided as observers. Finally SpecExplorer generates test cases that are based on restricted finite representation of a specification, so the properties are used to limit possible inputs, that is, to define the environment of systems.

The important aspect of all formally grounded analysis methods is their scalability, which is based on the size of the specification. In case of model-based test case generation this is also a problem. Although the used common model is rather straightforward, tools were not able to generate test cases for more than three compositions of such a system. TorX performed the best in terms of finding faults in models of SUTs, but it could not give confirmation that models are conformant. Other tools in terms of scalability achieved weaker results.

Generated test cases are executed on an implementation of a system, i.e., they should be applicable to the source code. In the set of the analyzed tools only TorX and SpecExplorer have such capabilities. For TorX wrapper code, which translates actions and output/input from and to a C program, needs to be implemented. In case of SpecExplorer more possibilities are offered and the tool requires only that there is a DLL library, which provides a source to define bindings between its methods and actions in the Spec# specification. If such bindings are provided, offline and online testing can be applied directly on an implementation. TGV and Cover do not provide any support to execute test cases. Therefore generated test cases must be manually translated to be used on an implementation.

4 Conclusions

This paper explores model-based test case generation for reactive systems. The paper contains two parts: the review of research and the comparative analysis of selected tools in the above area. In the first part we presented works that propose methods to generate test cases that rely on the given model of specification. The second part of the paper is the analysis of tools that implement model-based test case generation.

Model-based testing uses a model of required behaviors (a specification) to detect conformance with an implementation by executing on it a set of generated test cases (a test suite). Detection of the conformance increases the confidence that the given implementation is correct. In the first part we presented several variations of this idea with emphasis on the test case generation part. They differ in the following aspects:

1. Definition of conformance. The most common way is to consider all behaviors of a specification and to require that a conformant implementation has the same ones. However such a definition may be too restrictive and may lead to excessively large test suites. Therefore there are approaches that limit the conformance to consider only some specific parts of the specification. This limitation might be based on the structure (coverage criteria) or on some other properties of specifications as well as the environment in which the system works.

2. A modeling language used to express a specification. There are many formally defined languages that can represent a specification of a reactive system. Some of them, like FSMs or LTSs, are explicit and they specify all possible states of a model. However there are many others that provide more succinct representations, for example timed automata, Statecharts and ASMs. The methods to generate test cases from more comprehensive models usually adapt the methods proposed for more basic formalisms.

3. Underlying methods. Some of the presented algorithms were specifically designed to generate test cases, such as the W-method. However there are many works that reuse some other analysis methods for models. For instance search methods in graph-like structures with breadth- or depth-first search strategies. Another common underlying technique is model checking, which use counterexamples produced by model checkers as test cases.

The definition of conformance was used as the basic criterion to group works in the review part of this paper.

As presented in the review and in the analysis of tools, the main gain of using the model-based testing approach is automation of the testing process. This is especially evident for the methods for conformance based on complete models, because they analyze the whole specification. For methods that adapt a properties-based view of conformance, it is still required to manually define coverage criteria, test purposes or scenarios. The choice of the most useful ones in some cases is not trivial. Nevertheless, the task of specifying such properties is obviously less complex than defining the whole test suite.

The main flaw of the presented methods is their limited scalability. The analysis of tools showed that the tools where not able to generate test cases for a system consisting of more than 3 straightforward entities. The scalability issues are common to all methods based on any kind of a state space exploration. The test case generation is one of them and must in one way or another deal with all states that a specification may be in, or that properties require it to be in. This defines a tradeoff: either smaller part of specifications are tested and test suites are less complex or test suites are more complex, but larger parts of a specification are covered, so testing is more comprehensive. Model-based testing uses standard techniques to overcome the scalability problems such as: symbolic analysis (to avoid analysis of all values of variables) or online versions of algorithms (to avoid excessive storage).

Another limitation of the presented methods is that they are usually dedicated to only one modeling language. Therefore they assume that the whole system is specified using one formalism, for example with FSMs or Statecharts. Such an assumption holds for many software projects, but there are also projects that have heterogeneous specifications. This means that different parts of the system are modeled with different formalisms. The only way to reuse the methods presented in this paper is to transform such models into a selected common language. Such a solution is not appropriate for many large systems, so the presented techniques are not directly applicable to heterogeneous models.

Even with the above drawbacks the model-based testing approach is very attractive, especially with the increasing interest in the Model Driven Engineering [65] paradigm. This concept proposes to use models during the development at several levels of abstraction. The more abstract models are typically independent of any implementation details. They are transformed or refined and incorporate the platform dependent knowledge, so they become the basis of implementation. In this approach models are actively maintained, so they can also be used for the test case generation. Another benefit is that the generation can be performed at several stages of the development and therefore may be even more effective. In such settings model-based testing methods seem very suitable.

References

[1] Unified Modeling Language (UML), UML 2.0 Superstructure. http://www.uml.org/.

[2] A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604 – 15, 1991/11.

[3] R. Alur. Timed automata. In *Proceedings of 11th International Conference on Computer Aided Verification (CAV 1999) (Lecture Notes in Computer Science Vol.1633)*, pages 8 – 22, Trento, Italy, 1999.

[4] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.

[5] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In *Proceedings of REX Workshop.Real-Time: Theory in Practice (Lecture Notes in Computer Science Vol. 600)*, pages 74 – 106, 1992.

[6] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of Second International Conference on Formal Engineering Methods (ICFEM 1998)*, pages 46–54, Brisbane, Australia, 1998.

[7] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *Proceedings of 7th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2001)*, pages 212 – 21, Skövde, Sweden, 2001.

[8] P. Ammann, J. Offutt, and Wuzhi Xu. Coverage criteria for state based specifications. *An Outcome of the FORTEST Network. Formal Methods and Testing. Revised Selected Papers.*, 4949:118 – 56, 2008.

[9] Paul Ammann and Jeff Offutt. *Introduction to software testing.* New York, Cambridge University Press, 2008.

[10] B. Badban, M. Franzle, J. Peleska, and T. Teige. Test automation for hybrid systems. In *Proceedings of the Third International Workshop on Software Quality Assurance (SOQUA 2006)*, pages 14 – 21, Portland, OR, USA, 2006.

[11] F. Belli and A. Hollmann. Test generation and minimization with "basic" statecharts. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2008)*, pages 718 – 723, Fortaleza, Ceara, Brazil, 2008.

[12] P.E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 81 – 8, Grenoble, France, 2000.

[13] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for statecharts. In *Proceedings of International Workshop on Current Trends in Applied Formal Methods. Applied Formal Methods - FM-Trends 98. (Lecture Notes in Computer Science Vol.1641)*, pages 107 – 21, Boppard, Germany, 1999.

[14] S. Boroday, A. Petrenko, and R. Groz. Can a Model Checker Generate Tests for Non-Deterministic Systems? *Proceedings of the Third Workshop on Model Based Testing (MBT 2007). Electronic Notes in Theoretical Computer Science*, 190(2):3 – 19, 2007.

[15] L.B. Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *4th International Workshop on Formal Approaches to Software Testing (FATES 2004). Revised Selected Papers (Lecture Notes in Computer Science Vol.3395)*, pages 64 – 78, Linz, Austria, 2004.

[16] L.B. Briones and E. Brinksma. Testing real-time multi input-output systems. In *Proceedings of 7th International Conference on Formal Engineering Methods (ICFEM 2005). Formal Methods and Software Engineering. (Lecture Notes in Computer Science Vol. 3785)*, pages 264 – 79, Manchester, UK, 2005.

[17] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems. Advanced Lectures (Lecture Notes in Computer Science Vol.3472).* Springer Berlin / Heidelberg, 2005.

[18] J.R. Callahan, S.M. Easterbrook, and T.L. Montgomery. Generating test oracles via model checking. Technical report, NASA, 1998.

[19] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In *Proceedings of Second International Conference on Test and Proofs (TAP 2008) (Lecture Notes in Computer Science Vol. 4969)*, pages 66 – 83, Prato, Italy, 2008.

[20] R. Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350 – 71, 2000.

[21] W. Chen, C Tang, and S.T. Vuong. Improving the UIOv method for protocol conformance testing. *Computer Communications*, 18(9):609 – 19, 1995/09.

[22] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178 – 87, 1978/05.

[23] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, Mass., 1999.

[24] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Advanced NATO Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, pages 173 – 239, Le Chesnay, France, 1984.

[25] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of 32nd Design Automation Conference*, pages 427 – 32, New York, NY, USA, 1995.

[26] Confirmiq. *Confirmiq Qtronic*. http://www.conformiq.com/products.php.

[27] C. Constant, T. Jeron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558 – 74, 2007/08/.

[28] R. de Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211 – 37, 1987.

[29] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Automata, Languages and Programming, Lecture Notes in Computer Science*, 154:548–560, 1982.

[30] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *Proceedings 19th IEEE Real-Time Systems Symposium (RTSS 1988)*, pages 220 – 9, Madrid, Spain, 1998.

[31] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In *4th International Workshop on Formal Approaches to Software Testing (FATES 2004). Revised Selected Papers (Lecture Notes in Computer Science Vol.3395)*, pages 1 – 15, Linz, Austria, 2004.

[32] G. Fraser and F. Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Software Quality Journal*, 16(2):161 – 83, 2008/06.

[33] G. Fraser, F. Wotawa, and P. Ammann. Issues in using model checkers for test case generation. *Journal of Systems and Software*, 82(9):1403 – 1418, 2009.

[34] G. Fraser, F. Wotawa, and P.E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215 – 61, 2009/09.

[35] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591 – 603, 1991.

[36] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 1999)*, volume 24, pages 146 – 62, Toulouse, France, 1999.

[37] A. Gargantini and E. Riccobene. ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, 7:1050–1067, 2001.

[38] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to generate tests from ASM specifications. In *Proceedings of 10th International Workshop on Advances in Theory and Practice (ASM 2003) (Lecture Notes in Computer Science Vol. 2589)*, pages 263–277, Taormina, Italy, 2003.

[39] C. Gaston and D. Seifert. *Model-Based Testing of Reactive Systems. Advanced Lectures*, chapter Evaluating coverage based testing, pages 293 – 322. Springer-Verlag, Berlin, 2005.

[40] S. Gnesi, D. Latella, and M. Massink. Formal test-case generation for UML statecharts. In *Proceedings of 9th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2004)*, pages 75–84, Florence, Italy, 2004.

[41] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. *Software Engineering Notes*, 27(4):112 – 22, 2002/07.

[42] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE, Invited Paper.*, 79(9):1305 – 20, 1991/09.

[44] D. Harel. Statecharts: a visual formalism for complex system. *Science of Computer Programming*, 8(3):231 – 74, 1987/06.

[45] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293 – 333, 1996/10.

[46] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2004). Software Engineering Notes.*, volume 29, pages 129 – 32, Boston, USA, 2004/07/.

[47] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In *Proceedings of Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII)*, pages 23 – 38, London, UK, 1997.

[48] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proceedings of 19th International Colloquium on Automata, Languages and Programming (ICALP 1992)*, pages 545 – 58, Wien, Austria, 1992.

[49] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using sUPPAAL. *Formal Methods and Testing. An Outcome of the FORTEST Network. Revised Selected Papers.*, 5215:250–264, 2008.

[50] A. Hessel and P. Pettersson. A test case generation algorithm for real-time systems. In *Proceedings of 4th International Conference on Quality Software (QSIC 2004).*, pages 268 – 73, Braunschweig, Germany, 2004.

[51] A. Hessel and P Pettersson. A global algorithm for model-based test suite generation. In *Proceedings of Third Workshop on Model-Based Testing (MBT 2007)*, Paris, France, 2007.

[52] R.M. Hierons. Testing from a Z specification. *Software Testing, Verification and Reliability*, 7(1):19 – 33, 1997/03.

[53] R.M. Hierons, S. Sadeghipour, and H. Singh. Testing a system specified using Statecharts and Z. *Information and Software Technology*, 43(2):137 – 49, 2001/02.

[54] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[55] H. Hong, S. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proceedings of 25th International Conference on Software Engineering (ICSE 2003)*, pages 232 – 42, Portland, OR, USA, 2003.

[56] H. Hong, Y. Kim, S. Cha, D. Bae, and H. Ural. Test sequence selection method for statecharts. *Software Testing Verification and Reliability*, 10(4):203 – 227, 2000.

[57] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002) (Lecture Notes in Computer Science Vol.2280)*, pages 327 – 41, Grenoble, France, 2002.

[58] S.H. Hong and H. Ural. Dependence testing: extending data flow testing with control dependence. In *Proceedings of 17th IFIP TC6/WG 6.1 International Conference TestCom 2005. Testing of communicating systems (Lecture Notes in Computer Science Vol. 3502)*, pages 23 – 39, Montreal, Canada, 2005.

[59] IBM. *IBM Rational Rhapsody*. http://www.ibm.com/developerworks/rational/products/rhapsody/.

[60] INRIA Rhone-Alpes. *TGV (Test Generation with Verification technology)*. http://www-verimag.imag.fr/ async/TGV/index.shtml.en.

[61] F. Ipate and M. Holcombe. Generating test sets from non-deterministic stream X-machines. *Formal Aspects of Computing*, 12(6):443 – 58, 2000.

[62] C. Jard and T. Jeron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297 – 315, 2005/08.

[63] B. Jeannet, T. Jeron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005) (Lecture Notes in Computer Science Vol.3440)*, pages 349 – 64, Edinburgh, UK, 2005.

[64] Thierry Jron. Symbolic model-based test selection. *Electronic Notes in Theoretical Computer Science*, 240:167 – 184, 2009.

[65] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[66] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *Proceedings of 11th International SPIN Workshop on Model Checking Software (SPIN 2004) (Lecture Notes in Computer Science Vol. 2989)*, pages 109 – 26, Barcelona, Spain, 2004.

[67] D.R. Kuhn and V. Okun. Pseudo-exhaustive testing for software. In *30th Annual IEEE/NASA Software Engineering Workshop (SEW 2006)*, pages 153 – 158, Columbia, MD, USA, 2006.

[68] K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Proceedings of 18th IEEE Real-Time Systems Symposium (RTSS 1997)*, pages 14 – 24, San Francisco, CA, USA, 1997.

[69] D. Latella and M. Massink. A formal testing framework for uml statechart diagrams behaviours: from theory to automatic verification. In *Proceedings of Sixth IEEE International Symposium on High Assurance Systems Engineering (HASE 2001). Special Topic: Impact of Networking*, pages 11 – 22, Albuquerque, NM, USA, 2001.

[70] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of 26th Annual ACM Symposium on Theory of Computing*, pages 264 – 274, Victoria, BC, Canada, 1992.

[71] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 – 123, 1996/08.

[72] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings of International Symposium of Formal Methods Europe (FME 2002). Formal Methods-Getting IT Right. (Lecture Notes in Computer Science Vol. 2391)*, pages 21 – 40, Copenhagen, Denmark, 2002.

[73] H. Lötzbeyer and A. Pretschner. AutoFocus on constraint logic programming. In *Proceedings of (Constraint) Logic Programming and Software Engineering (LPSE 2000)*, London, UK, 2000.

[74] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* New York : Springer-Verlag, 1992.

[75] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATEL. In *Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 229 – 37, Grenoble, France, 2000.

[76] Microsoft Research. *SpecExplorer.* http://research.microsoft.com/en-us/projects/SpecExplorer/.

[77] Glenford Myers. *The art of software testing, second edition.* John Wiley & Sons, 2004.

[78] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, volume 29, pages 55 – 64, Boston, MA, USA., 2004/07.

[79] B. Nielsen and A. Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5(1):59 – 77, 2003.

[80] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of Second International Conference on Unified Modeling Language (UML 1999). Beyond the Standard. (Lecture Notes in Computer Science Vol.1723)*, pages 416 – 29, Fort Collins, CO, USA, 1999.

[81] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for Smart Cards. In *Proceedings of 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2003). Electronic Notes in Theoretical Computer Science*, volume 80, pages 170 – 184, 2003.

[82] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(3):241 – 84, 1987.

[83] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, pages 46–57, Providence, Rhode Island, USA, 1977.

[84] W. Prenninger and A. Pretschner. Abstractions for model-based testing. In *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004). Electronic Notes in Theoretical Computer Science*, volume 116, pages 59 – 71, Barcelona, Spain, 2005.

[85] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Proceedings of Formal Approaches to Testing of Software (FATES 2001)*, pages 47–60, Aalborg, Denmark, 2001.

[86] A. Pretschner, H. Lotzbeyer, and J. Philipps. Model based testing in incremental system development. *Journal of Systems and Software*, 70(3):315 – 29, 2004/03.

[87] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real : the inhouse card case study. *International Journal on Software Tools for Technology Transfer*, 5(2-3):140 – 57, 2004.

[88] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367 – 75, 1985/04.

[89] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, pages 200 – 9, Madrid, Spain, 1998.

[90] Reactive Systems. *Reactis. Model-based Testing and Validation.* http://www.reactive-systems.com/.

[91] V. Rusu, L. Du Bousquet, and T. Jeron. An approach to symbolic test generation. In *Proceedings of Second International Conference on Integrated Formal Methods (IFM 2000) (Lecture Notes in Computer Science Vol. 1945)*, pages 338 – 57, Schloss Dagstuhl, Germany, 2000.

[92] V. Rusu, H. Marchand, and T. Jeron. Automatic verification and conformance testing for validating safety properties of reactive systems. In *Proceedings of International Symposium of Formal Methods Europe (FM 2005) (Lecture Notes in Computer Science Vol. 3582)*, pages 189 – 204, Newcastle, UK, 2005.

[93] V. Santiago, A.S.M. do Amaral, N.L. Vijaykumar, Md.F. Mattiello-Francisco, E. Martins, and O.C. Lopes. A practical approach for automated test case generation using statecharts. In *Proceedings of 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, pages 183 – 188, Chicago, USA, 2006.

[94] B. Seljimi and I. Parissis. Using CLP to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *Proceedings of 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*, pages 105–116, Raleigh, NC, USA, 2006.

[95] D.P. Sidhu and T.-K. Leung. Formal methods for protocol testing: a detailed study. *IEEE Transactions on Software Engineering*, 15(4):413 – 26, 1989/04.

[96] J. Springintveld, F. Vaandrager, and P. R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225 – 257, 2001.

[97] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777 – 93, 1996/11.

[98] L. Tan, J. Kim, O. Sokolsky, and I. Lee. Model-based testing and monitoring for hybrid embedded systems. In *Proceedings of the 2004 IEEE Conference on Information Reuse and Integration (IRI 2004)*, pages 487 – 92, Las Vegas, NV, USA, 2004.

[99] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Proceedings of the 2004 IEEE Conference on Information Reuse and Integration (IRI 2004)*, pages 493 – 8, Las Vegas, NV, USA, 2004.

[100] Telcordia Technologies. *AETG Web.* http://aetgweb.argreenhouse.com/.

[101] J. Tretmans. Test generation with inputs, outputs, and quiescence. In *Proceedings of Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1996) (Lecture Notes in Computer Science Vol. 1055)*, pages 127 – 146, Passau, Germany, 1996.

[102] J. Tretmans. Model based testing - property checking for real. keynote speaker. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, Marseille, France, 2004. http://www-sop.inria.fr/everest/events/cassis04/.

[103] J. Tretmans. Model based testing with labelled transition systems. *Formal Methods and Testing*, 4949:1 – 38, 2008.

[104] University of Twente (UT). *TorX.* http://fmt.cs.utwente.nl/tools/torx/introduction.html.

[105] Uppsala University, Aalborg University. *UPPAAL Cover.* http://www.uppaal.com/CoVer.

[106] H. Ural and B. Yang. A test sequence selection method for protocol testing. *IEEE Transactions on Communications*, 39(4):514 – 23, 1991/04.

[107] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical report, Department of Computer Science, The University of Waikato, New Zealand, 2006.

[108] M. van der Bijl, A. Rensink, and J. Tretmans. Action refinement in conformance testing. In *Proceedings of 17th IFIP TC6/WG 6.1 International Conference TestCom 2005. Testing of communicating systems (Lecture Notes in Computer Science Vol. 3502)*, pages 81 – 96, Montreal, Canada, 2005.

[109] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with SpecExplorer. *Formal Methods and Testing*, 4949:39 – 76, 2008.

[110] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80 – 91, 1997/05.

[111] J.A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70 – 9, 2000/01/.

[112] D. Wijesekera, P. Ammann, L. Sun, and G. Fraser. Relating counterexamples to test cases in CTL model checking specifications. In *Proceedings of the 3rd international workshop on Advances in model-based testing (A-MOST 2007)*, pages 75–84, London, UK, 2007.