State Merging for Concolic Testing of Event-driven Applications

Maarten Vandercammen[a], Coen De Roover[a]

[a] *Vrije Universiteit Brussel, Pleinlaan 2, Brussels, 1050, Belgium*

Abstract

Symbolic execution has proven itself a successful technique for automatically testing applications. However, it suffers from the *state explosion* problem, where execution of the program generates an exponential number of execution states that must be explored to fully cover the program. This problem can be mitigated by incorporating *state merging* into the testing procedure, where execution states that are sufficiently similar are merged together.

Although state merging has been applied successfully to the testing of sequential applications, testing of event-driven applications brings with it unique challenges. In event-driven programs, the creation of new execution states is driven by both branch conditions encountered by the executor and the various permutations of the application's event sequence, as every event sequence potentially gives rise to a unique set of execution states that are created. This article presents the first description of how state merging can be applied in the context of symbolic execution of event-driven applications.

Furthermore, although state merging has been described extensively for *online* symbolic execution, no mechanism has yet explicitly been described for incorporating state merging into *offline* symbolic execution, such as concolic testing. Online symbolic executors enable forking of the execution state upon reaching a branch condition that depends on symbolic input. Such testers can explore multiple program paths simultaneously. They can hence fork and merge states whenever the opportunity arises. Offline symbolic execution on the other hand, explores each program path separately. The reduced flexibility complicates the application of state merging. This article provides the first explicit overview of how state merging can be incorporated in concolic testing.

We have implemented this approach to state merging in a concolic tester for event-driven JavaScript applications named StackFul. We evaluate this tester on a limited set of eight small, event-driven JavaScript web applications, and find that, within the scope of these eight applications, state merging successfully alleviates the state explosion problem, resulting in the concolic tester covering a larger part of the application in fewer test iterations.

Keywords: Concolic Testing, State Merging, Event-driven Applications

1. Introduction

In recent years, software testing techniques based on dynamic symbolic execution have risen to prominence. Their ability to complement symbolic reasoning over a program with concrete values that were observed while executing the program, combined with general advancements in SMT solving, enable these tools to achieve high code coverage. Examples of successful applications of dynamic symbolic execution include KLEE [9], PEX [31], SAGE [17], and Mayhem [5].

Testing tools based on these techniques test programs by iteratively performing concrete and symbolic execution simultaneously. Symbolic execution enables these testers to collect path constraints over the program's inputs. By mutating these constraints and having an SMT solver compute solutions for the program inputs that satisfy these new constraints, the tester can systematically explore new paths through the program. Executing the program concretely as well enables the tester to substitute symbolic expressions for their observed concrete value in cases where the expression is incomplete or computationally infeasible to solve.

Approaches to dynamic symbolic execution can be categorised as either *online* or *offline* (i.e., concolic) approaches [11, 7, 6]. Offline approaches explore each program path separately and restart the program between test runs. Online approaches on the other hand allow for exploring multiple program paths simultaneously by forking execution whenever a branch condition dependent on program input is encountered. Online approaches hence avoid re-executing the same program instructions that appear in multiple paths. However, this comes at the cost of increased memory consumption, as the tester has to keep multiple execution states in memory.

Both approaches suffer from the *state explosion problem*, where a linear number of conditional branches may give rise to an exponential number of execution states that must be covered. In order to address the state explosion problem, symbolic execution testers apply *state merging*, where they merge states that are sufficiently similar. Although this reduces the number of execution states, it increases the burden of the SMT solver, as the solver now has to reason over program constraints that feature increasingly complex symbolic expressions. Nevertheless, state merging can result in a net improvement in the time needed for a tester to explore a program [6].

For this reason, state merging has been applied to online symbolic execution of sequential programs. In non-sequential applications such as event-driven programs, however, non-determinism arises not only from the values for program inputs but also from the random order in which events, and their corresponding event handlers, may be triggered. The number of execution states that are created therefore grows even more quickly in event-driven programs, as every permutation of a program's event sequence may also give rise to the creation of different execution states. When merging two execution states in an event-driven program, the state merging algorithm therefore has to control for paths that fork because of branch conditions and those that fork because of different event orderings.

Furthermore, although state merging has been described extensively for online dynamic symbolic execution, to the best of the authors' knowledge, state merging has not yet been described for offline dynamic symbolic execution approaches, such as concolic testers. State merging for offline approaches raises additional challenges that are not apparent in online approaches as execution states cannot simply be cloned whenever necessary.

In this article, we present an approach for incorporating state merging into offline symbolic executors, such as concolic testers, and describe how state merging can applied to the testing of event-driven programs. More specifically, we implement state merging into StackFul, an existing concolic tester for JavaScript web applications [32]. The implementation of our approach in StackFul is available at `https://github.com/softwarelanguageslab/StackFul`. Our work makes the following contributions:

- We provide the first explicit description of how state merging can be incorporated into *offline*, i.e., concolic, symbolic execution and present a pseudocode implementation of our approach.
- We present a design for the integration of state merging into a concolic tester for event-driven JavaScript applications. We instantiated the design in StackFul, an existing concolic tester for event-driven JavaScript web applications.
- We evaluate this novel form of state merging on eight event-driven JavaScript web applications, and compare with a baseline version of the tester that does not incorporate state merging.

The remainder of this article is structured as follows. Section 2 presents background information on concolic testing of event-driven applications. Section 3 gives an overview of state merging for online symbolic execution of sequential applications that are not event-driven. Section 4 continues by transposing state merging to the setting of concolic testing. Section 5 takes this further by describing how state merging can be applied by concolic testers to event-driven applications. Section 6 evaluates the impact of state merging on a suite of eight small, event-driven JavaScript web applications, and Section 7 discusses these results. Section 8 presents technical limitations faced by our current prototype implementation. Section 9 describes related work, and Section 10 concludes.

2. Background

Concolic testing is a form of offline white-box fuzzing that tests a program over multiple test runs. Concolic testing uses symbolic execution to collect so-called *path constraints*. A path constraint describes the program conditions, expressed in the form of *symbolic expressions*, that must hold in order to follow a particular path through the program or code snippet under test.

By systematically negating these path constraints and using a satisfiability-modulo-theories (SMT) solver to compute solutions for program inputs that cause the negated constraint to become satisfied, the concolic tester can systematically exercise a new program path in each test run [6, 12]. In order to collect these path constraints, the concolic tester performs *concrete* and *symbolic* execution simultaneously. On the one hand the tester steers the *concrete* execution of the program along an extension of a previously-explored path and reports newly-encountered errors. On the other hand the *symbolic* execution gathers symbolic constraints over existing and newly-encountered non-deterministic variables (e.g., random values), user inputs, and program inputs that steer the program's execution so that subsequent test runs may explore different program paths.

Test run	Path constraint	x	y
1	$2y_0 \neq x_0$	3	5
2	$2y_0 = x_0 \land x_0 \leq y_0 + 10$	2	1
3	$2y_0 = x_0 \land x_0 > y_0 + 10$	30	15

Table 1: Summary of the three test runs performed during concolic testing of the code in Listing 1.

2.1. Concolic Testing of Sequential Applications

We illustrate the workings of a concolic tester via the JavaScript example depicted in Figure 1. Lines 14 and 15 assign random values to the variables `x` and `y`. In general, an expression that may evaluate to different concrete values over multiple test runs of the program, such as an expression producing a

```
1  function twice(v){
2    return v * 2;
3  }
4  function f(x, y){
5    let z = twice(y);
6    if(z === x) {
7      if(x > y + 10) {
8        throw new Error();
9      }
10   }
11 }
12 let x = randomInt();
13 let y = randomInt();
14 f(x, y);
```

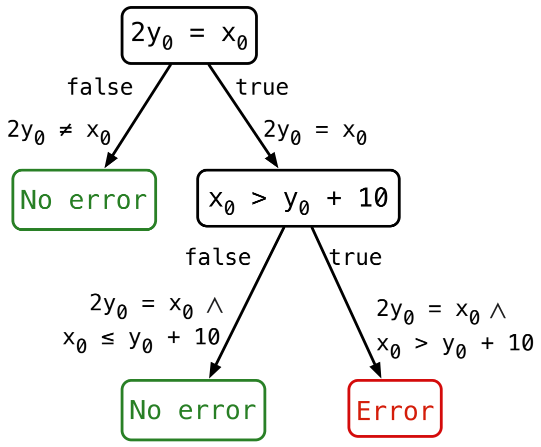


Figure 1: JavaScript program and its corresponding symbolic execution tree, based on Figure 1 of [8].

random value, reading a user or program input, performing an IO read, etc., is deemed a non-deterministic expression. A non-deterministic expression symbolically evaluates to a new symbolic input variable. The tester therefore symbolically represents the results of the calls to `randomInt`[1] as the symbolic input variables x_0 and y_0 respectively, which are then assigned to variables `x` and `y`.

Suppose that in the first test run, the concolic tester randomly assigns the values `3` to `x` and `5` to `y`. These values cause the branch condition on line 6 to be false, and the program terminates without errors. Simultaneously to this concrete execution, the tester also collects the symbolic representation of the conditional expression that was encountered on line 7 in the form of a so-called *path constraint*, i.e., $2y_0 \neq x_0$. After completing this run, the tester attempts to explore another path, such as the path leading to the `if` statement on line 7. To this end, the tester negates the path constraint and feeds the resulting constraint $2y_0 = x_0$ to an SMT solver, which finds a solution that assigns e.g., 2 to x_0 and 1 to y_0. The concolic tester re-executes the program and assigns the values `2` to `x` and `1` to `y`. Concrete execution reaches the `if` statement on line 8, then takes the `else` branch, and the program terminates again without errors. In the meantime, the symbolic

[1]JavaScript does not have a built-in function that returns a random integer value. For the purpose of this article, we therefore consider `randomInt` to be a hypothetical built-in function that returns a random integer value between `0` and `Number.MAX_SAFE_INTEGER`.

execution gathered the path constraint $2y_0 = x_0 \wedge x_0 \leq y_0 + 10$. The tester negates the last element of this path constraint and feeds the resulting path constraint $2y_0 = x_0 \wedge x_0 > y_0 + 10$ into the SMT solver, which finds e.g., the values 30 and 15 for `x` and `y` as a solution. A new test run is started with these values and the concrete execution reaches the error on line 8, which is reported by the concolic tester. As no new branches were encountered by the tester during this last run, the tester deduces that it has explored all feasible program paths and terminates. In practice, for realistic programs with a (near-)infinite number of program paths, the testing phase is terminated either upon exceeding a given time or search budget or when the desired level of code coverage has been reached. The three path constraints that were found can be collected in a symbolic execution tree that represents all possible executions of the program, as shown in Figure 1. A summary of the three concolic execution runs is presented in Table 1.

A concolic tester is often divided into a *test executor* and a *test selector* [20]. The test executor is the component that executes the program concretely and symbolically and that collects the path constraint over the course of a test run. The test selector constructs the symbolic execution tree from these path constraints over multiple test runs. Between test runs, it selects which path should be explored by the test executor in the subsequent run.

2.2. State Explosion in Sequential Applications

Constructing the symbolic execution tree naively gives rise to the problem of *state explosion*, where an exponential number of execution states are created and added to the tree. Consider Listing 1, which consists of four sequential, non-nested `if` statements. The predicates of the first three `if` statements are all independent from each other. On the other hand, the last `if` statement depends entirely on the first three `if` statements: its predicate depends on the values of the variables `a`, `b`, and `c`. Execution can hence only follow its `then` branch if all three previous `then` branches were followed.

The number of unique program paths that can be followed through this snippet, corresponding to the set of unique values that variables `a`, `b`, and `c` can assume, equals $2^3 = 8$ paths. Figure 2 depicts the symbolic execution tree for Listing 1, with unreachable nodes coloured black. In this figure, the left child edge of a node always corresponds to the `else` branch of the node's associated `if` statement, and the right edge corresponds to its `then` branch. In this tree, the symbolic predicate for the first `if` statement occurs only once, the symbolic predicate for the second `if` statement occurs twice, the third predicate is listed four times, and the last node, with predicate `a + b + c = 3`, is repeated eight times. For clarity, we also include the leaf

```
1  let a, b, c;
2  if (randomInt() === 0) {
3    a = 1;
4  } else {
5    a = 0;
6  }
7  if (randomInt() === 1) {
8    b = 1;
9  } else {
10   b = 0;
11 }
12 if (randomInt() === 2) {
13   c = 1;
14 } else {
15   c = 0;
16 }
17 if (a + b + c === 3) {
18   // program ends, via then branch
19 } else {
20   // program ends, via else branch
21 }
```

Listing 1: A small code snippet that results in an exponential increase in the number of program paths.

nodes `END T` and `END F` which indicate that program execution terminated either on line 18 or at 20. As traditional concolic testing explores each path separately, a traditional concolic tester hence requires eight test runs to explore all feasible paths through the snippet.

For the leftmost seven occurrences of the `a + b + c = 3` node, the child node that corresponds to their `then` branch (coloured black in the figure) is unreachable, as these occurrences correspond to program paths where at least one of the variables `a` through `c` is assigned `0`. For these seven nodes, execution can only reach their `else` branch. This situation is reversed for the rightmost occurrence of the `a + b + c = 3` node, where execution is forced along the `then` branch and the `else` branch is unreachable.

The state explosion arises from the fact that execution splits immediately after reaching the first `if` statement (i.e., the root node with symbolic predicate $i_0 = 0$) into two new paths: one that is created by following the `then` branch and another that is created by following the `else` branch. Not only do these two paths stay separate for the remainder of the program's execution, but they each subsequently split several times into sub-paths upon encountering each next `if` statement. Had these two initial paths been rejoined after their split at the $i_0 = 0$ node but before their next split at the $i_1 = 1$ node, the symbolic execution tree would have remained fairly small,

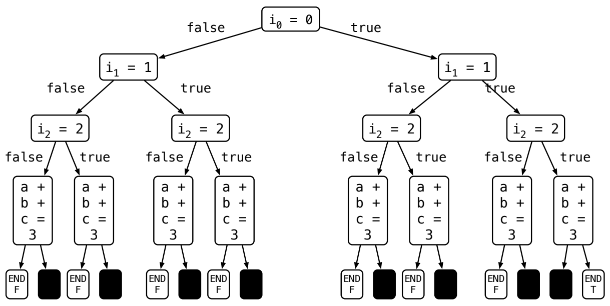


Figure 2: The symbolic execution tree for the code snippet in Listing 1.

instead of exploding into eight different `END` leaf nodes and an additional eight unreachable nodes.

2.3. Concolic Testing of Event-driven Applications

The previous section gave an overview of how concolic testing is performed on sequential programs. When testing event-driven code, the concolic tester must not only consider the conditional branches in the program, but also the sequences of events that may arise in the program's execution. Consider the event-driven code snippet in Listing 2 that one may find on a website. This snippet registers three event handlers, one for each of three buttons that are rendered on the site.

```
let b0 = 0, b1 = 0;
function b0Fun() {
  b0++;
}
function b1Fun() {
  b1++;
}
function b2Fun() {
  if (b0 > 0) {
    if (b1 > randomInt()) {
      foo();
    } else { ... }
  } else { ... }
  ...
}
document.getElementById("Button0").addEventListener("click", b0Fun);
document.getElementById("Button1").addEventListener("click", b1Fun);
document.getElementById("Button2").addEventListener("click", b2Fun);
```

Listing 2: An event-driven code snippet with three event handlers.

The exact behaviour of this code snippet depends on the order and the number of times these buttons are clicked, i.e., the sequence of events that are generated for the program. In fact, apart from the registration of these event handlers, no code is executed at all *until* a click event is triggered for a button. A complete exploration of all possible program paths must therefore consider both the sequence of events to be followed and the values of program inputs that may appear in the program, such as the call to `randomInt` on line 10.

To systematically explore all paths through an event-driven program, it is hence necessary to add to the symbolic execution tree the paths through the various event handlers and to encode the decisions of which event was triggered at a point in the program's execution. One possible extension to the symbolic execution tree is shown in Figure 3, which depicts part of the symbolic execution tree for Listing 2.

2.3.1. Extending the Symbolic Execution Tree

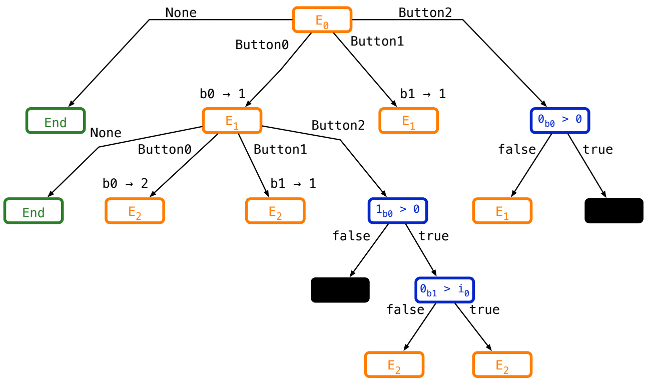


Figure 3: A partial symbolic execution tree for the code snippet in Listing 2. Conditional branching nodes are coloured blue, event branching nodes orange, end nodes green, and unreachable nodes black. For brevity, the figure includes only one subtree for an E_1 event branching node, while the subtrees for all E_2 nodes are omitted entirely.

In this extended symbolic execution tree, we distinguish between two different kinds of branching nodes. *Conditional branching nodes*, depicted as blue nodes in the figure, represent a point where control flow splits because of a branching condition such as an `if` statement or `while` loop. An example includes the node $0_{b0} > 0$ which was created when executing line 9. Note that the symbolic value 0 is subscripted with the name of the identifier that was read. *Event branching nodes*, depicted as orange nodes, represent a point in the execution where a new event can be triggered. Event branching nodes are of the form E_i, where i is the index of the corresponding event in the event sequence. These nodes determine which of the currently active event handlers should be executed next, i.e., for which currently active event handler the corresponding event should be triggered. The green `End` nodes indicate that program execution has ended, while black nodes are unreachable.

In the case of JavaScript, event handlers are *atomic*, i.e., no two event handlers can be executed simultaneously: a new event handler can only be executed when the previous handler has been completed [33]. This implies that the tester should only consider triggering a new event when no event handler is currently being executed. Event branching nodes are therefore only added at a point in the path in the symbolic tree where execution has already left an event handler.

Consider again Listing 2. In this program, no code is executed outside of the event handlers, apart from registering these handlers via calls to `button.addEventHandler`, which does not involve branching of either kind. The first branching point of the program, i.e., the root node of the tree depicted as event E_0 in Figure 3, is therefore an event branching node representing four choices: executing one of the three event handlers registered to a click on its corresponding button, or *not* invoking any event handler at all, in which case the program terminates immediately. If `button2` is clicked, the execution splits again into two possible paths, based the branch condition on line 9.

Whenever a button is clicked, once its corresponding event handler completes its execution, the previous four-fold choice is presented again for a potential second event E_1. In general, every event branching node is followed by $n + 1$ child nodes, where n is the number of event handlers that are active at the corresponding point in the program's execution.

Note that this extension implies that the size of the tree in an event-driven program is infinite, as the choice of which event handler to execute next can be repeated indefinitely. Indeed, every possible event sequence corresponds to a unique path through the tree. Since an event sequence may in theory grow to an infinite length, there are an infinite number of unique paths through the tree, even if many of these program paths do not reveal any "new" program behaviour.

2.4. State Explosion in Event-driven Applications

In event-driven programs, the number of execution states included in the symbolic execution tree grows exponentially both in function of the number of branch conditions and in function of the length of the event sequence. To illustrate the latter, for every i, every event branching node of the form E_i in Figure 3 can be considered a duplicate of another as they give rise to the same $n + 1$ branching. Thus, the only instance of the E_0 event branching node gives rise to three E_1 nodes, which further results in nine E_2 nodes and so on.

The duplication of both conditional branching nodes and event branching nodes conspire to exacerbate the state explosion problem in event-driven applications. Consider for example line 14 in Listing 2. For every invocation of event handler `b2Fun`, the execution state corresponding to this line will be duplicated in the tree thrice, as an instance of this state may be encountered after having executed the `else` branch of the `if` statement on line 9, or after having executed first the `then` branch on line 10 and then either the `then` branch or the `else` branch. However, as every occurrence of an E_i event branching node may give rise to an invocation of the `b2Fun` handler, the total number of occurrences of this state corresponds to $3N_E$, where N_E is the number of event branching nodes currently in the symbolic execution tree.

3. State Merging for Online Symbolic Execution

Broadly speaking, *execution states* correspond to the nodes in the symbolic execution trees presented in the previous section, although we revisit this definition of states in Section 3.2. For example, an occurrence of a node $i_2 = 2$ in the tree depicted in Figure 2 corresponds to a particular execution state that was created when execution reached the third `if` statement along a particular program path.

States may include path constraints, so an exponential increase in the number of paths (*"path explosion"*) corresponds to a likewise exponential increase in the number of states (*"state explosion"*) and vice versa [21]. The two terms are hence interchangeable. State merging alleviates the state explosion problem by merging program states that are sufficiently *similar* together. Instead of having states in the symbolic execution tree split whenever a branch condition is encountered, two or more states can be merged together, thereby also halving the total number of states that descend from these two states.

To illustrate this, consider the symbolic execution tree depicted in Figure 4 which features four program paths, i.e., four `END` states, when executed symbolically. Figure 5 depicts the result of merging the two states that feature the constraint $i_1 = 1$ together. By performing one state merge operation, the number of `END` states is halved.

Note that after applying the merge operation, the symbolic execution tree has been transformed into a symbolic execution directed acyclic graph (DAG). Furthermore, although the number of states has been reduced, no information on the path constraints through the DAG has been lost: the path constraints for the two `END` states now feature a logical disjunction that includes both branches of the root $i_0 = 0$ state. These two path constraints hence still precisely encode the information that was contained in the four path constraints for the `END` nodes of the unmerged symbolic execution tree. Although the number of `END` states has been reduced as a result of the merge operation, the complexity of their path constraints has hence increased.

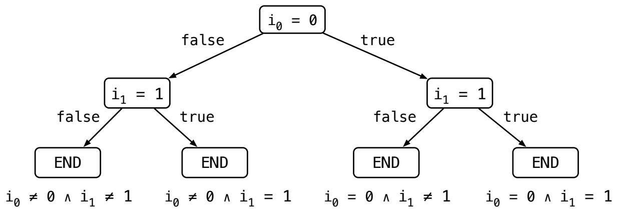


Figure 4: A symbolic execution tree with four program paths.

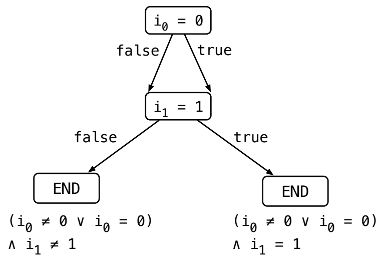


Figure 5: After merging two states together, the number of `END` states is halved.

3.1. Alleviating State Explosion through State Merging

Recall that the path explosion problem can be cast as a state explosion problem, which in turn can be solved by merging *similar* states together, i.e., states that correspond to the same `if` statement in the tree, rather than duplicating a state for each program path that reaches this statement. Figure 6 depicts what the symbolic execution DAG for Listing 1 would be if all states that correspond to the same `if` statement in the code were merged together.

Each program path gives rise to a unique combination of values assigned to the variables `a` through `c`. The difference in values for these variables is significant when the tester explores both branches of the last `if` statement on line 17 of Listing 1. Of the eight program paths that encounter this `if` statement, only one path visits the `then` branch, since only one path sets the value of all three variables to `1`. When joining states together into one merged state, the tester must hence be capable of precisely modelling the different values that each of the three variables assumes depending on whether the `then` branch or `else` branch was taken when assigning the variable.

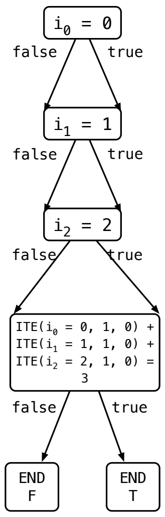


Figure 6: The symbolic execution DAG for Listing 1 with state merging.

State merging therefore employs symbolic *if-then-else* (ITE) expressions to model symbolic values that depend on some condition. For example, the value of the variable `a` can be represented as $ITE(i_0 = 0, 1, 0)$: this value evaluates to 1 if the predicate $i_0 = 0$ is true, and evaluates to 0 otherwise. Using these ITE expressions, the tester can represent the predicate of the last `if` statement as $ITE(i_0 = 0, 1, 0) + ITE(i_1 = 1, 1, 0) + ITE(i_2 = 2, 1, 0) = 3$.

Note that these ITE expressions may be computationally expensive to solve for an SMT solver, especially when they are included into path constraints that already feature long chains of logical disjunctions (as depicted in Figure 5). The increased complexity of these path constraints increases the burden on the SMT solver that must compute appropriate values for the symbolic input variables that are featured in these paths. In effect, state merging reduces the time that the tester spends exploring duplicate states, but increases the solving time of the SMT solver.

3.2. A Formal Definition of State Merging

State merging can be defined formally in terms of state triples $\langle P, PC, \sigma \rangle$, where PC represents the path constraint that leads to this state in the tree, σ the symbolic store mapping program variables to symbolic values, and P the *program point*. The program point encodes the current point in the program, e.g., a specific `if` statement, that was being executed when the state was created.

3.2.1. The State Merge Operation

The state merge operation $\sim$ is then defined on two states $\langle P, PC_1, \sigma_1\rangle \sim \langle P, PC_2, \sigma_2\rangle = \langle P, PC_1 \vee PC_2, \sigma_m\rangle$, where σ_m is defined as $\forall v \in \sigma_1 : \sigma_m[v] = \mathit{ITE}(PC_1, \sigma_1[v], \sigma_2[v])$.

In effect, this merged triple defines a state which can be reached by following either path constraint PC_1 or PC_2. Its symbolic store σ_m maps every program variable to an ITE expression. If the path constraint PC_1 of the first state evaluates to `true`, this ITE expression itself evaluates to the symbolic value observed by the first state and kept in its symbolic store σ_1. Note that this definition assumes that both symbolic stores include the exact same variables, though their values may be different.

If this merged state is merged again with another state $\langle P, PC_3, \sigma_3\rangle$, we apply the same merge operation. The merge operation then produces the state $\langle P, PC_3 \vee PC_1 \vee PC_2, \sigma_{m'}\rangle$, where $\sigma_{m'}$ is defined as $\forall v \in \sigma_3 : \sigma_{m'}[v] = \mathit{ITE}(PC_3, \sigma_3[v], \sigma_m[v])$, which is equivalent to $\forall v \in \sigma_3 : \sigma_{m'}[v] = \mathit{ITE}(PC_3, \sigma_3[v], \mathit{ITE}(PC_1, \sigma_1[v], \sigma_2[v]))$ Note that the three path constraints PC_1, PC_2, and PC_3 do not necessarily exhaustively cover all possible execution paths through a program. The negation of PC_3 and PC_1 therefore does not generally imply that PC_2 is true, and hence that any program variable `x` should receive the value σ_2[`x`] along a path that differs from PC_3 and PC_1. However, *with respect to the merged state's path constraint* $PC_3 \vee PC_1 \vee PC_2$, the symbolic store $\sigma_{m'}$ represents all three possibilities. As new states featuring the same program point are produced while testing the program, these states may also be joined together with this merged state.

3.2.2. Defining the Program Point

The merge operation is only applied to two states that share an identical program point P. This program point is defined only opaquely, but it is taken to refer to a specific point in the program's execution. In sequential programs without recursion or loops, such as the code snippet in Listing 1, it could be defined as just the code label (e.g., the line number) that was reached by the tester when the state was created. This makes it reasonable to assume that both stores include the same set of variables, as this definition of program points ensures that states that are merged together were created in the same lexical environment. We revisit the definition for program points in Sections 4.1.1 and 5.1.1.

3.2.3. Example of a State Merge Operation

As an example, consider the formal representation of both occurrences of the state $i_1 = 1$ from Listing 1. Defining program points as line numbers, the leftmost (as depicted in Figure 2) of these two states can be represented by the triple $\langle L_7, i_0 \neq 0, \sigma_1 \rangle$, where σ_1 is defined as a singleton map of the variable `a` to the symbolic value 0. The rightmost of these states is represented as the triple $\langle L_7, i_0 = 0, \sigma_2 \rangle$, with σ_2 mapping the variable `a` to the symbolic value 1. The result of the merge operation would then be defined as the triple $\langle L_7, i_0 = 0 \vee i_0 \neq 0, \sigma_m \rangle$, with σ_m mapping the variable `a` to the symbolic value $\mathit{ITE}(i_0 = 0, 1, 0)$.

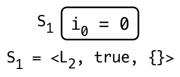
3.3. State Merging for Online Symbolic Execution

The formal definition of state merging can be used by any type of symbolic execution to merge two given states together. However, this definition is insufficient for understanding how state merging can be incorporated into the symbolic execution procedure at large, or how it affects the exploration of a program. We therefore give an overview of how state merging can be integrated with symbolic execution. As an introduction, we first detail how state merging can be incorporated into *online* symbolic execution of *sequential* applications, before describing how the state merging algorithm can be transposed to the setting of concolic testing (Section 4), and afterwards concolic testing of event-driven applications (Section 5).

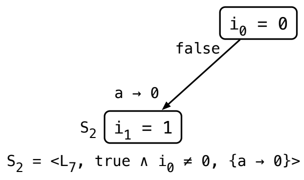
Online symbolic executors are symbolic executors that can fork their execution when reaching a branching point [6, 10]. These stand in contrast to *offline* symbolic executors, like concolic testers, which test an application by exploring one program path at a time. Online symbolic executors can select which state in the symbolic execution DAG to expand. Whenever the online executor has expanded a previously unexplored state in the symbolic execution DAG, it can compare its program point with that of all other explored states in the DAG. If it finds a previous state with an identical program point, the symbolic executor may opt to merge the two together before continuing its exploration of the remainder of the symbolic execution DAG.

Figure 7 illustrates an example of how an online symbolic executor can incorporate state merging while testing the code from Listing 1. As execution in this code snippet does not jump back to an earlier point in the program, via e.g., recursion or iteration, it is sufficient to represent program points as just the line number of the code statement being executed. For each new state that is added to the symbolic DAG, we list the triple representing that state.

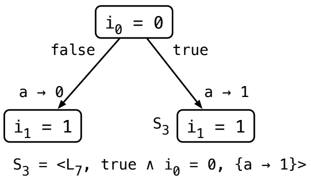
Apart from a `true` and `false` label, we also annotate the edges in the DAG with the assignments to program variables that were observed to have taken place when following the branch. We also annotate merged states where program variables have received a symbolic ITE expression as a result of the merge. An example of such a state annotation can be seen in e.g., Figure 7d, where the state annotation (coloured blue), indicates that the value of `a` is replaced by an ITE expression. The symbolic store for any state S can hence be constructed via the annotations along both the edges and the states on the path to S.



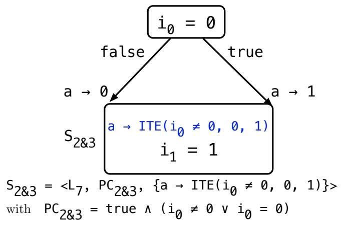
(a) After evaluating the branch condition on line 2, the root state is created.



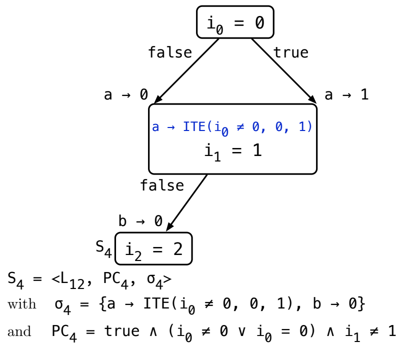
(b) The symbolic executor expands the `else` branch of the previous condition and adds a state corresponding to the branch condition on line 7.



(c) The symbolic executor expands the `then` branch of the previous condition and adds a state corresponding to the branch condition on line 7.


i0 = 0
false
true
a → 0
a → 1
S2&3
a → ITE(i0 ≠ 0, 0, 1)
i1 = 1
S2&3 = <L7, PC2&3, {a → ITE(i0 ≠ 0, 0, 1)}>
with PC2&3 = true ∧ (i0 ≠ 0 ∨ i0 = 0)

(d) Having expanded both child states, the symbolic executor immediately merges them.



(e) The symbolic executor continues testing the program by executing the `if` statement on line 12.

Figure 7: Online symbolic execution of the code in Listing 1, using state merging to prune states with duplicate program points.

The symbolic executor first adds a state S_1 for the first `if` statement of the code snippet (Figure 7a). As the root state in the symbolic DAG, the state features a path constraint with the default value `true` and an empty symbolic store.

It continues along the `else` branch of this `if` statement, injects the mapping from variable `a` to the symbolic value 0 into the symbolic store when observing the assignment to this variable, and then adds a new state S_2 $\langle L_7, \texttt{true} \wedge i_0 \neq 0, \{a \rightarrow 0\}\rangle$ to the symbolic DAG upon reaching the next `if` statement (Figure 7b). Before continuing to the child states of this `if` statement, the executor returns to the root state and explores the `then` branch of the first `if` statement. As before, the tester adds the mapping $\texttt{a} \rightarrow 1$ to its store and adds a duplicate state S_3 $\langle L_7, \texttt{true} \wedge i_0 = 0, \{a \rightarrow 1\}\rangle$ as the `then` child to the root (Figure 7c).

Since both states share the same program point, the symbolic executor merges them together into state $S_{1\&2}$. The path constraint of the merged state features the disjunction of both original states: $(\texttt{true} \wedge i_0 \neq 0) \vee (\texttt{true} \wedge i_0 = 0) \equiv \texttt{true} \wedge (i_0 \neq 0 \vee i_0 = 0)$. The variable `a` in the merged symbolic store is mapped to the symbolic value $\mathit{ITE}(i_0 \neq 0, 0, 1)$.

Upon completing the merge operation, the symbolic executor resumes normal symbolic execution of the program by exploring the `else` branch of the merged state. The constraint $i_1 \neq 1$ is added to the path constraint of the merged state, while the mapping $\texttt{b} \rightarrow 0$ is added to the symbolic store alongside the mapping for `a`.

Figure 8 depicts the final symbolic execution DAG that is produced once symbolic execution of the whole program finishes. The bottom state corresponds to the merged state for the `if` statement on line 17. The three variables that were used in this `if` statement have been replaced by a corresponding ITE expression. For clarity, we subscript ITE expressions that were produced as a result of reading a variable with the name of the variable that was inlined.

3.3.1. Practical Considerations

Note that it may not always be beneficial to merge states together. The introduction of symbolic ITE expressions increases the burden on the SMT solver, as it becomes more computationally intensive to solve queries containing these expressions [18, 28], especially since these ITE expressions may be nested. If merging introduces too many ITE expressions, the computational cost of solving queries may outweigh the cost of exploring duplicated program paths. Section 9 describes several techniques for finding a balance between the computational overhead introduced by state merging and the effort saved by the tester in exploring redundant paths. Of note is that it may be possible to heuristically predict the computational cost of replacing "concrete" symbolic values, such as 1 and 2, with more abstract symbolic

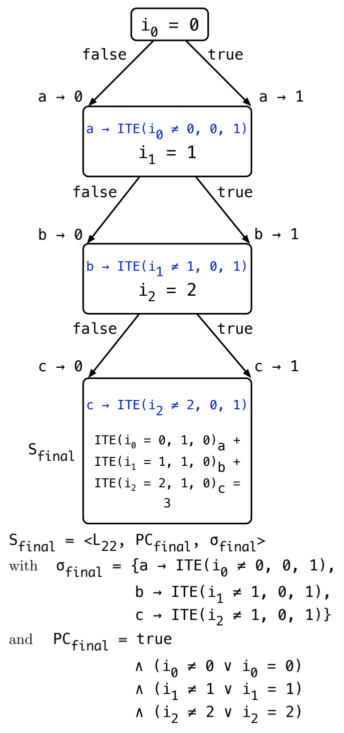

i0 = 0
false
true
a → 0
a → 1
a → ITE(i0 ≠ 0, 0, 1)
i1 = 1
false
true
b → 0
b → 1
b → ITE(i1 ≠ 1, 0, 1)
i2 = 2
false
true
c → 0
c → 1
c → ITE(i2 ≠ 2, 0, 1)
Sfinal
ITE(i0 = 0, 1, 0)a + ITE(i1 = 1, 1, 0)b + ITE(i2 = 2, 1, 0)c = 3
Sfinal = <L22, PCfinal, σfinal>
with σfinal = {a → ITE(i0 ≠ 0, 0, 1), b → ITE(i1 ≠ 1, 0, 1), c → ITE(i2 ≠ 1, 0, 1)}
and PCfinal = true ∧ (i0 ≠ 0 ∨ i0 = 0) ∧ (i1 ≠ 1 ∨ i1 = 1) ∧ (i2 ≠ 2 ∨ i2 = 2)

Figure 8: The complete symbolic execution DAG, produced after symbolic execution of the program has finished.

values such as $ITE(i_0 = 0, 1, 2)$ [21]. Merging then only takes place when the heuristic determines that the cost of replacing these values with an ITE expression is outweighed by the reduction in available program paths.

4. State Merging for Concolic Testing

The previous section gave an overview of how state merging is traditionally realised for online symbolic execution, where multiple program paths are explored simultaneously and where execution can be forked at every state. In this section, we give an overview of how state merging can be realised in the setting of an offline symbolic executor, i.e., a concolic tester, which is restricted to exploring a program one path per time.

4.1. Complications for Applying State Merging

The previous overview of state merging for online symbolic execution was aided by two factors:

- The program point could be represented as just a code label (e.g., the line number). In real-world applications, a code label does not suffice to precisely define the execution point of the state.

- The symbolic executor was able to backtrack to any previous state in the symbolic DAG, which in turn allowed for interleaving state merging and symbolic execution.

4.1.1. Defining Program Points

The code in Listing 3 demonstrates why the program point of a state cannot be modelled as just the code label of the corresponding branching point. The `if` statement on line 3 is reached via two different paths, namely the paths from functions `f` and `g`. Two different states would therefore be created in the symbolic execution tree to represent this branching point. If program points were to be defined as just the line number of the corresponding branching point, both states would be merged together. However, execution after this statement proceeds differently along both paths immediately upon returning from the `print` function. As the `if` statements encountered along both branches differ from each other, the structure of the subtrees of both states is also different. It may therefore be better to keep them separate and extend our example definition of program points to reflect the different nature of both states.

This problem can be solved by including the function stack in the program point. The program point for the state on line 3 that was reached via the function `f` is represented as the pair $\langle L_3, [\texttt{f}]\rangle$, whereas the program point for the same line reached via the function `g` is represented as $\langle L_3, [\texttt{g}]\rangle$. Since these program points are not identical, the symbolic executor does not consider the corresponding states as candidates for merging.

4.1.2. Interweaving Symbolic Execution and State Merging

Recall from the demonstration of state merging in the online symbolic execution of Listing 1 that the symbolic executor, after having explored the `else` branch of the root state (Figure 7b), backtracked to the root state to first explore the `then` branch of the root (Figure 7c), and then immediately joined both child states together (Figure 7d) before continuing the execution by exploring the `else` branch of the merged state (Figure 7e).

The symbolic executor was hence able to choose which states to explore in function of which states can be merged. This feature allows the symbolic executor to merge suitable child states as soon as they emerge, rather than first exploring an entire subtree for each state and having to merge these subtrees together. Backtracking hence enables the symbolic executor to avoid

```
let x, y;
function print(s) {
  if (...) {
    console.log(s);
  }
}
function f() {
  ... // elided
  print("f");
  if (x + y > 42) { ... } else { ... }
}
function g() {
  ... // elided
  print("g");
  if (randomInt() === 0) { ...} else { ... }
}
```

Listing 3: An example code snippet where it does not suffice to define a program point as just a code label.

having to merge entire subtrees. However, this backtracking is not possible for offline symbolic executors, such as concolic testers, as we will discuss below.

4.2. Mitigating the Path Determinacy Problem

In the context of offline symbolic execution, i.e., concolic testing, it is not possible to strictly interleave exploration of the symbolic execution DAG and merging of duplicated states. Conceptually, the problem is that the program is executed both concretely and symbolically, but symbolic execution is forced to follow alongside the concrete execution. Unlike online symbolic executors, concolic testers cannot halt exploration of a particular path, backtrack to an earlier state, and resume exploration from its alternative branch.

Recall from Section 2 that before each test run the test selector specifies a particular path to be explored in the run. The test selector then employs an SMT solver to compute appropriate concrete values for the program inputs appearing in the constraints along that path. When the test run commences, the concolic tester is forced to proceed along this exact path, without opportunities to deviate from it or to temporarily explore a different branch. We name this the *path determinacy problem*, as the path, and hence the states that it will encounter along the path, is selected before a test run is started and must be followed until completion.

In practical terms, this means that the tester is limited to merging the states that it will find along the preselected path. To partially mitigate this problem, our state merging algorithm mimics that of merging in online symbolic execution as closely as possible, by incrementally adding a new path to the DAG in segments, rather than adding the whole path in one operation as described in Section 3. Whenever a partial path has been added to the DAG, the algorithm attempts to find a duplicate state.

The test selector still determines which path must be explored in the next test run, but the test executor (cf. Section 2.1) attempts to perform a merge every time a new state is created, i.e., whenever it observes a branching condition. The `mergeState` function, which gives a pseudocode overview of the merging algorithm, is defined in Algorithm 3. Conceptually, the test executor invokes the `mergeState` function whenever a merge should be attempted. Once the `mergeState` function has returned, the tester continues executing the rest of the program path. As before, it will call `mergeState` whenever a state merge can be attempted, i.e., whenever it encounters a branching condition.

4.2.1. An Algorithmic Description of State Merging for Concolic Testing

`mergeState` (cf. Algorithm 3) receives as input the root state of the symbolic execution DAG, which is of type State. The definition of a State structure is provided by Algorithm 2, which follows the previously described tuple definition of a state. A State includes a ProgramPoint, defined in Algorithm 1, which follows the extended definition of a program point described in Section 4.1.1.

```
struct ProgramPoint
    lineNumber: A number.
    functionStack: A list of function labels.
end struct
```

Algorithm 1: Definition of a ProgramPoint struct.

```
struct State
    programPoint: A ProgramPoint.
    PC: The path constraint.
    store: A mapping of program variables to
           symbolic expressions.
end struct
```

Algorithm 2: Definition of a State struct.

`mergeState` receives as a second input the partial path that has been explored by the test executor in the current test run. `mergeState` first adds this partial path to the symbolic execution DAG (line 2) by calling the auxiliary function `addPathToDAG`. It then attempts to find a state previously inserted into the execution DAG with a program point that is identical to that of the last state of the partial path (line 3). If no duplicate program point can be found (line 4), the merge operation is aborted. If a suitable candidate state, named `duplicateState`, exists, the merge operation continues by constructing the merged symbolic store by calling the auxiliary function `mergeStores` (line 5).

`mergeStores` creates a new store by traversing all variables in both stores and comparing their values. If their values are different, the function inserts into the new store an ITE expression which uses as predicate the path constraint of `lastState`, and which has as `then` and `else` values the values from respectively the `lastState`'s store and duplicate state's store. If the values are the same, the function simply inserts this value into the new store. We elide the definition of `mergeStores`, but it aligns with the definition of the state merge operation presented in Section 3.2: $\forall v \in \sigma_1 : \sigma_m[v] = \mathit{ITE}(PC_1, \sigma_1[v], \sigma_2[v])$, where σ_m is the store produced by the `mergeStores` function, and σ_1 and σ_2 are the symbolic stores of respectively `lastState` and `duplicateState`.

The merged state itself is then constructed (line 7) by creating a new State that consists of the unchanged *programPoint* of the duplicate state, the disjuncted path constraints of the two original states (created by calling the auxiliary function `disjunct` on line 6), and the merged store.

As a consequence of the path determinacy problem, `duplicateState` likely had a subtree. The symbolic stores of all states in this subtree have to be updated to reflect the new values of the program variables in `mergedStore`. Note that `lastState` does not have a subtree, as it was the most recent node to be added to the symbolic execution tree. The symbolic stores of all the states in the duplicate's subtree are updated via the auxiliary function `applyStoreToChildren` (line 8). This function traverses the entire subtree, updating the stores of all states to reflect the merge operation that has taken place. While traversing the subtree, the function first replaces the store of the visited state with the merged store that was passed. After, the function applies to this store the variable assignment annotations (cf. Section 3.3) along the `then` or `else` edge from the state before recursively descending into respectively the `then` and `else` subtree of that state. Once this recursive process has completed, the function returns with an updated subtree in which all states' stores have been updated. This subtree is then added to `mergedState` (line 9).

Lastly, the original duplicate state and last state of the newly added path are replaced by having their parents (lines 10-11) replace these children with the newly merged state (lines 12-13).

```
input  : The root STATE that forms the root of the current symbolic
         execution tree.
input  : The partial path constraint path that has been observed by the
         test executor.
output: A boolean indicating whether the merge operation was
         successful.
Function mergeState(root, partialPath)
    lastState ← addPathToDAG(root, partialPath);
    duplicateState ← getDuplicate(root, lastState.ProgramPoint) ;
    if duplicateState ≠ null then
        mergedStore ← mergeStores (lastState.PC, lastState.store,
          duplicateState.store) ;
        disjunctedPC ← disjunct(duplicateState.PC, lastState.PC) ;
        mergedState ← new STATE(lastState.ProgramPoint, disjunctedPC,
          mergedStore) ;
        newSubtree ← applyStoreToChildren(mergedStore,
          duplicateState) ;
        setSubtree(mergedState, newSubtree) ;

        lastStateParent ← getParent(lastState) ;
        duplicateStateParent ← getParent(duplicateState) ;
        replaceChildWith(lastStateParent, lastState, mergedState) ;
        replaceChildWith(duplicateStateParent, duplicateState,
          mergedState) ;
        return true
    else
        return false
    end
end
```
Algorithm 3: A pseudo-code implementation of the state merging procedure for concolic testing.

Figure 9 illustrates how the state merging algorithm for concolic testing is applied to the code depicted in Listing 1 over the course of three test runs. As before, when a variable is read, the inlined ITE expression is subscripted with the name of the variable.

5. State Merging for Event-driven Applications

Whereas the previous section described how state merging can be incorporated into concolic testing of sequential applications, this section describes how state merging can be incorporated into the testing of event-driven JavaScript programs.

In sequential code, the state explosion problem is caused only by conditional branching nodes, which are created for branch conditions such as `if` statements. Testing of event-driven code, however, also introduces event branching nodes (cf. Section 2.3.1) that increase the number of program paths.

5.1. Considerations for State Merging in Event-driven Code

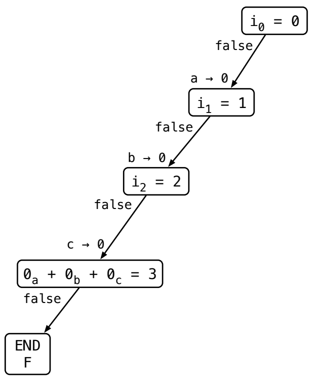
To apply state merging to event-driven code, we must solve the following problems:

- The program point must be redefined to include the event handler that was executed when the state was created.
- The event branching nodes encountered along the path towards a state must be translated into symbolic expressions, so that these expressions may appear in the predicate of ITE expressions. This enables constructing ITE expressions for variables whose value depends on the chosen event handler.

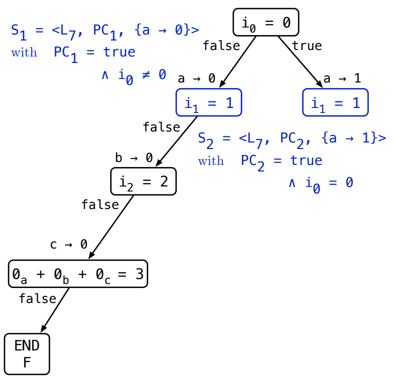
5.1.1. Redefining Program Points

Recall that the definition of a program point determines which states should be considered duplicates of each other, and which hence should be considered as candidate states for merging. Program points were defined in Section 4.1.1 as tuples of a code label, specifically the line number, and the function stack that was active when the state was observed.

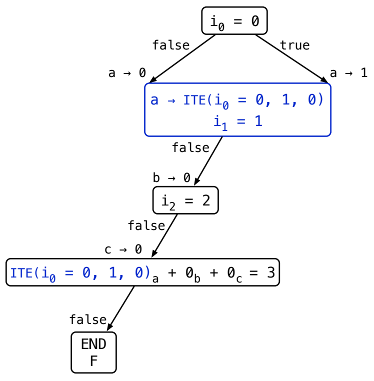
There are two reasons for extending this definition to include event handlers. First, as a technical complication, because of the inclusion of the code label, the program point is tied to a particular location in the code. It is therefore not possible to express a program point for event branching nodes, as these states are created whenever the tester has finished executing an event handler and is preparing to invoke the next handler. Furthermore,



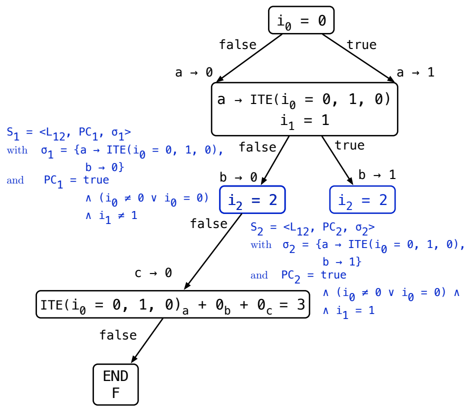
(a) Test run 1: The symbolic execution tree after adding the first program path.



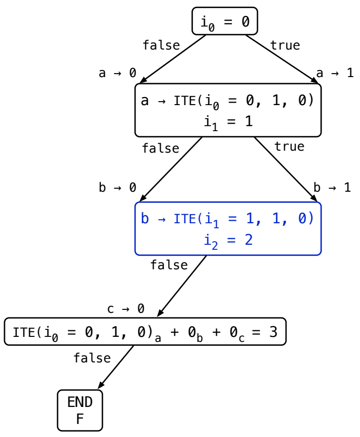
(b) Test run 2: The symbolic execution tree after adding part of the second program path, with the candidate states for merging coloured blue.



(c) Test run 2: The symbolic execution DAG after merging the candidate states from the previous step together.



(d) Test run 3: The symbolic execution DAG after adding part of the third program path.



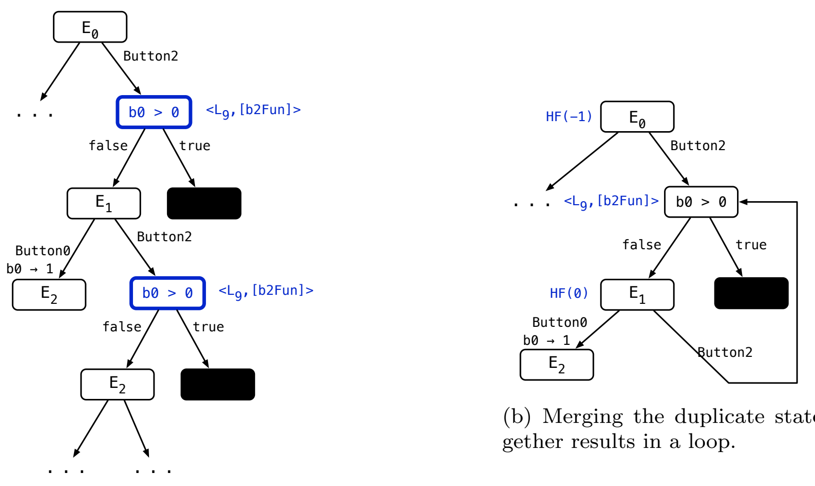
(e) Test run 3: The symbolic execution DAG after merging the candidate states from the previous step together.

Figure 9: Concolic testing of the code in Listing 1, using state merging to prune similar states.

because event handlers are atomic in JavaScript, the function stack for such a state would always be empty, as the JavaScript process cannot invoke a new event handler while another is still being executed. Since no code label can be included and the state's function stack would always be empty, we instead define a new type of program point `HandlerFinished(i)`, which we abbreviate to `HF(i)`. This state is used exclusively for event branching nodes, and expresses the point where the test executor has finished executing the i-th handler in the event sequence. By convention, we define a state of the form E_i to receive the program point `HF(i - 1)`, as E_i indicates that the previous handler, i.e., the handler whose corresponding event branching node has index $i - 1$, has just finished executing.

Second, it is necessary to distinguish between similar states of *different* invocations of the *same* event handler. Consider Figure 10a, which depicts a partial symbolic execution tree for Listing 2 without any merged states. The tree includes two instances, coloured blue, of a `b0` conditional branching node. One of these instances is created when handler `b2Fun` is invoked a first time, the other is created when it is invoked the second time. The program point for both states is defined as $\langle L_9, [\texttt{b2Fun}]\rangle$. As both states share the same program point, they could be merged together, which would result in the symbolic execution DAG in Figure 10b. However, doing so would introduce a loop in the symbolic execution DAG. DAGs are acyclic, so by introducing a loop, the resulting graph would no longer be a DAG. Not only does this complicate the traversal of the symbolic execution graph, but it also renders it possible for states to feature path constraints that have an infinite length. For example, in Figure 10b, node E_2 can be reached from node E_0 by either clicking `Button2` once, following the subsequent `else` branch, and clicking `Button0`, or by clicking `Button2` repeatedly, always following the `else` branch, before finally clicking `Button0`. Loops where `Button2` is clicked repeatedly in a path constraint, however, may interfere with the remainder of the path constraint, as it can no longer be said that node E_2 represents the choice for the third event handler in the event sequence. More importantly, symbolic execution trees are annotated with assignments to variables. Introducing loops into a DAG would therefore render the value of identifiers along a path indeterminate.

At the same time, however, we do not want to make the program point overly restrictive. For example, including in the program point the entire stack of event handlers that have been called so far would make the program point far too specific, as it would become impossible to ever merge states from the subtrees of different handlers. These problems can be avoided by distinguishing between different invocations of the same handler, i.e., by including in the program point tuple the global *index in the event sequence* of



(a) A partial symbolic execution tree for Listing 2 with subtrees for handler `b0Fun` expanded.

(b) Merging the duplicate states of the DAG together results in a loop.

Figure 10: An example of why it is necessary to include the current position in the event sequence as part of the program point. The program points are included in blue.

the current event handler invocation. This index does not refer to the specific event handler being invoked, but rather describes its position in the event sequence. It hence does not keep track of a history of which event handlers were invoked previously, thereby avoiding the problem of the program point being too restrictive. The program point for the top `b0 > 0` state of Figure 10a therefore becomes $\langle L_9, [\texttt{b2Fun}], 1\rangle$, while that of the bottom one becomes $\langle L_9, [\texttt{b2Fun}], 2\rangle$. Since the states now have different program points, the merging algorithm does not merge them together and the problem is avoided. If no event handler is currently being executed, we replace this index with the value `null`.

We therefore define a program point for event branching nodes as a `HF(i)` construct, and for conditional branching nodes as a triple consisting of the code label, the function stack, and either the index of the current event handler invocation or the value `null`.

5.1.2. Event Branching Nodes as Symbolic Expressions

As with sequential applications, the values of program variables may depend on the specific path that was followed. For example, the values of variables `b0` and `b1` depend on how often respectively the event handlers `b0Fun` or `b1Fun` have been executed.

Recall that when merging two separate states S_1 and S_2, the symbolic value for a program variable `x` in the merged state is replaced by an ITE expression of the form $ITE(PC_1, \sigma_1[\texttt{x}], \sigma_2[\texttt{x}])$, where PC_1 is the path constraint for S_1, and σ_1 and σ_2 are the symbolic stores for S_1 and S_2 respectively. The predicate values of such an ITE expression hence consist of the path constraint that lead to the merged states.

For sequential applications, a path constraint consists of the concatenation of the symbolic constraints corresponding to the conditional branching nodes encountered along that path. In the case of event-driven applications, however, the path is also determined by the event handlers that were executed. The path constraint must hence also encode the decision of which event handler was invoked when, which in turn means that the decision of which edge to take from an event branching node must be translated to a symbolic constraint.

This is possible by assigning a unique id to every event handler, named the *target id*, and to express event branching nodes of the form E_i as a constraint over a unique symbolic input variable e_i, named the *event input*. By following a branch from an event branching node, an event input is constrained to a specific target id. Event handlers `b0Fun`, `b1Fun`, and `b2Fun` can be assigned respectively the target ids 0, 1, and 2, based on the order in which they were registered. Following the `Button1` edge for event branching node E_0, which corresponds to clicking on `Button1` and invoking event handler `b1Fun` as the first event handler, is represented as the symbolic constraint $e_0 = 1$. As another example of this translation, the path constraint for the 1_{b0} `> 0` state on the third row of Figure 3, which can be reached by first clicking `Button0` and then `Button2`, is $e_0 = 0 \wedge e_1 = 2$.

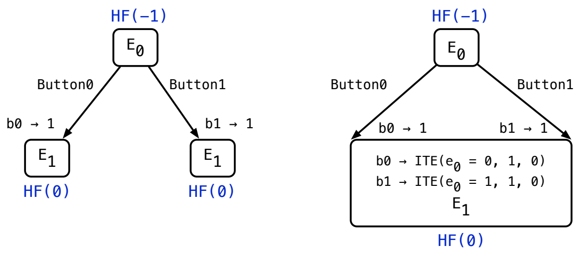


Figure 11: An example of how the program variables `b0` and `b1` are assigned different values along different paths in the unmerged symbolic execution tree (left) and the merged DAG (right). The program points are included in blue.

Having this symbolic representation of decisions on event branching nodes makes it possible to construct ITE expressions for program variables whose value depends on the chosen event handler. Consider Figure 11, which depicts a small part of both the unmerged symbolic execution tree for Listing 2 and the merged DAG. In the merged DAG, the symbolic value for `b0` can be expressed as $ITE(e_0 = 0, 1, 0)$. If this symbolic value were to appear in

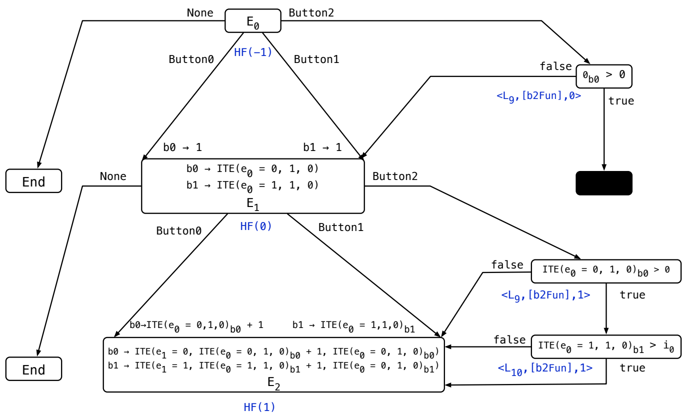


Figure 12: The symbolic execution DAG depicted in Figure 3 after applying state merging, with the states' program points included in blue.

another constraint, then the SMT solver can compute a mapping of event inputs to target ids to create an event sequence.

5.2. *Result of State Merging in Event-driven Code*

After taking these considerations into account, a part of the symbolic execution DAG generated by a concolic tester that incorporates state merging for the even-driven code of Listing 2 is depicted in Figure 12.

The DAG includes only one instance of an event branching node for every invocation of an event handler. Decisions over which event handler to invoke when, which affects the values of variables `b0` and `b1`, have been translated to symbolic constraints so that the symbolic value of these variables can be defined as ITE expressions. Note that in the first invocation of the event handler `b2Fun`, the call to `foo` on line 11 is not reachable, as the branch condition `b0 > 0` can only evaluate to `true` if event handler `b0Fun` has been invoked at least once.

6. Evaluation

In this section, we measure the effect of state merging on the concolic testing of event-driven applications. We implemented the state merging approach outlined in Section 5 in StackFul, a concolic tester for full-stack JavaScript web applications [32]. Full-stack JavaScript web applications are web applications which are comprised of at least one web client process and one server process and which communicate over technologies that are accessible via JavaScript, e.g., Websockets [32]. The web clients are event-driven, as they have to respond to user interactions, e.g., clicking a button.

We run two versions of StackFul, a baseline version that does not incorporate state merging which we name StackFul$_{\text{BASE}}$ and another that does which we name StackFul$_{\text{MERGE}}$ on eight full-stack JavaScript web applications. When testing the applications, both testers instantiate a single client process and a single server process. In all applications, the client process serves as the entry point into the application for the tester, and the client and the server process synchronously communicate with each other via bidirectional Websockets provided by the Socket.IO[2] library.

6.1. Overview of the Evaluation

We evaluate the effectiveness of state merging through the following research questions:

RQ1 Does state merging improve the code coverage of the tester per test run?

RQ2 Does state merging increase the computational overhead per test run, with respect to execution time and memory consumption?

RQ3 Does state merging offer an increase in coverage over the same unit of execution time? That is, does state merging make it possible to cover a larger part of the application in less time?

RQ4 How often does StackFul$_{\text{MERGE}}$ attempt to merge together two states? How often do these attempts succeed?

We answer these four research questions through the following metrics:

Line coverage We measure the number of lines of code that were executed at least once in any test run, expressed as a fraction of the total lines of code in the application.

Branch coverage We measure the total number of branch conditions of which both the `then` branch and the `else` branch were explored at least once [24].

Execution time We measure the overhead induced by state merging when testing an application by observing the execution time required by StackFul$_{\text{MERGE}}$ and StackFul$_{\text{BASE}}$ to complete the same number of test runs.

[2] `https://socket.io/`

Peak memory consumption We measure the peak amount of physical memory allocated over the course of an entire test session. Specifically, we consider the process's maximum resident set size.

Number of (successful) merge attempts performed We measure how frequently StackFul$_{\text{MERGE}}$ invokes the merging algorithm (cf. Algorithm 3), and how often the algorithm successfully merged two states.

Table 2: Characteristics of the web applications considered in our study.

Benchmark	LOC	# of Branches	# of Event Handlers	WMC
Calculator	181	16	15	18
Chat	236	39	1	27
Game of Life	237	64	6	32
Simple Chat	64	4	1	6
Slack Mockup	624	75	4	20
TOHacks	149	22	2	20
Totems	179	9	2	14
Whiteboard	137	16	4	14

To answer these research questions, we compare StackFul$_{\text{BASE}}$ versus StackFul$_{\text{MERGE}}$ on eight JavaScript web application programs that feature event-driven code: Calculator, Chat, Game of Life, Slack Mockup, Simple Chat, TOHacks, Totems, and Whiteboard. Whiteboard and Chat are demo applications[3] for the Socket.IO library. Game of Life, TOHacks, Totems, and Simple Chat were retrieved from a software gallery[4] featuring applications built with the Socket.IO library. Slack Mockup is a project that mimics some of the functionalities of the Slack communication platform. Table 2 reports the main characteristics of each web application: the number of lines of code, the number of branches in the code, the number of message and event handlers that are registered in the application, and the Weighted Methods per Class (WMC) metric which presents an indication of the application's complexity. Note, however, that this last metric does not take into account the complexity incurred by adding event handlers to an application.

[3]`https://github.com/socketio/socket.io/tree/master/examples`
[4]`https://devpost.com/software/built-with/socket-io`

6.1.1. Specifications

The test selector of both versions of the tester employs a search heuristic which prioritises exploring satisfiable paths that pass through uncovered branches. If no such paths are available, the selectors revert to a breadth-first search exploration of the symbolic execution tree. The test executor of both testers, implemented in TypeScript, uses NodeJS v18.7.0, while the test selector, implemented in Scala, uses Java SE Runtime Environment build 19.0.1+10-21, configured to run with a maximum heap size of 4GB. Both testers were executed on a 2.8 GHz Quad-Core Intel Core i7 CPU, with 16GB of 2133 MHz LPDDR3 RAM, running macOS 13.2. As an SMT solver, both testers use Z3 [13], version 4.8.5 - 64bit.

6.2. RQ1: Code Coverage per Test Run

The first research question asks whether, and to what extent, state merging improves the code coverage achieved by the tester per test run. To answer this question, StackFul$_{\text{MERGE}}$ and StackFul$_{\text{BASE}}$ are allocated the same *test run budget*, and we measure the coverage they achieve within this budget. The number of test runs to be completed is not fixed across all applications because the duration of a single test run heavily depends on the application itself. Using the same, fixed number of test runs would therefore result in testing of some applications finishing quickly, whereas other applications would not be finished in any reasonable timespan. We therefore find the test budget per application by having StackFul$_{\text{MERGE}}$ test an application for 5400 seconds (1.5 hours) and observing both the coverage level it obtains and the number of test runs it completes. StackFul$_{\text{BASE}}$ is then allocated the same test run budget. StackFul$_{\text{MERGE}}$ and StackFul$_{\text{BASE}}$ test an application in a deterministic manner. The code coverage that they achieve over a series of test runs is therefore also deterministic. Hence, it suffices to have both testers test each application only once.

To inspect the code coverage obtained by both testers, we employ the line coverage and branch coverage metrics.

6.2.1. Line Coverage

The results of the line coverage metric are shown in Figure 13. The X axis specifies how many test runs have been completed, and the Y axis specifies the fraction (presented as a percentage value) of the total lines of code in the application that have been executed at least once. StackFul$_{\text{BASE}}$ is represented by the orange line with triangles, and StackFul$_{\text{MERGE}}$ is represented by the blue line with circles.

The results indicate that the line coverage achieved by StackFul$_{\text{MERGE}}$ is generally higher than or equal to that of StackFul$_{\text{BASE}}$: it achieves a higher coverage for four out of eight applications (Calculator, Slack Mockup, Totems, and Whiteboard), a lower coverage for one application (Chat), and an ultimately equal coverage for the remaining three. We observe that for the Chat application, $\text{StackFul}_{\text{MERGE}}$ lags behind $\text{StackFul}_{\text{BASE}}$. Across all applications, $\text{StackFul}_{\text{MERGE}}$ shows an increase in line coverage of -3 %pt. to 23 %pt. over $\text{StackFul}_{\text{BASE}}$ at the end of the testing session.

Line coverage is generally high, especially considering some parts of the server are unreachable from that client of the application. After manual inspection of the covered lines, we determined that the remaining uncovered lines for Simple Chat were unreachable from the client. TOHacks also includes a fairly large number of unreachable lines, with many overlapping branch conditions.

In the case of the Chat application, the uncovered lines relies on an operation that is unsupported by the symbolic executor. The concolic tester may therefore not be able to steer execution towards both branches of the branch condition that features the unsupported operation. $\text{StackFul}_{\text{BASE}}$ achieves a 3%pt. increase in line coverage over $\text{StackFul}_{\text{MERGE}}$. We hypothesise this can be attributed to $\text{StackFul}_{\text{BASE}}$ happening upon a solution to the unsupported operation whereas $\text{StackFul}_{\text{MERGE}}$ did not. More concretely, as the structures of the symbolic execution trees computed by $\text{StackFul}_{\text{BASE}}$ and $\text{StackFul}_{\text{MERGE}}$ are different, the search heuristic employed by both versions' test selectors, though identical and deterministic, selects different paths. In some cases, this may cause $\text{StackFul}_{\text{BASE}}$ to cover some part of the program more quickly than $\text{StackFul}_{\text{MERGE}}$, although $\text{StackFul}_{\text{MERGE}}$ should be able to catch up over time.

State merging also does not achieve an improvement in line coverage for Game of Life. This application features several nested loops, which limit opportunities for state merging.

The highest increase in line coverage is attained for the Calculator application, which represents an online calculator application, where users formulate arithmetic expressions by clicking buttons on the client side until a valid expression is sent to the server upon clicking a `calculate` button. The web client of this application registers many event handlers, while relatively few lines of code are executed on the server of this application until a relatively complex sequence of events is invoked on the client. State merging not only reduces the number of test runs required by $\text{StackFul}_{\text{MERGE}}$ to explore the individual event handlers, it also allows for finding the appropriate sequence of events to access the server's message handler more quickly. The Slack Mockup similarly requires triggering a somewhat complex sequence of client-side events in order to reach certain parts of the server.

The Whiteboard application features relatively intricate constraints on user input, where malformed user inputs lead to early termination of the program. State merging manages to explore both branches of the user input validation more quickly than the baseline version. However, because the branches that reject user input feature relatively few lines of code, this results in only a modest improvement in line coverage.

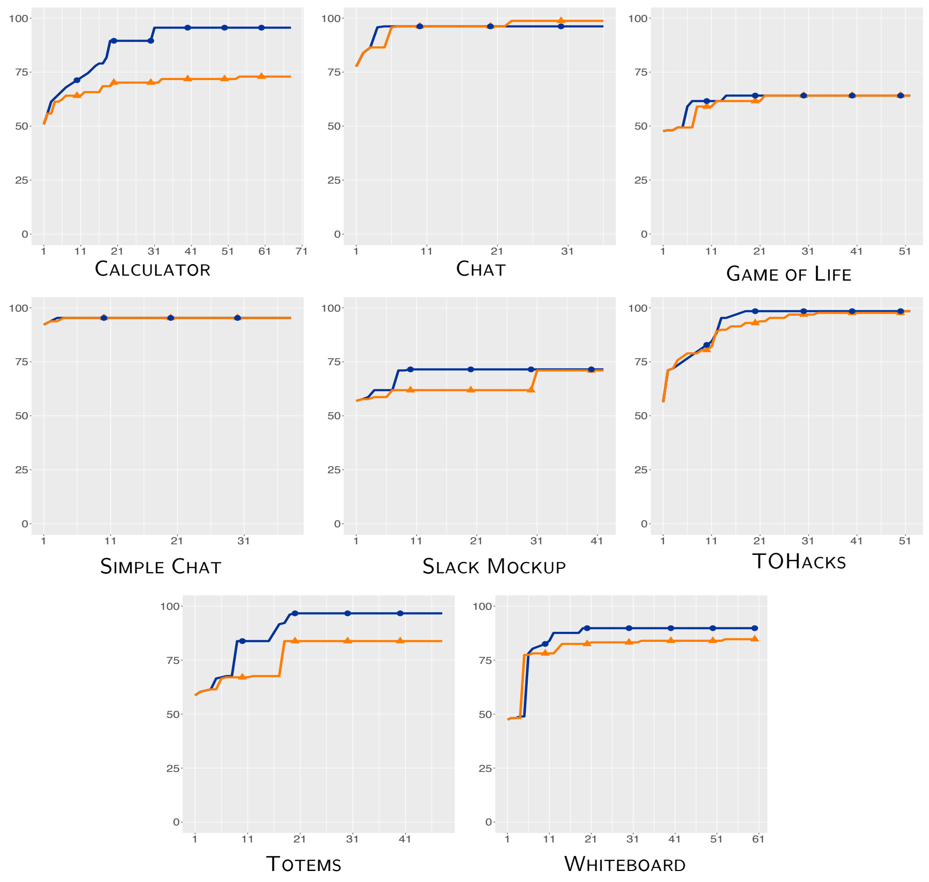


Figure 13: Line coverage achieved by both versions of the tester for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

6.2.2. Branch Coverage

As an alternative measure of coverage for sequential applications, we also observe branch coverage obtained by both testers on the eight applications. The results are depicted in Figure 14. As expected, these results mostly align with the line coverage results. StackFul$_{\text{MERGE}}$ achieves a higher or equal branch coverage for seven out of the eight applications, but again lags behind StackFul$_{\text{BASE}}$ for the Chat application. Across all applications, StackFul$_{\text{MERGE}}$ shows an increase in branch coverage of -25%pt. to 38%pt. over StackFul$_{\text{BASE}}$ at the end of the testing session.

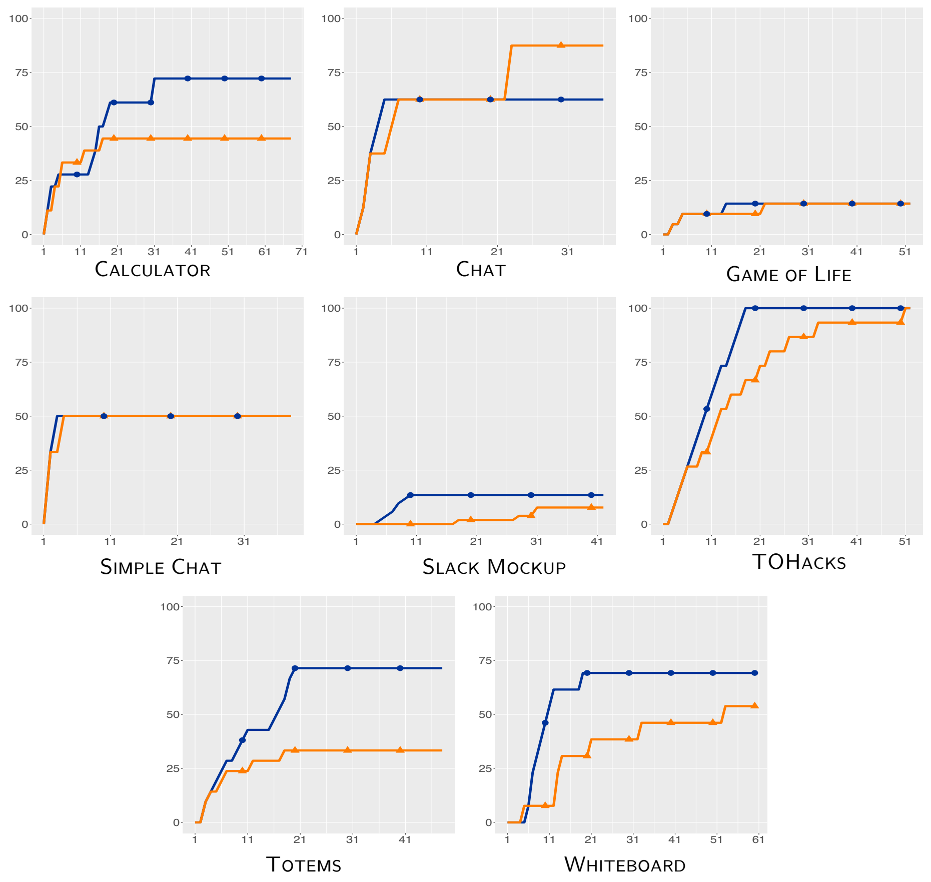


Figure 14: Branch coverage achieved by the tester for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

6.2.3. Conclusion

Both coverage metrics indicate that StackFul$_{\text{MERGE}}$ attains a higher or equal level of code coverage in fewer test runs. We conclude that state merging improves the code coverage achieved by the tester per test run, and we hence answer the first research question in the affirmative.

6.3. RQ2: Overhead per Test Run

The second research question concerns the computational overhead of employing state merging. It asks whether incorporating state merging increases the time required to complete a test run, and whether the memory consumption of the tester increases. We answer these questions by observing the execution time required to complete a number of test runs and measuring the peak memory consumption of both variants. As before, due to the large variations in execution time across all applications, we do not employ a fixed number of test runs. We run StackFul$_{\text{MERGE}}$ for 5400 seconds (1.5 hours) and record the number of test runs it has completed. Afterwards, we allocate the same test run budget to StackFul$_{\text{BASE}}$.

6.3.1. Execution Time

Figure 15 depicts the execution times required by both testers to complete their test run budget. We observe for all applications that state merging increases the execution time required to complete the same number of test runs. We also observe that the gap in execution time between both variants increases over time. We attribute this to the SMT solver requiring progressively more time to solve path constraints that grow increasingly complex. Furthermore, as more merge operations have taken place, the number of ITE expressions in these path constraints increases further. Solving these ITE expressions is known to be particularly computationally intensive for SMT solvers [21].

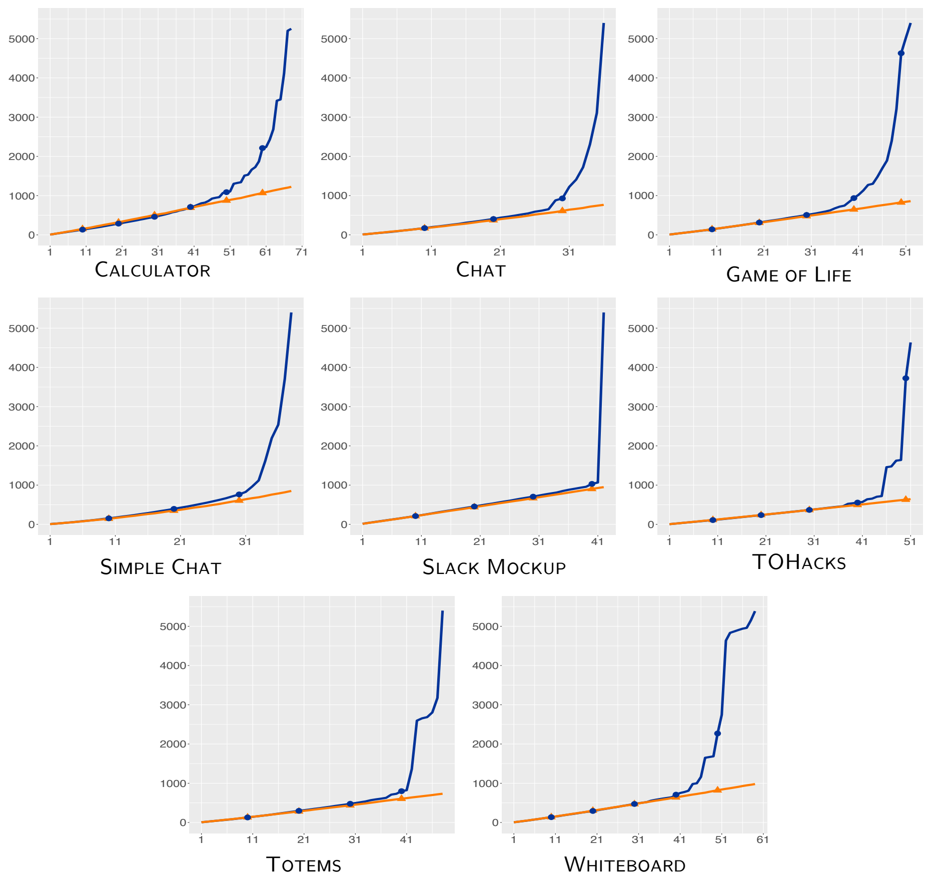


Figure 15: Execution time (in seconds) required by both versions of the tester for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

Across all applications, StackFul$_{\mathrm{MERGE}}$ shows an increase in execution time of 340%pt. to 637%pt. over StackFul$_{\mathrm{BASE}}$ at the end of the testing session.

We find that state merging has a detrimental effect on the execution time per test run because it increases the computational overhead of the tester per test run. We hence answer the second research question in the affirmative.

6.3.2. Merging Overhead

To investigate the causes for this computational overhead, we consider the time spent by StackFul$_{\mathrm{MERGE}}$'s test selector when testing an application. Apart from the computational overhead incurred by executing the state merging algorithm itself (cf. Algorithm 3), merging also affects the time spent selecting an appropriate path to explore in the next test run, as the structure of the symbolic execution tree has been transformed. We also measure the time spent translating StackFul$_{\text{MERGE}}$'s representation of symbolic expressions into the internal format used by its SMT solver (Z3) and the time required by Z3 to solve these constraints, as the introduction of ITE expressions into path constraints affects both processes.

Figure 16 depicts the time spent by StackFul$_{\text{MERGE}}$ in performing these tasks.

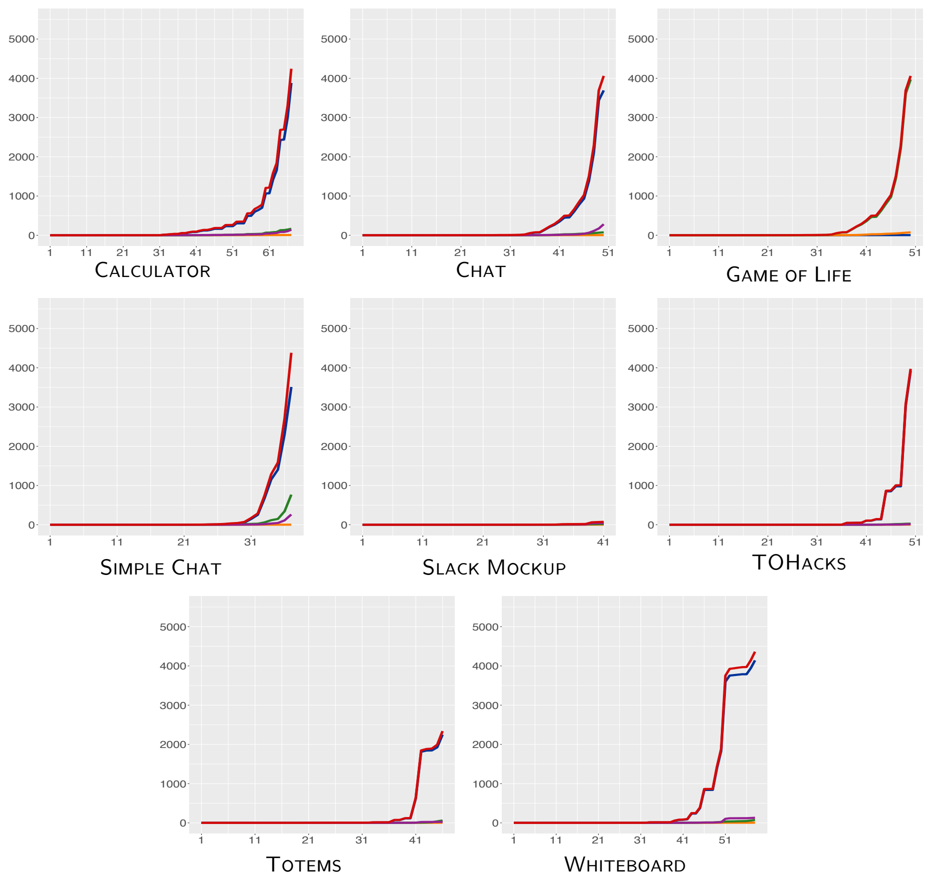


Figure 16: Overhead on the execution time (in seconds) when incorporating state merging. Time spent solving (purple), vs. time spent selecting a path (green), vs. time spent merging (orange), vs. time spent translating path constraints to Z3's internal format (blue), with total overhead time included (red).

The results indicate that the total computational overhead (represented via the red line) is dominated by the time spent translating path constraints, featuring deeply nested ITE expressions, into the constraint format used by Z3 (blue). The overhead of the state merging algorithm itself (orange), the time spent by Z3 in solving path constraints (purple), and the time spent by the search heuristic to select the next path to explore (green) are mostly negligible. However, in the cases of CALCULATOR, CHAT, SIMPLE CHAT there is a noticeable increase in the time spent solving and selecting a path towards the end of the test session. We discuss these results in more detail in Section 7.

At first glance, the performance overhead of the SLACK MOCKUP application appears to be almost zero. Upon closer inspection, however, we found that STACKFUL$_{\text{MERGE}}$ became stuck in finding a satisfiable path that could be explored. This also explains the large spike in SLACK MOCKUP's execution time in Figure 15. Although no timing data could be generated for its final test run, we can again ascribe the cause for this performance degradation to the time spent translating and solving path constraints.

6.3.3. Peak Memory Consumption

In addition to the overhead on the execution time per test run, we also investigate the effect of incorporating state merging on the amount of memory consumed by the tester. We measure the *peak* memory consumption attained by the tester over the course of a complete test session of 5400 seconds. We distinguish between the memory consumed by the test executor and the test selector. Figure 17 illustrates the results of this investigation.

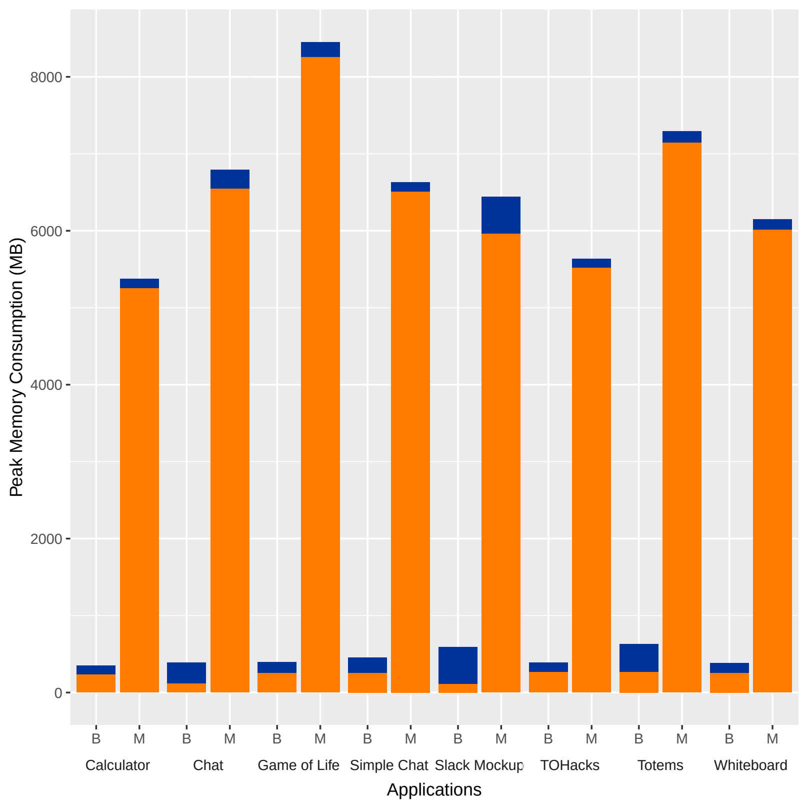


Figure 17: Peak memory consumption (expressed in MB) of StackFul$_{\text{BASE}}$ (B) and StackFul$_{\text{MERGE}}$ (M). For both variants, we distinguish between the memory consumed by the test executor (blue, top part), and the test selector (orange, bottom part).

We observe a significant increase in the memory consumed by StackFul$_{\text{MERGE}}$'s test selector over the baseline test selector. We attribute this increase to the large number of complex ITE expressions that are created when applying state merging. Furthermore, many of these ITE expressions are duplicated throughout the symbolic execution DAG or the path constraints. It may hence be worthwhile to consider a more memory-efficient approach which avoids creating duplicate ITE expressions.

In contrast, state merging does not significantly increase the memory consumed by the test executor. The test executor does not keep an internal representation of the symbolic execution DAG and must only store the path constraint of its current test run. It is hence not significantly affected by the introduction of ITE expressions.

6.4. RQ3: Code Coverage per Unit of Execution Time

We have concluded that state merging generally increases the code coverage attained by StackFul$_{\text{MERGE}}$ (RQ1), but also increases the execution time of each test run (RQ2). Our third research question combines both elements by asking directly whether state merging increases the code coverage attained by both versions of the tester over a unit of execution time, or whether a baseline tester achieves a higher coverage by virtue of the fact that it completes more test runs over the same time interval.

We answer this research question by measuring code coverage in function of the execution time. Rather than allocating a budget of test runs to both testers, we run both for 5400 seconds while recording the code coverage that they attain. We measure line coverage and branch coverage over time.

6.4.1. Line Coverage over Execution Time

Figure 18 depicts the line coverage attained by both testers in function of their execution times. Note that execution times are expressed on a log scale, as line coverage generally increases more quickly towards the beginning of a test execution than towards the end. As before, we find that the line coverage attained by StackFul$_{\text{MERGE}}$ is higher than that of StackFul$_{\text{BASE}}$ for four applications (Calculator, Slack Mockup, Totems, and Whiteboard), lower for one applications (Chat), and equal for the remaining three applications. Across all applications, StackFul$_{\text{MERGE}}$ shows an increase in line coverage over execution time of -3%pt. to 13%pt. over StackFul$_{\text{BASE}}$ at the end of the testing session.

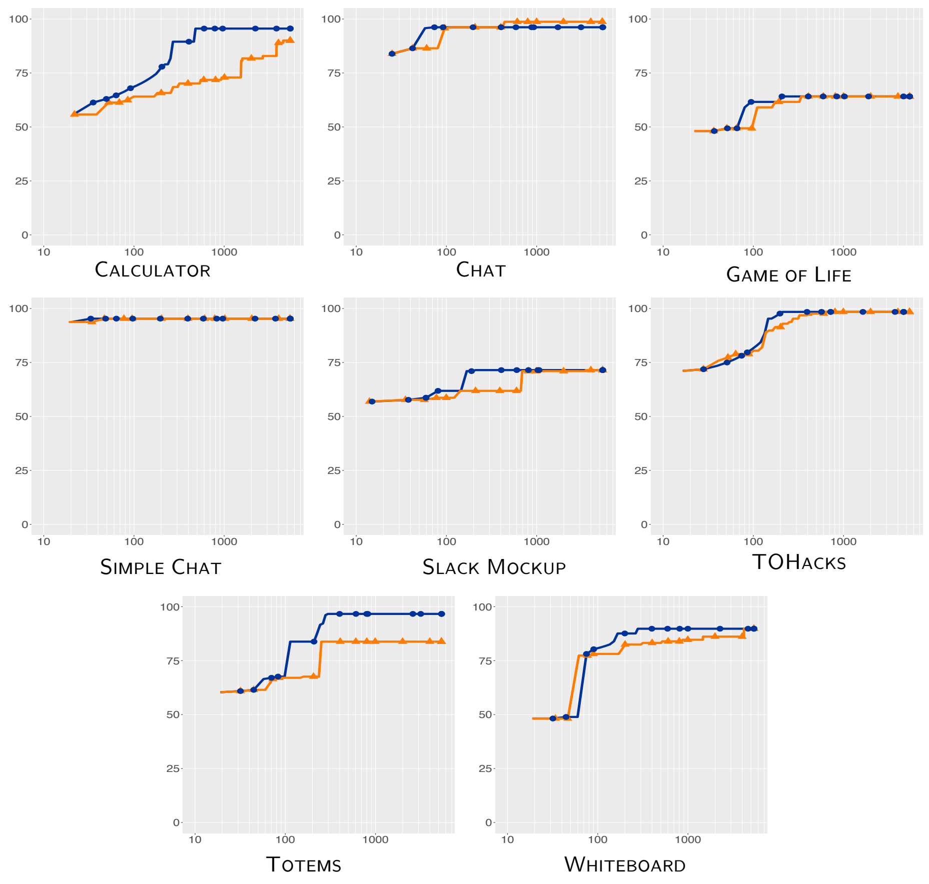


Figure 18: Line coverage attained by both versions of the tester in function of the execution time (in seconds) for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

6.4.2. Branch Coverage over Execution Time

Figure 19 depicts the branch coverage attained by both testers over the same span of execution time. Across all applications, StackFul$_{\text{MERGE}}$ shows an increase in line coverage over execution time of -25%pt. to 38%pt. over StackFul$_{\text{BASE}}$ at the end of the testing session.

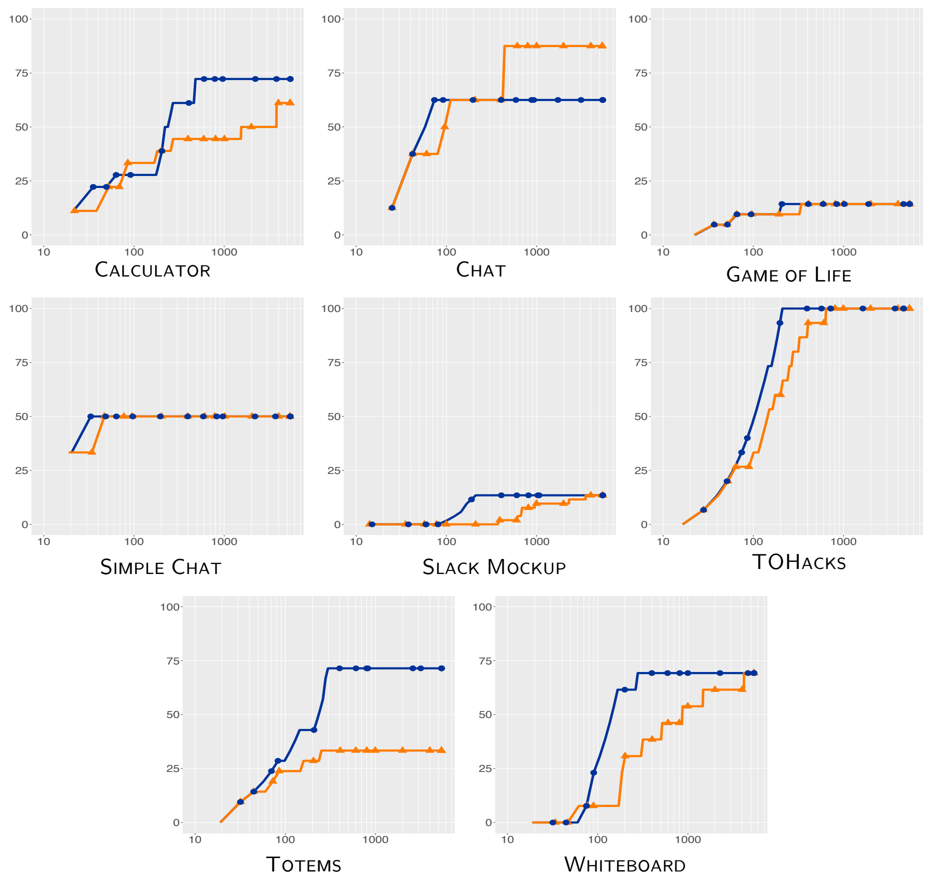


Figure 19: Branch coverage attained by both versions of the tester in function of the execution time (in seconds) for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

Having considered both the line coverage and branch coverage over time, we hence answer the third research question by proclaiming that state merging offers a higher or equal line and branch coverage for the majority of the applications.

6.5. RQ4: Frequency of State Merges

We also investigate the number of attempts that StackFul$_{\text{MERGE}}$ made over the course of each test session to merge two states together, and the number of those attempts that were successful. Figure 20 shows the cumulative number of total merging attempts (blue, with circles) and the cumulative number of successful merging attempts (orange, with triangles) for each input program.

Both numbers grow linearly over time. However, only a small fraction of the merge attempts actually succeed. As stated in Section 4.2, Stack-

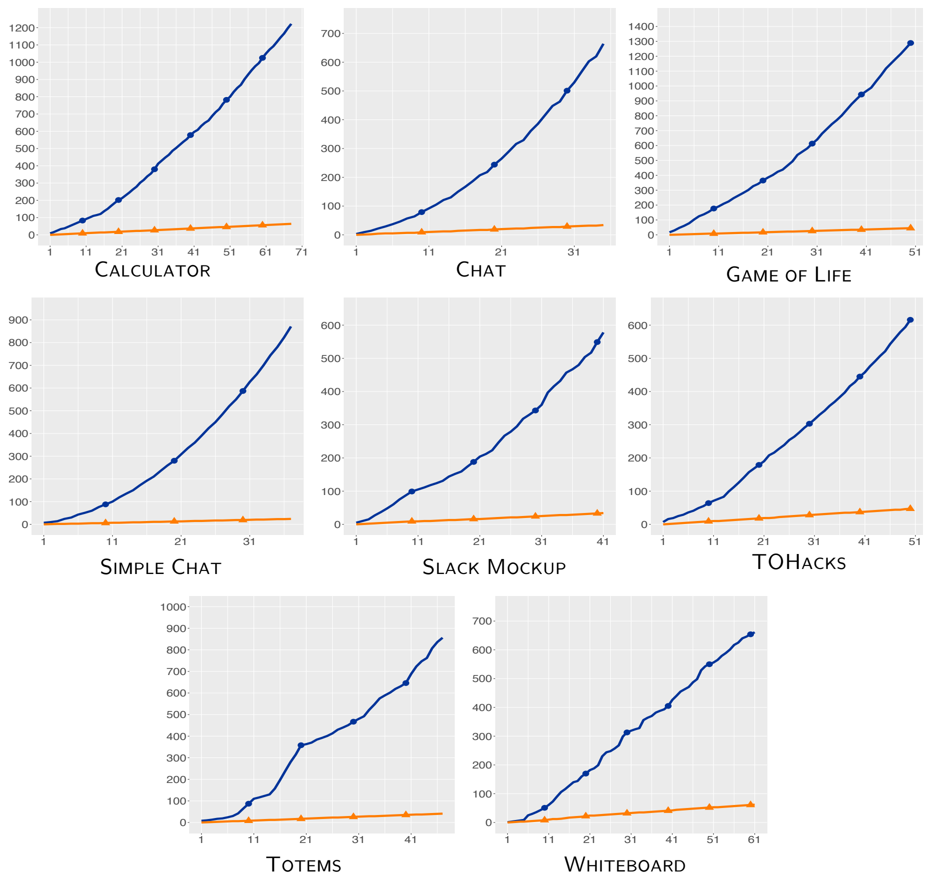


Figure 20: Cumulative number of merge attempts (blue, with circles) versus the number of successful merge (orange, with triangles).

FulMerge attempts a state merge whenever it encounters a branch condition. As the number of branching conditions grows roughly linearly over time, so does the number of merge attempts. Furthermore, Algorithm 3 ensures that no duplicate copies of a state can exist. The vast majority of merge invocations will therefore fail, as they cannot find a duplicate copy of the state.

We therefore answer the fourth research question by concluding that both the total number of merge attempts and the total number of successful merge attempts grow linearly over time.

6.6. Threats to Validity

We identify the limited scope of the evaluation as an important external threat to the validity of the results presented in this section. Although the eight applications that were included in the evaluation suite were retrieved from public software repositories, they are all small, ranging between 64 and

624 LOCs. These applications furthermore do not employ commonly used web frameworks or libraries such as Angular[5], React[6], or Vue[7], as these frameworks usually define idiosyncratic mechanisms for registering event handlers, which are difficult to support in StackFul. For example, event handler registrations that are described in an application's HTML page are not captured by StackFul, as the tester relies on instrumentation of the application's JavaScript code to observe when an event is registered.

Furthermore, many sources of nondeterminism encountered in real-world application cannot be controlled by StackFul because of a lack of support. In the case of web workers, for example, it is necessary to control the scheduling of the workers to ensure that operations are executed in the same order across consecutive test runs. A similar problem arises when testing both the server side and the client side of a full-stack web application, as both are executed by separate processes.

As large, real-world applications are not included in the evaluation suite, we may not generalise the results presented in this section to more complicated applications. It is therefore possible that the challenges imposed by such applications lower the effectiveness of StackFul$_{\text{MERGE}}$ to below the level offered by StackFul$_{\text{BASE}}$. For example, the memory overhead induced by the state merging approach is fairly large (cf. Section 6.3.3). It is likely that this problem is aggravated in more complex applications, to the point where the memory overhead becomes too high for StackFul$_{\text{MERGE}}$ to effectively test the application.

7. Discussion

From the results presented in Section 6, it is clear that incorporating state merging incurs a large performance overhead on our prototypical tester, which becomes especially impactful towards the end of each test session. The large overhead affects the scalability of StackFul$_{\text{MERGE}}$, rendering it difficult to test larger applications which feature more branch conditions. The performance overhead can be mostly attributed to the time spent by StackFul$_{\text{MERGE}}$ in translating the path constraints it has collected into the path constraint format employed by the Z3 solver [13]. StackFul$_{\text{MERGE}}$ cannot use this format directly, as StackFul$_{\text{MERGE}}$ must also track additional information that cannot be encoded in this format, such as identifiers or store

[5]`https://angular.dev/`

[6]`https://react.dev/`

[7]`https://vuejs.org/`

annotations (cf. Section 3.3). Furthermore, certain string operations supported by StackFul$_{\text{MERGE}}$ do not have a direct equivalent in Z3's format and must be translated into more basic operations.

Although this translation process hence cannot be avoided, the scalability of StackFul$_{\text{MERGE}}$ depends on whether this process can be optimised. Caching the translation of (sub)expressions will likely significantly reduce the performance overhead, but will also increase the memory overhead to a potentially intractable level.

However, it should be noted that the translation process is orthogonal to our approach for applying state merging for event-driven applications. The translation can hence be optimised without changing our approach for incorporating state merging in concolic testing for event-driven applications. We also note that, despite the large number of merge attempts conducted (cf. Section 6.5), the overhead of the merging algorithm itself is negligible. Furthermore, StackFul$_{\text{MERGE}}$ still generally attains a higher line and branch coverage than StackFul$_{\text{BASE}}$ over the same span of execution time, despite the large performance overhead.

8. Limitations

We now list two technical limitations that limit the scope of StackFul$_{\text{MERGE}}$: the merging of heap locations and merging that takes place inside a loop. Apart from specifying the technical challenge raised by these issues, we also describe potential solutions to these problems.

8.1. Heap Locations

One main limitation to StackFul$_{\text{MERGE}}$'s current implementation lies in how state merging handles the merging of *heap locations*. Heap locations are used in symbolic execution when e.g., creating an object, or reading from or writing to a field of an object [1, 14]. In symbolic execution, objects are stored in a symbolic heap $\mathcal{H}$, where they can be accessed via a heap location (i.e., an address). The symbolic heap is defined as a mapping of locations to objects, where an object is defined as a mapping of fields to values. When a new object is created, a new, unique heap location is created and the symbolic heap is extended with an entry that maps the location to the object.

For example, after executing the definition statement `let o = {x:1, y:2}`, a new symbolic heap location ℓ is created, the symbolic heap is extended with the entry $\ell \rightarrow \{\mathtt{x} \rightarrow 1, \mathtt{y} \rightarrow 2\}$, and the program variable `o` receives the symbolic heap location ℓ. Performing a read operation on an object's field, e.g., `o.x`, can then be performed symbolically by looking up the heap location to which `o` is bound in the symbolic heap, and then reading the field `x` from

```
1 let o1 = {x: 1}, o2 = {x: 2}, o3;
2 if (randomInt() === 1) {
3   o3 = o1;
4 } else {
5   o3 = o2;
6 }
7 if (o3.x === 1) { ... } else { ... }
```

Listing 4: A code snippet which results in merging of heap pointers.

the resulting object. A write operation, e.g., `o.x = 42`, can be performed by first applying a similar lookup procedure and then overwriting the value for the field `x` of the object.

Consider now the code snippet in Listing 4. After merging both program paths after the first `if` statement, the program variable `o3` is conditionally aliased with either the heap location of `o1` or that of `o2`, and should hence be represented symbolically as an ITE expression over these two locations. However, when reading the field `x` on line 7, the aforementioned lookup procedure can no longer be applied, as `o3` does not point to a unique location.

Applying state merging as described in Section 3 on two states where a program variable is conditionally bound with separate objects is hence unsound. However, we sketch a, currently unimplemented, solution in which we define a read operation from or a write operation to an object field of an ITE expression.

We represent a read operation on a field `f` via a symbolic expression X as $\mathcal{H}[X][\mathtt{f}]$, which is defined as follows:

$$\mathcal{H}[X][\mathtt{f}] = \begin{cases} \mathit{ITE}(c, \mathcal{H}[\ell_t][\mathtt{f}], \mathcal{H}[\ell_e][\mathtt{f}]) & \text{if } X \text{ evaluates to } \mathit{ITE}(c, \ell_t, \ell_e) \\ o[\mathtt{f}] & \text{if } X \text{ evaluates to a heap location } \ell \text{ and} \\ & \mathcal{H} \text{ contains the mapping } \ell \to o \end{cases}$$

For simplicity, we do not model the case where $\mathcal{H}$ does not contain an entry for ℓ. Note, however, that such a scenario would result in an exception being thrown at run time.

A write operation of the value v on a field `f` via a symbolic expression X is represented as $\mathcal{H}[X][\mathtt{f}]$ `:=` v, and updates the heap in the following way when executed:

$$\begin{cases} \mathcal{H}[\ell_t][\texttt{f}] \texttt{ := } \mathit{ITE}(c, v, v_{t\_old}) \text{ and} \\ \mathcal{H}[\ell_e][\texttt{f}] \texttt{ := } \mathit{ITE}(c, v_{e\_old}, v) & \text{if } X \text{ evaluates to } \mathit{ITE}(c, \ell_t, \ell_e)\text{, with } v_{t\_old} \text{ and } v_{e\_old} \text{ defined as the previous value for respectively } \mathcal{H}[\ell_t][\texttt{f}] \text{ and } \mathcal{H}[\ell_e][\texttt{f}] \\ \mathcal{H}[\ell][\texttt{f}] \texttt{ := } v & \text{if } X \text{ evaluates to } \ell \end{cases}$$

As before, we do not model the case where $\mathcal{H}$ does not contain an entry for ℓ.

Apart from redefining the read operation and write operation, incorporating this solution into StackFul$_{\text{MERGE}}$ would also require the edges of the symbolic execution DAG to be annotated with *heap updates*. These updates are analogous to the variable assignments (i.e., store updates) described in Section 3.3.

8.2. Merging inside Loops

Another limitation of the state merging algorithm described in this chapter relates to the unrolling of loops. Recall that in the context of event-driven applications, a program point is defined as a triple of the code label currently being executed, i.e., the line number, the function stack, and the index in the event sequence of the current event handler invocation. However, this does not account for code that is executed inside a program loop, where the same code label is observed multiple times using the same function stack and event handler invocation. Note that this is not a problem for recursive functions as, without tail recursion optimisation, recursive calls to a function are saved in the function stack of the program point. Different invocations of a recursive function hence correspond to different stacks.

Consider the code in Listing 5, where a loop with a fixed number of iterations is executed as part of the event handler for a button click. Without state merging, this loop gives rise to the creation of 1024 new paths in every invocation of the event handler. State merging is therefore necessary to prevent the number of states from exploding.

However, as the program point for the `if` statement on line 4 is identical across the ten loop iterations, our state merging algorithm would, in each loop iteration, attempt to merge the conditional branching node that was created in the current iteration with that of the previous iteration. This would introduce a cycle into the symbolic execution DAG, as the state now features a back-edge to itself.

Although currently not implemented, the problem can be avoided entirely by including the concrete loop iteration in the program point. Merging may

```
document.getElementById("button")
  .addEventListener("click", function () {
    for (let i = 0; i < 10; i++) {
      if (randomInt() === 0) { ... }
      else { ... }
    }
  });
```

Listing 5: A program loop inside an event handler.

then still take place across the same iteration, i.e., the `then` branch and `else` branch of the conditional branching node point to the same state, thereby avoiding the state explosion problem. In effect, this corresponds to unrolling the loop before state merging, similar to existing approaches [4, 21].

9. Related Work

State merging has received widespread attention because of its potential to prevent the state explosion problem. As mentioned in Section 3, however, merged states also increase the burden of the SMT solver to such an extent that it may not always be beneficial to merge states eagerly. Another problem is the fact that state merging may interfere with other optimisation techniques applied by the tester. State merging works best when similar states are merged as soon as possible, as this prevents the highest number of duplicate descendant states from being created. Ideally, if a state similar to one the tester has just explored is available, the tester would select that similar state so that it can be merged if found similar enough. However, the tester may employ a search strategy which prioritises exploration of a completely different state, for example because that state would lead the tester closer to a particular target. The tester therefore has to choose which optimisation technique to follow.

Kuznetsov et al. demonstrated two techniques for addressing these problems [21]:

- *Query count estimation* mitigates the burden placed on the SMT solver by taking into account the number of variables with differing values between two states when looking for similar states. Query count estimation attempts to minimise the number of ITE expressions that are created, as these expressions are computationally intensive to solve for the SMT solver. It does so by only merging states whose same variables with different values are expected to appear infrequently in future solver queries.

- *Dynamic state merging* does not interfere with existing search strategies but identifies opportune moments for merging dynamically. The tester's search strategy generally retains the privilege of selecting which state to explore next. However, if the dynamic state merging algorithm detects that a state scheduled for exploration at some future point will be sufficiently similar to an already explored state, the state merging algorithm overrides the decision of the search strategy and schedules the first state for immediate exploration.

MergePoint [4] alternates between static symbolic execution and concolic testing to take advantage of both approaches. It starts testing the program by performing concolic testing. When a branch condition is encountered, rather than forking the execution, MergePoint may switch to using static symbolic execution, since state merging can be applied more easily in static symbolic execution. While in static mode, MergePoint merges states together until performing static symbolic execution becomes too difficult, for example because of a system call or an indirect jump. At that point, MergePoint reverts to concolic testing and continues testing the program.

Jaffar et al. use *interpolation* to prune program paths by terminating the exploration of certain paths early, before they can spawn new paths [19]. With interpolation, the symbolic execution engine creates interpolants, which describe conditions necessary to trigger a particular bug. If during exploration of a path the engine can infer that the interpolant is not satisfied, it terminates exploration of the path.

As mentioned previously, using ITE expressions to represent merged symbolic values allows the tester to precisely model which values an expression evaluates to given a set of conditions, but this also increases the burden for the SMT solver. Rather than relying on ITE expressions, Scheurer et al. take inspiration from abstract interpretation to define a general lattice model [28]. When merging two values together, they perform a join operation on the values in the lattice and represent the merged value through this joined value. Although this loses precision, Scheurer et al. deem this an acceptable loss since queries to the SMT solver become easier to process.

MultiSE [29] takes the idea of merging program states even further, by using one shared, incrementally constructed representation for all execution states that have been explored by the tester. In this shared state, all variables, including the program counter, are described using *value summaries*. A value summary is a mapping of a set of conditions to the value that the variable will take if these conditions are true. In effect, MultiSE corresponds to merging program states at every assignment, rather than only when two states happen to collude.

9.1. Compositional Symbolic Execution

Like state merging, approaches based on compositional symbolic execution also try to address the state explosion problem.

Godefroid presented compositional symbolic execution for producing symbolic summaries of individual components of a program [15, 2]. In the first application of this approach, these components corresponded to function definitions. With compositional symbolic execution of function, the tester prioritises fully exploring each function that is called in the application, rather than exploring any particular part of the code base. While exploring a function, the tester constructs a symbolic summary that describes the behaviour of the function. This function summary consists of a disjunction of the individual paths through the function, mapping specific pre-conditions on the function's inputs to the corresponding post-conditions on the function's outputs. When the function is later called again in the application, the tester can reuse this summary rather than performing symbolic execution on the function anew, which would result in more branching of program paths. Similar symbolic summaries can also be created for loops, with pre-conditions and post-conditions on the variables read from and written to inside the loop [16, 30, 34, 35].

Lin et al. propose to go even further than summarising functions or loops, by constructing summaries at the level of basic blocks [23]. Symbolic summaries at such a fine-grained level of control would allow for more reuse of these summaries, and prevent a large amount of calls to the SMT solver, which would be especially beneficial if some of the paths in a summary are actually infeasible.

Compositional symbolic execution can be combined with forms of target-driven testing to efficiently find feasible program paths leading to a particular target statement [25, 2, 27]. Some target-driven compositional testers use backwards symbolic execution. These testers usually operate in two phases: in the first phase, the tester constructs symbolic summaries, while in the second phase, the tester performs symbolic execution backwards on the program, starting from the target statement and continuing until the tester reaches an entry point into the program [26, 22].

Although all of the previously mentioned work attempt to infer these summaries automatically, symbolic specifications could also be defined manually [3].

10. Conclusion

We described an approach for incorporating *state merging* in concolic testing of event-driven applications. This algorithm allows for merging similar states together, hence curbing the state explosion problem that arises during naive concolic testing. This is the first explicit description of how state merging can be incorporated into concolic testing, as well as the first approach that enables state merging to be performed in the setting of event-driven applications.

Naive concolic testing gives rise to the state explosion problem, where the number of states to be covered by the tester grows exponentially in function of the number of branch conditions that are executed. One approach for mitigating this problem is to merge similar states together, thereby reducing the total number of states in the symbolic execution DAG. When merging states, shared variables that have been assigned different values in each state can be represented as a single variable with an ITE expression as value.

We distinguished between state merging performed by *online* symbolic executors, and *offline* executors, such as concolic testers. Although state merging for concolic testers is more cumbersome than for online testers because of the path determinacy problem, we described an approach where offline state merging mimics the state merging performed by online testers. We also specified how state merging can be incorporated into concolic testing of event-driven applications, by extending the definition of a *program point* to incorporate event sequences and by translating event branching nodes into symbolic expressions that can be incorporated into ITE constructs. We evaluated StackFul$_{\text{MERGE}}$, which incorporates state merging, and StackFul$_{\text{BASE}}$, which does not, on a limited set of eight small full-stack JavaScript web applications. We compared both testers with respect to the code coverage that they achieve per test run, the computational overhead of state merging per test run, and the code coverage per unit of execution time. Our evaluation finds that StackFul$_{\text{MERGE}}$ achieves a higher code coverage for the same number of test runs, as well as for the same length of execution time on about half of the applications, while it attains an equal coverage for most of the other applications. However, the limited scale of these applications makes it difficult to generalise the results to more complicated, real-world applications. An important prospect for future work would therefore be to make it possible for StackFul to test real-world applications in order to validate the results presented in this article.

The implementation of our approach is publicly available at `https://github.com/softwarelanguageslab/StackFul`

Acknowledgements

The work of Maarten Vandercammen was supported in part by the Fonds voor Wetenschappelijk Onderzoek – Vlaanderen (FWO) Strategic Basic Research (SBO) BaseCamp Zero under Project S000323N.

References

[1] Saswat Anand and Mary Jean Harrold. Heap Cloning: Enabling Dynamic Symbolic Execution of Java Programs. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 33–42, 2011. doi: 10.1109/ASE.2011.6100071. URL `https://doi.org/10.1109/ASE.2011.6100071`.

[2] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 367–381, 2008.

[3] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with Veritesting. *Communications of the ACM*, 59(6):93–100, 2016.

[5] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alexandre Rebert, and Ned Williamson. The Mayhem Cyber Reasoning System. *IEEE Security & Privacy*, 16(2):52–60, 2018.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[7] Cristian Cadar and Martin Nowack. KLEE Symbolic Execution Engine in 2019. *International Journal on Software Tools for Technology Transfer*, 23(6):867–870, 2021.

[8] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2), February 2013.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[10] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 380–394, 2012.

[11] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. CRETE: A versatile binary-level concolic testing framework. In *Proceedings of the 21st International Conference on the Fundamental Approaches to Software Engineering FASE 2018*, pages 281–298, 2018.

[12] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the Art: Dynamic Symbolic Execution for Automated Test Generation. *Future Generation Computer Systems*, 29(7):1758–1773.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24.

[14] Xianghua Deng, Jooyong Lee, and Robby. Efficient and Formal Generalized Symbolic Execution. *Automated Software Engineering*, 19(3): 233–301, 2012. doi: 10.1007/s10515-011-0089-9. URL `https://doi.org/10.1007/s10515-011-0089-9`.

[15] Patrice Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 47–54, 2007.

[16] Patrice Godefroid and Daniel Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 23–33, 2011.

[17] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, 2012.

[18] Trevor Hansen, Peter Schachte, and Harald Søndergaard. State Joining and Splitting for the Symbolic Execution of Binaries. In *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, pages 76–92, 2009.

[19] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. Boosting Concolic Testing via Interpolation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 48–58, 2013.

[20] Kari Kähkönen, Roland Kindermann, Keijo Heljanko, and Ilkka Niemelä. Experimental Comparison of Concolic and Random Testing for Java Card Applets. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*, pages 22–39, 2010.

[21] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 193–204, 2012.

[22] Sihan Li, Farah Hariri, and Gul Agha. Targeted Test Generation for Actor Systems. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 8:1–8:31, 2018.

[23] Yude Lin, Tim Miller, and Harald Søndergaard. Compositional Symbolic Execution Using Fine-Grained Summaries. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*, pages 213–222, 2015.

[24] Glenford J. Myers. *The Art of Software Testing (3. Edition).* Wiley, 2011. ISBN 978-1118031964.

[25] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. MACKE: Compositional Analysis of Low-Level Vulnerabilities With Symbolic Execution. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 780–785, 2016.

[26] Saahil Ognawala, Fabian Kilger, and Alexander Pretschner. Compositional Fuzzing Aided by Targeted Symbolic Execution. *CoRR*, abs/1903.02981, 2019.

[27] Ivan Pustogarov, Thomas Ristenpart, and Vitaly Shmatikov. Using Program Analysis to Synthesize Sensor Spoofing Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, UAE, April 2-6, 2017*, pages 757–770, 2017.

[28] Dominic Scheurer, Reiner Hähnle, and Richard Bubel. A General Lattice Model for Merging Symbolic Execution Branches. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*, pages 57–73, 2016.

[29] Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 842–853, 2015.

[30] Jan Strejcek and Marek Trtík. Abstracting Path Conditions. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 155–165, 2012.

[31] Nikolai Tillmann and Jonathan de Halleux. Pex-White Box Test Generation for .NET. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153, 2008.

[32] Maarten Vandercammen, Laurent Christophe, Dario Di Nucci, Wolfgang De Meuter, and Coen De Roover. Prioritising Server Bugs via Interprocess Concolic Testing. *The Art, Science, and Engineering of Programming*, 5(2):5, 2020.

[33] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A Comprehensive Study on Real World Concurrency

Bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 520–531, 2017.

[34] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. S-looper: Automatic Summarization for Multipath String Loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 188–198, 2015.

[35] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 61–72, 2016.