# MultiSE: Multi-Path Symbolic Execution using Value Summaries

*Koushik Sen*
*George Necula*
*Liang Gong*
*Philip Wontae Choi*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-173
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-173.html

October 17, 2014

# MULTISE: Multi-Path Symbolic Execution using Value Summaries

Koushik Sen, George Necula, Liang Gong, Wontae Choi
EECS Department, University of California, Berkeley
{ksen, necula, gongliang13, wtchoi}@cs.berkeley.edu

## Abstract

Dynamic symbolic execution (DSE) has been proposed recently to effectively generate test inputs for real-world programs. Unfortunately, dynamic symbolic execution techniques do not scale well for large realistic programs, because often the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path.

In this paper, we propose MULTISE, a new technique for merging states *incrementally* during symbolic execution, *without using auxiliary variables*. The key idea of MULTISE is based on an alternative representation of the state, where we map each variable, including the program counter, to a set of guarded symbolic expressions called a *value summary*. MULTISE has several advantages over conventional DSE and state merging techniques: 1) value summaries enable sharing of symbolic expressions and path constraints along multiple paths, 2) value-summaries avoid redundant execution, 3) MULTISE does not introduce auxiliary symbolic values, which enables it to make progress even when merging values not supported by the constraint solver, such as floating point or function values.

We have implemented MULTISE for JavaScript programs in a publicly available open-source tool. Our evaluation of MULTISE on several programs shows that MULTISE can run significantly faster than traditional symbolic execution.

## 1. Introduction

Symbolic execution is a technique for automatically generating a symbolic model from a program. It has been used succesfully as a key component in a variety of applications, including generating high-coverage tests for C [11, 12, 17, 23, 48], C++ [37], C# [51], Java [3, 4, 32, 40, 46], PHP [5], JavaScript [45, 47], x86-binaries [6, 25, 50]. Symbolic execution has also been used in program verification tools, such as jStar [21] and KeY [1].

The key idea behind symbolic execution was introduced almost 40 years ago [18, 33]. In this paper we consider the dynamic variant of symbolic execution (DSE), in which a program is executed using symbolic values in place of concrete values for inputs. During the execution, the state of variables is represented using symbolic expressions over the symbolic input values. For each explored execution path of the program, symbolic execution generates a path constraint formula $\phi$ over the symbolic input values. A satisfying assignment to the path constraint $\phi$ denotes a concrete test input to the program on which the program executes along the corresponding path. Symbolic execution attempts to explore all feasible execution paths of a program systematically using a search strategy. When symbolic execution is used for test input generation, a constraint solver [19] is used to extract a satisfying assignment for each path constraint. When it is used for path-based program verification, the path constraint and the desired postcondition are passed to a theorem prover in the form of a path verification condition.

Symbolic execution techniques do not scale for large realistic programs because often the number of feasible execution paths of a program increases exponentially with the length of an execution path. To mitigate this path-explosion problem, a number of techniques [2, 6, 22, 26, 36, 52] have been proposed to merge states obtained from multiple paths converging at a join point. For example, if a variable `x` is assigned some values $v_1$ and $v_2$ along the two branches of an `if-then-else` statement, then after the conditional statement the states from the two paths are merged into a single state by introducing an auxiliary symbolic value for the value of `x`, say $x_0$. In the merged states, the variable `x` is mapped to $x_0$, and a symbolic constraint $(x_0 = v_1) \vee (x_0 = v_2)$ is added to the path constraint, stating that $x_0$ can either be $v_1$ or $v_2$. The advantage of merging paths at join points is that the number of paths that are explicitly explored remains polynomial in the length of an execution. However, this form of state merging can lead to difficulties if the resulting formulas are outside the scope of the theories supported by the constraint solver, which may happen, for example, if $v_1$ or $v_2$ in the above example are floating point values, function values, or objects. At the same time, this representation of the merged state prevents a common optimization in symbolic execution, which is to perform operations concretely if the operands are concrete. This would be the case in our example if $v_1$ and $v_2$ are constants; in the merged state the operations would have to be performed on the newly introduced auxiliary variable $x_0$, which means that they would have to be performed symbolically.

In this paper, we propose MULTISE, a new technique for merging states *incrementally* during symbolic execution, *without using auxiliary variables*. The key idea of MULTISE is based on an alternative representation of the state, where we map each variable, including the program counter, to a set of guarded symbolic expressions called a *value summary*. MULTISE improves upon conventional DSE while avoiding some of the drawbacks of conventional state-merging based on auxiliary variables, as follows:

1. Compared to conventional DSE, the value-summary representation of MULTISE is a powerful way of sharing symbolic expressions and path constraints along multiple paths. As an improvement over state merging, sharing using value summaries works even for paths that do not all end at the same join points, as long as some variables have the same values on those paths. Also, in MULTISE the sharing is achieved without having to explicitly identify join points, and without having to traverse the entire state when merging. Instead, the sharing is achieved *incrementally* at each statement in the program. We show in this paper that the sharing factor, i.e., the ratio of the number of paths to the number of distinct symbolic expressions in value summaries for each variable, ranges from 3 to 45 in our experiments.

```
1   var x = readInput;
2   var z = readInput;
3   var r = readInput;
4   x = 2*x;
5   if (x > 100)
6     if (z == 1)
7       r = 1.3;
8   if (r > 1)
9     z = r - 1;
10  halt
```

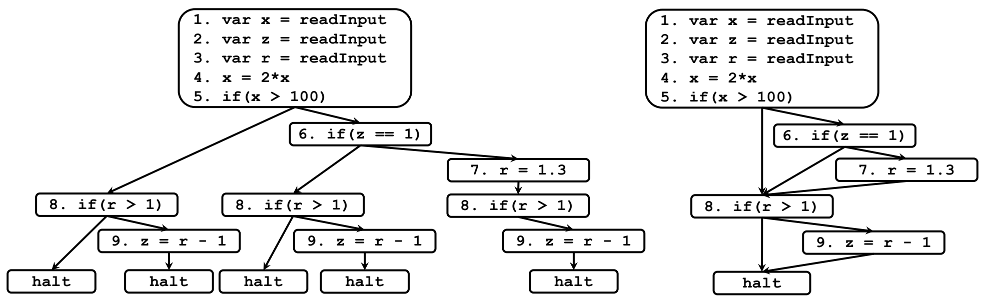

**Figure 1.** (a) A simple program to illustrate MULTISE; (b) Conventional symbolic execution tree; (c) MULTISE execution DAG.

2. Compared to conventional DSE, MULTISE can avoid doing redundant work, e.g., for statements that follow a join point and which operate on variables that have the same values on the joining paths. This follows naturally from the value-summary representation of the state that shares the parts of the state that are the same among different paths. We show in this paper that MULTISE executes between 1.3 to 87 times faster than conventional DSE. This is due to MULTISE performing between 2 to 47 times fewer operations, and correspondingly spending between 1.2 to 94 times less time in SMT solver invocations.
3. Compared to conventional state merging, MULTISE does not use auxiliary variables. This has several important advantages. First, symbolic execution can proceed even when joining values that are not supported by the constraint solver, e.g., depending on the solver: floating point values, objects, or function values. Existing symbolic execution techniques deal with such situations by discarding one of the paths and continuing the execution with a concrete value, while MULTISE can often carry out the symbolic execution for all paths while staying within the scope of the constraint solver. Second, if the values being merged are function values, and the merged value is invoked, the MULTISE value-summary representation encodes naturally the various possible functions that may be invoked. In contrast, in conventional state merging if the value being invoked is represented as an auxiliary variable, an SMT solver must be used to figure out what function should be invoked to proceed with the symbolic evaluation. These kinds of operations are quite common in dynamically typed programming languages such as JavaScript, Python, and Ruby. We show in Section 5.2 that about half of our benchmarks would require auxiliary variables of type other than integer or string if executed with conventional state merging, sometimes in the thousands, for up to 60% of the joins; MULTISE avoids all these problematic auxiliary variables.
4. Value-summary based symbolic execution can be formulated in a way that generalizes both conventional DSE and state-merging algorithms for symbolic execution. We show in this paper that both these variants can be obtained from MULTISE by varying the choice of when and to what extent value summaries are compacted based on sharing of symbolic values. We also show that we can vary the way MULTISE chooses which state to explore next, and depending on this choice, we can get different search strategies in symbolic execution, such as breadth-first, or depth-first. This flexibility of MULTISE makes it a general framework for describing various heuristics used in symbolic execution and state merging.

We have implemented MULTISE for JavaScript programs in a publicly available open-source tool (`https://github.com/SRA-SiliconValley/jalangi` under branch `symfront`). We use binary decision diagrams (BDDs) [10] to concisely represent and to efficiently manipulate path constraints and guards of value summaries. Our evaluation of MULTISE on several programs shows that MULTISE can run significantly faster than traditional symbolic execution.

## 2. Overview

We introduce the concepts of conventional symbolic execution and its state representation informally and then we describe the main elements of the MULTISE symbolic execution. We will use the program in Figure 1(a) as a running example, which is written in a JavaScript-like language. A statement of the form `var v = e;` declares and initializes a variable `v` with the value of the expression `e`. The execution of the statement `var x = readInput;` receives an integer input from the environment and assigns it to the variable `x`.

### 2.1 Conventional Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) executes a program using symbolic expressions for the program variables and memory locations. These expressions are in terms of fresh symbolic values that are introduced upon execution of `readInput` expressions. DSE executes one path at a time, and it maintains the current symbolic state that includes: the program counter, a mapping of program variables to symbolic expressions, and a symbolic path constraint $\phi$, which is a quantifier-free propositional formula over symbolic expressions.

For example, after executing statements 1–4 from our example the symbolic execution state is as follows:

| path | $\phi$ | *pc* | x | z | r |
|---|---|---|---|---|---|
| 1-5 | *true* | 5 | $2x_0$ | $z_0$ | $r_0$ |

where $x_0$, $z_0$, and $r_0$ are the symbolic values introduced for the result of the `readInput` expressions in lines 1–3, respectively.

Each row in these tables corresponds to the symbolic execution state of a path.[1] Informally, for any concrete input values (concrete values for the symbolic values) that satisfy the path constraint, the concrete execution on those input values will follow the path given in the table. Also, if we evaluate the symbolic expressions on the same input values for the symbolic variables, we obtain the value of the variable in the concrete execution at the end of the path. In our example so far, for any set of input values the program will follow the path 1–5, and if the input value for $x_0$ is 10, then the value of `x` at line 5 will be 20.

Upon encountering a branch and if both sides of the branch are feasible, DSE replaces the current symbolic state with two copies of the state with updated values of $pc$ and of the path constraints. One of these copies is placed in a backtracking set and the other becomes the new current state. Continuing our example, we can

[1] The `path` component of the state is shown here for clarity, but is not explicitly maintained during symbolic execution.

represent the state after executing the conditional in line 5 in a consolidated manner, as follows:

| path | $\phi$ | *pc* | x | z | r |
|---|---|---|---|---|---|
| 1-5,8 | $2x_0 \leq 100$ | 8 | $2x_0$ | $z_0$ | $r_0$ |
| 1-5,6 | $2x_0 > 100$ | 6 | $2x_0$ | $z_0$ | $r_0$ |

In general, the path constraints are conjunctions of symbolic boolean expressions corresponding to the branches taken to follow the path specified in the first column.

At every step, DSE will pick one state from the consolidated state, and will update the values of variables and the value of the program counter according to the statement at the program counter for that state. In the case of a conditional statement a copy of the state, with updated *pc* and path constraint, is added to the consolidated state. We can consider that DSE is exploring the execution tree (Figure 1(b)), and depending on the strategy DSE uses to pick the state to advance next, we can have different exploration orders in symbolic execution, such as depth-first, breadth-first, or best-first.

Eventually, DSE will finish exploring all states, and will terminate with the consolidated state shown below, with each of the five states corresponding to one of the five feasible paths shown in Figure 1(b).

| path | $\phi$ | *pc* | x | z | r |
|---|---|---|---|---|---|
| 1-5,8-10 | $\phi_1 = 2x_0 \leq 100 \wedge r_0 > 1$ | 10 | $2x_0$ | $r_0 - 1$ | $r_0$ |
| 1-5,8,10 | $\phi_2 = 2x_0 \leq 100 \wedge r_0 \leq 1$ | 10 | $2x_0$ | $z_0$ | $r_0$ |
| 1-5,6,8-10 | $\phi_3 = 2x_0 > 100 \wedge z_0 \neq 1 \wedge r_0 > 1$ | 10 | $2x_0$ | $r_0 - 1$ | $r_0$ |
| 1-5,6,8,10 | $\phi_4 = 2x_0 > 100 \wedge z_0 \neq 1 \wedge r_0 \leq 1$ | 10 | $2x_0$ | $z_0$ | $r_0$ |
| 1-10 | $\phi_5 = 2x_0 > 100 \wedge z_0 = 1$ | 10 | $2x_0$ | 0.3 | 1.3 |

We make several observations about this state. First, the path constraints for any two paths in a consolidated state are disjoint. Second, DSE will evaluate eagerly expressions containing concrete values. In our example, when the conditional `if (r > 1) ...` is executed in the path that includes line 7, the boolean condition is evaluated concretely for the value 1.3 for `r`. The condition for the "then" branch is $1.3 > 1$, which evaluates to *true*, and is thus not shown above in the path constraint for the path 1–10. Third, symbolic execution aggressively rules out unfeasible paths by checking the satisfiability of the path constraint for the two branches of a conditional using an SMT solver. For the same conditional as before, the path constraint for the "else" includes the conjunct $1.3 \leq 1$ that evaluates to *false* and makes the path constraint unsatisfiable.

## 2.2 MULTISE Value-Summary State Representation

The MULTISE representation of the symbolic execution state is based on the key observation that by considering a consolidated view of the execution state, including the current state and also the states saved for backtracking, we expose a significant opportunity for sharing of path constraints and symbolic expressions.

Consider the final consolidated state of DSE, as shown above. We can obtain a more compact representation if we represent it by variables, i.e., by columns. For each variable, and for each distinct symbolic expression of the variable, we construct the disjunction of the corresponding path constraints. For example, for *pc* the only symbolic expression is 10 with the disjunction of path constraints $\phi_1 \vee \phi_2 \vee \phi_3 \vee \phi_4 \vee \phi_5$ which is equivalent to *true*. Consequently, we represent the consolidated value of *pc* as the pair $(\mathit{true}, 10)$. We call such a pair, a *guarded symbolic expression*. For variables that take different symbolic expressions on different paths we represent their value as a set of pairs, with one pair for every distinct symbolic expression. We call such a set of guarded symbolic expressions a *value summary*. The MULTISE state is a mapping that maps each variable to a value summary. The path constraints of different guarded expressions for a given variable are disjoint and their disjunction is *true*. The MULTISE representation of the final state for our example program is:

$$\begin{array}{l} \{ \quad pc \mapsto \{(\mathit{true}, 10)\}, \\ \quad \mathtt{x} \mapsto \{(\mathit{true}, 2x_0)\}, \\ \quad \mathtt{z} \mapsto \{(\phi_1 \vee \phi_3, r_0 - 1), (\phi_2 \vee \phi_4, z_0), (\phi_5, 0.3)\}, \\ \quad \mathtt{r} \mapsto \{(\neg\phi_5, r_0), (\phi_5, 1.3)\} \\ \} \end{array} \qquad \text{(Final State)}$$

A MULTISE final state describes compactly the final values of all variables in all feasible concrete executions, as follows. Given any assignment of integer input values to the symbolic values corresponding to the program inputs, exactly one of the path constraints will hold for each variable. The corresponding symbolic expression, evaluated at the given program inputs, gives the value of the variable at the end of the execution of the program on the given program inputs.

There are several advantages to the MULTISE value-summary representation. The obvious one is its more compact form. As we will show in our experiments there is a significant amount of sharing for the symbolic expressions of variables among the many execution paths. The less obvious but more important advantage is that this representation achieves a natural form of state merging, which in turn can reduce dramatically the number of statements that must be executed symbolically, as we discuss in the next section and we show experimentally in Section 5.

## 2.3 MULTISE: Symbolic Execution with Value Summaries

To illustrate the operation of MULTISE, consider the state when the symbolic execution has explored all three paths up to the conditional in line 8. For a conventional DSE the state would be:

| path | $\phi$ | *pc* | x | z | r |
|---|---|---|---|---|---|
| 1-5,8 | $2x_0 \leq 100$ | 8 | $2x_0$ | $z_0$ | $r_0$ |
| 1-5,6,8 | $2x_0 > 100 \wedge z_0 \neq 1$ | 8 | $2x_0$ | $z_0$ | $r_0$ |
| 1-5,6-8 | $\phi_5 = 2x_0 > 100 \wedge z_0 = 1$ | 8 | $2x_0$ | $z_0$ | 1.3 |

This state representation with three separate rows corresponds to the three separate instances of execution paths ending in the statement at line 8 shown in DSE execution tree from Figure 1(b).

The corresponding MULTISE value-summary representation of the state is:

$$\begin{array}{l} \{ \quad pc \mapsto \{(\mathit{true}, 8)\}, \\ \quad \mathtt{x} \mapsto \{(\mathit{true}, 2x_0)\}, \\ \quad \mathtt{z} \mapsto \{(\mathit{true}, z_0)\}, \\ \quad \mathtt{r} \mapsto \{(\neg\phi_5, r_0), (\phi_5, 1.3)\} \\ \} \end{array} \qquad \text{(Intermediate State 8)}$$

where, $\phi_5 = (2x_0 > 100 \wedge z_0 = 1)$. Note that the guard for the value $r_0$ of the variable `r`, can be written either as $2x_0 \leq 100 \vee (2x_0 > 100 \wedge z_0 \neq 1)$, or the logically equivalent $\neg\phi_5$.

This value summary represents a merge of the three separate conventional DSE states, corresponding to three separate executions paths. This allows MULTISE to evaluate the conditional in line 8 twice (i.e., once for each value of `r` in the value summary), instead of three times for conventional DSE, as shown in the MULTISE execution DAG (directed acyclic graph) shown in Figure 1(c).

MULTISE symbolic execution in this state first considers the value summary for the program counter. It picks one of the values, in this case 8, guarded by the path constraint *true*, and executes the statement `if (r > 1) ...`. This requires the computation of the value of the expression `r > 1`.

The symbolic execution of the expression `r > 1` goes over each guarded expression in the value summary for variable `r`, applies the operation $>$ on the expression part of each guarded expression, and computes the value summary $\{(\neg\phi_5, r_0 > 1), (\phi_5, \mathit{true})\}$. Note that we add the conjunct *true* to each guard, to account for the current path constraint for the program counter. Note that the second guarded expression for `r > 1` contains the symbolic expression "*true*", which is obtained from $1.3 > 1$. MULTISE eagerly simplifies the parts of symbolic expressions that do not depend on symbolic values.

Essentially, we want to compute the value of the binary expression `r > 1` only for the paths matching the path constraint from the value summary of *pc* .

Next MULTISE processes the actual conditional statement. We compute the condition for the computed value of `r > 1` to be *true*, as a disjunction over the guarded expressions in the value summary for `r > 1`. We must also add a conjunction for the current path constraint (*true*). We will denote this condition as $\phi_6$:

$$\phi_6 = \textit{true} \wedge ((\neg\phi_5 \wedge r_0 > 1) \vee (\phi_5 \wedge \textit{true}))$$

Therefore, after the execution of the conditional statement at line 8, in the new state *pc* maps to the value summary $\{(\phi_6, 9), (\neg\phi_6, 10)\}$, where $\neg\phi_6$ is logically equivalent to $\neg\phi_5 \wedge r_0 \leq 1$, the condition for the computed value of `r > 1` to be *false*. The value summary representing compactly *both* the "then" and the "else" branches can be written as:

$$\begin{aligned} \{ \quad & pc \mapsto \{(\phi_6, 9), (\neg\phi_6, 10)\}, \\ & \mathtt{x} \mapsto \{(\textit{true}, 2x_0)\}, \\ & \mathtt{z} \mapsto \{(\textit{true}, z_0)\}, \\ & \mathtt{r} \mapsto \{(\neg\phi_5, r_0), (\phi_5, 1.3)\} \\ \} & \end{aligned} \qquad \text{(Intermediate State 9+10)}$$

Note that this value summary represents five paths, two of which end at line 10 after taking the "else" branch at line 8, and the remaining three paths end at line 9. $\neg\phi_6$ denotes the combined path constraint of the two paths ending at line 10 and $\phi_6$ denotes the combined path constraint of the three paths ending at line 9.

Every time a new guarded symbolic expression is added to the value summary for *pc* , MULTISE invokes a quick BDD satisfiability check followed by an SMT solver satisfiability check for the path constraint. This is important in order to avoid exploring unfeasible paths. For the value summaries of other variables, only a BDD satisfiability check is used, to reduce the overall cost of SMT solving, which is a significant fraction of the overall cost.

One of the most interesting aspects of MULTISE is that it performs *incremental state merging* at every assignment statement to obtain a new consolidated representation of states using value summaries. To illustrate this aspect, we continue with the above MULTISE state. Say that for the program counter, MULTISE picks the guarded value $(\phi_6, 9)$ and executes line 9 next, with the path constraint $\phi_6$. First, we symbolically evaluate the right-hand side of the assignment (`r - 1`), and we obtain the guarded value:

$$\{(\neg\phi_5, r_0 - 1), (\phi_5, 0.3)\}$$

Since line 9 is guarded by the path constraint $\phi_6$, symbolic execution of the assignment `z = r - 1` should only update the value of `z` for those paths for which $\phi_6$ is *true*. The value of `z` must remain unchanged in the symbolic state for the other paths. This is achieved by computing the new value of `z` using a *guarded value-summary union*, where we preserve the previous value of `z` with the additional guard $\neg\phi_6$ (the negation of the current path constraint) to which we add the value summary for the right-hand side with the additional guard $\phi_6$. By applying a conjunction of $\neg\phi_6$ to the guards of the current value summary stored in `z`, we keep unchanged the portion of the value summary for the other two paths (whose combined path constraint is $\neg\phi_6$).

The resulting value summary for `z` is:

$$\{(\neg\phi_6, z_0), (\phi_6 \wedge \neg\phi_5, r_0 - 1), (\phi_6 \wedge \phi_5, 0.3)\}$$

which is logically equivalent with the value summary we have used in (Final State) for the final value of `z`. Finally, the value summary stored in *pc* is also updated to $\{(\neg\phi_6, 10), (\phi_6, 10)\}$, which simplifies to $\{(\textit{true}, 10)\}$. Therefore, after the execution of the statement `z = r - 1` at line 9, the state becomes the same as the (Final State).

## 2.4 Advantages of MULTISE

We highlight the key advantages of MULTISE over existing techniques for symbolic execution.

**First**, the MULTISE state representation using guarded symbolic expressions is a powerful way of sharing symbolic expressions and path constraints among many different paths. We show in Section 5 that the sharing factor, i.e., the ratio of the number of paths to the number of distinct symbolic expressions in value summaries, ranges from 3 to 45 in our experiments.

Some amount of sharing is also accomplished by previous techniques for symbolic execution using state merging [6, 36], but in MULTISE there is sharing for all states, not just those at control-flow join points, as shown, for example, in the (Intermediate State 9+10).

MULTISE proposes a novel technique for *incrementally updating* the consolidated symbolic execution state. We have seen an example of this incremental update in the previous section when we showed the state update for the assignment on line 9. The right-hand side of the assignment is computed only for the paths matching the current path constraint by conjoining the guards of the used variables with the current path constraint. The new value summary for the variable includes this computed value summary for the right-hand side, along with the old value summary conjoined with the negation of the current path constraint, to model the preservation of the value of the variable on paths not matching the path constraint. Thus the MULTISE state is at *all times consolidated* over all the paths being explored.

This incremental state update is in contrast with how state merging is conventionally implemented. At join points, state merging needs to iterate over the part of the symbolic state that has been modified by the paths converging at the join point and merge that part of the state. Identifying the join points, keeping track of the modified part of the state, and merging the modified state could pose various implementation challenges which are not present in MULTISE.

Furthermore, because in MULTISE sharing is automatic for all paths at all times, it takes effect even for programs where *the join points are not known statically*, such as programs with exceptions, or computed control-flow, or for binary programs with unstructured control-flow where the join points are non-trivial to compute. In fact, in Section 3 we present the MULTISE algorithm for an assembly-like language with computed jumps, which shows that MULTISE can be used effectively for very low-level languages. In contrast, state merging techniques need explicit knowledge of the join points to trigger the merging operation that achieves sharing.

**Second**, MULTISE avoids redundant computation. This feature is directly due to maintaining the value summaries in a consolidated form at all times. At each step, MULTISE picks one of the guarded expressions in the program counter value summary and executes that statement. For example, in conventional DSE, the same statement `z = r - 1` at line 9 will get executed three times along three paths reaching the statement, as shown in Figure 1(b). MULTISE executed the statement only once, as shown in Figure 1(c), because the new state representation (Intermediate State 8) merges the three paths. In contrast to state merging techniques, which need to identify statically the join points, in MULTISE we can achieve sharing, and thus effectively merging, even in a language with computed jumps. We show in Section 5 that MULTISE executes between 1.3 to 87 times faster than conventional DSE. This is due to DSE performing between 2 to 47 times more operations, and also spending between 1.2 to 94 times more time in SMT solver invocations.

**Third**, MULTISE achieves sharing without introducing auxiliary symbolic values during state merging at join points. This has three advantages:

1. Execution can proceed even if certain theories are not supported by the constraint solver.
2. Execution can carry out most operations concretely.
3. There is no need for expensive constraint solver calls where conventional state merging introduces auxiliary symbolic values for functions and subsequently functions denoted by those auxiliary symbolic values are called.

To illustrate these advantages, we first need to take a look at how existing symbolic execution techniques for merging state work. Existing techniques introduce auxiliary symbolic values to represent the value of a variable computed along two or more paths merging at a point. For example, consider the intermediate DSE state of the example program at line 8 where three paths merge.

| path | $\phi$ | *pc* | x | z | r |
|---|---|---|---|---|---|
| 1-5,8 | $2x_0 \leq 100$ | 8 | $2x_0$ | $z_0$ | $r_0$ |
| 1-5,6,8 | $2x_0 > 100 \land z_0 \neq 1$ | 8 | $2x_0$ | $z_0$ | $r_0$ |
| 1-5,6-8 | $\phi_5 = 2x_0 > 100 \land z_0 = 1$ | 8 | $2x_0$ | $z_0$ | 1.3 |

Here the symbolic expression for the variable `r` along the three paths are not all the same. Conventional DSE stores only one symbolic expression for each variable. Therefore, conventional state merging would introduce an auxiliary variable $r_1$ to denote the value of the variable `r`, and would add to the path constraint the relationship between $r_1$ and the different symbolic expressions for `r` along the merged paths, as follows:

| path | $\phi$ | *pc* | x | z | r |
|---|---|---|---|---|---|
| …,8 | $((2x_0 \leq 100 \land r_1 = r_0)$ $\lor(2x_0 > 100 \land z_0 \neq 1 \land r_1 = r_0)$ $\lor(2x_0 > 100 \land z_0 = 1 \land r_1 = 1.3))$ | 8 | $2x_0$ | $z_0$ | $r_1$ |

The problem with this approach is that the new path constraint containing the auxiliary variable has a predicate $r_1 = 1.3$. However, if the constraint solver does not support floating point constraints, then symbolic execution cannot merge the paths to generate a path constraint that is beyond the scope of the constraint solver.[2]

In MULTISE, we never introduce auxiliary symbolic values. Therefore, path constraints in MULTISE are always formulas over the input symbolic values, which we restrict to integer and string types. Concrete values of data types that are not supported by the constraint solver remains in the state as concrete values guarded by symbolic predicates. This also implies MULTISE can perform more operations concretely than existing techniques, as demonstrated below for functions as values.

The fact that MULTISE does not introduce auxiliary symbolic values while merging paths also helps MULTISE to efficiently handle function values, which are often first-class objects in dynamic languages such as JavaScript, Python, and Ruby. We illustrate this using the following program:

```
1  var x = readInput;
2  var f, r = 0;
3  if (x > 0)
4    f = function f1() { return 1;}
5  else
6    f = function f2() { return -1;}
7  r = f();
```

[2] The same problems arise if we write the path constraint using *ITE* (if-then-else): $r_1 = ITE(2x_0 \leq 100, r_0, ITE(z_0 \neq 1, r_0, 1.3))$.

In this program, `x` gets an input from the environment. Depending on whether `x>0`, `f` is assigned the function `f1` or `f2`. Then the function stored in `f` is called and the value returned by the call is stored in `r`.

Consider a conventional DSE state with two paths both of which end at line 7:

| path | $\phi$ | *pc* | x | f | r |
|---|---|---|---|---|---|
| 1-3,4,7 | $x_0 > 0$ | 7 | $x_0$ | f1 | 0 |
| 1-3,6,7 | $x_0 \leq 0$ | 7 | $x_0$ | f2 | 0 |

If we merge the two paths using existing path merging techniques, then the state becomes:

| path | $\phi$ | *pc* | x | f | r |
|---|---|---|---|---|---|
| …,7 | $((x_0 > 0 \land f_0 = \texttt{f1})$ $\lor(x_0 \leq 0 \land f_0 = \texttt{f2}))$ | 7 | $x_0$ | $f_0$ | 0 |

Merging introduces an auxiliary variable $f_0$ and the path constraint now refers to the function objects `f1` and `f2`. If we treat `f1` and `f2` as symbolic references to the two functions, then when symbolic execution executes the statement `r = f()` at line 7, it needs to resolve what are the possible function values that may be invoked. This is typically done by invoking a constraint solver to find all satisfying assignments to $f_0$ given the path constraint [6]. Invoking a constraint solver to obtain all satisfying assignments is expensive.

MULTISE requires no such constraint solving as it explicitly stores both `f1` and `f2` as separate guarded expressions in the value summary denoted by `f`. Specifically, the state of MULTISE will be:

$$
\begin{aligned}
\{\quad & pc \mapsto 7,\\
& \texttt{x} \mapsto x_0,\\
& \texttt{f} \mapsto \{(x_0 > 0, \texttt{f1}), (x_0 \leq 0, \texttt{f2})\},\\
& \texttt{r} \mapsto 0\\
\}
\end{aligned}
$$

In this state, and others that follow, we simplify the notation and drop the constraint *true* from a guarded expression. Symbolic execution of the statement `r = f();` will then create two paths corresponding to the invocation of the two functions stored in the value summary denoted by `f`. MULTISE's mechanism of explicitly storing all uninterpreted objects as values in value summaries allows us to avoid repeated constraint solver calls.

Keeping the values along different paths separate is very helpful when dealing with memory addresses and pointers, since it allows MULTISE to maintain in a natural way the set of memory addresses that a variable may point to, which in turn will make it possible to lookup and update memory addresses directly in many cases. Consider the following example program:

```
1  var r = [ 2 ]; // A new array with one element
2  var s = [ 3 ]; // A new array with one element
3  var x = readInput;
4  if (x > 0)
5    t = r;
6  else
7    t = s;
8  t[0] = 4; // Store at first location in array
```

At each memory allocation, MULTISE returns a new concrete memory address, such as $a_0$ and $a_1$ in this example[3], and keeps value summaries for the value stored at each address symbol, just as for variables. The MULTISE state before the store statement in line 8 will be:

[3] $a_0$ and $a_1$ are not auxiliary symbolic values. We could have replaced $a_0$ and $a_1$ with concrete addresses, say 0x3242 and 0x3246, had they been known to us.

$$\begin{aligned}\{\quad & pc \mapsto 8, \mathtt{x} \mapsto x_0, \mathtt{r} \mapsto a_0, \mathtt{s} \mapsto a_1,\\ & \mathtt{t} \mapsto \{(x_0 > 0, a_0), (x_0 \leq 0, a_1)\},\\ & a_0 \mapsto 2, a_1 \mapsto 3\\ \}\end{aligned}$$

When processing the store statement on line 8, MULTISE will resolve the address being written (`t[0]`) to either $*a_0$ or $*a_1$. Thus the value summaries for $a_0$ and $a_1$ are modified to contain a combination of their previous values and their values as updated by the store statement. After processing the store statement on line 8, the state becomes:

$$\begin{aligned}\{ & \\ & pc \mapsto 9, \mathtt{x} \mapsto x_0, \mathtt{r} \mapsto a_0, \mathtt{s} \mapsto a_1,\\ & \mathtt{t} \mapsto \{(x_0 > 0, a_0), (x_0 \leq 0, a_1)\},\\ & a_0 \mapsto \{(x_0 > 0, 4), (x_0 \leq 0, 2)\},\\ & a_1 \mapsto \{(x_0 \leq 0, 4), (x_0 > 0, 3)\}\\ \}\end{aligned}$$

When $x_0 > 0$, the variable $t$ contains address $a_0$, which is updated to 4 under this path constraint. In the alternative, the variable $t$ contains address $a_1$, which is updated to 4.

This allows MULTISE value summaries to maintain precise aliasing information and to perform strong updates and strong reads, updating and accessing directly the memory locations that may be involved in the memory operation, without having to resort to encoding constraints for the theory of arrays. This is in contrast with state merging techniques that use auxiliary variables. For example, if the value of `t` is represented using the auxiliary variable $t_1$, along with the constraint $(x_0 > 0 \wedge t_1 = a_0) \vee (x_0 \leq 0 \wedge t_1 = a_1)$, then symbolic execution would have to either invoke a solver to enumerate the possible addresses that $t_1$ refers to, or must defer the reasoning about the memory operations to a solver using the theory of arrays. We show in Section 5.2 that about half of our benchmarks would require auxiliary variables of type other than integer or string if executed with conventional state merging, sometimes in the thousands, for up to 60% of the joins; MULTISE avoids all these problematic auxiliary variables.

**Fourth**, value-summary based representations provide a general framework for symbolic execution by making explicit two sources of non-determinism:

- if two guarded expressions $(\phi, v)$ and $(\phi', v)$ are present in a value summary, an implementation may or may not merge them,
- at each step, an implementation can pick non-deterministically a guarded expression from the value summary stored in $pc$.

Depending on how the first set of non-deterministic choices are resolved, we get various degrees of merging in symbolic execution. If we do not perform any merging, we get conventional symbolic execution. If we want to keep our guards simpler (i.e. avoid too many levels of nesting of disjunctions and conjunctions), we can avoid merging two guarded expressions if their guards are already complex. The second set of non-deterministic choices could be resolved to derive various search strategies in symbolic execution.

As further evidence of how general is the formulation of symbolic execution with value summaries, we show in Section 3.4 that by changing only the choice of how value summaries are merged when they contain multiple occurrences of the same symbolic expression, the operation of MULTISE degenerates into DSE.

## 3. Algorithm

In this section, we formally describe MULTISE using a simple programming language.

| | | |
|---|---|---|
| *Pgm* | ::= | $(\ell\colon\ stmt\ ;)^*$ |
| *stmt* | ::= | $x = c$ |
| | | $x =$ *readInput* |
| | | $z = x \bowtie y$ |
| | | if $x$ goto $y$ |
| | | $y = *x$ |
| | | $*x = y$ |
| | | error |
| | | halt |
| where | | |
| $V$ | | is a set of variables |
| $C$ | | is the set of constants and statement labels |
| $A$ | | is a set of memory addresses |
| $x, y, z$ | | are elements of $V$ |
| $pc$ | | an element of $V$ denoting the program counter |
| $c$ | | is an element of $C \cup A \cup L$ |
| $\ell$ | | is an element of $L$ |
| $\bowtie$ | | is a binary operator |

**Figure 2.** Syntax of a simple imperative language

### 3.1 Syntax

The syntax of the language is shown in Figure 2. A program in the language is a sequence of labelled statements. We use $x =$ *readInput* to denote that $x$ gets an input from the environment. $*x$ denotes the memory cell whose address is stored in $x$. The language is similar to a simple untyped assembly language. Objects, references, and functions can be modeled using memory and memory address arithmetic: the heap grows from lower address to higher addresses and the call stack grows from higher address to lower addresses. Structured and unstructured control-flow, as well as exceptions, jump tables, can be modeled using if $x$ goto $y$ with computed jumps. Variables can be thought as the registers of the machine. The special variable $pc$ contains the program counter and $\ell_0$ is the label of the first statement of the program.

### 3.2 MULTISE Symbolic Execution Semantics

We use the following notations to describe the semantics of MULTISE execution:

- $S$ is the set of symbolic input values,
- $E$ is the set of all symbolic expressions built using the binary operators $\bowtie$ over elements of $S$, constants $C$, addresses $A$, and labels $L$,
- $F$ is the set of all propositional logical predicates over elements of $E$; we use $\phi$, $\phi'$, $\phi_i$ to denote a predicate in $F$,
- If $\ell$ is a statement label, then $Pgm(\ell)$ returns the statement in the program whose label is $\ell$.

The state of MULTISE is denoted by a mapping for variables and addresss to *value summaries*. A value summary is a set of guarded symbolic expressions, each consisting of a symbolic predicate along with a symbolic expression:

$$\Sigma \in (A \cup V) \to 2^{F \times E}$$

The predicate in a pair of a value summary is called a *path constraint*. Note that the program counter is represented as any other variable, which allows MULTISE to deal naturally with computed control flow constructs.

A key advantage of using a value summary is that often times a state can be represented in a concise form due to the following three observations:

Guarded Update

$$\frac{}{\{(\phi^a_i, v^a_i)\}_i \uplus_\phi \{(\phi^b_j, v^b_j)\}_j = \{(\neg\phi \wedge \phi^a_i, v^a_i)\}_i \uplus \{(\phi \wedge \phi^b_j, v^b_j)\}_j}$$

NextPC

$$\frac{}{\mathit{NextPC}(\Sigma, \phi, \ell) = (\Sigma(\mathit{pc}) \setminus \{(\phi, \ell)\}) \uplus \{(\phi, \ell + 1)\}}$$

Constant

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad Pgm(\ell) = (x = c)}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(\mathit{true}, c)\}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Symbolic Input

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad Pgm(\ell) = (x = \mathit{readInput}) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(\mathit{true}, s)\}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Binary Operation

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad Pgm(\ell) = (z = x \bowtie y) \qquad \Sigma(x) = \{(\phi^x_i, v^x_i)\}_i \qquad \Sigma(y) = \{(\phi^y_j, v^y_j)\}_j \qquad \phi^{x \bowtie y}_{ij} = \phi^x_i \wedge \phi^y_j \qquad v^{x \bowtie y}_{ij} = v^x_i \bowtie v^y_j}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(\phi^{x \bowtie y}_{ij}, v^{x \bowtie y}_{ij})\}_{ij}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Conditional

$$\frac{\begin{array}{c}(\phi, \ell) \in \Sigma(\mathit{pc}) \\ Pgm(\ell) = (\text{ if } x \text{ goto } y) \qquad \Sigma(x) = \{(\phi^x_i, v^x_i)\}_i \qquad \Sigma(y) = \{(\phi^y_j, \ell^y_j)\}_j \qquad s = \{(\phi^x_i \wedge v^x_i \wedge \phi^y_j, \ell^y_j)\}_{ij} \uplus \{((\phi^x_i \wedge \neg v^x_i), \ell + 1)\}_i\end{array}}{\Sigma \longrightarrow \Sigma[\mathit{pc} \mapsto (\Sigma(\mathit{pc}) \setminus \{(\phi, \ell)\}) \uplus_\phi s}$$

Load

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad Pgm(\ell) = (y = *x) \qquad \Sigma(x) = \{(\phi^x_i, v^x_i)\}_i \qquad \Sigma(v^x_i) = \{(\phi_{ij}, v_{ij})\}_j}{\Sigma \longrightarrow \Sigma[y \mapsto \Sigma(y) \uplus_\phi \{(\phi^x_i \wedge \phi_{ij}, v_{ij})\}_{ij}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Store

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad Pgm(\ell) = (*x = y) \qquad \Sigma(x) = \{(\phi^x_i, v^x_i)\}_i \qquad \Sigma(y) = \{(\phi^y_j, v^y_j)\}_j}{\Sigma \longrightarrow \Sigma[v^x_i \mapsto \Sigma(v^x_i) \uplus_{\phi \wedge \phi^x_i} \{\phi^y_j, v^y_j)\}_j)]_i[\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

**Figure 3.** Alternative Symbolic Execution Semantics using Value Summaries

- if $s$ is a value summary and $(\phi, v)$ and $(\phi', v')$ are any two distinct elements of $s$ such that $v = v'$, then we can replace the two elements with $\{(\phi \vee \phi', v)\}$ to obtain the equivalent value summary $s \setminus \{(\phi, v), (\phi', v')\} \cup \{(\phi \vee \phi', v)\}$.
- if $(\mathit{false}, v)$ is an element of a value summary, then it can be removed from the value summary to get an equivalent value summary.
- each guard in a value summary can be represented and manipulated efficiently using a binary decision diagram (or a BDD), which we discuss in detail later in the paper.

We take advantage of the above simplification rules by way of a special value-summary union operation. We write $s1 \uplus s2$ for the value summary obtained from the union of $s1$ and $s2$ followed by removing guarded expressions with guards that are unsatisfiable, i.e. guards that are equivalent to *false*, and replacing guarded expressions with the same symbolic expression with a single guarded expression using the union of the guards. Note that with an alternative implementation of $\uplus$ that does not do coalescing of repeated symbolic expressions we obtain an algorithm that operates essentially like conventional DSE, as explained in Section 3.4.

Figure 3 gives the operational semantics of MultiSE symbolic execution as a transition relation between MultiSE states:

$$\Sigma \longrightarrow \Sigma'$$

The execution starts from an initial state that maps each variable, except $\mathit{pc}$, to the value summary $\{(\mathit{true}, \bot)\}$, where ($\bot$ denotes the undefined value), and maps $\mathit{pc}$ to $\{(\mathit{true}, \ell_0)\}$, where $\ell_0$ denotes the first statement label.

The crucial operation used in the definition of the MultiSE algorithm is $s_1 \uplus_\phi s_2$, which given two value summaries $s_1$ and $s_2$ computes a value summary that should behave as $s_1$ on paths where $\neg\phi$ holds, and as $s_2$ on paths where $\phi$ holds. This function is defined in the rule Guarded Update.

The NextPC defines the function *NextPC* that is used to update the value summary for the program counter when advancing to the next statement. The Constant and Symbolic Input are simple rules that update the value of the assigned variable and the program counter. As for all assignments, we use the function $\uplus_\phi$ to ensure that we represent the fact that the assignment takes effect only on paths that satisfy the current path constraint $\phi$.

The rule Binary Operation triggers for a statement of the form $z = x \bowtie y$. The value summary for the right-hand side is computed by combining each symbolic expression for the variable $x$ with each symbolic expression for the variable $y$.

The Conditional rule is a bit more involved. For a computed jump of the form if $x$ goto $y$ we compute a value summary $s$ for the possible destination labels, including the cases when the jump is taken and those when it is not. For the cases when the jump is taken we consider every combination of the value summaries for $x$ and $y$, adding to the path constraint the condition that the symbolic expression for $x$ holds. For the cases when the jump is not taken, we consider every guarded expression in the value summary for $x$, along with the condition that $x$ is *false*. Finally, as shown in the conclusion of the rule, we do a guarded union of this set with the existing value summary for $\mathit{pc}$ .

The Load rule shows the lookup operation. For the statement $y = *x$, we first consider the value summary for `x` to obtain the possible addresses that `x` refers to. Then, we get the value summaries for these addresses as the value of $*x$.

The Store rule for statement $*x = y$ also considers first the value summary for `x` to obtain the possible addresses being written. Each of these addresses is updated with the value summary for `y`. Note the guard $\phi \wedge \phi^x_i$ in the guarded update for the address $v^x_i$, to model accurately the condition under which $v^x_i$ should be updated.

### 3.3 Approximation in MULTISE

There are several situations when MULTISE may need to approximate a concrete execution, in the sense that not all concrete execution paths will be represented in the symbolic state.

First, if the program contains a loop or a recursive function, and the loop termination condition or the recursion base case are input dependent, then MULTISE symbolic execution could run forever. In such cases we may want to stop the symbolic execution after a certain number of iterations. This is a typical problem with any kind of symbolic execution. This can be handled in MULTISE by simply dropping guarded symbolic expressions from the value summary of the program counter, e.g., when a label has been visited more than a certain number of times.

Second, it is possible for MULTISE to generate a symbolic expression that is outside the scope of the theories supported by the associated SMT solver, e.g., a product of symbolic expressions (if we assume that the associated SMT solver cannot handle non-linear arithmetic). Consider, for example, the following MULTISE state:

$$\begin{array}{l} \{ \\ \quad pc \mapsto \{\ldots,(\phi,\ell),\ldots\}, \\ \quad \mathtt{x} \mapsto \{(\phi_x,2),(\neg\phi_x,x_0)\}, \\ \quad \mathtt{y} \mapsto \{(\phi_y,3),(\neg\phi_y,y_0)\} \\ \} \end{array}$$

This state suggests that the variables `x` and `y` have been initialized with constants on some paths and with `readInput` on other paths. The label $\ell$, pointing to statement `z = x * y`, is reached under path constraint $\phi$. When evaluating the binary expression `x * y` under path constraint $\phi$, MULTISE combines the symbolic expressions from the value summaries of `x` and `y`, and one of the resulting guarded expressions will be $(\neg\phi_x \wedge \neg\phi_y, x_0 * y_0)$. If we assume that non-linear arithmetic is not supported by our SMT solver, MULTISE approximates it as follows. First, we find a satisfying assignment for $\phi \wedge \neg\phi_x \wedge \neg\phi_y$, from which we extract a possible concrete value for `x`, e.g., $x_0 = 5$. At this point we approximate by dropping from further consideration the concrete paths where $\phi \wedge \neg\phi_x \wedge \neg\phi_y \wedge x_0 \neq 5$. We do this by refining the path constraint for $pc$ to $\phi \wedge (\phi_x \vee \phi_y \vee x_0 = 5)$, to obtain the following MULTISE symbolic state:

$$\begin{array}{l} \{ \\ \quad pc \mapsto \{\ldots,(\phi \wedge (\phi_x \vee \phi_y \vee x_0 = 5),\ell+1),\ldots\}, \\ \quad \mathtt{z} \mapsto \{\ (\phi_x \wedge \phi_y, 3),(\phi_x \wedge \neg\phi_y, 2y_0), \\ \qquad\quad (\neg\phi_x \wedge \phi_y, 3x_0),(\neg\phi_x \wedge \neg\phi_y \wedge x_0 = 5, 5y_0)\} \\ \quad \ldots \\ \} \end{array}$$

This sort of simplification allows MULTISE to make progress and get around the limitations of the underlying SMT solver. When such an approximation happens, we set a flag `incomplete` to true indicating that MULTISE cannot guarantee full coverage of the code. This approach has the same end result as the simplification approach proposed in DART [23]. In DART, there was no need to use an SMT solver to find a concretization because DART could read the concrete value of $v$ from the concrete execution.

### 3.4 Conventional DSE as a Special Case of MULTISE

We show in this section that the rules shown in Figure 3 can be used to also model conventional DSE, although using a value-summary state representation, as long as we use a different definition for the value-summary union operation $\uplus$. Consider for example the following consolidated DSE state in a program with two paths ending at statement label 7:

| path | $\phi$ | $pc$ | x | y |
|---|---|---|---|---|
| $\ldots,7$ | $\phi$ | 7 | $x_0$ | $y_0$ |
| $\ldots,7$ | $\neg\phi$ | 7 | $x_0$ | $y_1$ |

This consolidated DSE state suggests that `x` has the same value on both paths, while `y` was assigned different values. In MULTISE, this consolidated state would be represented as:

$$\begin{array}{l} \{ \\ \quad pc \mapsto \{(\phi \vee \neg\phi, 7), \\ \quad \mathtt{x} \mapsto \{(\phi \vee \neg\phi, x_0)\}, \\ \quad \mathtt{y} \mapsto \{(\phi, y_0),(\neg\phi, y_1)\}, \\ \} \end{array}$$

or, more precisely in a form where the disjunctions are simplified to *true*. The disjunctions arise from the definition of value-summary union $\uplus$, which collapses together the guarded expressions with the same symbolic expression (e.g., 7 for $pc$ , and $x_0$ for `x`), and replaces the guard with the disjunction of the collapsed guards. If we use an alternate implementation of $\uplus$ that does not do not this minimization step, we would get a state as follows:

$$\begin{array}{l} \{ \\ \quad pc \mapsto \{(\phi, 7),(\neg\phi, 7), \\ \quad \mathtt{x} \mapsto \{(\phi, x_0),(\neg\phi, x_0)\}, \\ \quad \mathtt{y} \mapsto \{(\phi, y_0),(\neg\phi, y_1)\} \\ \} \end{array}$$

We argue that with such an implementation of the $\uplus$ operation, MULTISE mirrors essentially the operation of DSE. First, most importantly, the statement at program counter 7 would be processed twice, once for each element of the value summary for $pc$ . Furthermore, other rules behave just as DSE. For example, if the statement at label 7 is `x = x + y`, the rule BINARY OPERATION updates the value summary for `x` along the path with path constraint $\phi$ to:

$$\begin{array}{ll} & \{\quad (\neg\phi \wedge \phi, x_0),(\neg\phi \wedge \neg\phi, x_0)\} \\ \uplus & \{\quad (\phi \wedge \phi \wedge \phi, x_0 + y_0),(\phi \wedge \phi \wedge \neg\phi, x_0 + y_1), \\ & \qquad (\phi \wedge \neg\phi \wedge \phi, x_0 + y_0),(\phi \wedge \neg\phi \wedge \neg\phi, x_0 + y_1)\} \end{array}$$

The first line in the above value summary is from the left-hand side of the $\uplus_\phi$ operator applied on the previous value of `x` along with the $\neg\phi$ guard. The four guarded values in lines 2 and 3 are from the calculation of `x + y` by combining the value summaries of `x` and `y`, along with the conjunct $\phi$ from the right-hand side of the $\uplus_\phi$ operator. Once we simplify the boolean expressions we retain the value summary $\{(\neg\phi, x_0),(\phi, x_0 + y_0)\}$, which is a correct representation of the DSE state after processing `x = x + y` under the path constraint $\phi$ for the program counter. After one more step, when we process the $pc$ guarded value $(\neg\phi, 7)$, we obtain the state:

$$\begin{array}{l} \{ \\ \quad pc \mapsto \{(\phi, 8),(\neg\phi, 8), \\ \quad \mathtt{x} \mapsto \{(\phi, x_0 + y_0),(\neg\phi, x_0 + y_1)\}, \\ \quad \mathtt{y} \mapsto \{(\phi, y_0),(\neg\phi, y_1)\}, \\ \} \end{array}$$

We point out that, modulo the minimization of the value summary for $pc$ , this is the same as the state MULTISE would have arrived at in *only one step* due to the fact that it maintains the value summary in minimized form. We used such a configuration of MULTISE to evaluate the reduction in the number of operations performed, and in the total running time of MULTISE compared with conventional DSE. We discuss the results in Section 5.2.

## 4. Soundness of MULTISE Symbolic Execution

There are two correctness results that we desire for MULTISE, which we first summarize informally:

- Soundess w.r.t. concrete executions: Any program behavior encoded in the final symbolic state of MULTISE corresponds to a concrete program behavior, and
- Soundness and completeness w.r.t. DSE: The final symbolic state of MULTISE encodes exactly the same set of behaviors as the final symbolic state of DSE.

We note also that the only reason completeness does not hold w.r.t. concrete executions is due to the approximations discussed in Section 3.3. Those approximations drop concrete paths from the symbolic representation. However, the paths that are kept are still faithfully represented in the symbolic state.

To state these results formally, we assume without loss of generality that all the `readInput` statements occur as consecutive statements at the start of the program. Let $\Sigma_0$ be the MULTISE state after these statements have been executed symbolically. Essentially, $\Sigma_0$ will map the program counter to $\{(\mathit{true}, \ell_0)\}$, will map the input variables to distinct symbolic values, and other variables and addresses to the undefined value ($\perp$). We are going to refer to $\Sigma_0$ as the initial symbolic state.

The concrete executions of the program can be formalized by a transition relation between concrete states. A concrete state is denoted as $\rho \colon (A \cup V) \to (A \cup C \cup L)$. The definition of this concrete transition relation $\rho \longrightarrow \rho'$ is standard, and we do not show it here.

We define the meaning of symbolic expressions and predicates by a denotation function $[\![\cdot]\!]$ that given a symbolic expression (a member of $E$) and a mapping of symbolic values to integer constants (a member of $S \to C$), yields a value. Thus,

$$[\![\cdot]\!] : E \to (S \to C) \to (A \cup C \cup L)$$

We lift the $[\![\cdot]\!]$ function to MULTISE states. Given a mapping $V$ of symbolic values to constants, a MULTISE state $\Sigma$ denotes the concrete state $[\![\Sigma]\!]V$ defined as follows, where $x$ is a variable or address:

$$[\![\Sigma]\!]Vx = \begin{cases} [\![v_i]\!]V & \text{if } \exists(\phi_i, v_i) \in \Sigma(x) \text{ such that } [\![\phi_i]\!]V \\ \perp & \text{otherwise} \end{cases}$$

Note that given any mapping $V$, the concrete state $[\![\Sigma_0]\!]$ is a concrete initial state, where $\Sigma_0$ is the symbolic evaluation state after all `readInput` statements, as discussed above.

The first soundness result states that a symbolic state denotes only actual concrete executions, in the following sense:

THEOREM 1 (Soundness w.r.t. the concrete executions). *If $\Sigma$ is a symbolic state obtained from the initial state $\Sigma_0$, i.e., $\Sigma_0 \longrightarrow^* \Sigma$, then for any mapping $V$ of symbolic values to integer constants such that $[\![\Sigma]\!]V\ pc \neq \perp$ we have that $[\![\Sigma]\!]V$ is a concrete state that is reached from the initial state $[\![\Sigma_0]\!]V$ in an actual execution, i.e., $[\![\Sigma_0]\!]V \longrightarrow^* [\![\Sigma]\!]V$.*

The proof of this soundness theorem can be done by induction on the length of the MULTISE derivation $\Sigma_0 \longrightarrow^* \Sigma$, and the inductive case by case analysis on the transition rules in Figure 3.

The second correctness result follows from the fact that the only difference between MULTISE and DSE is the implementation for $\uplus$: in DSE we do not merge guarded expressions in $\uplus$, whereas in MULTISE we do. Value summaries obtained from the two implementations of $\uplus$ are logically equivalent.

## 5. Implementation and Evaluation

We have implemented a prototype framework for MULTISE execution for JavaScript using the Jalangi framework [47] and we made it publicly available under Apache 2.0 open-source licence (`https://github.com/SRA-SiliconValley/jalangi` under branch `symfront`). We use CVC3 [7] for constraint solving, to handle the theory of integer linear arithmetic and strings (with append, length, equality check, parseInt, and regular expression matching). We encode string operations in terms of integer linear arithmetic after bounding the lengths of strings. These theories are often sufficient for handling integer and string inputs in JavaScript.

In conventional DSE, at join points, state merging needs to iterate over the part of the symbolic state that has been modified by the paths converging at the join point and merge that part of the state. Identifying the join points, keeping track of the modified part of the state, and merging the modified state in conventional DSE require one to implement a symbolic interpreter. While implementing MULTISE we observed that due to our incremental state merging approach we do not need to keep track of variables and memory addresses that has been updated along different paths before a join. Moreover, since we do not rely on join points, we do not need to compute the static control-flow graph of the program. These observations immensely simplified the implementation of MULTISE and we managed to finish our implementation through instrumentation without going through the expensive path of implementing a full-fledged symbolic interpreter for JavaScript. We believe that MULTISE can easily be implemented for x86 and other low-level languages, such as LLVM and Java bytecode, and other high-level languages such as Python and Ruby.

The implementation provides a general framework where we can choose when to merge two guards in a value summary. Note that the soundness result for MULTISE holds irrespective of whether we merge guards or not. In our framework, we can set a flag to indicate if we want to merge guards. If we do not merge guards, we get conventional symbolic execution. At the end of conventional symbolic execution $pc$ maps to a value summary where for each feasible path we have a statement label guarded by the path constraint for the path. We use this flag to execute and compare MULTISE and DSE on several unit test programs.

Every time a new guarded symbolic expression is added to the value summary for $pc$, we invoke a quick BDD satisfiability check followed by an SMT solver satisfiability check for the path constraint. This is important in order to avoid exploring unfeasible paths. For the value summaries of other variables, only a BDD satisfiability check is used, to reduce the overall cost of SMT solving. During both MULTISE and DSE execution we generate an input for each satisfiable SMT solver call made by the respective techniques at a conditional statement. We generate inputs only at conditional statements because one of the key goals of symbolic execution is to generate a set of inputs that give maximal branch coverage. Generating inputs that forces program execution along both branches of a conditional statement ensures that we maximize branch coverage for the particular conditional statement.

As explained earlier, MULTISE does not prescribe a particular search order. In our implementation, we perform *function-bounded depth-first search*. In this search strategy, we completely explore all paths inside a function at the top of the call stack using the depth-first search strategy before returning to the caller. During this search process, MULTISE naturally merges $pc$ variables at the function boundaries. The values of all other variables are merged incrementally during assignments.

### 5.1 Using Binary Decision Diagrams To Represent Guards

During a MULTISE execution, we perform a lot of disjunction, conjunction, and negation operations on symbolic predicates. For example, in the rule CONDITIONAL we compute $\phi \wedge \bigvee_i(\phi_i^x \wedge \neg v_i^x)$. These formulas could easily become complex if they are not simplified on-the-fly. In our initial implementation of MULTISE, we didn't simplify the formulas. As a result we ended up generating huge path constraints and MULTISE execution ran out of memory even for simple programs such as quick sort.

We use binary decision diagrams (BDDs) to represent path constraints and guards in value summaries as follows. Note that symbolic predicates arise from conditional expressions, as shown in the CONDITIONAL rule. This happens via $v_i^x$ when we compute $s$ in the rule. For each unique symbolic predicate $v_i^x$ generated in a

| Test | LOC | MULTISE | | | | | | DSE/ MULTISE ratio | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Total time (s) | BDD time (%) | Solver time (%) | Avg. value summary size | Avg. value summary sharing factor | Auxiliary variables avoided | Time ratio (× speedup) | Solver time ratio | Avg. solver call time ratio | # Operations ratio |
| Find Max | 32 | 5.0 | 1.2 | 97.9 | 1.9 | 23.0 | 0 | 10.0 | 9.9 | 0.8 | 28.7 |
| Kadane Subarray | 38 | 6.5 | 1.0 | 98.4 | 2.4 | 3.2 | 0 | 2.7 | 2.6 | 0.9 | 6.9 |
| Array Index | 56 | 11.7 | 5.3 | 93.4 | 9.1 | 9.1 | 0 | 3.7 | 3.9 | 0.9 | 3.3 |
| Calc Parser | 66 | 35.5 | 8.9 | 90.2 | 20.4 | 9.8 | 0 | 1.6 | 1.6 | 1.0 | 2.8 |
| Stack | 81 | 0.6 | 6.2 | 89.0 | 2.4 | 7.7 | 44 (41.5%) | 26.2 | 29.2 | 1.2 | 9.1 |
| Queue | 85 | 0.3 | 0 | 93.1 | 1.0 | 5.4 | 0 | 6.7 | 7.2 | 1.1 | 4.2 |
| Heap Sort | 87 | 4.0 | 1.5 | 96.7 | 1.7 | 5.6 | 0 | 2.5 | 2.5 | 1.0 | 8.5 |
| Quick Sort | 93 | 15.1 | 4.6 | 94.6 | 3.6 | 7.1 | 0 | 2.6 | 2.7 | 3.0 | 3.7 |
| PL/0 Parser | 135 | 246.4 | 18.7 | 80.4 | 29.3 | 45.8 | 0 | 1.3 | 1.2 | 0.9 | 2.7 |
| Linked List | 148 | 2.5 | 3.6 | 95.1 | 2.8 | 5.3 | 124 (56.8%) | 11.1 | 11.6 | 0.9 | 5.1 |
| Priority Queue | 190 | 0.9 | 3.2 | 92.3 | 1.2 | 31.5 | 10 (10%) | 87.7 | 94.5 | 1.3 | 47.5 |
| Binary Search Tree | 386 | 6.5 | 2.4 | 96.6 | 2.4 | 9.4 | 188 (78.3%) | 7.3 | 7.4 | 0.9 | 5.6 |
| Symbolic Arithmetic | 475 | 1.5 | 9.1 | 82.3 | 1.8 | 39.3 | 168 (28.3%) | 49.3 | 51.4 | 40.4 | 34.0 |
| BDD | 623 | 6.2 | 68.2 | 19.6 | 2.5 | 6.4 | 15838 (59.2%) | 7.5 | 29.6 | 24.3 | 5.4 |
| Red Black | 1061 | 37.1 | 11.3 | 88.0 | 3.5 | 43.6 | 1088 (57.9%) | 6.5 | 7.1 | 0.7 | 8.8 |

**Table 1.** Results: DSE vs MULTISE

conditional statement, we introduce a Boolean variable. A guard or a path constraint is expressed as a Boolean formula in terms of these Boolean variables. We use binary decision diagrams (BDDs) to compactly represent Boolean formulas over these Boolean variables. Conjunction, disjunction, and negation of Boolean formulas are computed by performing the corresponding operations on the BDDs denoting the formulas. If we need to check if a guard or a path constraint $\phi$ is satisfiable, we first check if its BDD representation is not *false* and then we replace each Boolean variable in the formula by its corresponding symbolic predicate and check the satisfiability of the resulting formula using an SMT solver. The ordering on the Boolean variables in a BDD is the same as the order in which they are created. We noticed that the use of BDDs helped us to effectively maintain and manipulate the guards. We use an unoptimized textbook implementation of BDD in JavaScript. Despite being unoptimized, we noticed that the total overhead due to BDD manipulation is significantly lower that the total overhead due to SMT solving. For example, in our experiments, on an average we spend less than 10% of total execution time in BDD manipulation, whereas over 85% of total execution time is spent in SMT solving.

## 5.2 Evaluation

We experimented with the prototype implementation of MULTISE to evaluate its effectiveness at sharing values in the value summaries, at avoiding the need for auxiliary variables, and to measure the total cost of symbolic evaluation and how much of it is due to BDD or to SMT solver calls. We also measured the speedup over conventional DSE.

For the evaluation we ran MULTISE on several test harnesses created from publicly available JavaScript libraries. A symbolic test harness for a library is created by sequentially calling the methods of the library (possibly with repetitions) with inputs marked as `readInput`. Even when the tested library is small the execution trees can be quite large if the test harnesses contain several library invocations.

Table 1 shows various statistics that we collected by running MULTISE and conventional DSE on our benchmark programs. The programs used in our evaluation include parsers (calculator parser and PL/0 parser), data structures (red-black tree, binary-decision diagrams, linked list, stack, priority queue, binary search tree and queue), standard algorithms (quick sort, heap sort, Kadane maximum subarray), and small programs (find max and array index).

The first few columns in Table 1 summarize MULTISE performance and effectiveness, while the columns on the right side of the double vertical line summarize comparisons between DSE and MULTISE. The experiments were performed on a laptop with 2.3 GHz Intel Core i7 and 16 GB RAM.

The "Total time" column reports the total running time of MULTISE in seconds, and the columns "BDD time" and "Solver time" report the percentage of time spent in BDD manipulation and in SMT solving running time, respectively. We observe that even though MULTISE involves numerous boolean predicate constructors, the overall time spent in the BDD library is negligible. The SMT solver time takes most of the time. Note that we use SMT solver only in CONDITIONAL rule when we perform $\uplus_\phi$ to update the value of $pc$ . Another observation is that compared to the SMT time, the time actually spent in interpreting statements and constructing symbolic expressions is also very small.

The column "Avg. value summary size" reports the cardinality of the value-summary set, averaged over all variables during the execution of MULTISE. We observe that in many cases the value summaries are small (between 1 and 30). The smaller the size of a value summary, the more efficient it is to perform an operation on the value summary. This is especially true for statements that involve multiple variables, such as binary operations and conditionals, where we need to process all combinations of the value summaries involved. A related measure is shown in the "Avg. value summary sharing factor". This column contains the ratio between the number of paths to a point in the program and the size of the value summary, averaged over all variables and all program points. Recall that the size of a value summary is given by the number of distinct symbolic values for a variable at a point in the program. Our experiments show that the distinct values are shared on average between 3 to 45 paths. This validates our premise that there is a significant opportunity for a representation based on sharing.

The column "Auxiliary variables avoided" is showing how many auxiliary variables of types other than integer or string would be introduced by conventional state-merging techniques. These auxiliary variables would be problematic for most SMT solvers. We note that for several of the benchmarks there would be such variables, sometimes in the thousands, and each variable would force a state-merging based tool to drop paths from symbolic execution. MULTISE never introduces auxiliary variables and can proceed along all paths even when dealing with variables of types not supported by the constraint solver.

The right side of Table 1 shows how much value-summary based symbolic execution improves over conventional DSE. In the "Time ratio" column we show how much more time it takes to run

DSE compared to MULTISE. We observe a significant speedup, between 1.3× and up to 87×. This speedup is due to two related factors. First, DSE performs a lot more operations than MULTISE because it processes statements following a join multiple times, as shown in the "# Operations ratio". This column shows how many more operations DSE has to perform compared to MULTISE. Note that for each statement processed by MULTISE we count as many operations as the size of the value summary at that statement. Second, DSE spends significant more time in SMT solver calls, as shown in "Solver time ratio" (DSE/MultiSE). Finally, the column "Avg. solver call time ratio" shows the ratio between the average duration of a call to the SMT solver in DSE vs. MULTISE. We present this number to show that even in the face of more complicated constraints in MULTISE the cost of an individual SMT call is not higher. The real problem is the higher number of SMT calls that DSE must make.

## 6. Related Work

Recently several techniques for state merging [2, 6, 22, 26, 34, 36, 52] have been proposed to tackle the path-explosion problem. Dynamic state merging [36] uses a novel method called QCE (query count estimation) which determines statically when merging two states is advantageous, while at the same time allowing the state exploration to be guided by arbitrary strategies. MergePoint [6] alternates between path-based exploration of DSE and state-merging based exploration of static symbolic execution. State merging is only performed for code that does not contain system calls, indirect jumps, or other statements that are difficult to reason about precisely. Both of these techniques introduce auxiliary symbolic values and cannot merge states when there are unstructured control-flow and operations that introduce outside-theory constraints over auxiliary symbolic values. Rozzle [34] does not introduce auxiliary variables and performs merging at join points, but could give rise to formulas outside the domain of a constraint solver. Rossette [52] also does state merging and manages to avoid some of the auxiliary variables. Rosette's state merging happens at join points and is type-based where two data-structure values, such as two lists having same length, are merged recursively to further compact the merged state. Recursive merging of data-structure values only works for immutable data-structures and cannot be applied to get state compaction in mutable data-structures used in imperative languages. SMART [22] performs compositional test generation by computing summaries of all program functions. The summary of a function is computed by exploring all paths of the function using DSE and then by merging the symbolic states of those paths via symbolic auxiliary variables. For real programs, SMART can generate function summaries that are outside the theories that can be handled by an SMT solver. In such situations, it simplifies the summaries at the cost of completeness. Demand-driven compositional symbolic execution [2] was subsequently proposed to incrementally construct partial summaries to avoid analyzing unnecessary paths in functions. SMASH [26] incorporates both symbolic execution summaries (must summaries) and static analysis summaries (may summaries). SMASH performs reachability analysis to reason about possible buggy program states and to prune out group of uninteresting execution paths. All the above three techniques inherit the same limitation: they introduce auxiliary symbolic values at function interfaces. Therefore, they can reason about a function precisely only if the function's behavior can be captured by the given decidable theories.

Another line of work [8, 14, 30, 31, 38, 39, 53] tries to mitigate the path-explosion problem by pruning out redundant or unnecessary executions. Most of these techniques are orthogonal to compositional reasoning and state merging. A subset of these techniques avoid redundant executions by checking whether the current symbolic program state has been visited before. JPF [53] first uses state matching to avoid redundant state exploration. Boonstoppel et al. [8] uses read and write sets to relax state matching condition. McMillan [39] proposed the idea to store interpolants as a generalization of visited states and to check inclusion instead of exact state matching. Tracer [30, 31], a symbolic execution engine targeting C programs, proposes to use interpolants to mitigate the path explosion problem by subsuming paths that can be proved to be safe. Another subset of these techniques try to decompose the program execution space into a number of independent sub spaces. For example, Majumdar and Xu [38] and Chakrabarti and Godefroid [14] applied program slicing ideas to cluster the program execution space. Recently, Santelices et al. [44], Qi et al. [41], Godefroid et al. [24], and Yang et al. [56] suggested incremental symbolic execution techniques to reduce the cost of regression testing of gradually evolving programs.

Function summaries [42, 49] have been used extensively in static program analysis. Graph-reachability based analysis [42], constraint-based analysis [29, 55], pointer analysis [15, 54], alias analysis [20], shape analysis [35], separation logic [13, 27], and abstract interpretation [9, 43] have incorporated function summaries for scalability. More recently, Gulwani et al. [28] and Yorsh et al. [57] investigated a general framework for summary-based static analysis. In general, function summaries in static analysis use interface symbolic values at function boundaries. Summaries are instantiated by replacing interface variables with real variables at call sites. Therefore, static analysis faces the same problem: a function behavior can be captured precisely only if it can be described in underlying decidable theories. However, this is not a serious limitation for static analysis because summaries can always be over approximated. Dynamic symbolic execution cannot over approximate a summary since over-approximation leads to loss of soundness.

Saturn [55] is one of the most closely related static analysis techniques. For intra-procedural analysis, Saturn and MULTISE have a number of commonalities: both perform symbolic execution, use guarded values to track values in a path sensitive manner, and maintain a path constraint. However, Saturn introduces fresh symbolic variables at function interfaces and updates guards of values at join points. MULTISE, in contrast, never introduces auxiliary symbolic values and performs join at every assignment to maintain a consolidated state throughout the execution. Guarded value flow analysis [16] is another closely related work. It performs value flow analysis using both path constraints and guards on values. Value can flow from a source to a sink if the conjunction of the guard on a value at the sink and the path constraint at the sink is satisfiable. However, the technique computes path constraint and guards on demand, while MULTISE performs symbolic execution to obtain guarded values at every execution point.

In general, one of the biggest advantages of static analysis techniques using summaries is that they can over-approximate a summary if it falls outside the scope of decidable theories; dynamic symbolic execution cannot over-approximate because it needs to know the exact path constraint for test generation.

## 7. Acknowledgments

The first and last author did part of this work while visiting Samsung Research America, San Jose. The work presented in this paper is supported in part by NSF grants CCF-1017810, CCF-0747390, CCF-1018729, CCF-1423645, and CCF-1018730, and gifts from Samsung and Mozilla.

## References

[1] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The key platform for verification and analysis of Java programs. In D. Giannakopoulou and D. Kroening,

editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, Lecture Notes in Computer Science. Springer-Verlag, 2014.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.

[3] S. Anand and M. J. Harrold. Heap cloning: Enabling dynamic symbolic execution of java programs. In *ASE*, pages 33–42, 2011.

[4] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS'07*, 2007.

[5] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA'08*, 2008.

[6] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[7] C. Barrett and C. Tinelli. CVC3. In $19^{th}$ *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302, 2007.

[8] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, 2008.

[9] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On interprocedural analysis of programs with lists and data. In *PLDI*, pages 578–589, 2011.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[11] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, Sept. 2008.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, Dec 2008.

[13] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.

[14] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *EMSOFT*, pages 262–271, 2006.

[15] B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

[16] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI*, 2007.

[17] V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2, 2012.

[18] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[19] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54:69–77, Sept. 2011.

[20] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

[21] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In G. E. Harris, editor, *OOPSLA*. ACM, 2008.

[22] P. Godefroid. Compositional dynamic test generation. In *POPL'07*, Jan. 2007.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'05*, June 2005.

[24] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, pages 112–128, 2011.

[25] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'08*, Feb. 2008.

[26] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL'10*, 2010.

[27] B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori. Bottom-up shape analysis using lisf. *ACM Trans. Program. Lang. Syst.*, 33(5):17, 2011.

[28] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, pages 253–267, 2007.

[29] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *PLDI*, 2003.

[30] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, New York, NY, USA, 2013. ACM.

[31] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Tracer: A symbolic execution tool for verification. In *CAV*, pages 758–766, 2012.

[32] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *In NFM'09*, Apr. 2009.

[33] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[34] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 443–457, Washington, DC, USA, 2012. IEEE Computer Society.

[35] J. Kreiker, T. W. Reps, N. Rinetzky, M. Sagiv, R. Wilhelm, and E. Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. In *Programming Logics*, pages 414–445, 2013.

[36] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204, 2012.

[37] G. Li, I. Ghosh, and S. P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *CAV*, pages 609–615, 2011.

[38] R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV'09*, LNCS, pages 555–569, 2009.

[39] K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, pages 104–118, 2010.

[40] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA'08*, July 2008.

[41] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/FSE*, pages 33–42, 2009.

[42] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[43] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186, 2011.

[44] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, pages 218–227, 2008.

[45] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528. IEEE, 2010.

[46] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV'06*, 2006.

[47] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE'13*, August 2013. To appear.

[48] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*, Sep 2005.

[49] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[50] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS'08*, Dec. 2008.

[51] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *TAP'08*, Apr 2008.

[52] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM.

[53] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA'06*, July 2006.

[54] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

[55] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363, 2005.

[56] G. Yang, S. Khurshid, and C. S. Pasareanu. Memoise: a tool for memoized symbolic execution. In *ICSE*, pages 1343–1346, 2013.

[57] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234, 2008.