




Test Model Coverage Analysis Under Uncertainty

I. S. W. B. Prasetya^(✉)  and Rick Klomp

Utrecht University, Utrecht, The Netherlands
s.w.b.prasetya@uu.nl

Abstract. In model-based testing (MBT) we may have to deal with a non-deterministic model, e.g. because abstraction was applied, or because the software under test itself is non-deterministic. The same test case may then trigger multiple possible execution paths, depending on some internal decisions made by the software. Consequently, performing precise test analyses, e.g. to calculate the test coverage, are not possible. This can be mitigated if developers can annotate the model with estimated probabilities for taking each transition. A probabilistic model checking algorithm can subsequently be used to do simple probabilistic coverage analysis. However, in practice developers often want to know what the achieved aggregate coverage is, which unfortunately cannot be re-expressed as a standard model checking problem. This paper presents an extension to allow efficient calculation of probabilistic aggregate coverage, and moreover also in combination with k -wise coverage.

Keywords: Probabilistic model based testing · Probabilistic test coverage · Testing non-deterministic systems

1 Introduction

Model based testing (MBT) is considered as one of the leading technologies for systematic testing of software [5,6,17]. It has been used to test different kinds of software, e.g. communication protocols, web applications, and automotive control systems. In this approach, a model describing the intended behavior of the system under test (SUT) is first constructed [27], and then used to guide the tester, or a testing algorithm, to systematically explore and test the SUT's states. Various automated MBT tools are available, e.g. JTorX [4,26], Phact [11], OSMO [14], APSL [24], and RT-Tester [17].

There are situations where we end up with a non-deterministic model [13,17,23], for example when the non-determinism within the system under test, e.g. due to internal concurrency, interactions with an uncontrollable environment (e.g. as in cyber physical systems), or use of AI, leads to observable effects at the model level. Non-determinism can also be introduced as byproduct when we apply abstraction on an otherwise too large model [20]. Models mined from

executions logs [7, 21, 28] can also be non-deterministic, because log files only provide very limited information about a system’s states.

MBT with a non-deterministic model is more challenging. The tester cannot fully control how the SUT would traverse the model, and cannot thus precisely determine the current state of the SUT. Obviously, this makes the task of deciding which trigger to send next to the SUT harder. Additionally, coverage, e.g. in terms of which states in the model have been visited by a series of tests, cannot be determined with 100% certainty either. This paper will focus on addressing the latter problem—readers interested in test cases generation from non-deterministic models are referred to e.g. [13, 16, 25]. Rather than just saying that a test sequence *may* cover some given state, we propose to *calculate the probability* of covering a given coverage goal, given modelers’ estimation on the local probability of each non-deterministic choice in a model.

Given a probabilistic model of the SUT, e.g. in the form of a Markov Decision Process (MDP) [3, 22], and a test σ in the form of a sequence of interactions on the SUT, the most elementary type of coverage goal in MBT is for σ to cover some given state s of interest in the model. Calculating the probability that this actually happens is an instance of the probabilistic reachability problem which can be answered using e.g. a probabilistic model checker [3, 10, 15]. However, in practice coverage goals are typically formulated in an ‘aggregate’ form, e.g. to cover at least 80% of the states, without being selective on which states to include. Additionally, we may want to know the aggregate coverage over pairs of states (the transitions in the LTS), or vectors of states, as in k -wise coverage [1], as different research showed that k -wise greatly increases the fault finding potential of a test suite [9, 18]. Aggregate goals cannot be expressed in LTL or CTL, which are the typical formalisms in model checking. Furthermore, both types of goals (aggregate and k -wise) may lead to combinatorial explosion.

This paper **contributes**: (1) a concept and definition of probabilistic test coverage; as far as we know this has not been covered in the literature before, and (2) an algorithm to calculate probabilistic coverage, in particular of aggregate k -wise coverage goals.

Paper Structure. Section 2 introduces relevant basic concepts. Section 3 introduces the kind of coverage goals we want to be able to express and how their probabilistic coverage can be calculated. Section 4 presents our algorithm for efficient coverage calculation. Section 5 shows the results of our benchmarking. Related work is discussed in Sect. 6. Section 7 concludes.

2 Preliminary: Probabilistic Models and Simple Coverage

As a running example, consider the labelled transition system (LTS) [2] in Fig. 1 as a model of some SUT. The transitions are labelled with actions, e.g. a and b . A non- τ action represents an interaction between the SUT and its environment. In our set up such an action is assumed to occur *synchronously* a la CSP [12] (for an action a to take place, both the SUT and the environment first need to agree on doing a ; then they will do a together). The action τ represents an internal action by the SUT, that is not visible to the environment.

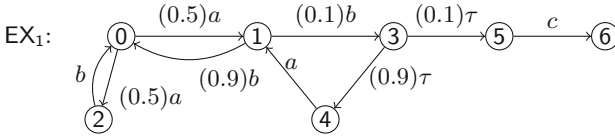


Fig. 1. An example of a probabilistic model of some SUT called EX_1 .

To test the SUT, the tester controls the SUT by insisting on which action it wants to synchronize; e.g. if on the state t the SUT is supposed to be able to either do a or b , the tester can insist on doing a . If the SUT fails to go along with this, it is an error. The tester can also test if in this state the SUT can be coerced to do an action that it is not supposed to synchronize; if so, the SUT is incorrect. We will assume a black box setup. That is, the tester cannot actually see the SUT’s state, though tester can try to infer this based on information visible to him, e.g. the trace of the external actions done so far. For example after doing a from the state 0 on the SUT EX_1 above, the tester cannot tell whether it then goes to the state 1 or 2. However, if the tester manages to do abc he would on the hind sight know that the state after a must have been 1.

When a state s has multiple outgoing transitions with the same label, e.g. a , this implies non-determinism, since the environment cannot control which a the SUT will take (the environment can only control whether or not it wants to do a). We assume the modeler is able estimate the probability of taking each of these a -transitions and annotate this on each of them. E.g. in Fig. 1 we see that in state 1, two a -transitions are possible, leading to different states, each with the probability of 0.5. Similarly, in state 3 there are two τ -transitions leading to states 4 and 5, with the probability of 0.9 and 0.1 respectively. A probabilistic model such as in Fig. 1 is also called a Markov Decision Process (MDP) [3].

Let M be an MDP model, with finite number of transitions, and a single initial state. Let s, t be states, and a an action. We write $s \in M$ to mean that s is a state in M . The notation $s \xrightarrow{a} t$ denotes a transition that goes from the state s to t and is labelled with a . We write $s \xrightarrow{a} t \in M$ to mean that $s \xrightarrow{a} t$ is a transition in M . $P_M(s \xrightarrow{a} t)$ denotes the probability that M will take this particular transition when it synchronizes over a on the state s .

To simplify calculation over non-deterministic actions, we will assume that M is τ -normalized in the following sense. First, a state cannot have a mix of τ and non- τ outgoing transitions. E.g. a state s with two transitions $\{s \xrightarrow{\tau} t, s \xrightarrow{a} u\}$ should first be re-modelled as $\{s \xrightarrow{\tau} t, s \xrightarrow{\tau} s', s' \xrightarrow{a} u\}$ by introducing an intermediate state s' , and the modeler should provide estimation on the probability of taking each of the two τ transitions. Second, M should have no state whose all incoming and outgoing transitions are τ transitions. Such a state is considered not interesting for our analyses. Third, M should not contain a cycle that consists of only τ transitions. In a τ -normalized model, non-determinism can only be introduced if there is a state s with multiple outgoing transitions labelled by the same action (which can be τ).

We define an *execution* of the SUT as a finite path ρ through the model starting from its initial state. A *trace* is a finite sequence of external actions.

The trace of ρ , $\text{tr}(\rho)$, is the sequence external actions induced by ρ . A *legal trace* is a trace that can be produced by some execution of the SUT. A *test-case* is abstractly modeled by a trace. We will restrict to test-cases that form legal traces, e.g. ab , aba , and $ababc$ are test cases for EX_1 in Fig. 1. Negative tests can be expressed as legal traces by adding transitions to an error state. A set of test cases is also called a *test suite*.

Since the model can be non-deterministic, the same test case may trigger multiple possible executions which are indistinguishable from their trace. If σ is a trace, $\text{exec}(\sigma)$ denotes the set of all executions ρ such that $\text{tr}(\rho) = \sigma$, and moreover is τ -maximal: it cannot be extended without breaking the property $\text{tr}(\rho) = \sigma$. Assuming τ -maximality avoids having to reason about the probability that ρ , after being observed as σ , is delayed in completing its final τ transitions.

2.1 Representing a Test Case: Execution Model

The probability that a test case σ covers some goal ϕ (e.g. a particular state s) can in principle be calculated by quantifying over $\text{exec}(\sigma)$. However, if M is highly non-deterministic, the size of $\text{exec}(\sigma)$ can be exponential with respect to the length of σ . To facilitate more efficient coverage calculation we will represent σ with the subgraph of M that σ induces, called the *execution model* of σ , denoted by $E(\sigma)$. $E(\sigma)$ forms a Markov chain; each branch in $E(\sigma)$ is annotated with the probability of taking the branch, under the premise that σ has been observed. Since a test case is always of finite length and M is assumed to have no τ -cycle, $E(\sigma)$ is always acyclic. Typically the size of $E(\sigma)$ (its number of nodes) is much less than the size of $\text{exec}(\sigma)$. For example, the execution model of the test case aba on EX_1 is shown in Fig. 2. An artificial state denoted with $\#$ is added so that $E(\sigma)$ has a single exit node, which is convenient for later.

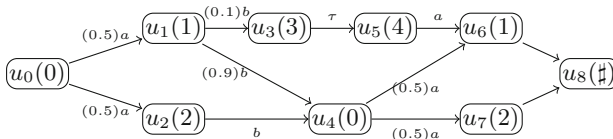


Fig. 2. The execution model of the test case aba on EX_1 .

To identify the states in $E(\sigma)$ we assign IDs to them ($u_0 \dots u_8$ in Fig. 2). We write $u.st$ to denote u 's state label, which is the ID of a state in M that u represents (so, $u.st \in M$); in Fig. 2 this is denoted by the number between brackets in every node.

Importantly, notice that the probability of the transitions in $E(\sigma)$ may be different than the original probability in M . For example, the transition $u_3 \xrightarrow{\tau} u_5$ in the above execution model has probability 1.0, whereas in the original model EX_1 this corresponds to the transition $3 \xrightarrow{\tau} 4$ whose probability is 0.9. This is because the alternative $3 \xrightarrow{\tau} 5$ could not have taken place, as it leads to an

execution whose trace does not correspond to the test case aba (which is assumed to have happened).

More precisely, when an execution in the model $E(\sigma)$ reaches a node u , the probability of extending this execution with the transition $u \xrightarrow{\alpha} v$ can be calculated by taking the conditional probability of the corresponding transition in the model M , given that only the outgoing transitions specified by $E(\sigma)$ could happen. So, $P_{E(\sigma)}(u \xrightarrow{\alpha} v)$ is $P_M(u.st \xrightarrow{\alpha} v.st)$ divided by the the sum of $P_M(u.st \xrightarrow{\alpha} w.st)$ of all w such that $u \xrightarrow{\alpha} w \in E(\sigma)$.

Let $E = E(\sigma)$. Since E is thus acyclic, the probability that SUT traverses a path/execution ρ in $E(\sigma)$ when it is given σ can be obtained by multiplying the probability of all the transitions in the path:

$$P_E(\rho) = \prod_{s \xrightarrow{\alpha} t \in \rho} P_E(s \xrightarrow{\alpha} t) \quad (1)$$

Simple Coverage Analyses. As an example of a simple analysis, let's calculate the probability that a test case σ produces an execution that passes through a given state s , denoted by $P(\langle s \rangle \mid \sigma)$. This would then just be the sum of the probability of all full executions in $E(\sigma)$ that contain s . So:

$$P(\langle s \rangle \mid \sigma) = \sum_{\rho \text{ s.t. } \rho \in E(\sigma) \wedge s \in \rho} P_{E(\sigma)}(\rho) \quad (2)$$

For example, on the execution model EX_1 , $P(\langle 1 \rangle \mid aba) = 0.525$, $P(\langle 2 \rangle \mid aba) = 0.475$, $P(\langle 4 \rangle \mid aba) = 0.05$, whereas $P(\langle 5 \rangle \mid aba) = 0$.

3 Coverage Under Uncertainty

Coverage goals posed in practice are however more complex than goals exemplified above. Let us first introduce a language for expressing 'goals'; we will keep it simple, but expressive enough to express what is later called 'aggregate k -wise' goals. A goal of the form $\langle 0, 2, 0 \rangle$ is called a *word*, expressing an intent to cover the subpath $\langle 0, 2, 0 \rangle$ in the MDP model. We will also allow disjunctions of words and sequences of words (called *sentences*) to appear as goals. For example: $(\langle 0, 2 \rangle \vee \langle 1, 0 \rangle)$; $\langle 1 \rangle$ formulates a goal to first cover the edge $0 \rightarrow 2$ or $1 \rightarrow 0$, and then (not necessarily immediately) the node 1.

The typical goal people have in practice is to cover at least $p\%$ of the states. This is called an *aggregate goal*. We write this a bit differently: a goal of the form ${}^1 \geq N$ expresses an intent to cover at least N different states. Covering at least $p\%$ can be expressed as ${}^1 \geq \lfloor p * K/100 \rfloor$ where K is the number of states in the model. To calculate probabilistic coverage in k -wise testing [1], the goal ${}^k \geq N$ expresses an intent to cover at least N different words of length k . Formally:

Definition 1. A coverage goal is a formula ϕ with this syntax:

$$\begin{aligned}
 \phi &::= S \mid A && \text{(goal)} \\
 S &::= C \mid C; S && \text{(sentence)} \\
 A &::= {}^k\geq N && \text{(aggregate goal), with } k \geq 1 \\
 C &::= W \mid W \vee C && \text{(clause)} \\
 W &::= \langle s_0, \dots, s_{k-1} \rangle && \text{(word), with } k \geq 1
 \end{aligned}$$

A sentence is a sequence $C_0; C_1; \dots$. Each C_i is called a *clause*, which in turn consists of one or more words. A *word* is denoted by $\langle s_0, s_1, \dots \rangle$ and specifies one or more connected states in an MDP.

Let ρ be an execution. If ϕ is a goal, we write $\rho \vdash \phi$ to mean that ρ covers ϕ . Checking this is decidable. For a word W , $\rho \vdash W$ if W is a segment of ρ . For a clause $C = W_0 \vee W_2 \vee \dots$, $\rho \vdash C$ if $\rho \vdash W_k$ for some k . Roughly, a sentence $C_0; C_1; \dots$ is covered by ρ if all clauses C_i are covered by ρ , and furthermore they are covered in the order as specified by the sentence. We will however define it more loosely to allow consecutive clauses to overlap, as follows:

Definition 2 (Sentence Coverage). Let S be a sentence. (1) An empty ρ does not cover S . (2) If S is a just a single clause C , then $\rho \vdash S$ iff $\rho \vdash C$. (3) If $S = C; S'$ and a prefix of ρ matches one of the words in C , then $\rho \vdash S$ iff $\rho \vdash S'$. If ρ has no such a prefix, then $\rho \vdash S$ iff $\text{tail}(\rho) \vdash S$.

An aggregate goal of the form ${}^k\geq N$ is covered by ρ if ρ covers at least N different words of size k . While sentences are expressible in temporal logic, aggregate goals are not. This has an important consequence discussed later.

Let ϕ be a coverage goal and σ a test case. Let's write $P(\phi \mid \sigma)$ to denote the probability that ϕ is covered by σ , which can be calculated analogous to (2) as follows:

Definition 3. $P(\phi \mid \sigma)$ is equal to $P(\phi \mid E)$ where $E = \mathbf{E}(\sigma)$, $P(\phi \mid E) = \sum_{\rho \text{ s.t. } \rho \in \text{exec}(E) \wedge \rho \vdash \phi} P_E(\rho)$, and where $P_E(\rho)$ is calculated as in (1).

For example, consider the test case *aba* on the SUT EX_1 . Figure 2 shows the execution model of *aba*. $P(\langle 2, 0 \rangle \mid \text{aba})$ is the probability that *aba*'s execution passes through the transition $2 \rightarrow 0$; this probability is 0.5. $P(\langle \langle 2 \rangle \vee \langle 3 \rangle \rangle; \langle 1 \rangle \mid \text{aba})$ is the probability that *aba* first visits the state 2 or 3, and sometime later 1; this probability is 0.75. $P({}^1\geq 4 \mid \text{aba})$ is the probability that the execution of *aba* visits at least four different states; this is unfortunately only 0.05.

Due to non-determinism, the size of $\text{exec}(\sigma)$ could be exponential with respect to the length of σ . Simply using the formula in Definition 3 would then be expensive. Below we present a much better algorithm to do the calculation.

4 Efficient Coverage Calculation

Coverage goals in the form of sentences are actually expressible in Computation Tree Logic (CTL) [3]. E.g. $\langle s, t \rangle; \langle u \rangle$ corresponds to $\text{EF}(s \wedge t \wedge \text{EF}u)$. It follows that

the probability of covering a sentence can be calculated through probabilistic CTL model checking [3, 10]. Unfortunately, aggregate goals are not expressible in CTL. Later we will discuss a modification of probabilistic model checking to allow the calculation of aggregate goals. We first start with the calculation of *simple sentences* whose words are all of length one.

Let S be a simple sentence, σ a test case, and $E = \mathbf{E}(\sigma)$. In standard probabilistic model checking, $P(S|\sigma)$ would be calculated through a series of multiplications over a probability matrix [3]. We will instead do it by performing labelling on the nodes of E , resembling more to non-probabilistic CTL model checking. This approach is more generalizable to later handle aggregate goals.

Notice that any node u in E induces a unique subgraph, denoted by $E@u$, rooted in u . It represents the remaining execution of σ , starting at u . When we label E with some coverage goal ψ , the labelling will proceed in such a way that when it terminates every node u in E is extended with labels of the form $u.\text{lab}(\psi)$ containing the value of $P(\psi \mid E@u)$. The labelling algorithm is shown in Fig. 3, namely the procedure `label(..)`—we will explain it below. In any case, after calling `label(E, S)`, the value of $P(S \mid \sigma)$ can thus be obtained simply by inspecting the `lab(S)` of E 's root node. This is done by the procedure `calcSimple`.

```

1: procedure calcSimple( $E, S$ )
2:   label( $E, S$ )
3:   return root( $E$ ).lab( $S$ )
4: end procedure

5: procedure label( $E, S$ )
6:    $u_0 \leftarrow$  root( $E$ )
7:   case  $S$  of
8:      $C \rightarrow$  label1( $u_0, C$ )
9:      $C; S' \rightarrow$  label( $E, S'$ ) ; label1( $u_0, S$ )
10: end procedure

11: procedure checkClause( $u, C$ )
12:    $\triangleright$  the clause  $C$  is assumed to be of
      this form, with  $k \geq 1$  :
13:   let  $\langle s_0 \rangle \vee \dots \vee \langle s_{k-1} \rangle = C$ 
14:    $isCovered \leftarrow$   $u.st \in \{s_0, \dots, s_{k-1}\}$ 
15:   return  $isCovered$ 
16: end procedure

17: procedure label1( $u, S$ )
18:    $\triangleright$  recurse to  $u$ 's successors :
19:   forall  $v \in u.next \rightarrow$  label1( $v, S$ )
20:    $\triangleright$  pre-calculate  $u$ 's successors' total
      probability to cover  $S$  :
21:    $q' \leftarrow \sum_{v \in u.next} u.pr(v) * v.lab(S)$ 
22:    $\triangleright$  calc.  $u$ 's probability to cover  $S$  :
23:   case  $S$  of
24:      $C \rightarrow$  if checkClause( $u, C$ )
                then  $q \leftarrow 1$ 
                else  $q \leftarrow q'$ 
25:      $C; S' \rightarrow$  if checkClause( $u, C$ )
                       then  $q \leftarrow u.lab(S')$ 
                       else  $q \leftarrow q'$ 
26:   end case
27:    $\triangleright$  add the calculated probability as
      a new label to  $u$  :
28:    $u.lab(S) \leftarrow q$ 
29: end procedure

```

Fig. 3. The labeling algorithm to calculate the probability of simple sentences.

Since S is a sentence, it is a sequence of clauses. The procedure `label(E, S)` first recursively labels E with the tail S' of S (line 9), then we proceed with the labelling of S itself, which is done by the procedure `label1`. In `label1`, the following notations are used. Let u be a node in E . Recall that $u.st$ denotes the ID of the state in M that u represents. We write $u.next$ to denote the set of

u 's successors in E (and not in M !). For such a successor v , $u.pr(v)$ denotes the probability annotation that E puts on the arrow $u \rightarrow v$. A label is a pair (ψ, p) where ψ is a coverage goal and p is a probability in $[0..1]$. The notation $u.lab$ denotes the labels put so far to the node u . The assignment $u.lab(\psi) \leftarrow p$ adds the label (ψ, p) to u , and the expression $u.lab(\psi)$ returns now the value of p .

The procedure `label1`(ψ) will perform the labelling node by node recursively in the bottom-up direction over the structure of E (line 19). Since E is acyclic, only a single pass of this recursion is needed. For every node $u \in E$, `label1`(u, S) has to add a new label (S, q) to the node u where q is the probability that the goal S is covered by the part of executions of σ that starts in u (in other words, the value of $P(S \mid E@u)$). The goal S will be in one of these two forms:

1. S is just a single clause C (line 24). Because S is a simple sentence, C is a disjunction of singleton words $\langle s_0 \rangle \vee \dots \vee \langle s_{k-1} \rangle$, where each s_i is an ID of a state in M . If u represents one of these states, the probability that $E@u$ covers C would be 1. Else, it is the sum of the probability to cover C through u 's successors (line 20). As an example, Fig. 4 (left) shows how the labeling of a simple sentence $\langle 1 \rangle$ on the execution model in Fig. 2 proceeds.
2. S is a sentence with more than one clause; so it is of the form $C; S'$ (line 25) where C is a clause and S' is the rest of the sentence, we calculate the coverage probability of $E@u$ by basically following the third case in Definition 2. As an example, Fig. 4 (right) shows how the labeling of $S = \langle 0 \rangle; \langle 1 \rangle$ proceeds. At every node u we first check if u covers the first word, namely $\langle 0 \rangle$. If this is the case, the probability that $E@u$ covers S would be the same as the probability that it covers the rest of S , namely $\langle 1 \rangle$. The probability of the later is by now known, calculated by `label` in its previous recursive call. The result can be inspected in $u.lab(\langle 1 \rangle)$.

If u does *not* cover S , the probability that $E(u)$ covers S would be the sum of the probability to cover S through u 's successors (calculated in line 21).

Assuming that checking if a node locally covers a clause (the procedure `checkClause` in Fig. 3) takes a time unit, the time complexity of `label1` is $\mathcal{O}(|E|)$, where $|E|$ is the size of E in terms of its number of edges. The complexity of `label` is thus $\mathcal{O}(|E| * |S|)$, where $|S|$ is the size of the goal S in terms of the number of clauses it has. The size of E is typically just linear to the length of the test case: $\mathcal{O}(N_{sucs} * |\sigma|)$, where N_{sucs} is the average number of successors that each state in M has. This is a significant improvement compared to the exponential run time that we would get if we simply use Definition 3.

4.1 Non-simple Sentences

Coverage goals in k -wise testing would require sentences with words of length $k > 1$ to be expressed. These are thus *non-simple* sentences. We will show that the algorithm in Fig. 3 can be used to handle these sentences as well.

Consider as an example the sentence $\langle 0, 2, 0 \rangle; \langle 4, 1, \# \rangle$. The words are of length three, so the sentence is non-simple. Suppose we can treat these words as if they

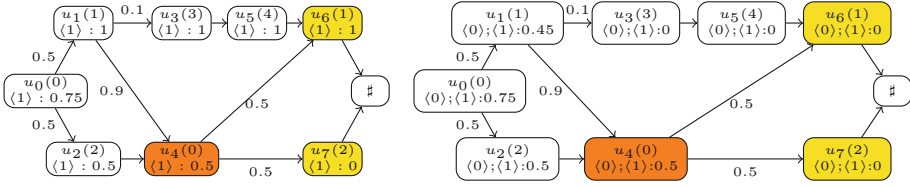
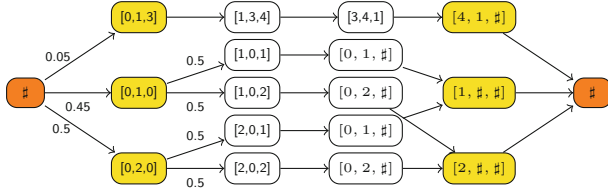


Fig. 4. The **left graph** shows the result of $\text{label}(\langle 1 \rangle)$ on the execution model of *aba* in Fig. 2. For simplicity, the action labels on the arrows are removed. The probability annotation is kept. In turn, $\text{label}()$ calls label1 , which then performs the labelling recursively from right to left. The nodes u_6 and u_7 (yellow) are base cases. The probabilities of $\langle 1 \rangle$ on them are respectively 1 and 0. This information is then added as the labels of these nodes. Next, label1 proceeds with the labelling of u_4 and u_5 . E.g. on u_4 (orange), because $u_4.\text{st}$ is not 1, for u_4 to cover $\langle 1 \rangle$ we need an execution that goes through u_6 , with the probability of 0.5. So the probability of $\langle 1 \rangle$ on u_4 is 0.5. The **right graph** shows the result of $\text{label}(\langle 0 \rangle; \langle 1 \rangle)$ on the same execution model. This will first call $\text{label}(\langle 1 \rangle)$, thus producing the labels as shown in the left graph, then proceeds with $\text{label1}(\langle 0 \rangle; \langle 1 \rangle)$. Again, label1 performs the labelling recursively from right to left. The base cases u_6 and u_7 do not cover $\langle 0 \rangle; \langle 1 \rangle$, so the corresponding probability there is 0. Again, this information is added as labels of the corresponding nodes. Node u_4 (orange) has $u_4.\text{st} = 0$. So, any execution that starts from there and covers $\langle 1 \rangle$ would also cover $\langle 0 \rangle; \langle 1 \rangle$. The probability that u_4 covers $\langle 1 \rangle$ is already calculated in the left graph, namely 0.5. So this is also the probability that it covers $\langle 0 \rangle; \langle 1 \rangle$. (Color figure online)

are singletons. E.g. in $\langle 0, 2, 0 \rangle$ the sequence 0, 2, 0 is treated as a single symbol, and hence the word is a singleton. From this perspective, any non-aggregate goal is thus a simple sentence, and therefore the algorithm in Fig. 3 can be used to calculate its coverage probability. We do however need to pre-process the execution model to align it with this idea.

The only part of the algorithm in Fig. 3 where the size of the words matters is in the procedure checkClause . Given a node u in the given execution model E and a clause C , $\text{checkClause}(u, C)$ checks if the clause C is covered by E 's executions that start at u . If the words in C are all of length one, C can be immediately checked by knowing which state in M u represents. This information is available in the attribute $u.\text{st}$. Clauses with longer words can be checked in a similar way. For simplicity, assume that the words are all of length k (note: shorter words can be padded to k with wildcards $*$ that match any symbol). We first restructure E such that the st attribute of every node u in the new E contains a word of length k that would be covered if the execution of E arrives at u . We call this restructuring step *k-word expansion*. Given a base execution model E , the produced new execution model will be denoted by E^k . As an example, the figure below shows the word expansion with $k = 3$ of the execution model in Fig. 2 (for every node v we only show its $v.\text{st}$ label, which is an execution segment of length 3). Artificial initial and terminal states are added to the new execution model, labelled with $\#$. When a word of length k cannot be formed, because the

corresponding segment has reached the terminal state $\#$ in E , we pad the word with $\#$'s on its the end until its length is k .



4.2 Coverage of Aggregate Goals

We will only discuss the calculation of aggregate goals of the form $k \geq N$ where $k=1$. If $k > 1$ we can first apply a k -word expansion (Sect. 4.1) on the given execution model E , then we calculate $1 \geq N$ on the expanded execution model.

Efficiently calculating $1 \geq N$ is more challenging. The algorithm below proceeds along the same idea as how we handled simple sentences, namely by recursing over E . We first need to extend every node u in E with a new label $u.A$. This label is a set containing pairs of the form $V \mapsto p$ where V is a set of M 's states and p is the probability that $E@u$ would cover *all* the states mentioned in V . Only V 's whose probability is non-zero need to be included in this mapping. After all nodes in E are labelled like this, the probability $1 \geq N$ can be calculated from the A of the root node u_0 :

$$P(1 \geq N \mid \sigma) = \sum_{V \mapsto p \in u_0.A} \text{if } |V| \geq N \text{ then } p \text{ else } 0 \tag{3}$$

The labelling is done recursively over E as follows:

1. The base case is the terminal node $\#$. The A label of $\#$ is just \emptyset .
2. For every node $u \in E$, we first recurse to all its successors. Then, we calculate a preliminary mapping for u in the following *multi-set* A' :

$$A' = \{ V \cup \{u.st\} \mapsto p * P_E(u \mapsto v) \mid v \in u.next, V \mapsto p \in v.A \}$$

As a multi-set note that A' may contain duplicates, e.g. two instances of $V \mapsto p_0$. Additionally, it may contain different maps that belong to the same V , e.g. $V \mapsto p_1$ and $V \mapsto p_2$. All these instances of V need to be merged by summing up their p 's, e.g. the above instances is to be merged to $V \mapsto p_0 + p_0 + p_1 + p_2$. The function `merge` will do this. The label $u.A$ is then just: $u.A = \text{merge}(A') = \{ V \mapsto \sum_{V \mapsto p \in A'} p \mid V \in \text{domain}(A') \}$, where $\text{domain}(A')$ is the set of all unique V 's that appear as $V \mapsto$. in A' .

The recursion terminates because E is acyclic.

The above algorithm can however perform worse than a direct calculation via Definition 3. The reason is that `merge` is an expensive operation if we do it literally at every node. If we do not merge at all, and make the A 's multi-sets instead of sets, we will end up with $u_0.A$ that contains as many elements as the

number of paths in E , so we are not better of either. Effort to merge is well spent if it delivers large reduction in the size of the resulting set, otherwise the effort is wasted. Unfortunately it is hard to predict the amount of reduction we would get for each particular merge. We use the following merge policy. We only merge on nodes at the $B - 1$ -th position of ‘bridges’ where B is the length of the bridge at hand. A bridge is a sequence of nodes v_0, \dots, v_{B-1} such that: (1) every v_i except the last one has only one outgoing edge, leading to v_{i+1} , and (2) the last node v_{B-1} should have more than one successor. A bridge forms thus a deterministic section of E , that leads to a non-deterministic section. Merging on a bridge is more likely to be cost effective. Furthermore, only one merge is needed for an entire bridge. Merging on a non-deterministic node (a node with multiple successors) is risky. This policy takes a conservative approach by not merging at all on such nodes. The next section will discuss the performance of our algorithm.

5 Experimental Results

In the following experiment we benchmark the algorithm from Sect. 4 against the ‘brute force’ way to calculate coverage using Definition 3. We will use a family of models M_m in Fig. 5. Despite its simplicity, M_m is highly non-deterministic and is designed to generate a large number of executions and words.

We generate a family of execution models $E(i, m)$ by applying a test case tc^i on the model M_m where $m \in \{0, 2, 8\}$. The test case is:

$$tc^i = ac^i ab^i ac^i a$$

The table in Fig. 6 (left) shows the statistics of all execution models used in this experiment. Additionally we also construct $E(i, m)^3$ (applying 3-word expansion). The last column in the table shows the number of nodes in the corresponding $E(i, m)^3$ (the number of executions stays the same, of course).

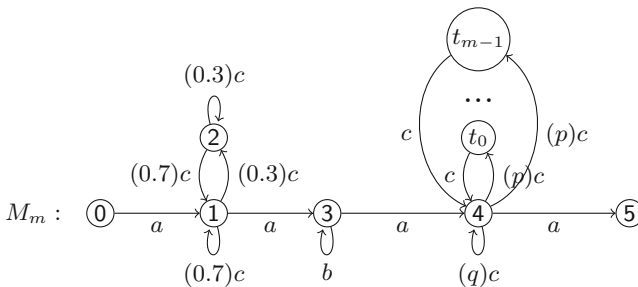


Fig. 5. The model M_m used for the benchmarking. If $m = 0$ then there is no states t_i and $q = 1$. If $m > 0$ then we have states $t_0 \dots t_{m-1}$; $p = 0.3/m$ and $q = 0.7$.

The number of possible executions in the execution models correspond to their degree of non-determinism. The test case tc^i has been designed as such that increasing i exponentially increases the non-determinism of the corresponding execution model (we can see this in Fig. 6 by comparing $\#paths$ with the i index of the corresponding $E(i, m)$).

All the models used (M_0 , M_2 , and M_8) are non-deterministic: M_0 is the least non-deterministic one whereas M_8 is very non-deterministic. This is reflected in the number of possible executions in their corresponding execution models, with $E(i, 8)$ having far more possible executions than $E(i, 0)$.

The following four coverage goals are used:

goal	type	word expansion
$f_1 : \langle 2 \rangle; \langle t_0 \rangle$	simple sentence	no
$f_2 : \langle 1, 1, 1 \rangle; \langle 4, 4, 4 \rangle$	non-simple sentence	3-word
$f_3 : \overset{1}{\geq}_8$	aggregate	no
$f_4 : \overset{3}{\geq}_8$	aggregate	3-word

We let our algorithm calculate the coverage of each of the above goals on the execution models $E(5, 0) \dots E(9, 8)$ and measure the time it takes to finish the calculation. For the merging policy, n is set to 1 when the goal does not need word expansion, and else it is set to be equal to the expansion parameter. The experiment is run on a Macbook Pro with 2,7 GHz Intel i5 and 8 GB RAM. Figure 6 (right) shows the results. For example, we can see that f_1 can be calculated in just a few milli seconds, even on $E(12, m)$ and $E(i, 8)$. In contrast, brute force calculation using Definition 3 on e.g. $E(11, 2)$, $E(12, 2)$, $E(8, 8)$, and

	$ tc $	$\#nodes$	$\#paths$	$\#nodes^3$		f_1	f_2	f_3	f_4
$E(5, 0)$	20	26	16	103(4)	$E(5, 0)$	0.001	0.002	0.001	0.002
$E(6, 0)$	23	30	32	144(5)	$E(6, 0)$	0.001	0.002	0.001	0.002
$E(7, 0)$	26	34	64	223(7)	$E(7, 0)$	0.001	0.003	0.001	0.003
$E(8, 0)$	29	38	128	381(10)	$E(8, 0)$	0.001	0.004	0.001	0.005
$E(9, 0)$	32	42	256	422(10)	$E(9, 0)$	0.001	0.005	0.002	0.006
$E(10, 0)$	35	46	512	501(11)	$E(10, 0)$	0.001	0.006	0.003	0.008
$E(11, 0)$	38	50	1024	659(13)	$E(11, 0)$	0.001	0.008	0.004	0.012
$E(12, 0)$	41	54	2048	700(13)	$E(12, 0)$	0.001	0.008	0.009	0.024
$E(5, 2)$	20	34	336	185(5)	$E(5, 2)$	0.001	0.002	0.002	0.004
$E(6, 2)$	23	40	1376	306(8)	$E(6, 2)$	0.001	0.004	0.002	0.01
$E(7, 2)$	26	46	5440	435(9)	$E(7, 2)$	0.001	0.005	0.003	0.039
$E(8, 2)$	29	52	21888	695(13)	$E(8, 2)$	0.001	0.01	0.005	0.138
$E(9, 2)$	32	58	87296	944(16)	$E(9, 2)$	0.001	0.014	0.01	0.44
$E(10, 2)$	35	64	349696	1073(17)	$E(10, 2)$	0.001	0.012	0.019	1.09
$E(11, 2)$	38	70	1397760	1333(19)	$E(11, 2)$	0.001	0.018	0.041	3.13
$E(12, 2)$	41	76	5593088	1582(21)	$E(12, 2)$	0.001	0.023	0.091	10.68
$E(5, 8)$	20	58	3600	863(15)	$E(5, 8)$	0.001	0.011	0.006	0.032
$E(6, 8)$	23	70	29984	2760(39)	$E(6, 8)$	0.001	0.04	0.034	0.279
$E(7, 8)$	26	82	175168	4287(52)	$E(7, 8)$	0.001	0.076	0.073	1.38
$E(8, 8)$	29	94	1309824	8261(88)	$E(8, 8)$	0.002	0.154	0.266	12.04
$E(9, 8)$	32	106	8225024	23726(224)	$E(9, 8)$	0.002	0.46	0.539	219

Fig. 6. Left: the execution models used in the benchmark. $\#nodes$ and $\#paths$ are the number of nodes and full paths (executions) in the corresponding execution model; $\#nodes^3$ is the number of nodes in the resulting 3-word expansion model. The number between brackets is $\#nodes^3 / \#nodes$. **Right:** the run time (seconds) of our coverage calculation algorithm on different execution models and coverage goals.

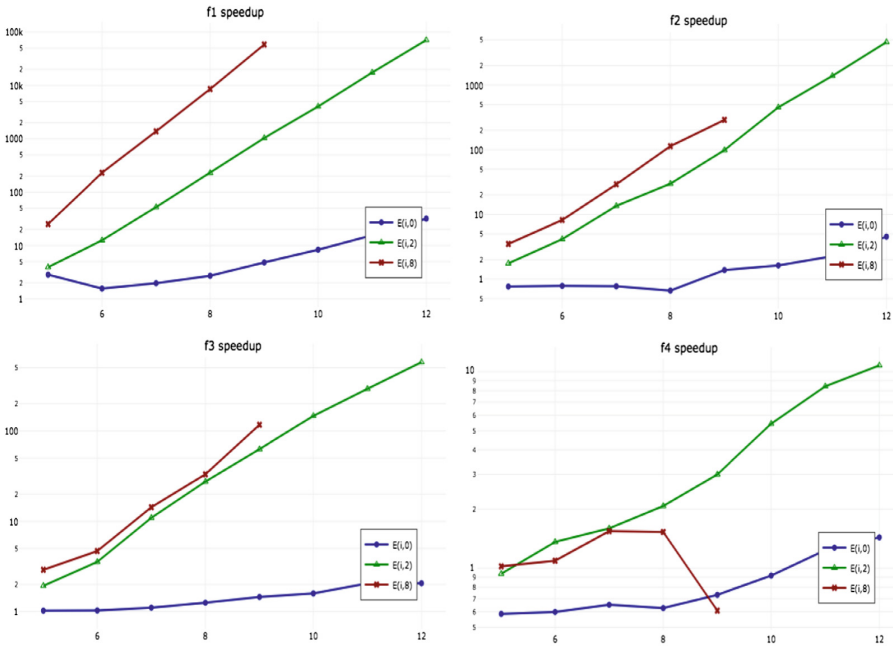


Fig. 7. The graphs show our algorithm’s speedup with respect to the brute force calculation on four different goals: f_1 (top left), f_2 (top right), f_3 (bottom left), and f_4 (bottom right). f_1 and f_2 are non-aggregate, whereas f_3 and f_4 are aggregate goals. Calculating f_1 and f_3 does not use word expansion, whereas f_2 and f_4 require 3-word expansion. Each graph shows the speedup with respect to three families of execution models: $E(i, 0)$, $E(i, 2)$, and $E(i, 8)$. These models have increasing degree of non-determinism, with models from $E(i, 8)$ being the most non-deterministic ones compared to the models from other families (with the same i). The horizontal axes represent the i parameter, which linearly influences the length of the used test case. The vertical axes show the speedup in the **logarithmic** scale. (Color figure online)

$E(9, 8)$ would be very expensive, because it has to quantify over more than a million paths in each of these models.

Figure 7 shows the speedup of our algorithm with respect to the brute force calculation—note that the graphs are set in logarithmic scale. We can see that in almost all cases the speedup grows exponentially with respect to the length of the test case, although the growth rate is different in different situations. We can notice that the speed up on $E(i, 0)$ is much lower (though we still have speedup, except for f_4 which we will discuss below). This is because $E(i, 0)$ ’s are not too non-deterministic. They all induce less than 2100 possible executions. The brute force approach can easily handle such volume. Despite the low speedup, on all $E(i, 0)$ ’s our algorithm can do the task in just few milli seconds (1–24 ms).

The calculation of f_1 is very fast (less than 2 ms). This is expected, because f_1 is a simple sentence. The calculation of f_2 , on the other hand, which is a

non-simple sentence, must be executed on the corresponding 3-word expanded execution model, which can be much larger than the original execution model. E.g. $E(9, 8)^3$ is over 200 times larger (in the number of nodes) than $E(9, 8)$. Despite this we see the algorithm performs pretty well on f_2 .

f_3 and f_4 are both aggregate goals. The calculation of f_3 is not problematical, however we see that f_4 becomes expensive on the models $E(12, 2)$, $E(8, 8)$, and $E(9, 8)$ (see Fig. 6 right). In fact, on $E(9, 8)$ the calculation of f_4 is even worse than brute force (the dip in the red line in Fig. 7). Recall that $f_4 = \binom{3}{\geq 8}$; so, calculating its coverage requires us to sum over different sets of words of size 3 that the different executions can generate. $E(12, 2)$, $E(8, 8)$, and $E(9, 8)$ are large (over 70 states) and highly non-deterministic. Inevitably, they generate a lot of words of size 3, and therefore the number of possible sets of these words explodes. E.g. on $E(8, 8)$ and $E(9, 8)$ our algorithm ends up with about 1.2M and 6.7M sets of words to sum over. In contrast, the number of full paths in these models are about respectively 1.3M and 8.2M. At this ratio, there is not much to gain with respect to the brute force approach that simply sums over all full paths, whereas our algorithm also has to deal with the overhead of book keeping and merging. Hypothetically, if we always merge, the number of final sets of words can be reduced to respectively about 500K and 2M, so summing over them would be faster. We should not do this though, because merging is expensive, but the numbers do suggest that there is room for improvement if one can figure out how to merge more smartly.

6 Related Work

To the best of our knowledge the concept of probabilistic coverage has not been well addressed in the literature on non-deterministic MBT, or even in the literature on probabilistic automata. A paper by Zu, Hall, and May [30] that provides a comprehensive discussion on various coverage criteria does not mention the concept either. This is a bit surprising since coverage is a concept that is quite central in software testing. We do find its mentioning in literature on statistical testing, e.g. [8, 29]. In [29] Whittaker and Thomason discussed the use of Markov chains to encode probabilistic behavioral models. The probabilities are used to model the usage pattern of the SUT. This allows us to generate test sequences whose distribution follows the usage pattern (so-called ‘statistical testing’). Techniques from Markov chain are then used to predict properties of the test sequences if we are to generate them in this way, e.g. the probability to obtain a certain level of node or edge coverage, or conversely the expected number of test runs needed to get that level of coverage. In contrast, in our work probabilities are used to model SUT’s non-determinism, rather than its usage pattern. We do not concern ourselves with how the tester generates the test sequences, and focuses purely on the calculation of coverage under the SUT’s non-determinism. Our coverage goal expressions are more general than [29] by allowing words of arbitrary length (rather than just words of length one or two, which would represent state and respectively edge coverage), clauses, and sentences to be specified as coverage

goals. Coverage calculation in both [8,29] basically comes down to the brute force calculation in Definition 3.

Our algorithm to calculate the coverage of simple sentences has some similarity with the probabilistic model checking algorithm for Probabilistic Computation Tree Logic (PCTL) [10,15]. Although given a formula f a model checking algorithm tries to decide whether or not f is valid on the given behavior model, the underlying probabilistic algorithm also labels for every state in the model with the probability that any execution that starts from that state would satisfy f . Since we only need to calculate over execution models, which are acyclic, there is no need to do a fixed point iteration as in [15]. From this perspective, our algorithm can be seen as an instance of [15]. However we also add k -word expansion. In addition to simplifying the algorithm when dealing with non-simple sentences, the expansion also serves as a form of memoisation (we do not have to keep calculating the probability for a state u to lead to a word w). In particular the calculation of aggregate coverage goals benefits from this memoisation. Though, the biggest difference between our approach with a model checking algorithm is that the latter does not deal with aggregate properties (there is no concept of aggregate formulas in PCTL). Our contribution can also be seen as opening a way to extend a probabilistic model checking algorithm to calculate such properties. We believe it is also possible to generalize over the aggregation so that the same algorithm can be used to aggregate arbitrary state attributes that admit some aggregation operator (e.g. the cost of staying in various states, which can be aggregated with the ‘+’ operator).

In this paper we have focused on coverage analyses. There are other analyses that are useful to mention. In this paper we abstract away from the data that may have been exchanged during the interactions with the SUT. In practice many systems do exchange data. In this situation we may also want to do data-related analyses as well. E.g. the work by Prasetya [19] discussed the use of an extended LTL to query temporal relations between the data exchanged through the test sequences in a test suite. This is useful e.g. to find test sequences of a specific property, or to check if a certain temporal scenario has been covered. The setup is non-probabilistic though (a query can only tell whether a temporal property holds or not), so an extension would be needed if we are interested in probabilistic judgement. Another example of analyses is risk analyses as in the work by Stoelinga and Timmer [23]. When testing a non-deterministic system, we need to keep in mind that although executing a test suite may report no error, there might still be lurking errors that were not triggered due to internal non-determinism. Stoelinga and Timmer propose to annotate each transition in a model with the estimated probability that it is incorrectly implemented and the entailed cost if the incorrect behavior emerges¹. This then allows us to calculate

¹ We gloss over the complication that the transition might be in a cycle. A test case may thus exercise it multiple times. Each time, exercising it successfully would arguably decrease the probability that it still hides some hidden erroneous behavior. This requires a more elaborate treatment, see [23] for more details.

the probability that a successful execution of a test suite still hides errors, and the expected cost (risk) of these hidden errors.

7 Conclusion

We have presented a concept of probabilistic coverage that is useful to express the coverage of a test suite in model-based testing when the used model is non-deterministic, but has been annotated with estimation on the probability of each non-deterministic choice. Both aggregate and non-aggregate coverage goals can be expressed, and we have presented an algorithm to efficiently calculate the probabilistic coverage of such goals. Quite sophisticated coverage goals can be expressed, e.g. sequence (words) coverage and sequence of sequences (sentences) coverage. We have shown that in most cases the algorithm is very efficient. A challenge still lies on calculating aggregate k -wise test goals on test cases that repeatedly trigger highly non-deterministic parts of the model. Such a situation is bound to generate combinatoric explosion on the possible combinations of words that need to be taken into account. Beyond a certain point, the explosion becomes too much for the merging policy used in our algorithm to handle. Analyses on the data obtained from our benchmarking suggests that in theory there is indeed room for improvement, though it is not yet clear what the best course to proceed. This is left for future work.

References

1. Ammann, P., Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, Cambridge (2016)
2. Arnold, A.: *Finite Transition Systems*. International Series in Computer Science (1994)
3. Baier, C., Katoen, J.P., Larsen, K.G.: *Principles of Model Checking*. MIT Press, Cambridge (2008)
4. Belinfante, A.: *JTorX: exploring model-based testing*. Ph.D. thesis, University of Twente (2014)
5. Bringmann, E., Krämer, A.: Model-based testing of automotive systems. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 485–493. IEEE (2008)
6. Craggs, I., Sardis, M., Heuillard, T.: AGEDIS case studies: model-based testing in industry. In: *Proceedings of 1st European Conference on Model Driven Software Engineering*, pp. 129–132 (2003)
7. Dallmeier, V., Lindig, C., Wasykowski, A., Zeller, A.: Mining object behavior with ADABU. In: *Proceedings of the International Workshop on Dynamic Systems Analysis (WODA)*, pp. 17–24. ACM (2006). <https://doi.org/10.1145/1138912.1138918>
8. Denise, A., Gaudel, M.C., Gouraud, S.D.: A generic method for statistical testing. In: 15th International Symposium on Software Reliability Engineering ISSRE, pp. 25–34. IEEE (2004)
9. Grindal, M., Offutt, J., Andler, S.F.: Combination testing strategies: a survey. *Softw. Test. Verif. Reliab.* **15**(3), 167–199 (2005)

10. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects Comput.* **6**(5), 512–535 (1994)
11. Heerink, L., Feenstra, J., Tretmans, J.: Formal test automation: the conference protocol with phact. In: Ural, H., Probert, R.L., v. Bochmann, G. (eds.) *Testing of Communicating Systems. IAICT*, vol. 48, pp. 211–220. Springer, Boston, MA (2000). https://doi.org/10.1007/978-0-387-35516-0_13
12. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River (2004)
13. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. *Int. J. Softw. Tools Technol. Transf.* **7**(4), 297–315 (2005)
14. Kanstrén, T., Puolitaival, O.P.: Using Built-in Domain-Specific Modeling Support to Guide Model-Based Test Generation. *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*, pp. 295–319 (2012)
15. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) *SFM 2007. LNCS*, vol. 4486, pp. 220–270. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72522-0_6
16. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. In: *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 55–64. ACM (2004)
17. Peleska, J.: Industrial-strength model-based testing - state of the art and current challenges. In: *Proceedings 8th Workshop on Model-Based Testing (MBT)*, pp. 3–28 (2013). <https://doi.org/10.4204/EPTCS.111.1>
18. Petke, J., Cohen, M.B., Harman, M., Yoo, S.: Practical combinatorial interaction testing: empirical findings on efficiency and early fault detection. *IEEE Trans. Softw. Eng.* **41**(9), 901–924 (2015)
19. Prasetya, I.: Temporal algebraic query of test sequences. *J. Syst. Softw.* **136**, 223–236 (2018)
20. Pretschner, A., Philipps, J.: 10 methodological issues in model-based testing. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) *Model-Based Testing of Reactive Systems. LNCS*, vol. 3472, pp. 281–291. Springer, Heidelberg (2005). https://doi.org/10.1007/11498490_13
21. Schur, M., Roth, A., Zeller, A.: Mining behavior models from enterprise web applications. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pp. 422–432. ACM (2013). <https://doi.org/10.1145/2491411.2491426>
22. Stoelinga, M.: An introduction to probabilistic automata. *Bull. EATCS* **78**(2), 176–198 (2002)
23. Stoelinga, M., Timmer, M.: Interpreting a successful testing process: risk and actual coverage. In: *3rd International Symposium on Theoretical Aspects of Software Engineering TASE*, pp. 251–258. IEEE (2009)
24. Tervoort, T., Prasetya, I.S.W.B.: APSL: a light weight testing tool for protocols with complex messages. *Hardware and Software: Verification and Testing. LNCS*, vol. 10629, pp. 241–244. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_20
25. Tretmans, G.J.: A formal approach to conformance testing. Ph.D. thesis, Twente University (1992)
26. Tretmans, J., Brinksma, E.: TorX: automated model-based testing. In: *1st European Conference on Model-Driven Software Engineering* (2003)
27. Utting, M., Pretschner, A., Legiard, B.: A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.* **22**(5), 297–312 (2012)

28. Vos, T., et al.: Fittest: a new continuous and automated testing process for future internet applications. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), pp. 407–410. IEEE (2014)
29. Whittaker, J.A., Thomason, M.G.: A Markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.* **20**(10), 812–824 (1994)
30. Zhu, H., Hall, P.A., May, J.H.: Software unit test coverage and adequacy. *ACM Comput. Surv.* **29**(4), 366–427 (1997)