Visualization and Abstractions for Execution Paths in Model-based Software Testing

Rui Wang[1], Cyrille Artho[2], Lars Michael Kristensen[1], Volker Stolz[1]

[1] Department of Computing, Mathematics, and Physics,
Western Norway University of Applied Sciences, Bergen, Norway
Email: {rwa,lmkr,vsto}@hvl.no
[2] School of Computer Science and Communication,
KTH Royal Institute of Technology, Stockholm, Sweden
Email: artho@kth.se

Abstract. This paper presents a technique to measure and visualize execution-path coverage of test cases in the context of model-based software systems testing. Our technique provides visual feedback of the tests, their coverage, and their diversity. We provide two types of visualizations for path coverage based on so-called state-based graphs and path-based graphs. Our approach is implemented by extending the Modbat tool for model-based testing and experimentally evaluated on a collection of examples, including the ZooKeeper distributed coordination service. Our experimental results show that the state-based visualization is good at relating the tests to the model structure, while the path-based visualization shows distinct paths well, in particular linearly independent paths. Furthermore, our graph abstractions retain the characteristics of distinct execution paths, while removing some of the complexity of the graph.

1 Introduction

Software testing is a widely used, scalable and efficient technique to discover software defects [17]. However, generating sufficiently many and diverse test cases for a good coverage of the system under test (SUT) remains a challenge. *Model-based testing* (MBT) [21] addresses this problem by automating test-case generation based on deriving concrete test cases automatically from abstract (formal) models of the SUT. In addition to allowing automation, abstract test models are often easier to develop and maintain than low-level test scripts [20]. However, for models of complex systems, an exhaustive exploration of all possible tests is infeasible, and the decision of how many tests to generate is challenging.

Visualizing the degree to which tests have been executed can be helpful in this context: visualization can show if different parts of the model or SUT have been explored equally well [13], if there are redundancies in the tests [13], and if there are parts of the system that are hard to reach, e. g., due to preconditions that do not hold [3]. In this paper, we focus on the visualization of test paths on the test model, as this provides a higher level of abstraction than the SUT.

The main contribution of this paper is to present a technique to capture and visualize execution paths of the model covered by test cases generated with MBT. Our approach records execution paths with a trie data-structure and visualizes them with the aid of lightweight abstractions as *state-based graphs* (SGs) and *path-based graphs* (PGs). These abstractions simplify the graphs and help us to deal with complexity in moderately large systems. The visual feedback provided by our technique is useful to understand to what degree the model and the SUT are executed by the generated test cases, and to understand execution traces and locate weaknesses in the coverage of the model. Being based on the state machine of the model, the state graph focuses on the behaviors of a system in relation to the test model. The path graph shows paths as transition sequences and eliminates crossing edges.

Our second contribution is to provide a path coverage visualizer based on the Modbat model-based API tester [2]. Our tool extends Modbat and enables the visualization of path coverage without requiring modifications of the models. Users of the tool can choose to visualize all execution paths in the SGs and PGs, or limit visualization to subgraphs of the SGs and PGs for models of large and complex systems. Our third contribution is an experimental evaluation on several model-based test suites. We analyze the number of executed paths against quantitative properties of the graphs. We show how edge thickness and colors help to visualize the frequency of transitions on executed paths, what kinds of paths have higher coverage than others, and what kinds of tests succeed or fail. We also compare the resulting SGs and PGs with *full state-based graphs* (FSGs) and *full path-based graphs* (FPGs). The FSGs and FPGs are the graphs without applying abstractions; they are used only in this paper for comparison with the SGs and PGs. We show that our abstraction technique helps to reduce the number of nodes and edges to get concise and abstracted graphs.

The rest of this paper is organized as follows. Section 2 gives background on extended finite state machines and Modbat. In Sect. 3, we give our definition of execution paths and the trie data structure used for their representation. In Sect. 4, we introduce our approach for the path coverage visualization. In Sect. 5, we present the two types of graphs and the associated abstractions used. Section 6 presents our experimental evaluation of the path coverage visualizer tool and analyzes path coverage of selected test examples. In Sect. 7, we discuss related work, and in Sect. 8 we sum up conclusions and discuss future work.

2 Extended Finite State Machines and Modbat

We use extended finite state machines (EFSMs) as the theoretical foundation for our models and adapt the classical definition [6] to better suit its implementation as an embedded language, and several extensions that Modbat [2] defines.

Definition 1 (Extended Finite State Machine). *An* extended finite state machine *is a tuple* $M = (S, s_0, V, A, T)$ *such that:*

- S *is a finite set of states, including an initial state* s_0*.*

- $V = V_1 \times \ldots \times V_n$ *is an n-dimensional vector space representing the set of values for variables.*
- *A is a finite set of actions* $A : V \to (V, R)$*, where* $res \in R$ *denotes the* result *of an action, which is either* successful*,* failed*,* backtracked*, or* exceptional*. A successful action allows a test case to continue; a failed action constitutes a test failure and terminates the current test; a backtracked action corresponds to the case where the enabling function of a transition is* false *[6]; exceptional results are defined as such by user-defined predicates that are evaluated at run-time, and cover the non-deterministic behavior of the SUT. We denote by* $Exc \subset R$ *the set of all possible exceptional outcomes.*
- *T is a transition relation* $T : S \times A \times S \times E$*; for a transition* $t \in T$ *we denote the left-side (origin) state by* $s_{origin}(t)$ *and the right-side (destination) state by* $s_{dest}(t)$*, and use the shorthand* $s_{origin} \to s_{dest}$ *if the action is uniquely defined. A transition includes a possible empty mapping* $E : Exc \to S$*, which maps exceptional results to a new destination state.*

Compared to the traditional definition of an EFSM [6], we merge the enabling and update functions into a single action $\alpha \in A$, and handle inputs and outputs inside the action. Actions deal with preconditions, inputs, executing test actions on the SUT, and its outputs. An action may also include assertions; a failed assertion causes the current test case to fail. Finally, transitions support non-deterministic outcomes in our definition.

Modbat. Modbat is a model-based testing tool aimed at performing online testing on state-based systems [2]. Test models in Modbat are expressed as EFSMs in a domain-specific language based on Scala [18]. The model variables can be arbitrarily complex data structures. Actions can update the variables, pass them as part of calls to the SUT, and use them in test oracles.

Fig. 1 (left) shows the ChooseTest model that we will use as a simple running example to introduce the basic concepts of Modbat and our approach to execution path visualization and abstraction. A valid execution path in a Modbat model starts from the initial state and consists of a sequence of transitions. The first declared state automatically constitutes the initial state. Transitions are declared with a concise syntax: "*origin*" → "*dest*" := {*action*}. The ChooseTest model in Fig. 1 consists of three states: "*ok*", "*end*, and "*err*". It also uses `require` in the action part as a precondition to check if a call to the random function `choose` returns `0` (10% chance). Only in that case is the transition from "*ok*" to "*err*" enabled. Function `assert` is then used to check if a subsequent call to `choose` returns non-zero. If `0` is returned (10% chance), the assertion fails. Thus, transition "*ok*" → "*err*" is rarely enabled; and if enabled, it fails only infrequently.

Choices. Modbat supports two kinds of *choices*: (1) Before a transition is executed, the choice of the next transition is available. (2) Within an action, choices can be made on parameters that can be used as inputs to the SUT or for computations inside the action. The latter are *internal choices*, which can be choices

```
class ChooseTest extends Model {
  "ok" -> "ok" := skip
  "ok" -> "end" := skip
  "ok" -> "err" := {
    require(choose(0, 10) == 0)
    assert(choose(0, 10) != 0)
  }
}
```

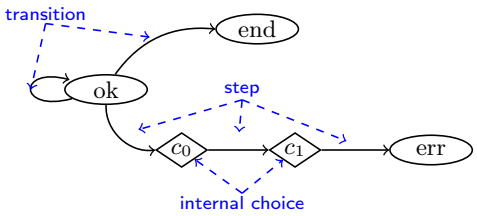


Fig. 1: Model ChooseTest (left) with steps and internal choices (right).

over a finite set of numbers or functions. These choices are obtained in Modbat by calling the function `choose`. In our example, the action in transition "*ok*" to "*err*" has two internal choices shown as c_0 and c_1 in Fig. 1 (right).

Transitions and steps. We divide an action into smaller *steps* to distinguish choices between transitions from internal choices inside an action. A step is a maximal-length sequence of statements inside an action that does not contain any choices. Our definition of choices corresponds to the semantics of Modbat, but also that of other tools, such as Java Pathfinder [22], a tool to analyze concurrent software that may also contain non-deterministic choices on inputs.

Action results. Modbat actions (which execute code related to transitions) have four possible outcomes: successful, backtracked, failed, or exceptional. A successful action allows a test case to continue with another transition, if available. An action is *backtracked* and resets the transition to its original state if any of its preconditions are violated. An action *fails* if an assertion is violated, if an unexpected exception occurs, or if an expected exception does not occur. In our example, the action of transition "*ok*" to "*err*" is backtracked if the `require`-statement in the action evaluates to `false`, and the action *fails* if the `assert`-statement evaluates to `false`. *Exceptional results* are defined by custom predicates that may override the destination state ($s_{dest}(t)$) of a transition t; see above. If no precondition or assertion is violated, and no exceptional result occurs, the action is *successful*.

3 Execution Paths and Representation

Path coverage concerns a sequence of branch decisions instead of only one branch at a time. It is a stronger measurement than branch coverage, since it considers combinations of branch decisions (or statements) with other branch decisions (or statements), which may not have been tested according to the plain branch or statement coverage [16]. It is hard to reach 100% path coverage, as the number of execution paths usually increases exponentially with each additional branch or cycle [15].

A finite *execution path* is a sequence of transitions starting from the initial state and leading to a terminal state. A *terminal state* in our case is a state

without outgoing transitions, or a state after a test failed. We denote by $S_{terminal}$ the set of terminal states.

Definition 2 (Execution Path). *Let $M = (S, s_0, V, A, T)$ be an EFSM. A* finite execution path *p of M is a sequence of transitions, which constitute a path $p = t_0 t_1 \dots t_n, \quad t_n \in T$, such that $s_{origin}(t_0) = s_0$, the origin and destination states are linked: $\forall i, 0 < i \leq n, s_{origin}(t_i) = s_{dest}(t_{i-1})$, and $s_{dest}(t_n) \in S_{terminal}$.*

We first represent the executed paths in a data structure based on the transitions executed by the generated test cases, and then use this to visualize path coverage of a test suite in the form of state-based and path-based graphs.

We record the path executed by each test case in a *trie* [5]. A trie is a prefix tree data structure where all the descendants of a node in the trie have a common prefix. Each trie node n stores information related to an executed transition, including the following: t (executed transition); ti (transition information); trc (transition repetition counter) to count the number of times transition t has been executed repeatedly without any other transitions executing in between during a test-case execution, with a value of 1 equalling no repetition; tpc (transition path counter) to count the number of paths that have this transition t executed trc times in a test suite; Ch, the set of children of node n; and lf, a Boolean variable to decide if the current node is a leaf of the tree. The transition information ti consists of the $s_{origin}(t)$ and $s_{dest}(t)$ states of the transition, a transition identifier tid, a counter cnt to count the number of times this transition is executed in a path, an action result res, which could be successful, backtracked, or failed, and sequences of transition-internal choices C for modeling non-determinism.

As an example, consider a test suite consisting of three execution paths: $p_0 = [a \to b, b \to b, b \to c, c \to d]$, $p_1 = [a \to b, b \to b, b \to b, b \to c, c \to d]$, and $p_2 = [a \to b, b \to b, b \to e]$, where a, b, c, d, and e are states. These execution paths can be represented by the trie data structure shown in Fig. 2 where the node labeled root represents the root of the trie. Note that this data structure is not a direct visual representation of the paths and it is not the trie data structure that we eventually visualize in our approach. Each non-root node in the trie in Fig. 2 has been labeled with the transition it represents. As an example, node 1 represents the transition $a \to b$ and node 2 represents the transition $b \to b$. This reflects that all the three execution paths stored in the trie have $a \to b$ followed by $b \to b$ as a (common) prefix. Each non-root node also has a label representing the transition counters associated with the node. For the transition counters, the value before the colon is trc (transition repetition counter), while the value after the colon is tpc (transition path counter). For example, the transition $b \to b$ associated with node 2 has been taken three times in total. Two paths, (p_0 and p_2) execute this transition once (label trc=1:tpc=2), while one path p_1 executes it twice (label trc=2:tpc=1). A parent node and a child node in the trie are connected by a mapping $\langle tid, res \rangle \mapsto n$ in each node which associates a transition identifier and action result (res) with a child (destination) node n.

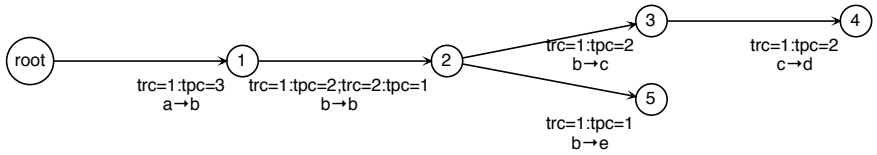
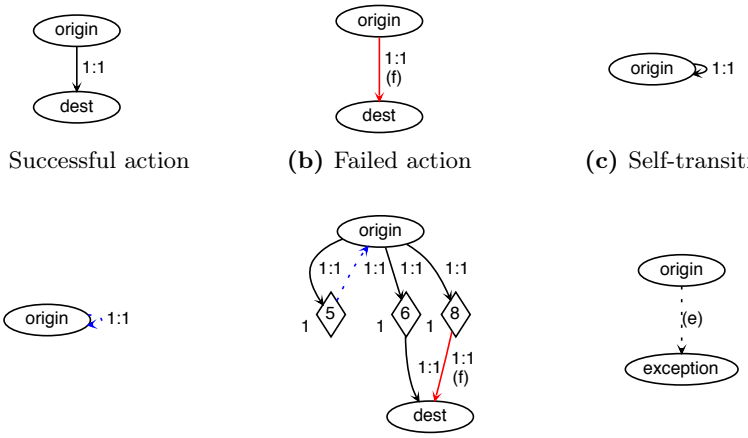


Fig. 2: Example trie data structure representing three executed paths.



(a) Successful action **(b)** Failed action **(c)** Self-transition

(d) Backtracked transition **(e)** Action with choices **(f)** Exception result

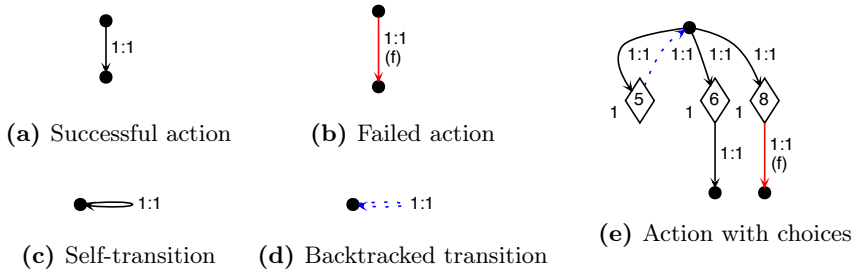
Fig. 3: Basic visualization elements of the state-based graphs (SGs).

4 Path Coverage Visualization

Our path coverage visualizer can produce two types of directed graphs: state-based graphs (SGs) and path-based graphs (PGs). These two types of graphs are produced based on the data stored in the trie data structure representing the executed paths of the testing model. Fig. 3 and Fig. 4 illustrate the basic visualization elements of the SGs and PGs, respectively, with the help of the DOT Language [9], which can be used to create graphs with Graphviz tools [4].

The SGs and PGs have common node and edge styles (shape, color and thickness) to indicate different features of the path- execution coverage visualization.

Node Styles. We use three types of node shapes in the graphs for path coverage visualization. Elliptical nodes ⬭ represent states in the SG as shown in Fig. 3. Point nodes • represent the connections between transitions/steps in the PG as shown in Fig. 4. Diamond nodes ◊ visualize internal choices in both the SGs and PGs as shown in Fig. 3e and Fig. 4e. Each diamond node has a value inside indicates the chosen value. There is also an optional counter value label aside each diamond node to show how many times this choice has been taken. The edge labels of the format $n : m$ will be discussed later.



(a) Successful action **(b)** Failed action **(c)** Self-transition **(d)** Backtracked transition **(e)** Action with choices

Fig. 4: Basic visualization elements of the path-based graphs (PGs).

Edge Styles. A directed edge in both the SGs and PGs represents an executed transition and its related information as stored in the trie structure. We distinguish different kinds of edges based on the action results, using shape and color styles. Black solid edges are used to represent successful transitions (Fig. 3a and Fig. 4a). Blue dotted edges are used to visualize backtracked transitions (Fig. 3d and Fig. 4d). Red solid edges labeled (f) are used to visualize failed transitions (shown in Fig. 3b and Fig. 4b). Black solid loops represent self-transitions (Fig. 3c and Fig. 4c) and are used when $s_{dest}(t)$ and $s_{origin}(t)$ of a transition t are the same state. Black dotted edges labeled (e) are used to represent exceptional results for the SG (shown in Fig. 3f). This allows the visualization to distinguish between the normal destination state $s_{dest}(t)$ and the exception destination state. For the PG, this kind of edge is ignored by merging the point nodes of $s_{origin}(t)$ and the exception destination state of a transition t into one point node. If a transition t consists of multiple *steps* (Fig. 3e and Fig. 4e), we only apply the edge styles to the last step edge which connects to $s_{dest}(t)$, while other step edges use a black solid style.

Each edge may have a label for additional information, such as transition identifier *tid*, and values of the counters *trc* and *tpc*. Here we use the format *trc* : *tpc*. It is optional to show these labels. For example, in both Fig. 3 and Fig. 4, the values of counters are all 1 : 1 indicates that each transition in a test case is executed only once without any repetitions, and there is only one path that has this transition executed.

The thickness of an edge indicates how frequently a transition is executed for the entire test suite. The thicker an edge is, the more frequently is its transition executed. Let *nTests* be the total number of executed test cases. Then, the thickness of an edge is given by $\ln(\frac{\sum count*100}{nTests} + 1)$, where the value of *count* is the *tpc* value of a transition in each path if there are no internal choices for this transition. If a transition has internal choices, then we use the value of the counter for each internal choice as the value of *count*. Since we merge edges in the graphs corresponding to the same transitions or the same choices from different paths, we then compute the sum of values of *counts* obtained for the transition or choice.

5 State-based and Path-based Graphs

We now present the details of the state-based (SG) and path-based (PG) graphs with abstractions that form the foundation of our visualization approach. These abstractions underly the reduced representation of the execution paths.

McCabe [12] proposed basic path testing and gave the definition of a *linearly independent path*. A linearly independent path is any path through a program that contains at least one new edge which is not included in any other linearly independent paths. A subpath q of an execution path p is a subsequence of p (possibly p itself), and an execution path p traverses a subpath q if q is a subsequence of p. In this paper, for the visualization of execution paths, we merge subpaths from different linearly independent paths in both SG and PG with the aid of the trie data structure.

5.1 State-based Graphs

An SG is a directed graph $SG = (N_s, E_t)$, where $N_s \equiv \{n_{s_0}, n_{s_1}, \ldots, n_{s_i}\}$ is a set of nodes including both elliptical nodes representing states with their names and diamond nodes representing internal choices with their values as discussed in Sect. 4. Elliptical nodes use the name of their state as node identifier; diamond nodes are identified by a tuple $\langle v, cn \rangle$, where v is the value of the choice, and cn is an integer number starting from 1 and increasing with the number of diamond nodes. $E_t \equiv \{e_{t_0}, e_{t_1}, \ldots, e_{t_i}\}$ is a set of directed edges representing both transitions and steps. These edges connect nodes according to node identifiers.

An SG is an abstracted graph of the unabstracted full state-based graph (FSG). An FSG may have redundant edges representing the same transition/step between two states; it may also contain choices with the same choice value appearing more than once. These situations, in general, contribute to making the FSG large, complex and difficult to analyze, especially for large and complex systems. Note that the FSG is only used by us to show its complexity in this paper for comparison with the SG. The FSG for the ChooseTest model (discussed in Sect. 3) is already very dense after only 100 test cases (see Fig. 5).

In order to reduce the complexity of graphs such as Fig. 5, we abstract the FSG to get the SG, and use edge thickness to indicate the frequency of transitions in the executed paths. We use the ChooseTest model with 1000 executed test cases as an example to show how the SG is obtained in four abstraction steps:

1. *Merge edges of subpaths*: the trie data structure is used to merge subpaths of linearly independent paths when storing transitions in the trie. As discussed for Fig. 2, transition $a \rightarrow b$ followed by $b \rightarrow b$ is a (common) prefix for all the three execution paths p_0, p_1 and p_2. In other words, all these three

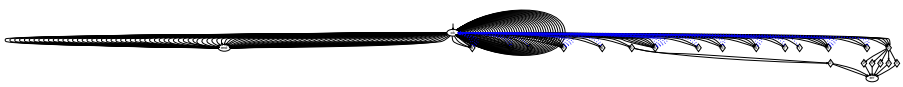


Fig. 5: FSG for 100 test cases of ChooseTest.

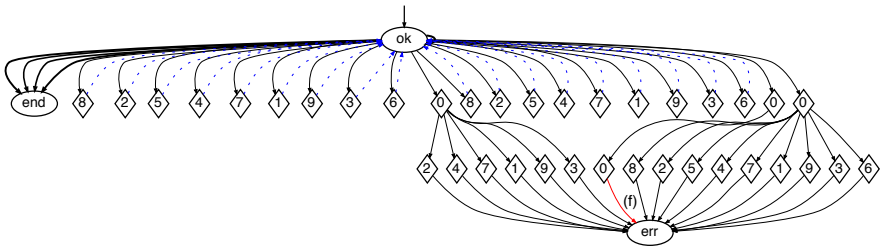


Fig. 6: The graph for 1000 test cases of ChooseTest after merging subpaths.

execution paths traverse the subpaths $a \to b$ and $b \to b$. Therefore, to obtain the SG, edges representing transition $a \to b$ and $b \to b$ from three execution paths are merged into one edge by the trie data structure. We then use an edge label of the form "$trc : tpc$" to show how a transition represented by this edge is executed. (Here, we do not show edge labels due to space limitations.) After merging edges of subpaths, we get only linearly independent paths in the graph. Fig. 6 shows the graph of the ChooseTest model after merging subpaths. There are seven linearly independent paths: $p_0 = [ok \to end]$, $p_1 = [ok \to ok, ok \to end]$, $p_2 = [ok \to ok, ok \to err(backtracked), ok \to end]$, $p_3 = [ok \to ok, ok \to err]$, $p_4 = [ok \to err(backtracked), ok \to end]$, $p_5 = [ok \to err(failed)]$ and $p_6 = [ok \to err]$.

2. *Merge edges of linearly independent paths*: from Fig. 6, it can be noticed that after merging edges of subpaths, the graph may still have redundant edges between two states that represent the same transition with the same action result from different linearly independent paths. For example, there are four edges between the "ok" and "end" states, from four linearly independent paths: p_0, p_1, p_2 and p_4. We merge such edges into one single edge. We also aggregate the path coverage counts. The aggregated counts can be shown as an optional edge label on the form "$trc : tpc$", using ";" as the separator, e. g., "$1 : 304; 1 : 158; 1 : 177; 1 : 290$" for the edge between the "ok" and "end" states after merging p_0, p_1, p_2 and p_4.
3. *Merge internal choice nodes*: internal choice nodes of a transition are merged in two ways. First, based on Step 1, when storing transitions in the trie, each transition has recorded choice lists; we merge choice nodes from different choice lists if these choice nodes have the same choice value and they are a (common) prefix of choice lists. For example, for choice lists $[0, 1, 2]$ and $[0, 1, 3]$ (0, 1, 2, 3 are choice values), we notice that these two choice lists both have choice nodes with value 0 and 1, and they are a (common) prefix for these two lists. We then merge choice nodes with value 0 and 1 to become one choice node, respectively, when storing transitions in the trie. Second, if there are still choice nodes of a transition from different linearly independent paths, with the same value appearing more than once, such as choices in Fig. 6, then we merge them into one choice node during Step 2. For both approaches, we get the result of the sum of the values of counters of merged choice nodes. This result denotes the total number of times a choice value appears in the

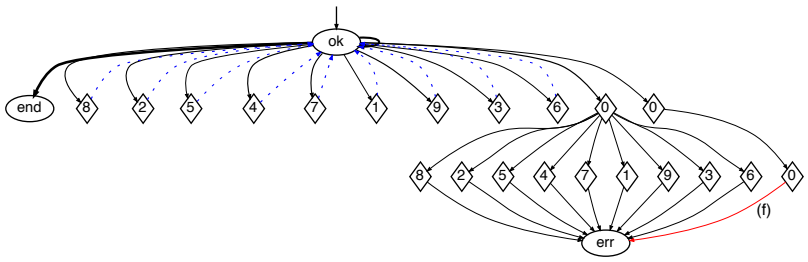


Fig. 7: SG for 1000 test cases of ChooseTest with all abstractions applied.

SG, and it can then be shown in addition to the outcome of the choice on the label of the final choice node after merging. Note that to avoid visual clutter, we elide showing the target state for backtracked transitions.

4. *Merge loop edges*: loop edges represent self-transition loops and backtracked transitions; they are merged if they represent the same transition with the same action result.

Fig. 7 illustrates the final SG with all abstractions after 1000 test cases. One characteristic of the SG is that it is a concrete instance of its underlying state machine graph. The EFSM shows potential transitions, whereas the SG shows the actually executed steps of actions, and internal choices for non-determinism. For example, Fig. 7 is a concretization of the state machine shown in Fig. 1. As shown in Fig. 1, the transition "*ok*" $\rightarrow$ "*err*" has a precondition with an internal choice over values 0 to 9 (see Fig. 7). Only choice 0 enables this transition; the transition is backtracked the original state "*ok*" otherwise, as shown with the blue dotted edges. If the transition is enabled by a successful choice with value 0, the assertion, which is another internal choice that fails only for value 0 out of 0 to 9, is executed; its failure is shown by the red solid (failing) edge in Fig. 7.

5.2 Path-based Graphs

The PG is a directed graph $PG = (N_p, E_t)$, where $N_p \equiv \{n_{p_0}, n_{p_1}, \ldots, n_{p_i}\}$ is a set of nodes, including point nodes representing connections between transitions and diamond nodes representing internal choices with their values (see Sect. 4); and $E_t \equiv \{e_{t_0}, e_{t_1}, \ldots, e_{t_i}\}$ is a set of directed edges representing both transitions and steps. They connect nodes using the identifiers of nodes.

The nodes in PG in contrast to SG do not correspond to states in the EFSM; instead, each node corresponds to a step in a linear independent path through the EFSM. Therefore, each point node is allocated a point node identifier pn, an integer starting from 0 (we elide the label in the diagrams here). The value of pn increases with the number of point nodes. Each diamond node is identified by a tuple $\langle v, cn \rangle$, similar to the diamond nodes in the SG. For the edges representing transitions, they are connected by point nodes according to their identifiers, which results in constructing paths one by one. All constructed paths start with the same initial point node and end in different final point nodes. The number of constructed paths in the PG indicates the number of linearly independent paths.

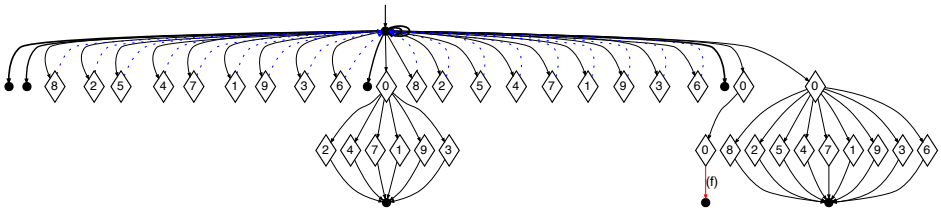


Fig. 8: Path-based graph for 1000 test cases of the ChooseTest model.

An PG is an abstracted graph of the full state-based graph (FPG) without any abstractions. (Here, we do not show the FPG of the ChooseTest model due to space limitations.) As was the case for SG, we apply abstractions to reduce the complexity of the FPG to obtain the PG and use thickness to indicate the frequency of transitions taken by executed paths. The reduction is based on three abstractions:

1. Merge edges of subpaths using the same approach as for step 1 of the SG.
2. Merge internal choice nodes with the first approach of step 3 used to merge choice nodes for the SG.
3. Merge loop edges representing self-transition loops and backtracked transitions as for the SG.

Unlike for the SG, we do not merge edges of linearly independent paths and choice nodes from different linearly independent paths for the PG, since our goal for the PG is to show linearly independent paths after applying the abstractions.

Fig. 8 shows the abstracted PG for the ChooseTest model. It can be seen that there are seven black final point nodes from seven paths, which indicate that seven linearly independent paths have been executed. The information about the number of linearly independent paths is one characteristic of the PG, and this information is not easy to derive from the SG shown in Fig. 7.

5.3 User-defined search function

As the SG and PG graphs might become unwieldy for complex testing models, the user can specify a *selection function* to limit the visualization to a subgraph. After completion of the tests, the user can filter the graph into a subgraph by providing a query in the form of a quadruple $\langle tid, res, l, ptid \rangle$ to locate a recorded transition in the trie data structure, where *tid* is the transition identifier for the transition that the user wants to locate; *res* is the action result of this transition; l is the level of this transition in the trie; *ptid* is the transition identifier for this transition's parent in the trie. With this selection function, users can select a subtrie to generate both SG and PG with the corresponding root node in lieu of an interactive user-interface. It should be noted that this projection only affects visualization, and not the number of executed tests.

6 Experimental Evaluation

We have applied and evaluated our path coverage visualization approach on a collection of Modbat models. The list of models includes the Java server socket implementation, the coordinator of a two-phase commit protocol, the Java array list and linked list implementation, and ZooKeeper [11]. The array and linked list models, as well as the ZooKeeper model, consist of several parallel EFSMs, which are executed in an interleaving way [2].

Table 1 summarizes the results. For each Modbat model, we have considered configurations with 10, 100, 200, 500 and 1000 randomly generated test cases. The table first lists the statistics reported by Modbat: the number of states (S) and transitions (T) covered for each model (including their percentage), and the number of test cases (TC) and failed test cases (FC). The second part of the table shows the metrics of the graphs we generate. For both SGs and PGs, we list: the total number of Nodes (including both state nodes and choice nodes); the total numbers of Edges (E), the number of failed edges (FE), and loops (L). In addition to these graph metrics, for the PGs, our path coverage visualizer calculates the numbers of linearly independent paths (LIP), the longest paths (LP), the shortest paths (SP), the average lengths of paths (AVE), and the corresponding standard deviation (SD).

In Table 1, when comparing the results of the SG and PG obtained from all the models, we can see that for any increase in the number of test cases by going from 10 to 1000, the SG has a smaller number of nodes and edges than the PG. This shows that the SG is constructed in a more abstract way than the PG and is useful for giving an overview of the behavior. For the PG, although there are more nodes and edges in the graph compared to the SG, we can directly see the information about the number of linearly independent paths (LIP column in Table), so that we know how execution paths are constructed and executed from the sequences of transitions executed. This information cannot be easily seen from the SG.

In addition, the results in Table 1 indicate what degree the models are executed by the generated test cases. For example, for the coordinator model, the numbers of nodes and edges in both the PG and SG do not increase after 100 test cases are executed, and there are no failed edges. This gives us confidence about how well this model is explored by the tests. The same situation occurs for the array and linked list models. For the Java server socket and Zookeeper models, the number of failed edges for each model keeps increasing with more tests. This indicates that for these kinds of complex models, there are parts that are hard to reach and explore, so there might be a need to increase the number or quality of the tests. Moreover, we can see from Table 1 that for some models such as the ZooKeeper model, there are very large numbers of nodes and edges in both the SG and PG for, e. g., 1000 test cases executed. To deal with such large and complex models, we can use the user-defined search function discussed in Sect. 5.3 to limit the visualization to a subgraph. We do not show any subgraphs due to space limitations.

Table 1: Experimental results for the Modbat models.

Model	S	T	TC	FC	Path-based (PG)									State-based (SG)			
					Nodes	Edges			Paths					Nodes	Edges		
						E	FE	L	LIP	LP	SP	AVE	SD		E	FE	L
JavaNio ServerSocket	7/ 7 (100%)	17/17 (100%)	10	2	57	79	1	17	8	14	3	9.25	4.18	9	23	1	6
			100	3	177	243	1	48	30	15	2	7.87	3.84	9	23	1	6
			200	8	363	528	4	111	53	29	2	9.68	6.24	10	25	1	7
			500	14	779	1147	8	247	105	29	2	10.51	5.29	11	27	1	8
			1000	28	1269	1904	15	439	168	29	2	10.80	4.79	11	27	1	8
Coordinator Test	7/ 7 (100%)	6/ 6 (100%)	10	0	17	20	0	0	1	6	6	6.00	0.00	17	20	0	0
			100	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			200	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			500	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
			1000	0	21	27	0	0	1	6	6	6.00	0.00	21	27	0	0
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 5/11 (45%) 12/29 (41%)	10	0	174	542	0	276	6	99	12	58.17	38.75	34	85	0	38
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 9/11 (81%) 13/29 (44%)	100	0	1171	3222	0	1571	75	181	2	23.93	29.74	102	216	0	94
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 17/29 (58%)	200	1	3369	10474	1	4848	138	181	2	45.35	47.46	204	423	1	184
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 25/29 (86%)	500	1	10438	29730	1	14024	319	181	2	48.96	43.55	467	955	1	417
ArrayList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	11/11 (100%) 10/11 (90%) 27/29 (93%)	1000	14	29056	86871	1	40609	649	406	2	70.87	64.17	896	1812	1	815
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 1/ 2 (50%)	18/19 (94%) 8/11 (72%) 5/29 (17%)	10	0	216	718	0	348	9	191	10	56.11	72.01	34	85	0	36
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 1/ 2 (50%)	19/19 (100%) 9/11 (81%) 7/29 (24%)	100	0	1190	3348	0	1679	83	191	2	23.51	38.05	148	312	0	131
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 19/29 (65%)	200	0	6266	17140	0	7549	178	191	2	54.45	49.14	405	824	0	295
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 22/29 (75%)	500	0	15091	43303	0	19797	406	257	2	60.17	61.56	699	1413	0	522
LinkedList Iterator ListIterator	1/ 1 (100%) 2/ 2 (100%) 2/ 2 (100%)	19/19 (100%) 9/11 (81%) 24/29 (82%)	1000	0	39391	113155	0	52461	825	257	2	74.66	67.19	1404	2819	0	1083
ZKServer ZKClient	4/ 4 (100%) 9/13 (69%)	4/ 4 (100%) 28/54 (51%)	10	0	488	536	0	6	10	27	17	24.60	2.65	158	203	0	6
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 38/54 (70%)	100	7	4628	5160	7	76	98	31	4	22.57	5.99	862	1110	5	75
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 39/54 (72%)	200	9	9869	9869	9	138	197	31	4	22.88	5.67	1532	1964	5	135
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 40/54 (74%)	500	26	27208	31918	25	325	480	31	4	22.79	5.31	3057	3910	10	320
ZKServer ZKClient	4/ 4 (100%) 11/13 (84%)	4/ 4 (100%) 43/54 (79%)	1000	47	63524	76090	44	648	937	31	4	23.01	5.07	5719	7201	16	643

We use the Java server socket model to further discuss our experimental results based on the graphs obtained. The static visualization of the EFSM (see Fig. 9) shows the transition system and uses red edges to show expected exceptions, since the notion of failed tests does not apply. After applying abstractions, Fig. 10 shows the SG and PG for the Java server socket model with ten test cases executed, including failed transitions in red and labeled with (f).

Compared to the EFSM in Fig. 9, the SG in Fig. 10 shows the concrete executions instead of possible executions as shown by the EFSM. We see from the SG that all states have been visited after ten test cases; the SG also provides

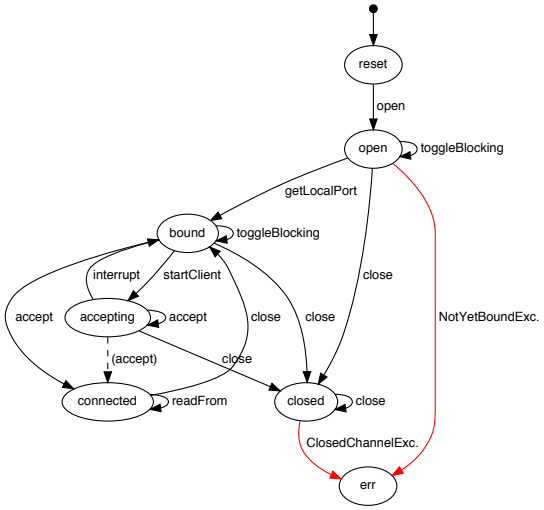


Fig. 9: EFSM for the Java server socket model.

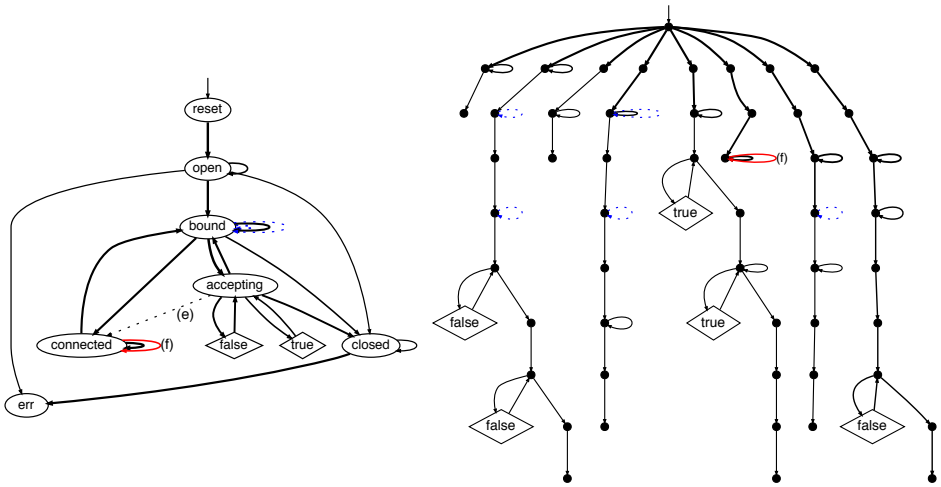


Fig. 10: SG (left) and PG (right) for the above model after ten tests.

information about possible exceptions and failures occurred, actual paths and choices taken; the edge thickness indicates how often transitions were taken.

A good path-coverage-based testing strategy requires that the test cases execute as many linearly independent paths as possible. For the PG in Fig. 10, we can directly see that there are eight linearly independent paths. Each linearly independent path has a sequence of edges which represent executed transitions of the path. This gives us a simpler way of showing the paths as transition sequences, at the expense of a graph that has more nodes and edges overall. In addition, all loops, backtracked edges and taken choices are directly shown with their related linearly independent paths in the PG, and there is one linearly

independent path which shows a failed test in the graph. Also, like the SG, the edge thickness in the PG indicates how often transitions were taken.

To show how our abstraction reduces the complexity of graphs, we use the Java server socket model as an example. Fig. 11 shows the FSG without applying any abstractions for the Java server socket model with 10 test cases executed. This graph should be contrasted with the SG shown in Fig. 10 (left). From this FSG, we notice that the FSG has many redundant edges between both state nodes and choice nodes, and it also has more choice nodes, as opposed to the SG in Fig. 10. Here, we do not show the FPG for the Java server socket model due to space limitations, but we give the detailed comparison between the SG and FSG and between the PG and FPG for the Java server socket model in Table 2. For instance, with 1000 test cases, the PG has three times fewer edges than the FPG; the SG has only 11 nodes and 27 edges, as compared to 61 nodes and 5491 edges in the FSG. This comparison shows that with the help of abstractions, the SG and PG are much more concise and less complex than the FSG and FPG.

7 Related Work

Coverage analysis is an important concern in software testing. It can be used as an exit criterion for testing and deciding whether additional test cases are needed, and related to which aspects of the SUT. For source code coverage, tools generally only report a verdict on which line of code has been executed how often. In the tool Tecrevis, a visual representation of redundancy in unit

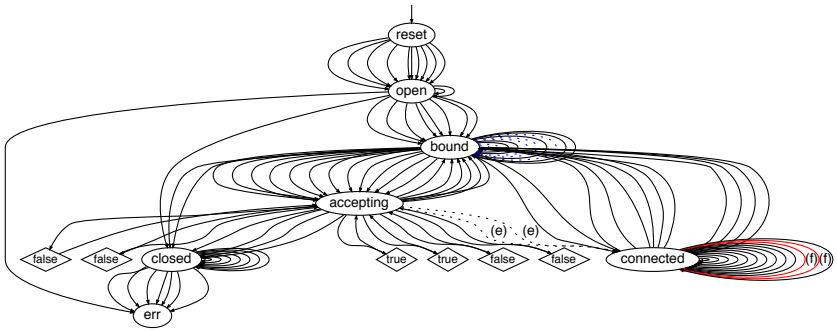


Fig. 11: FSG for the Java server socket model after ten tests.

Table 2: Comparison between PG and FPG, and between SG and FSG of the Java server socket model.

Model	TC	PG				FPG				SG				FSG			
		Nodes	Edges			Nodes	Edges			Nodes	Edges			Nodes	Edges		
			E	FE	L		E	FE	L		E	FE	L		E	FE	L
JavaNio ServerSocket	10	57	79	1	17	70	97	2	21	9	23	1	6	13	109	2	31
	100	177	243	1	48	376	497	3	100	9	23	1	6	15	545	3	145
	200	363	528	4	111	811	1101	8	221	10	25	1	7	26	1239	8	353
	500	779	1147	8	247	1943	2613	14	520	11	27	1	8	40	2910	14	816
	1000	1269	1904	15	439	3721	4982	28	996	11	27	1	8	61	5491	28	1510

tests provides a graphical mapping between each test case and the artifacts in the SUT (here: methods) that indicates which tests exercise the same component [13]. In path coverage, the underlying graph is usually derived from the source code, the *control flow graph*, or from the *call graph* of the SUT when considering function calls. In our approach, we are not directly concerned with visualizing paths of the SUT, but rather, paths on the testing model used for test-case generation. Correspondingly, our graphs are usually more concise than the control flow graph, as not all branches of the SUT may need to be modeled at the level of ESFMs. In particular, with respect to related work and the coverage analysis domain, visualization is usually an orthogonal concern to quantifying coverage, and not often considered.

Visualization makes coverage information understandable. Ladenberger and Leuschel address the problem of visualizing large state spaces in the ProB tool [14]. They introduce *projection diagrams,* which through a user-selectable function partition the states into equivalence classes. A coloring scheme for states and transitions indicates whether the state space has been exhausted, or all collapsed transitions share the same enablement. As their diagrams are based on the actually explored state space, they do not directly visualize coverage of the underlying model as in our approach. Moreover, they do not cover multiple transitions between the same pair of states as in our application scenario; however, this could be accounted for by adjusting the thickness of edges by the number of collapsed edges. Similarly, Groote and van Ham [10] applied an automated visualization to examples from the Very Large Transition System (VLTS) Benchmark set [8]. A relation between the graphical representation of the underlying model (in the form of UML sequence diagrams) and a set of paths from test cases is presented by Rountev et al. [19]. Their goal is deriving test cases, and as such they are not concerned with a representation of the paths.

The basic visualization elements of both SG and PG we have defined in this paper are based on the concept of *simple path* proposed by Ammann and Offutt [1]. An execution path is a simple path if there are no cycles in this path, with the exception that the first and last states may be identical (the entire path itself is a cycle) [1]. Based on this definition, any execution path can be composed of simple paths. Therefore, in this paper, the concept of the simple path is applied by considering only transitions from $s_{origin}(t)$ to $s_{dest}(t)$ (or $s_{origin}(t)$ if t is a self-transition or backtracked transition).

8 Conclusions and Future Work

The main contribution of this paper is to present an approach to capture and visualize test-case-execution paths through models. This is achieved by first recording execution paths with a trie data structure, then visualizing them using state-based graphs (SGs) and path-based graphs (PGs) obtained by applying abstractions. The SG conveys the behavior of the model well. The PG only shows executed paths, without providing detail. It avoids crossing edges, which makes

the PG more scalable, even though it contains more nodes and edges as such. Also, the PG directly indicates the number of linearly independent paths.

To obtain the SGs and PGs, we have proposed abstractions as our initial technique to reduce the size and complexity of graphs. We have implemented our approach as a path coverage visualizer for the Modbat model-based API tester. An experimental evaluation on several model-based test suites shows that our abstraction technique reduces the complexity of graphs, and our visualization of execution paths helps to show the frequency of transitions taken by the executed paths and to distinguish successful from failed test cases.

Future work includes investigating other techniques and tools to support more visualization features in the SGs and PGs, using more abstractions for the further reduction of larger graphs and applying the SGs and PGs not only for visualizing execution paths of models, but also for the SUT. Another direction of future work is to investigate approaches to perform state space exploration efficiently for selecting good test suites and visualizing execution paths. Although our current visualization approach has been applied to the Modbat tester, it is also possible to use it for other testing platforms. Furthermore, additional coverage metrics such as branch-coverage of boolean subexpressions within preconditions and assertions, or the more detailed *modified condition/decision coverage* (MC/DC) [7] could be used to refine the intermediate execution steps even further. In essence, many of the coverage techniques available at the SUT-level could be lifted to the model level.

References

1. P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, 2016.
2. C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Haifa Verification Conference*, volume 8244 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2013.
3. C. Artho, G. Rousset, and Q. Gros. Precondition coverage in software testing. In *Proc. 1st Int. Workshop on Validating Software Tests (VST 2016)*, Osaka, Japan, 2016. IEEE.
4. AT&T Labs Research. Graphviz - Graph Visualization Software. `https://www.graphviz.org`.
5. P. Brass. *Advanced Data Structures.* Cambridge University Press, 2008.
6. K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th Intl. Design Automation Conf.*, DAC, pages 86–91, Dallas, USA, 1993. ACM.
7. J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.
8. CWI and INRIA. The VLTS benchmark suite. `https://cadp.inria.fr/resources/vlts/`, 2019. Last accessed: 2019-05-20.
9. E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot. `http://www.graphviz.org/pdf/dotguide.pdf`, 2006.
10. J. F. Groote and F. van Ham. Interactive visualization of large state spaces. *Intl. Journal on Software Tools for Technology Transfer*, 8(1):77–91, Feb 2006.

11. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In P. Barham and T. Roscoe, editors, *2010 USENIX Annual Technical Conference*. USENIX Association, 2010.
12. P. C. Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013.
13. N. Koochakzadeh and V. Garousi. Tecrevis: a tool for test coverage and test redundancy visualization. In *Intl. Academic and Industrial Conf. on Practice and Research Techniques (TAIC PART)*, volume 6303 of *Lecture Notes in Computer Science*, pages 129–136. Springer, 2010.
14. L. Ladenberger and M. Leuschel. Mastering the visualization of larger state spaces with projection diagrams. In M. J. Butler, S. Conchon, and F. Zaïdi, editors, *Formal Methods and Software Engineering - 17th Intl. Conf. on Formal Engineering Methods, ICFEM 2015*, volume 9407 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2015.
15. J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel. How well do professional developers test with code coverage visualizations? An empirical study. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 53–60. IEEE, 2005.
16. S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *Proc. of the 39th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 38–52. IEEE Computer Society, 2006.
17. G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.
18. Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne. The Scala Programming Language. `https://www.scala-lang.org`.
19. A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th Intl. Conf., FASE 2005*, volume 3442 of *Lecture Notes in Computer Science*, pages 289–304. Springer, 2005.
20. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
21. M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22:297–312, 2012.
22. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.