Test Sequence Generation with Cayley Graphs

Sylvain Hallé, Raphaël Khoury

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

***Abstract*—The paper presents a theoretical foundation for test sequence generation based on an input specification. The set of possible test sequences is first partitioned according to a generic "triaging" function, which can be created from a state-machine specification in various ways. The notion of coverage metric is then expressed in terms of the categories produced by this function. Many existing test generation problems, such as t-way state or transition coverage, become particular cases of this generic framework. We then present algorithms for generating sets of test sequences providing guaranteed full coverage with respect to a metric, by building and processing a special type of graph called a *Cayley graph*. An implementation of these concepts is then experimentally evaluated against existing techniques, and shows it provides better performance in terms of running time and test suite size.**

I. Introduction

Specification-based testing is the process of generating sets of inputs to be sent to a system under test (SUT), with the goal of checking that the SUT fulfills the specification [32]. As with any form of software testing, it is intrinsically an incomplete technique that cannot provide any formal compliance guarantee; it is rather aimed at helping developers find *defects* —specific inputs for which the specification is violated, and which can then be used to fix the system's implementation. Despite this incompleteness, specification-based testing has proved its bug-finding capability in numerous use cases over the years [2], [15].

A large part of existing literature on specification-based testing focuses on what we call *static* systems, where each test consists of a single input given to the SUT, which in turn produces a single output that is then evaluated by some test oracle. Another line of work focuses on specification-based testing of *reactive* systems, whose interaction is often done through multiple "calls" or "requests". In such a case, one is interested in generating not single inputs, but *sequences* of actions. In this paper, we shall focus on one specific type of specification, expressed in the form of a labeled finite-state automaton where vertices represent observable states, and edge labels represent actions to be performed on the system.

Intuitively, the principle of all specification-based generation methods can be summarized as the search for a set of inputs that "covers" the specification. For example, on a system specified as a list of simple if-then rules, an input generation technique could produce one test representative of each "if" case, in order to make sure that the system behaves as expected in each of the stipulated eventualities. This intuitive notion of coverage can be concretely defined in multiple ways, depending on the specification and the formal notation used to express it. In the specific case of finite-state automata, state and transition coverage are two notions that naturally spring to mind, but a quick survey of literature reveals there are many others: t-way transition coverage, state residual coverage, etc.

It could be interesting, from a theoretical standpoint, to ask whether there exists a uniform way of generating a set of sequences that satisfy any of these coverage criteria. In the current state of things, existing studies provide *ad hoc* algorithms to generate sequences for a single one of these coverage metrics; hence, an algorithm developed for state coverage cannot be used for transition coverage. In almost all cases, these works provide no formal guarantee that full coverage is always achieved. What is more, many notions of coverage mentioned above still lack a systematic algorithm to generate sequences according to them.

This, in a nutshell, is the issue addressed by the present paper. More precisely, we describe a systematic technique to efficiently generate test sequences based on a reactive specification expressed as a labeled finite-state automaton, according to an arbitrary coverage criterion. In Section II, we formally introduce the problem of specification-based test sequence generation, and through numerous examples, discuss a variety of coverage criteria expressed on labeled finite-state automata taken from existing literature. We also present an overview of related works on the subject.

In Section III, we introduce a generic theoretical framework for test sequence generation based on the notion of *triaging function*. A triaging function associates every sequence to an abstract object called a category; it can be seen as a way of classifying all possible sequences in a specific way. The problem of coverage can then be reduced to the problem of generating one sequence belonging to each category. We show how the many coverage criteria introduced earlier can actually be expressed as specific cases of triaging functions, and how other criteria, unexplored in past literature, can also be taken into account.

In Section IV, we then present an algorithm for generating a set of sequences providing complete coverage with respect to a given criterion, by constructing a structure borrowed from abstract algebra called a Cayley graph. The problem is solved by reduction to the Directed Steiner Tree problem on the Cayley graph induced by a given finite-state automaton and triaging function. The construction is generic, and can apply for any

coverage criterion that satisfies the conditions presented in the paper.

Finally, in Section V, we describe an implementation of these concepts, by extending an existing open source test sequence generation library. We provide a comparison and analysis of our proposed approach on 33 problem instances taken from existing sources. These experiments show that, despite solving a problem using a generic algorithm, our proposed approach outperforms existing state-of-the-art sequence generation tools, in terms of running time, coverage obtained, and test suite size.

II. Specification-Based Test Sequence Generation

We shall first formally define the problem of specification-based test sequence generation, and then give examples of coverage criteria on these specifications that have been presented in past literature. We shall end by a review of existing works that focus on this particular problem and related notions.

A. Specification of Reactive Systems

Let Q be an abstract set of *states*, and A be a set of *actions*. A reactive system specification is a tuple $\mathcal{M} = \langle Q, A, q_0, \delta, \omega \rangle$, where $\delta : Q \times A \to Q$ is a transition function, $q_0 \in Q$ is the unique initial state, and $\omega : Q \to \{\top, \bot, ?\}$ is an oracle function. The oracle function associates each state of the specification to a ternary Boolean verdict that can either be true ($\top$), false ($\bot$), or inconclusive (?). Intuitively, elements of A represent positive and controllable actions that can be done by a user on a system, while elements of Q represent the observable states of the system resulting from these actions. By convention, if δ is not total, we shall assume the presence of an implicit sink state $q_\bot \in Q$ such that $\omega(q_\bot) = \bot$, and assume that $\delta(q, a) = q_\bot$ for all pairs (q, a) not defined originally in δ. In such a way, any transition left undefined by the specification is assumed to be a failure. Since this paper is only concerned with generating sequences, the actual definition of ω will not matter.[1]

A *test sequence* is a finite sequence of alternating states and actions $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} q_n$ such that for every $i > 0$, $q_i = \delta(q_{i-1}, a_{i-1})$. We shall equate such a sequence with a word over $\Sigma \subseteq (Q \times A \times Q)^*$, where Σ contains all finite sequences such that the i-th element element is the triplet of the form (q_{i-1}, a_{i-1}, q_i). The execution of a test sequence consists of performing the sequence of actions $a_0, \ldots, a_n$ over a system under test (SUT) and observing the resulting states. A test sequence is said to be *passing* if all observed system states correspond to the expected ones, and ends in a state q such that $\omega(q) = \top$. The sequence is said to be failing if some observed state of the SUT does not correspond to the expected state in the sequence, or if it ends in a state q such that $\omega(q) = \bot$. In all other cases, the test sequence is said to be inconclusive.

Despite being simple, this model of reactive specifications is nevertheless very general, as expressions in many other formal notations can be transformed into equivalent finite-state automata of this form; this includes, among others, Linear Temporal Logic on finite prefixes and UML state diagrams. Moreover, states and actions, although represented by atomic symbols, can actually model arbitrary finite structures with the proper amount of syntactical sugar. For example, with few adaptations, such a model can be used to represent navigation state machines [17], where states represent web pages and actions represent clicks on specific elements of a page. Simple objects such as a wristwatch or a vending machine have options that can be accessed by pressing various buttons, and hence be modeled in a similar way [20].

[1]Other works on model-based testing specify reactive systems as Mealy machines where the SUT produces an output on each transition (e.g. [38]), whereas our model is actually a Moore machine. This distinction is irrelevant since both models are equivalent in terms of expressiveness.

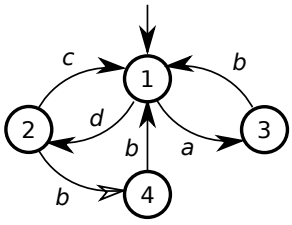


Figure 1: A simple reactive specification.

We call *specification-based test sequence generation* any operation which, from the set of all possible test sequences Σ, picks a subset $\Sigma_T \subseteq \Sigma$ called the *test suite*. Typically, test sequence generation intends to choose elements of Σ by taking advantage that the specification is known, and selects test sequences that are representative of various "cases" described by that specification. This is what we call *coverage-based* test sequence generation. It has been argued that such a technique has the potential of finding classes of faults, such as missing special cases, that are difficult to find by only considering the implementation of a system [21].

B. Coverage Criteria

In the present context, a coverage criterion is a condition on the set of sequences that is used to decide whether this set sufficiently reveals the underlying specification. To illustrate the various coverage criteria, we shall use the reactive specification shown in Figure 1, where $Q = \{1, 2, 3, 4\}$ and $A = \{a, b, c, d\}$. To avoid clutter, we use the notation conventions defined above and do not represent the implicit sink state $q_\bot$ and the transitions leading to it. We assume all other states are success states.

1) State Coverage: A first natural possibility is to declare a set of sequences as complete if the sequences it contains, taken together, are such that each *state* of the specification is visited at least once. In this example, the set $S_{s,1} = \{a, db\}$ satisfies this criterion, and so does the set $S_{s,2} = \{abdb\}$. These two sets, however, generate only valid sequences according to our specification. Since testing also involves generating error conditions, state coverage would then require that the implicit sink $q_\bot$ is also visited at least once, leading to solutions such as the set $S_{s,3} = \{aa, db\}$.

2) Action Coverage: There exist situations where visiting each state of the specification may not be an appropriate measure of coverage. Note that in the previous example, none of the three sets shown execute action c. A second coverage

criterion, called *action* coverage, stipulates that each action label be present in at least one sequence. Here, the set $S_{a,1} = \{ab, dc\}$ satisfies this criterion.

3) Transition Coverage: A stronger coverage criterion is called *transition* coverage; as its name implies it requires that each transition be taken at least once [30]. For example, $2 \xrightarrow{b} 4$ is a different transition than $3 \xrightarrow{b} 1$ even though they have the same action label. In general, one cannot generate a set of sequences for transition coverage by simply taking the union of a set achieving state coverage with a set achieving action coverage.

4) Residual Coverage: Suppose that the system under test makes a difference between the last state of a sequence and an intermediate state. In this context, one would like not only to visit every state, but to produce sequences that *end* on each state. This is called *state residual* coverage [11]. A set like $S_{sr,1} = \{\epsilon, a, d, db\}$ satisfies this criterion. Note how this example introduces the use of ϵ, which represents the empty sequence of actions. Intuitively, this would amount to starting the test procedure, performing no action, and ending the test procedure.

The same notion of residual coverage can be translated to actions and transitions, although this has seldom been done in the literature. Action residual coverage requires a set of sequences such that each action is the last element of at least one sequence; for example, $S_{ar,1} = \{a, ab, d, dc\}$. Transition residual coverage requires that sequences end in each possible transition; a set like $S_{tr,1} = \{a, ab, d, dc, db, dbb\}$ satisfies this criterion.

5) Combinatorial Coverage: Covering actions, states or transitions exactly once may not be a strong enough guarantee that the SUT complies with the specification. One possibility is to increase the *strength* of the tests —for example, by generating sequences such that every possible sequence of t successive states is visited at least once. One can see that the state coverage criterion presented above is simply the particular case where $t = 1$. As an example, a possible set of sequences for 3-way state coverage (i.e. $t = 3$) is $S_{ts,1} = \{aba, abdc, dbbd, dbba\}$; these sequences indeed cover the 8 sequences of three states that are actually possible in the specification.

As for the other coverage metrics seen so far, t-way coverage can also be considered for actions; for example, 2-way action coverage requires to exercise every possible sequence of two successive actions (a solution being $S_{ta,1} = \{aba, cdc, abd, dbbd\}$). Finally, t-way transition coverage applies the same condition on transitions instead of actions. For $t = 2$, this criterion has been called *transition-pair coverage* by Offutt *et al.* [30].

C. State of the Art in Test Sequence Generation

From a test case generation perspective, many works relate to the problem studied in this paper. They can be divided into a few broad categories, which we discuss in the following.

1) Static and Combinatorial Test Sequence Generation: A first broadly related line of work is test case generation for the "static" case. In this context, the goal is to generate inputs (i.e. parameter values) to be fed to a function or a system under test. We call this process *static*, in the sense that each test instance consists of a single assignment of values to each parameter. Nevertheless, static test input generation shares some similarities with the present problem. For example, in equivalence class partitioning (ECP), also called category partition testing [34], the principle is to partition the space of values for each input parameter into a number of equivalence classes, and to generate tests so that all combinations of equivalence classes for each input appears in some test. ECP is particularly useful when testing inputs on infinite domains, as these domains can be abstracted by a finite number of classes. We shall see that one of the contributions of this paper is precisely to reframe test sequence generation as a form of ECP.

The t-way test sequence generation problem was introduced as a dynamic counterpart to static combinatorial testing. Given a set of actions A, the classical problem is defined as the generation of a set of traces where each action occurs exactly once in each trace, such that all sequences of t actions occur somewhere in one of the traces. Kuhn *et al.* [26, Chapter 10] describe a greedy algorithm which generates a large number of tests, scores each by the number of previously uncovered sequences it covers, and selects the highest scoring test. While this technique generates sequences of actions, it is not based on a specification; we shall see in the experiments detailed in Section V that it generates a disproportionate number of invalid sequences and struggles to achieve full coverage according to any metric.

2) Conformance Testing: The generation of test sequences falls more closely into the field of model-based testing for so-called *reactive* systems [3]. In this field, *conformance testing* has led to a very large body of works. The problem consists of discovering whether an unknown implementation behaves in the same way as a specification given as a finite-state machine with inputs and outputs, i.e. that the same input sequences produce the same output sequences in both the SUT and the specification [10], [28]. Methods for testing in such a context have been reviewed by Ural [37] and Dorofeeva *et al.* [12]; they include the W, HSI, H, SPY, UIOv and P methods [7], [16], [31].

It shall be noted that these techniques operate under much stronger constraints than the problem addressed in this paper. They assume the internal state of the SUT is not observable, and can only be partially deduced by the outputs it produces. As a matter of fact, the central task of most of the testing methods enumerated above is to traverse the specification in various ways in order to progressively discover the states of the SUT, and to ensure that every state of the specification exists in the implementation. This is done through the progressive construction of so-called "synchronizing", "homing" or "distinguishing" sequences [27]. For this reason, the sequences generated by these methods are typically long, as guessing the internal state can often only be done through convoluted traversals of several input-output pairs and frequent returns to a known state. In counterpart, they also provide much stronger guarantees; typically, if the specification can be modeled as an FSM with

n states, and the (unknown) implementation has at most $m \geq n$ states, they can derive a test suite such that the implementation passes this test suite if and only if it conforms (i.e. is equivalent) to the specification. A more detailed discussion of the difference between test sequence generation and conformance testing can be found in [38, §5.1.4].

3) Specification-Based Test Sequence Generation: We end this section by discussing the works that tackle the same problem as ours, which is to generate sequences that cover a finite-state machine specification. Despite being considered one of the simplest forms of model-based testing, the generation of test sequences from a finite-state machine specification is often handled through *ad hoc* means, and has been the subject of few formal studies. Li *et al.* describe an algorithm to solve the Minimum Cost Test Paths Problem (MCTP), which consists of finding a set of sequences (here called "test paths") that cover a given set of sub-paths in a graph, using a greedy set covering algorithm [29]; it reports on experiments but the implementation is not made available.

Chander *et al.* consider a specific automaton and generate a set of sequences such that all sequences of t states present in the specification occur somewhere in the set [5]. The authors describe how this can be formulated, and solved optimally, as an integer linear programming (ILP) problem. However, the approach is only illustrated on a few examples for the single case $t = 1$.

Kim *et al.* [24] use model checking to generate test sequences from UML statecharts. In this work, each coverage criterion is expressed as a CTL formula; performing model checking of the statechart against that formula produces a counter-example trace which can then be used as a test sequence. The writing of the statecharts into NuSMV input files, and the expression of the coverage criteria as CTL formulæ, however, seem to be done by hand. Moreover, for criteria such as state coverage, one CTL formula per state needs to be written, each producing one distinct trace. The fact that a single trace visits more than one state is not exploited, leading to an unnecessarily large number of sequences.

Generating test sequences from a specification using multi-agent systems has been suggested by Kruse [25]. Srivastava *et al.* use ant colony optimization to generate test sequence that achieve either state coverage or transition coverage of a given FSM [33]. In this latter case, probabilities are associated to each transition, making the model a Markov chain. The test generation problem takes into account these probabilities when generating the test sequences. In both works, however, the presented algorithm only supports t-way state and transition coverage for $t = 1$, and guarantee neither full coverage nor the optimality of the solution.

Ferrer *et al.* [14] study search-based approaches and propose two algorithms. First, the Genetic Test Sequence Generator (GTSG) constructs an entire test suite by evolving a population of solutions in each iteration until a given coverage criterion is fulfilled. The algorithm tries to find the tests that maximize the coverage, then it sequentially adds them to the solution. Second, the ACO Test Sequence (ACOts) algorithm is an adaptation of an ant colony algorithm that can deal with the construction of large graphs of unknown size. Experimental analysis claims that these two search-based approaches are better than the greedy deterministic approach, especially in the most complex instances. However, the paper provides no publicly available implementation and no benchmark dataset.

Swain *et al.* convert a statechart into an intermediate representation called a State-Activity Diagram (SAD), which can take into account the lifecycle of multiple interacting objects [35]. Kansomkeat and Rivepiboon [22] transform a UML statechart into a flattened structure called a Testing Flow Graph (TFG), which is then traversed to generate test cases. Again, in these two works, the coverage metrics supported are t-way state and transition coverage for the single case $t = 1$. Unfortunately, no implementation of these techniques is provided in their respective papers.

A sequence generation library called SealTest [18] implements a pseudo-random greedy algorithm to generate sequences according to a coverage metric, based on the random greedy algorithm described by [26]. Analysis of the source code shows that this algorithm generates k random walks of a randomly selected length inside the specification, and adds to its test suite the sequence that produces the largest increase in coverage; if no sequence increases the coverage, it regenerates a new set of k candidates. The process repeats for a maximum number of iterations n. Each random walk favors transitions that have not yet been explored when possible. The values of k and n must be appropriately guessed by the user, and the process does not guarantee that full coverage can be achieved. GraphWalker[2] is an industrial-grade tool that operates in a similar manner, by performing walks inside the finite-state machine specification and favoring transitions that increase coverage.

III. An Abstract Definition of Coverage

As one can see, there are relatively few works that focus on the specification-based sequence generation problem, as defined in this paper. In addition, although these works display a wide variety of techniques for sequence generation, they all focus on a narrow set of coverage metrics (t-way state and transition coverage, with $t = 1$ for most of them); our discussion in Section II-B has shown that many other coverage criteria could be considered, and for which the presented algorithms cannot be applied. A single of these works provides a construction that guarantees the generated test sequences achieve full coverage with respect to the criterion it considers. To address this issue, we start by providing the basis of a generic formal framework for describing coverage criteria. These criteria will be expressed as *triaging functions*, which will classify test sequences extracted from a reactive specification into abstract "categories". This generic definition will make it possible to model a large number of coverage criteria into a uniform formal model.

[2]https://graphwalker.org

A. Triaging Functions and Coverage

We recall from Section II-A that any test sequence can be equated to a list of triplets $(q_0, a_0, q_1), (q_1, a_1, q_2), \ldots$, where q_0 is the initial state of the specification, and $q_i = \delta(q_{i-1}, a_{i-1})$ for every $i > 0$; note how, with the exception of q_0, each state is the last element of a triplet and the first element of the next one. Let Σ be the set of all possible runs with respect to a reactive specification. In the following, we shall denote by $\overline{\sigma} \in \Sigma$ a test sequence, and by $\overline{\sigma}[i]$ the i-th triplet of the sequence.

Let C be a set of elements called *categories*. A triaging function is a function $\kappa : \Sigma \to C$, which associates each test sequence to a particular category. Without loss of generality, we assume that κ is surjective: each category has at least one sequence associated to it. A triaging function κ can be seen as a categorization function over sequences, with respect to a given specification. We consider the equivalence relation $\sim_\kappa$ lifted from κ as $\overline{\sigma} \sim_\kappa \overline{\sigma}'$ if and only if $\kappa(\overline{\sigma}) = \kappa(\overline{\sigma}')$. The *kernel* of κ is the partition of Σ induced by the quotient $\Sigma/\sim_\kappa$; each subset contains all traces that are categorized in the same way by κ. The subscript κ is omitted when clear from context.

In some cases, one may wish to associate each test sequence with all the categories given to its prefixes; this is what we call the *prefix closure*. Formally, the prefix closure of a triaging function $\kappa : \Sigma \to C$ is the triaging function $\kappa' : \Sigma \to 2^C$, such that:

$$\kappa'(\overline{\sigma}) \triangleq \bigcup_{\overline{\sigma}' \preccurlyeq \overline{\sigma}} \left\{\kappa(\overline{\sigma}')\right\}$$

where the notation $\overline{\sigma}' \preccurlyeq \overline{\sigma}$ indicates that $\overline{\sigma}'$ is a prefix of $\overline{\sigma}$. We shall note by $\int\kappa$ the prefix closure of κ; intuitively, this function accumulates into a set the categories of all prefixes of a sequence given by κ. For example, if $C = \mathbb{N}$, and if $\kappa(a) = 0$, $\kappa(ab) = 1$ and $\kappa(abc) = 2$, then $\int\kappa(abc) = \{0, 1, 2\}$. When a triaging function is defined as the prefix closure of some other triaging function, the sets of categories it returns will be called *families*, i.e. a family is a set of categories of some other triaging function.

For a triaging function κ and a set of sequences $\Sigma_T \subseteq \Sigma$, one can say that a category $c \in C$ is "covered" if there exists a sequence $\overline{\sigma} \in \Sigma_T$ such that $c = \kappa(\overline{\sigma})$; this is what we call *category coverage*. For a prefix closure $\int\kappa$, we say that c is covered if there exists a sequence $\overline{\sigma} \in \Sigma_T$ such that $c \in \int\kappa(\overline{\sigma})$; this is what we call *family coverage*. We shall use the notation $c \sqsubset_\kappa \Sigma_T$ and $c \sqsubset_{\int\kappa} \Sigma_T$ to indicate that c is (category- or family-) covered by the set Σ_T. Again, we shall omit the subscript when the triaging function is clear in the context.

This notion of coverage for single sequences can be used to associate a numerical value to a set of sequences. For a given triaging function (either κ or $\int\kappa$), the coverage ratio of a test suite Σ_T, noted $\rho(\Sigma_T)$ is defined as:

$$\rho(\Sigma_T) \triangleq \frac{|\{c \in C : c \sqsubset \Sigma_T\}|}{|C|}$$

Intuitively, $\rho(\Sigma_T)$ is a value between 0 and 1 that corresponds to the proportion of categories covered by Σ_T with respect to the total number of categories one can cover using all possible test sequences. A coverage of 1 indicates that each category is "represented" somewhere in the test suite, and that the specification has been fully "covered" by it.

B. Coverage Criteria Revisited

We can now return to the coverage criteria described in Section II-B and provide a formal definition for each of them using the concepts we just described.

State residual coverage can be defined as a triaging function $\kappa_{sr} : \Sigma \to Q$, such that $\kappa_{sr}((q_0, a_1, q_1) \cdots (q_{n-1}, a_{n-1}, q_n)) \triangleq q_n$. In this case, the set of categories is the set of states Q of the reactive specification, and the function associates a sequence to the state of the specification it ends at. It is straightforward to see that a set Σ' that covers all categories produced by κ_{sr} if it contains one test sequence that ends in each state of the specification, which indeed corresponds to the state residual coverage criterion.

Classical state coverage, noted κ_s, is nothing but the prefix closure of state residual coverage, i.e. $\kappa_s \triangleq \int\kappa_{sr}$. This is in line with the intuition that a state q of the specification is considered covered as long as it has been visited at some point by some test sequence $\overline{\sigma}$ —in other words, that a prefix of $\overline{\sigma}$ ends in q. In this case, the function ρ, applied on a set of sequences, computes the fraction of all states of the reactive specification that are visited by the tests.

Coverage criteria on actions and transitions can easily be defined using the same pattern. A function $\kappa_{ar} : \Sigma \to A$, such that $\kappa_{sr}((q_0, a_1, q_1) \cdots (q_{n-1}, a_{n-1}, q_n)) \triangleq a_{n-1}$ associates a sequence with the last action it contains; this corresponds to the action residual coverage criterion. Action coverage, noted κ_a, is the prefix closure of action residual coverage, i.e. $\kappa_a \triangleq \int\kappa_{ar}$; according to this criterion, an action is covered if it appears somewhere in one of the test sequences. This time, ρ calculates the fraction of actions that are exercised by the set of test sequences. Finally, a function $\kappa_{tr} : \Sigma \to (Q \times A \times Q)$, such that $\kappa_{sr}((q_0, a_1, q_1) \cdots (q_{n-1}, a_{n-1}, q_n)) \triangleq (q_{n-1}, a_{n-1}, q_n)$ associates a sequence with the last transition it contains, and corresponds to the transition residual coverage criterion. Again, transition coverage, noted κ_t, is the prefix closure of transition residual coverage.

We shall now turn to combinatorial, or so-called "t-way" coverage criteria, and show how they can be expressed as triaging functions as well. We describe the construction only for t-way state coverage; by now the reader should have grasped how similar criteria for actions and transitions can likewise be defined. We start with the t-way state residual triaging function. Let $\kappa_{t,sr} : \Sigma \to \prod_{i=0}^{t} Q^i$ be a function that produces a list of at most t states from a test sequence, defined as follows:

$$\kappa_{t,sr}((q_0, a_1, q_1) \cdots (q_{n-1}, a_{n-1}, q_n)) \triangleq \begin{cases} \langle q_0, \ldots, q_n \rangle \text{ if } n < t \\ \langle q_{n-t}, q_{n-t-1}, \ldots, q_n \rangle \\ \qquad \text{if } n \geq t \end{cases}$$

Intuitively, $\kappa_{t,sr}$ returns the ordered list of the last t visited states in the sequence, or a shorter list if the sequence has

a length smaller than t. Note that this definition is such that the empty trace, ϵ, is assumed to visit the initial state of the specification, q_0. This fulfills our definition of a triaging function, and corresponds to t-way state residual coverage. The t-way state coverage criterion is, again, its prefix closure: $\kappa_{t,s} \triangleq \int \kappa_{t,sr}$. One can see that, in this case, the family associated to a sequence $\overline{\sigma}$ is the set of all t-tuples of successive states that have been visited by the sequence.[3] For example, with $t = 2$, the sequence *dca* will result in the family $\{\langle 1\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 1,3\rangle\}$.

Other triaging functions must be omitted due to lack of space. However, it should be clear from the examples above that our proposed framework provides a generic and very flexible notation to describe triaging functions of various kinds, including and extending those already studied in past literature on specification-based test sequence generation.

IV. Sequence Generation with Cayley Graphs

In this section, we provide a procedure that solves the test sequence generation problem in the general case. The procedure can be divided into three main steps: generating a Cayley graph G from a reactive specification and a triaging function, computing a set of important vertices V_I, and finding a Steiner tree of G that reaches all vertices of V_I.

A. Generating Cayley Graphs

The notion of equivalence classes on sequences is closely linked to an abstract structure called a *Cayley graph*.

Definition 1. *Let κ be a triaging function $\kappa : \Sigma \to C$ for some reactive specification $\mathcal{M}$. Let $G_\kappa = \langle Q, q_0, \delta, \ell\rangle$ be a deterministic Moore machine with Q, q_0 and δ defined as usual, and $\ell : Q \to C$ a state labeling function, associating each state to a category in C. G_κ is called a* Cayley graph *of κ if for every $\overline{\sigma} \in \Sigma$, $\ell(\delta(q_0, \overline{\sigma})) = \kappa(\overline{\sigma})$.*

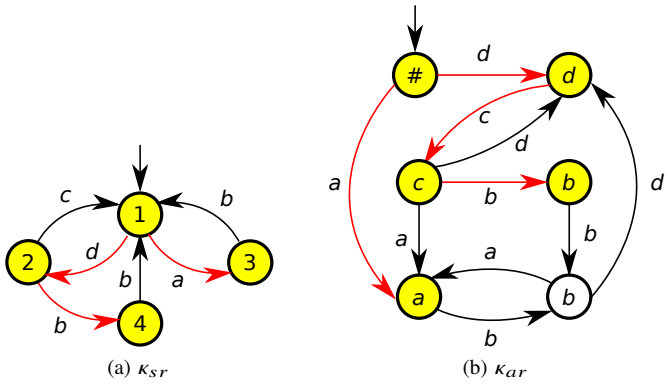
For example, Figure 2 shows Cayley graphs for a few of the triaging functions we defined in the previous section. As one can see, each state of the graph is labeled with one of the categories. One can also observe that, for any path taken in this graph, the label of the state the path ends with indeed corresponds to the category that is associated to this path by the triaging function.

Cayley graphs were first introduced in the study of algebraic data structures [4]. While our definition allows more than one vertex to be associated to a category $c \in C_\kappa$, the original definition of Cayley graph imposes that ℓ be injective. When the context requires it, we will use the term "classical" to denote this restricted type of graph.

Algorithm 1 generates a Cayley Graph from a triaging function κ and a reactive specification $\mathcal{M}$. The algorithm actually builds a graph whose vertices are pairs $(c, q) \in C \times Q$, where c is a category of κ and q is a state of $\mathcal{M}$; once the graph is built, the states in each vertex can simply be ignored. Starting from the empty trace ϵ, it adds to a set E quadruplets of the form $(\kappa(\epsilon), q_0, \epsilon, a)$ for all $a \in A$. This represents a set of transitions to visit, from a given source pair (c, q), a given "history" sequence and a given action to append to that history. The algorithm then repeatedly picks (i.e. removes) one quadruplet of the form $(c, q, \overline{\sigma}, a)$ from E. It then computes $\kappa(\overline{s} \cdot a) = c'$, and the new state $\delta(q, a) = q'$. Then, the transition $(c, q) \xrightarrow{a} (c', q')$ is added to the graph. If the pair (c', q') has never been seen before (i.e. is not already in V), a new vertex labeled (c', q') is added to the graph. Finally, the quadruplets $(c', q', \overline{\sigma} \cdot a, a)$ are added to the set of transitions to explore. Note that the algorithm assumes without loss of generality that δ is total, as per the remarks we made in Section II-A.

We must omit the proof that this algorithm terminates and does produce a Cayley graph, due to lack of space. A quick analysis shows that the number of quadruplets that can be iterated over in the loop of line 6 is bounded by $O(|C| \cdot |Q| \cdot |A|)$.

[3] With the exception of the first $t-1$ prefixes of $\overline{\sigma}$, where fewer than t states will be contained in the tuple.



(a) κ_{sr} (b) κ_{ar}

Figure 2: The Cayley graphs of some of the triaging functions defined in this paper, using the automaton of Figure 1 for $\mathcal{M}$. Colored nodes represent a possible set of important vertices, and colored edges represent a possible Steiner tree for these vertices.

Algorithm 1 An algorithm for generating a Cayley Graph from a triaging function κ and a reactive specification $\mathcal{M} = \langle Q, A, q_0, \delta, \omega\rangle$.

```
1:  procedure CayleyGraph(M, κ)
2:      δ', E ← ∅; σ̄ ← ϵ; V ← {(κ(ϵ), q0)}
3:      for a ∈ A do
4:          E ← E ∪ {(κ(ϵ), q0, ϵ, a)}
5:      end for
6:      while E ≠ ∅ do
7:          pick (c, q, σ̄, a) in E
8:          c' ← κ(σ̄ · a)
9:          q' ← δ(q, a)
10:         δ' ← δ' ∪ {((c, q), a, (c', q'))}
11:         if (c', q') ∉ V then
12:             V ← V ∪ {(c', q')}
13:             for a' ∈ A do
14:                 E ← E ∪ {(c', q', σ̄ · a, a')}
15:             end for
16:         end if
17:     end while
18:     return ⟨V, δ', (κ(ϵ), q0)⟩
19: end procedure
```

Indeed, one can observe that throughout its execution, the set E will contain at most one quadruplet $(c, q, \overline{\sigma}, a)$ for given $c \in C$, $q \in Q$ and $a \in A$. This entails the algorithm is linear in the number of categories of the triaging function κ, the number of actions and the number of states in the reactive specification.[4].

B. Building Test Suites for Triaging Functions

Once a Cayley graph has been computed for a given triaging function κ, the second step of the operation is to pick vertices of this graph we call the *important* vertices. The set of important vertices must be a set $V_I = \{q_1, \ldots, q_n\}$, where $n = |C|$, and such that for every $c \in C$, there exists some $1 \leq i \leq n$ such that $q_i = c$. In other words, the set of important vertices contains one vertex labeled with each category in C. For example, consider the Cayley graph of κ_a in Figure 2b. A set of important vertices is any set that includes each possible action; one possible set is identified in yellow.[5] Note that nodes that are not chosen may nevertheless be part of the Steiner tree if they are on the way to an important leaf that must be reached.

The last step of the process is to generate sequences from the Cayley graph and the important vertices. Let $G = \langle V, E \rangle$ be a directed graph with a vertex $v_0 \in V$ called the *root*. Let $V' \subseteq V$ be a subset of edges of V. A *directed Steiner tree* is a minimal subgraph $G' \subseteq G$ where the root is connected to every vertex $v \in V'$. In the present case, the set V' is simply the set of important vertices identified in the previous step. In Figure 2, edges in red in each graph represent a possible Steiner tree. In the general case, finding the Steiner tree of the smallest size is known to be NP-hard, although it can be approximated in polynomial time [6].

Once the Steiner tree has been computed, the last step is trivial. The set of test sequences is simply the set of all paths in G that start from the root and, taking only vertices of the Steiner tree, end in all important nodes. For example, in the case of Figure 2a, such a set is $\{\epsilon, a, d, db\}$. Notice how this corresponds exactly to the set $S_{sr,1}$ we presented in Section II-B, and how this set indeed achieves state coverage. A similar reasoning could be made for the remaining Cayley graphs.

We can observe that a test suite generated by this method ensures complete category coverage by construction, since the Steiner tree is such that there is one path that ends in each of the important vertices, and hence one sequence that produces each category of κ. Note that, thanks to the genericity of our construction, full coverage is guaranteed for category coverage of all triaging functions at once. When the labeling function ℓ of the Cayley graph is injective, one can observe that all vertices are important. In this particular case, the Steiner tree degenerates into the problem of finding the minimum spanning tree of G. Then by definition, there is no way to cover each vertex of G using fewer edges (i.e. fewer actions).

[4]However, for some functions, the number of categories $|C|$ may itself be exponential in the size of the specification. This will be discussed in Section IV-C

[5]Although our simple examples look like almost all vertices must be chosen, this is no longer the case for more complex specifications.

C. Dealing with Prefix Closure

We shall now turn to triaging functions κ' that are expressed as the prefix closure of some other function (i.e. $\kappa' = \int\kappa$ for some κ). Algorithm 1 could in principle be used to generate a Cayley graph by directly using $\int\kappa$ as the triaging function; however, since the families of prefix closures are sets of categories, their size is typically exponentially larger than the set of categories of the original function –which would result in extremely large Cayley graphs. For example, categories for t-way residual state coverage are sequences of t states, whose number is bounded by $|Q|^t$. Prefix closure is t-way state coverage, where categories are sets of sequences of t-states, whose number is bounded by $2^{|Q|^t}$. Moreover, the choice of important vertices on the resulting graph differs, since a prefix closure produces families and requires family coverage instead of category coverage. One would hence need to pick vertices in such a way that each category of κ is *contained* in the family of at least one important vertex of $G_{\int\kappa}$.

A simpler way to work around this problem is to realize that any test suite that achieves full category coverage with respect to κ, by construction, also achieves full family coverage with respect to $\int\kappa$. Indeed, for any $c \in C$, if $c = \kappa(\overline{\sigma})$, then $c \in \int\kappa(\overline{\sigma})$, since $\overline{\sigma}$ is obviously a prefix of itself. Therefore, any test suite Σ_T generated for a triaging function κ also works for its prefix coverage. Note however that, in the general case, this test suite will be larger than needed, as κ is a stronger condition on a test suite than $\int\kappa$: the first requires that each category be produced at the end of a test sequence, while it suffices for the second that each category be produced somewhere along some sequence.

Therefore, an optional filtering step can be applied in order to account for this fact. To this end, from a set of test sequences $\Sigma_T = \{\overline{\sigma}_1, \ldots, \overline{\sigma}_n\}$, we shall create a hypergraph H_{Σ_T}. As a reminder, a *hypergraph* is a tuple $G = \langle V', E \rangle$, where V' is a set of vertices and $E \subseteq 2^{2^V}$ is a set of edges. A hypergraph generalizes a classical graph by having edges that may link more than two vertices. An *edge covering* of some hypergraph is a set of edges $E' \subseteq E$, such that for every vertex $v \in V'$, there exists an edge $e = \{v_0, v_1, \ldots, v_k\} \in E'$ such that $v \in e$. Hence every vertex of the graph is adjacent to at least one hyperedge in E'.

In the present case, we let $V' \triangleq \Sigma_T$, the set of all sequences produced in the test suite produced for κ. The set of hyperedges E is defined as:

$$E \triangleq \bigcup_{c \in C} \{\overline{\sigma} \in \Sigma_T : c \in \int\kappa(\overline{\sigma})\}$$

In other words, there exists one hyperedge for each category c produced by κ; this hyperedge connects the test sequences associated to a family that contains c. Finding a vertex covering of this hypergraph results in a subset $E' \subseteq \Sigma_T$ of the test sequences of κ where each category is produced at least once. This problem is NP-complete [23]; however, in practice it can be approximated in polynomial time by a greedy procedure that repeatedly picks the hyperedge containing the most uncovered vertices [8]. In our running example, we have seen that a

possible test suite for residual state coverage κ_{sr} is $S_{sr,1} = \{\epsilon, a, d, db\}$. State coverage is the prefix closure of state residual coverage; using the hypergraph vertex covering approach, we obtain the subset $\{a, db\}$, which, as it turns out, is exactly the set we had shown for that triaging function.

The advantage of this construction is that it avoids enumerating the families of $\int\kappa$ and does not require the explicit construction of its Cayley graph. This is the approach we chose to follow in our implementation of these principles, which will be discussed next.

V. Implementation and Experiments

In this section, we report on the implementation and experimental evaluation of the concepts described earlier. To the best of our knowledge, this paper is the first that compares multiple test sequence generation tools on the same input specifications.

A. Implementation

We chose to implement the concepts described in this paper by extending the open source Java library called SealTest mentioned in Section II-C. [18]. This choice was made for two reasons. First, the modular organization of SealTest makes it possible to implement new trace generators easily. Second, as we shall discuss later, the library constitutes the single other implementation of a specification-based sequence generator that was publicly available for comparison. Our additions have been integrated directly into SealTest's code base.

The library provides constructs for defining coverage metrics and test sequence generators for `AtomicEvents`, which are actions represented by a single symbol; an `AtomicTrace` is a finite sequence of atomic events, and a `TestSuite` is a set of atomic traces. A reactive specification can be created programmatically, or read form a variety of text file formats.

SealTest provides the `TraceGenerator` interface, which allows users to implement their own trace generation algorithms. It currently provides a simple method for generating traces, using a greedy random algorithm we already described in Section II-C. The extension to SealTest we implemented is an alternate generator, based on the Cayley graph/Steiner tree construction presented in this paper. Here, `T` is the generic type of the events inside a sequence, and `U` is the type of the categories returned by the function. The new class needs to implement `getStartCategory()`, which returns the equivalence class of the empty trace, and `processTransition()`, which returns the equivalence class of the current sequence to which a new event is to be appended.

B. Results

Equipped with this implementation, we proceeded to experimentally evaluate our proposed approach. We focus on two important dimensions of the problem, quality and performance. Quality is measured by the size of the generated test suites and their associated coverage ratio, for various specifications and coverage metrics. Performance is the ability of a test generation technique to produce results in reasonable time, and to scale well to large specifications. It is important to stress that our experiments do not intend to assess the relative merits of the coverage criteria themselves, in terms of their capacity to reveal bugs. We recall that our focus is on providing a uniform theoretical framework to *generate* test sequences according to a variety of criteria.

Alas, only two of the works mentioned in Section II-C provide an implementation. In addition, for those related works that give experimental results, many of the input specifications that have been used could not be found either. Finally, some papers report data only for state coverage, while some others report figures only for transition coverage; those that report on t-way coverage consider only the case $t = 1$. This makes any meaningful empirical comparison between our proposed approach and these related works flatly impossible.[6] Among the remaining contenders, only SealTest v0.2 and GraphWalker v4.3.0 could be reliably tested against our approach.

Consequently, we prepared a set of experiments that compares our Cayley-based approach to these two available implementations. To remedy the absence of a preexisting benchmark, we gathered 33 reactive specifications from various sources. This includes temporal specification patterns introduced by Dwyer *et al.* [13] and which occur commonly in the specification of concurrent and reactive systems; all the finite-state specifications that could be obtained from aforementioned works on test sequence generation (less than a dozen); classical examples of specifications such as the *Qui-Donc* protocol [38] and the microwave FSM [9]; FSMs obtained from common regular expressions, such as validating e-mail addresses and date formats; and a collection of FSM from the Büchi Store [36]. The reactive specifications range in size between 2 and 33 states, and have up to 397 transitions. Counting all combinations of state/action/transition residual/non-residual t-way triaging functions on all specifications, this corresponds to a set of 1452 distinct problem instances which generate a total of 5808 measurements. The experiments were implemented using the LabPal testing framework [19], and their source code is publicly available, including all input specifications.[7] Each problem instance was given a timeout of 15 seconds.

1) Solution Quality: The first factor we measured is the size of the generated test sequences. This can be measured as both the number of different sequences required to achieve a certain coverage, and the total number of actions in the generated sequences. We compared the size of generated test sequences on all 33 automata specifications for the SealTest (greedy) algorithm and our proposed method. Figure 3 shows a graphical rendition of this comparison. Each dot represents a problem instance; the length of the Cayley-based suite is the position on the x-axis, and the length of the greedy solution is the position on the y-axis. Any dot that lies over the line $x = y$ therefore corresponds to a problem where the Cayley approach produced a smaller (i.e. better) test suite. On average over all problem instances, our proposed approach generates

[6] Implementations of tools that perform *conformance testing*, such as JTorX [1] must also be discarded as they address a different problem; see §II-C2.

[7] https://bitbucket.org/sylvainhalle/cayley-fsm-benchmark

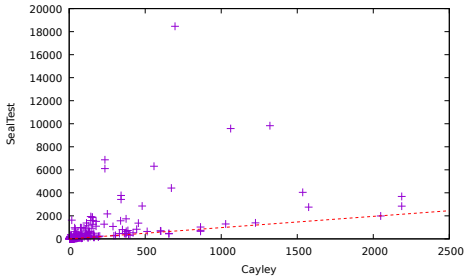


Figure 3: Test suite total length for each problem instance, for the Cayley graph method (x) and the random greedy method (y). The red line represents $x = y$.

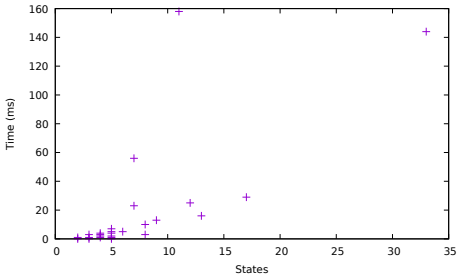


Figure 4: Test sequence generation time with respect to specification size (in states), for the state coverage metric.

test suites that are 4.73 times smaller than SealTest, and 4.11 times smaller than GraphWalker.

These experiments made us discover that both SealTest and GraphWalker fail to obtain 100% coverage for many problem instances; (in 23.0% and 38.0% of all problem instances, respectively). When such a situation occurs, these tools repeatedly generate candidate sequences to add to a test suite, but find that none of them increases the existing coverage. As a result, they exhaust their iterations and return a suite that is artificially small, but also well below full coverage (sometimes as little as 25%). This is why, in comparison, the Cayley-based test suites are sometimes larger —in counterpart they always have 100% coverage.

Note that increasing the maximum number of iterations of the greedy algorithm is of no help. In our benchmark, the greedy algorithm was given an upper limit of $1000 \times t$ iterations[8], yet none of the generated test suites contains more than 275 sequences. Thus, the random greedy algorithm already runs for hundreds of iterations without succeeding at generating a single new sequence that increases coverage.

As for the number of sequences inside a test suite, we observed that the random greedy approach produces fewer sequences than the Cayley approach, by an average factor of 0.542; this number decreases to 0.529 when including only test suites with full coverage. This is again a consequence of the behavior of the greedy algorithm when it fails to achieve full coverage. We recall that our proposed approach nevertheless produces test suites that contain a clearly smaller total number of actions to be performed on the SUT.

2) Performance: A second factor we measured is the required time it takes to generate a test sequence from a given specification. On the same problem instances as before, we measured the relative time taken by the greedy vs. the Cayley approach to generate a test sequence. The plot cannot be included due to lack of space, but the results are unequivocal as the Cayley approach is on average 238.0 times faster. This can, again, be explained by the fact that the greedy random algorithm spends a lot of time trying to find new sequences that increase coverage (and often fails, as we have seen above). In absolute numbers, our proposed approach took at most 1793 ms to generate a test suite across the whole benchmark.

We also studied the impact of increasing the size of the specification on the running time of the algorithm. An excerpt of the results is plotted in Figure 4, for the state coverage metric; it shows the roughly linear increase we expected from our discussion about the complexity of Algorithm 1. Similar trends were observed with respect to the number of states and number of categories of the underlying triaging function.

VI. Conclusion and Future Work

This paper described a theoretical framework for the categorization and generation of test sequences based on a reactive specification. Given a triaging function κ and its associated Cayley graph, we have shown a generic technique to create a set of test sequences with respect to multiple kinds of coverage metrics, all based on the actual function κ being used. We then presented a generic algorithm for generating a set of test sequences that achieves complete coverage with respect to any of these metrics. Experimental results on a large set of reactive specifications have shown that our method is usable and efficient in practice. It is faster than existing tested solutions, produces smaller test suites on average, and produces guaranteed 100% coverage for all metrics and all input specifications.

Some limitations of the proposed framework need to be mentioned. First, it would be advisable to characterize the coverage criteria that *can* be expressed as triaging functions; the fact that those studied in this paper are amenable to such a formulation is not proof that they all can. Second, all the models we have hown are deterministic. It would be interesting to study how non-determinism could be handled in the proposed framework. For example, is it possible to generate adaptive, tree-shaped test cases in a similar way?

This theoretical framework lends itself to many extensions and improvements. First, there remain some coverage criteria for which a formulation into triaging functions has yet to be provided, in particular prime path coverage. Second, the model could be extended to take into account actual inputs and outputs, instead of abstract transition labels; this could make it possible to express the more complex conformance testing problem in the form of triaging functions. In addition, one could think of assigning a different *cost* to each action; this would correspond to a scenario where performing an

[8]With t being the strength of the coverage, and $t = 1$ for non-combinatorial coverage criteria.

action on a system is more complex or resource-intensive than for other actions. Adapting our proposed technique to this situation should be straightforward, as this would translate into solving the corresponding problems on the Cayley graph for the *weighted* case. So far, we also considered events as atomic units; one could handle richer types of events on which arbitrary predicates could be evaluated, and upgrade the triaging functions and test sequence generation techniques to this more general setting.

Finally, a promising approach would consist of adapting this theoretical framework to other formalisms for sequences of actions, such as temporal logic or regular expressions. To this end, it would suffice to provide appropriate definitions of a triaging function for these notations; this work is currently under way and will be the subject of an upcoming publication.

References

[1] A. Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In P. G. Frankl, editor, *ISSTA*, pages 123–133. ACM, 2002.

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

[4] A. Cayley. Desiderata and suggestions: No. 2. the theory of groups: graphical representation. *Amer. J. Math.*, 1(2):174–6, 1878.

[5] A. Chander, D. Dhurjati, K. Sen, and D. Yu. Optimal test input sequence generation for finite state models and pushdown systems. In *ICST*, pages 140–149. IEEE Computer Society, 2011.

[6] M. Charikar, C. Chekuri, T. yat Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed Steiner problems. *Journal of Algorithms*, 33(1):73–91, October 1999.

[7] T. Chow. Testing software designs modeled by finite-state machines. *IEEE Trans. on Software Engineering*, SE-4(3):178–187, May 1978.

[8] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, August 1979.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[10] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Trans. Software Eng.*, 33(8):558–574, 2007.

[11] F. Denis, A. Lemay, and A. Terlutte. Residual finite state automata. In A. Ferreira and H. Reichel, editors, *STACS*, volume 2010 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 2001.

[12] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Information & Software Technology*, 52(12):1286–1297, 2010.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[14] J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba. Search based algorithms for test sequence generation in functional testing. *Information & Software Technology*, 58:419–432, 2015.

[15] Y. Fu and W. Choosilp. Specification based testing of on Android systems. *International Journal of Wireless & Mobile Networks*, 9(5), 2017.

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

[17] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE*, pages 235–244. ACM, 2010.

[18] S. Hallé and R. Khoury. SealTest: a simple library for test sequence generation. In T. Bultan and K. Sen, editors, *ISSTA*, pages 392–395. ACM, 2017.

[19] S. Hallé, R. Khoury, and M. Awesso. Streamlining the inclusion of computer experiments in a research paper. *IEEE Computer*, 51(11):78–89, 2018.

[20] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[21] R. M. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Softw. Test. Verification Reliab.*, 10(4):201–202, 2000.

[22] S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams. In *SAICSIT*, pages 296–300. ACM, 2003.

[23] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proc. of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[24] Y. G. Kim, H. S. Hong, D. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4):187–192, 1999.

[25] P. M. Kruse. Test sequence generation from classification trees using multi-agent systems, 2011.

[26] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman and Hall/CRC, 2013.

[27] N. Kushik, N. Yevtushenko, I. Burdonov, and A. Kossatchev. Deriving synchronizing and homing sequences for input/output automata. *Aut. Control Comp. Sci.*, 52:589, 2018.

[28] D. Lee and M. Yannakakis. Principles and methods of testing FSMs: A survey. *Proceedings of the IEEE*, 84:1089–1123, 1996.

[29] N. Li, F. Li, and J. Offutt. Better algorithms to minimize the cost of test paths. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 280–289. IEEE Computer Society, 2012.

[30] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13(1):25–53, 2003.

[31] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das. Nondeterministic state machines in protocol conformance testing. In O. Rafiq, editor, *Protocol Test Systems VI*, volume C-19 of *IFIP Transactions*, pages 363–378. North-Holland, 1993.

[32] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[33] P. R. Srivastava, N. Jose, S. Barade, and D. Ghosh. Optimized test sequence generation from usage models using ant colony optimization. *Int. J. of Softw. Engineering and Applications*, 1(2):1089–1123, 2010.

[34] P. Stocks and D. A. Carrington. A framework for specification-based testing. *IEEE Trans. Software Eng.*, 22(11):777–793, 1996.

[35] S. K. Swain, D. P. Mohapatra, and R. Mall. Test case generation based on state and activity models. *Journal of Object Technology*, 9(5):1–27, 2010.

[36] Y. Tsay, M. Tsai, J. Chang, Y. Chang, and C. Liu. Büchi store: an open repository of ω-automata. *Int. J. Softw. Tools Technol. Transf.*, 15(2):109–123, 2013.

[37] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992.

[38] M. Utting and B. Legeard. *Practical Model-Based Testing. A Tools Approach*. Morgan Kauffmann, 2006.