# Memoized Symbolic Execution

Guowei Yang
University of Texas at Austin
Austin, TX 78712, USA
guoweiyang@utexas.edu

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley
NASA Ames, M/S 269-2
Moffett Field, CA 94035, USA
corina.s.pasareanu@nasa.gov

Sarfraz Khurshid
University of Texas at Austin
Austin, TX 78712, USA
khurshid@ece.utexas.edu

## ABSTRACT

This paper introduces *memoized symbolic execution* (*Memoise*), a new approach for more efficient application of forward symbolic execution, which is a well-studied technique for systematic exploration of program behaviors based on bounded execution paths. Our key insight is that application of symbolic execution often requires several successive runs of the technique on largely similar underlying problems, e.g., running it once to check a program to find a bug, fixing the bug, and running it again to check the modified program. Memoise introduces a *trie*-based data structure that stores the key elements of a run of symbolic execution. Maintenance of the trie during successive runs allows re-use of previously computed results of symbolic execution without the need for re-computing them as is traditionally done. Experiments using our prototype implementation of Memoise show the benefits it holds in various standard scenarios of using symbolic execution, e.g., with iterative deepening of exploration depth, to perform regression analysis, or to enhance coverage using heuristics.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

## General Terms

Verification, Algorithms

## Keywords

Symbolic Execution, Incremental Analysis, Trie Data Structure, Constraint Solving

## 1. INTRODUCTION

Forward symbolic execution [15, 7, 10, 22, 19, 6] is a powerful technique that is gaining popularity for systematic exploration of program behaviors. The technique enumerates the program paths (of interest) and records as formulas the conditions on the inputs to follow the different paths, as dictated by the branches in the code. Off-the-shelf constraint solvers are used to reason about the formulas to discard those paths whose conditions are unsatisfiable. In practice, the technique can be costly to apply due to its inherent high time and space complexity. There are two key factors that determine its cost: (1) the number of paths that need to be explored and (2) the cost of constraint solving.

Recent years have seen substantial advances in raw computation power and constraint solving technology [1], as well as in basic algorithmic approaches for symbolic execution [4, 25]. These advances have made symbolic execution applicable to a diverse class of programs and enable a range of analyses, including bug finding using automated test generation – a traditional application of this technique – as well as other novel applications, such as program equivalence checking [23], regression analysis [17], and continuous testing [27]. All these applications utilize the same path-based analysis that lies at the heart of symbolic execution. As such, their effectiveness is determined by the two factors that determine the cost of the symbolic execution, and at present, reducing the cost of symbolic execution remains a fundamental challenge.

This paper introduces *memoized symbolic execution* (*Memoise*), a new approach that addresses both factors to enable more efficient applications of symbolic execution. Our key insight is that applying symbolic execution often requires several successive runs of the technique on largely similar underlying problems, e.g., running it once to check a program to find a bug, fixing the bug, and running it again to check the modified program. Memoise leverages the similarities to reduce the total cost of applying the technique by maintaining and updating the *state* of a symbolic execution run. Specifically, Memoise uses a *trie* [8, 28] – an efficient tree-based data structure – for a compact representation of the symbolic paths generated during a symbolic execution run. Essentially, the trie records the *choices* taken when exploring different paths, together with bookkeeping information that maps each trie node to the corresponding condition in the code. Maintenance of the trie during successive runs allows *re-use* of previously computed results of symbolic execution without the need for re-computing them as is traditionally done. Constraint solving is turned off for the paths that were previously explored and the search is guided by the choices recorded in the trie. Moreover, the search is pruned for the paths that are deemed to be no longer of interest for the analysis. To keep the cost of Memoise small we further define two operations on the trie: *compression* (to discard "un-interesting" trie branch sequences) and *merging* (of compressed tries obtained in successive runs of Memoise).

We developed a prototype tool for memoized symbolic execution of Java programs. The implementation uses the Symbolic PathFinder tool [19], part of the Java PathFinder open-source framework. Experiments show that the space and time cost of storing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ISSTA*'12, July 15–20, 2012, Minneapolis, MN, USA
Copyright 2012 ACM 978-1-4503-1454-1/12/05 ...$15.00

```
1 int compute(int curr, int thresh, int step){
2   int delta = 0;
3   if (curr < thresh){
4     delta = thresh - curr;
5     if ((curr + step) < thresh)
6       return -delta;
7     else
8       return 0;
9   } else {
10      int counter = 0;
11      while (curr >= thresh) {
12        curr = curr - step;
13        counter++;
14      }
15      return counter;
16  }
17 }
```

**Figure 1: Example program**

and retrieving the trie is small and that it is far out-weighted by the benefits Memoise obtains in standard scenarios of using symbolic execution, such as, with iterative deepening of exploration depth, regression analysis, and heuristic search to enhance coverage. The approach introduced by Memoise also holds much promise in optimizing a variety of other analyses based on symbolic execution, as discussed in Section 4.

## 2. BACKGROUND

Symbolic execution [15, 7] is a program analysis technique that uses symbolic values, instead of actual data, as inputs to execute a program fragment, e.g. a program or a method within a program. The technique represents the values of program variables as symbolic expressions and it computes the outputs as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables and a path constraint ($PC$). The path constraint is a (quantifier free) Boolean formula over the symbolic inputs; it accumulates the constraints on the inputs in order for an execution to follow the particular associated path. A *symbolic execution tree* characterizes the paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

We illustrate symbolic execution on the program in Figure 1, that we will use as a running example throughout the paper. Method `compute` has three integer inputs: `curr` (current), `thresh` (threshold) and `step`; it calculates the relationship between the current and the threshold, in increments given by the step value.

Figure 2 shows the corresponding symbolic execution tree. Initially, $PC$ is *true* and `curr`, `thresh` and `step` have symbolic values $S_1$, $S_2$ and $S_3$, respectively. Program variables are assigned expressions in terms of these symbolic inputs; e.g., after executing statement 4, `delta` becomes $S_2 - S_1$. At each branch point, there is a *choice* in the execution and $PC$ is updated with assumptions about the inputs, to choose between alternative paths. For example, after the execution of statement 3, both `then` and `else` alternatives of the `if` statement are possible, and $PC$ is updated accordingly. Whenever $PC$ is updated, it is checked for satisfiability using off-the-shelf decision procedures. If $PC$ becomes false (there are no inputs that satisfy it) it means the state is un-reachable, and symbolic execution does not continue for that path. This happens when the `while` statement at line 11 is executed the first time: the $PC$ corresponding to the condition for exiting the loop is unsatisfiable.Test inputs are generated by solving the collected $PC$s.

Symbolic execution of looping programs may result in an infinite symbolic execution tree (see Figure 2 where the expansion of the right-most leaf in the tree may continue forever). For this reason, one needs to put a limit on the depth of the search for symbolic paths, and iteratively increase that depth until either an error is found or the desired testing coverage has been achieved.

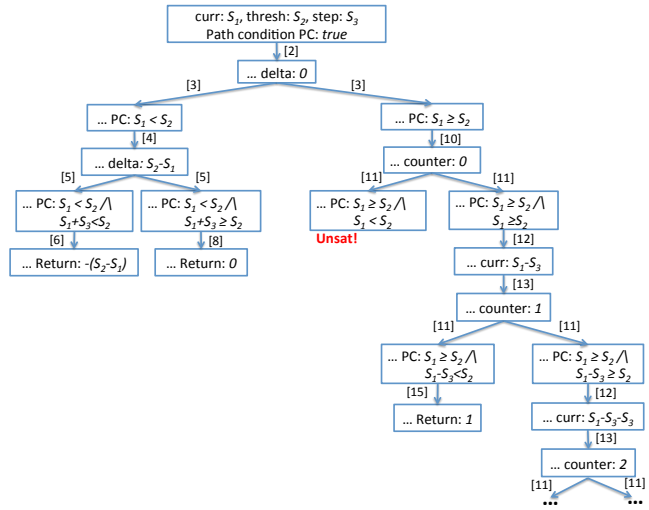


**Figure 2: Symbolic execution tree**

## 3. MEMOIZED SYMBOLIC EXECUTION

Given program $p$ and execution depth bound $b$, *memoized symbolic execution* (*Memoise*) addresses the problem of running symbolic execution on *problem instance* $\langle p, b\rangle$ given that symbolic execution was already performed on problem instance $\langle p_{old}, b_{old}\rangle$. Memoise leverages the results of running symbolic execution on $\langle p_{old}, b_{old}\rangle$ by caching them and re-using them when running symbolic execution on $\langle p, b\rangle$. To cache a symbolic execution run, Memoise builds an efficient *trie* data structure [8, 28] for representing compactly the global state of a symbolic execution run, i.e. the choices taken during symbolic execution. A trie (prefix tree) is an ordered tree that enables efficient re**trie**val of the information stored in it. The benefits of using the trie are two-fold: first, users can easily retrieve the symbolic execution results of the same system repeatedly; second, if the system undergoes development or it is checked with a greater bound, only part of the data structure needs to be maintained, which should be cheaper than re-running the symbolic execution from scratch.

We say that trie $t$ is *complete* for program $p$ and bound $b$ if it encodes all the choices taken during the symbolic execution of $p$ up to bound $b$. The Memoise approach has three basic steps:

- **Initialization.** An initial run of Memoise performs standard symbolic execution as well as builds the trie on-the-fly and saves it on the disk for future re-use.
- **Memoized analysis.** The trie built during the initialization run or a previous run of Memoise is loaded in memory and it is used to guide analysis for iterative deepening (Section 4.1.1), regression (Section 4.1.2), or application of heuristics (Section 4.1.3). During the analysis, a new trie is built/updated on-the-fly, which is saved back on the disk. As an optimization, the memoized analysis performs a *compression* on the input trie to remove the components that are irrelevant in the context of the particular application scenario.
- **Trie merging.** The (compressed) trie built during memoized analysis is (optionally) *merged* with the old trie to obtain a complete trie for $\langle p, b\rangle$ .

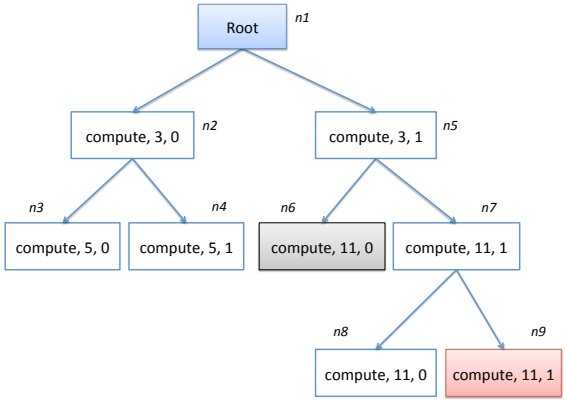


**Figure 3: Example trie**

## 3.1 Initialization

The trie is built on-the-fly during symbolic execution. We represent the trie using a recursive data structure – a tree of nodes where each node has a list of children nodes. Whenever a conditional instruction is symbolically executed a trie node is created, recording the location of the symbolic conditional, i.e., method and the instruction offset, and the choice taken by the execution. The trie stores just enough information to guide symbolic execution in future runs (through the stored choices) and to map back the nodes to the program constructs (for e.g. regression analysis). Memoise does not require the trie to explicitly store the actual path conditions, which are simply re-created in future runs. However, should such storage be desired by the user, the trie data structure facilitates a compact storage using a distributed representation of path conditions (similar to the internal representation used in SPF [18, 19]). To facilitate future runs of symbolic execution, a subset of the leaf nodes in a trie is partitioned into a set of *boundary* nodes, which are leaf nodes because of the chosen depth bound, and a set of *unsatisfiable* nodes, which are leaf nodes due to unsatisfiable path conditions. Figure 3 shows the (complete) trie for our running example (for depth 3). In this figure $n9$ is a boundary node and $n6$ is an unsatisfiable node.

The search order can be e.g. depth-first search or breadth-first-search. The search order does not really matter as long as the same order is used during re-execution.

## 3.2 Memoized Analysis

Memoized symbolic execution enables efficient re-execution based on the results cached in the trie structure.

### 3.2.1 Node Marking

The first step in memoized execution is to *mark* nodes of *interest*. Specifically, we characterize parts of the old trie that may be updated using *candidate* nodes, which represent roots of sub-trees potentially updated during memoized execution. Given the candidate nodes, we mark nodes on paths that need re-execution – all nodes on any path from the trie root to a *candidate* node are marked (while the rest of the nodes remain unmarked). The exact classification of candidate nodes depends on the particular analysis that is performed. For example, for iterative deepening, the boundary nodes are the candidate nodes (e.g., $n9$ in Figure 3). regression analysis the nodes that are impacted by the program change are considered as candidate ones (the impacted nodes are found by an impact analysis as described in Section 4.1.2). The node marking is reset at the beginning of memoized analysis.

```
1 void compress(Trie trie) {
2   List<Node> worklist = new LinkedList<Node>();
3   worklist.add(trie.root);
4   while (!worklist.isEmpty()) {
5     Node curr = worklist.remove(0);
6     if (!curr.isMarked()) {
7       curr.removeChildren();
8       continue;
9     }
10    if (curr.children.size() > 0)
11      worklist.addAll(0, curr.children);
12  }
13 }
```

**Figure 4: Trie compression procedure**

### 3.2.2 Trie-Guided Symbolic Execution

Memoise monitors the symbolic execution of the program and whenever a conditional instruction is executed symbolically, it makes the corresponding traversal in the trie. Furthermore, Memoise turns off constraint solving for the portion of the path whose information has already been stored in the trie. When an unmarked node is encountered, the traversal backtracks and at the same time requests the symbolic execution to backtrack as well, thus "pruning" the search for the unmarked nodes. When a candidate node is encountered, constraint solving is turned on. The part of the trie rooted at the candidate node is then built while new states are explored, using traditional symbolic execution. Constraint solving is turned off again when the traversal backtracks from a candidate node.

## 3.3 Trie Compression

As mentioned, a trie node is created when a conditional instruction is symbolically executed. Thus, the size of the trie is proportional to the number of executions of symbolic conditionals. We note that the trie is a very light-weight representation of the explored symbolic state-space since it only records the choices taken during symbolic execution plus the method and instruction offset. However, since the trie needs to be stored to and loaded from disk to be used across different runs of symbolic execution, the trie for a large exploration space may also be quite large and may take much time. To address this problem, we define a compression operation on tries as described below. Furthermore, we expect the cost of building and maintaining the trie to be amortized during multiple successive applications of symbolic execution.

As an optimization, the memoized analysis performs a *compression* operation on the input trie to remove the nodes that are irrelevant in the context of the particular application scenario (see Figure 4). The analysis simply removes all the children of the nodes that are found to be *not* marked by the marking procedure described in Section 3.2.1, and then uses this compressed version of the trie to perform the memoized analysis.

The resulting trie that is saved on the disk may no longer be "complete" therefore we define a trie merging procedure to recover a complete trie as described in Section 3.4. We note that as a result of compression, the trie will still contain unmarked nodes that are the immediate successors of marked nodes. These unmarked (pruning hanging) nodes are necessary for the merging procedure as described next.

## 3.4 Trie Merging

As a result of compression, the trie $t$ for $\langle p, b\rangle$ may no longer be complete. While some applications, such as symbolic execution with iterative deepening, may not need to maintain a complete trie, others, like regression analysis, do require a complete trie. We therefore define an optional merging operation $merge(t_{old}, t_{new})$ that returns a complete trie for $\langle p_{new}, b_{new}\rangle$. The nodes from the

```
1 void merge(Trie t_old, Trie t_new) {
2   // assume trie nodes are marked; updates t_new
3   List<Node> worklist_old = new LinkedList<Node>();
4   List<Node> worklist_new = new LinkedList<Node>();
5   worklist_old.add(t_old.root);
6   worklist_new.add(t_new.root);
7   while(!worklist_new.isEmpty()&&!worklist_old.isEmpty()){
8     Node curr_old = worklist_old.remove(0);
9     Node curr_new = worklist_new.remove(0);
10    if (curr_new.isCandidate()) continue;
11    if (!curr_new.isMarked()) {
12      curr_new.setChildren(curr_old.children);
13      continue;
14    }
15    worklist_old.addAll(0, curr_old.children);
16    worklist_new.addAll(0, curr_new.children);
17  }
18 }
```

**Figure 5: Trie merging procedure**

root node down to either candidate nodes or the pruning hanging (unmarked) nodes, are shared by both $t_{old}$ and $t_{new}$. While the parts rooted at candidate nodes in $t_{new}$ are updated and there is no need to bring back the obsolete parts from $t_{old}$, the parts rooted at pruning hanging nodes must get recovered from $t_{old}$. The merging procedure shown in Figure 5 performs a synchronous traversal on the two tries and merges the parts from $t_{old}$ to the pruning hanging nodes in $t_{new}$ at lines 11-14. It does not check the parts rooted at candidate nodes (line 10).

## 3.5 Correctness

The correctness of our proposed approach is based on two assumptions: first, constraint solving is deterministic, i.e., given a constraint, the underlying constraint solver or decision procedure would always give a unique answer on satisfiability; second, the order among the branches of a symbolic conditional is uniquely determined. The first assumption assures that different runs of symbolic executions on a particular program path would always yield to same result. Therefore, we have the following two observations based on this assumption.

**Observation 1.** If a path constraint exists in the previous run of symbolic execution and was solved previously, it does not need to be solved again, and the solving result for it from the previous run can be reused.

**Observation 2.** If path constraints for all paths that continue from some point in the exploration space remain the same as those in the space previously explored, the subspace rooted at that point can be pruned, and the solving results for these path constraints from previous run of symbolic execution can be reused.

The second assumption maintains the correspondence of the executions of program paths across different runs of symbolic execution, and makes feasible the reuse of symbolic execution results. As long as the same search order is used during re-execution, the symbolic execution tree corresponding to the same program executions remain the same, and this assures the correctness of trie-guided symbolic execution. Merging is correct since the executions corresponding to the removed parts remain the same in re-execution and will yield to the same sub-trie, and thus the removed parts can be brought back from the old trie.

# 4. ENABLED APPLICATIONS

We envision many applications that can be optimized using memoized symbolic execution. We describe in detail how Memoise enables three "standard" applications of symbolic execution: symbolic execution with iterative deepening, regression analysis and symbolic execution guided by heuristics to enhance program coverage. We further discuss four other applications.

```
void mark(Trie trie) {
List<Node> worklist = new LinkedList<Node>();
  worklist.add(trie.root);
  while (!worklist.isEmpty()) {
    Node curr = worklist.remove(0);
    if (curr.isBoundary()) {
      trie.markAncestor(curr);
      continue;
    }
    if (curr.children.size() > 0)
      worklist.addAll(0, curr.children);
  }
}
```

**Figure 6: Marking procedure for iterative deepening**

## 4.1 Three Representative Applications

### 4.1.1 Iterative Deepening

Memoized symbolic execution enables an efficient iterative deepening approach by re-using the results from smaller depths when exploring paths at larger depths. The approach works as follows. In the first iteration, we explore paths exhaustively up to a certain depth and store the choices from the symbolic execution tree in the trie structure defined in the previous section. When the search depth bound is hit, the current trie node at that point is a boundary node. Then, to perform memoized symbolic execution at a deeper depth, we first execute the marking procedure described in Figure 6. Boundary nodes are candidate nodes in this particular analysis, and the marking procedure traverses the input trie and marks all ancestor nodes of the boundary nodes. The paths that lead to candidate nodes are then selected (their nodes are marked) and, guided by the trie, are executed up to the next depth bound. Note that the paths whose nodes were not marked would not be re-executed (e.g. the paths who ended at smaller depths can not have successors at the new bigger depth). During re-execution we turn off constraint solving for the portion of the path that has been already explored in the previous iteration, and the exploration is only guided by the choices recorded in the trie. The process repeats until all paths are explored, or the new bound is reached.

For example, if we get the trie of Figure 3 in the first iteration and want to explore paths at a larger depth in the next iteration, we only select the path $n1 \rightarrow n5 \rightarrow n7 \rightarrow n9$ to re-execute since only $n9$ is a candidate node. We turn off constraint solving for the portion from $n1$ to $n9$, and turn on constraint solving after $n9$ is encountered in trie traversal. If compression is performed, the unmarked nodes are further removed from the trie (except the immediate successors of marked nodes). For example nodes $n3$ and $n4$ in Figure 3 would get removed. We note that the trie merging would not be necessary for iterative deepening, since we know that the removed paths from the compressed trie for depth $b$ ended at smaller bounds, and therefore can not have successors at a depth larger than $b$.

### 4.1.2 Regression Analysis

Programs evolve during development or maintenance. Reapplying full symbolic execution to programs as they evolve may be impractical or infeasible. In regression analysis, program differences are utilized to make symbolic execution more efficient on the subsequent program version. The results generated by regression analysis should be complete, i.e., they should be the same as the results generated by regular symbolic execution.

Memoise enables regression analysis by only allowing the paths impacted by the program change to be re-executed. A change impact analysis is used to identify the *impacted* trie nodes, which represent roots of sub-trees potentially changed by the execution of the

```
1void mark(Trie trie, Program p_old, Program p_new) {
2  CFG g_old = new CFG(p_old);
3  CFG g_new = new CFG(p_new);
4  trie.mark(CFG.impact(g_old, g_new));
5}
```

**Figure 7: Marking procedure for regression analysis**

change during memoized execution. Thus, the impacted trie nodes are candidate nodes in this particular analysis, and the marking procedure marks all the nodes along any path from the trie root node to a candidate node (see Figure 7). As before, for the portion of the path up to the impacted node, constraint solving is turned off, and only the part rooted at the impacted node needs to be rebuilt while it is explored with constraint solving turned on.

The control flow graph (CFG) of the program together with the trie are used to calculate the impacted trie nodes, and hence to guide symbolic execution to only execute paths with impacted trie nodes. Given a changed node in the CFG, we use backward reachability analysis to find the first symbolic conditional branch on each path from the changed node to the entry node in the CFG, and the trie nodes corresponding to the branch(s) are impacted. For example, assume a change is made to line 6 of the program shown in Figure 1, where `delta` instead of `-delta` is returned. Tracing the change towards the entry of the program in the CFG, we can find that the true branch of the symbolic conditional instruction at line 5 is the nearest symbolic branch leading to the change. We map this to the trie, and find the corresponding node $n3$ in Figure 3, which represents the first choice, i.e., index 0. $n3$ is the impacted node, and only the execution of $n3$ would lead to execution of the change. Therefore, we select the trie path $n1 \rightarrow n2 \rightarrow n3$ to guide the exploration; the execution corresponding to the other trie paths can be pruned; constraint solving is turned off for the execution corresponding to the selected path; it is turned on when $n3$ is encountered.

If compression is performed, the unmarked nodes are further removed from the trie (except the immediate successors of marked nodes). For example nodes $n6$, $n7$, $n8$ and $n9$ in Figure 3 would be removed. For regression analysis a complete trie needs to be maintained, since one can not anticipate which parts of the trie will be impacted with the next program change. The trie merging procedure (see Figure 5) is performed to combine the old trie with the current compressed trie.

In previous work [17] we have used the differences between program versions to make symbolic execution more efficient for evolving programs. A comparison between that work and the regression analysis enabled by Memoise is provided in the next section.

### 4.1.3 Heuristics-Guided Symbolic Execution

The iterative-deepening approach described above can be further extended to perform a heuristic search of program paths, as guided by the testing coverage achieved so far. At each iteration, the approach discovers those paths that may lead to increased code coverage, and selects only those paths for re-execution up to larger depths in subsequent iterations. The analysis computes the coverage achieved by the explored paths on the control flow graph of the program and maintains a mapping between the program control flow graph and the symbolic execution trie. We next give some basic definitions and then describe our heuristics.

DEFINITION 1. ***Control flow graph*** *(CFG): A CFG of a method in the program is a directed-graph represented formally by a tuple* $\langle N, E \rangle$. *$N$ is the set of nodes, where each node is labeled with a unique program location identifier. The edges, $E \subseteq N \times N$, represent possible flow of execution between the nodes in the CFG. Each CFG has a single* begin, $n_{begin}$, *and* end, $n_{end}$, *node. All the nodes in the CFG are reachable from the $n_{begin}$ and the $n_{end}$ node is reachable from all nodes in the CFG.*

DEFINITION 2. ***Reachability****: A node $n_1$ in CFG for method $m_1$ is reachable from a node $n_2$ in CFG for method $m_2$ if at least one of the following conditions is satisfied:*
*(1) $m_1$ and $m_2$ are the same method, and $n_1$ is reachable from $n_2$ in the CFG.*
*(2) In CFG for $m_2$, there is a node $n_3$ reachable from $n_2$, and the invocation of $m_1$ is located at $n_3$.*
*(3) In CFG for $m_2$, there is a node $n_3$ reachable from $n_2$, the invocation of $m_3$ is located at $n_3$, and $m_1$ is reachable in the call graph from $m_3$.*

We define the following two heuristics:

- Reachability: A reachability analysis is performed to determine which paths may potentially reach the uncovered nodes. Only those paths are then selected for re-execution at the next iteration. A simplified version of this heuristic could be just favoring paths that end in certain methods, assuming that the target is reachable from those methods.

- Counter: Sometimes, the execution of the uncovered part of the program depends on certain number of executions of specific statements, and intuitively the more those specific statements are executed, the more likely the uncovered part would get covered. This situation occurs in reactive programs that interact continuously with the environment or, more generally, in programs with looping constructs. We can count how many times the specific statements are executed for each path, and select paths with the maximum counter to re-execute. For example, we use the mapping of CFG and trie to find the nearest executed symbolic conditional branch that leads to the uncovered part, and count how many times the branch is executed on each path using the trie.

Other heuristics that utilize the information from a partial symbolic execution run to guide subsequent runs can be defined.

## 4.2 Other Applications

### 4.2.1 Continuous Testing

Memoise can further enable interesting new applications such as "continuous testing" [21]. Similar to "continuous compilation" in modern IDEs, continuous testing uses excess CPU cycles on a software developer's workstation to continuously test the code, while the developer works on writing it. In the original continuous testing approach [21], the test cases were provided explicitly by the user. One can use memoized symbolic execution to continuously and *incrementally* generate the tests automatically, while the code is being written.

Similar to regression analysis, a differential analysis could monitor for program changes and determine the impacted nodes in the trie structure. That information can be used to drive the symbolic execution of the impacted parts of program and re-generate parts of the trie and the corresponding tests, while unchanged parts of the trie and the corresponding tests are still there for reuse. In this way, the trie structure and tests generated from symbolic execution can be efficiently maintained when the program evolves.

### 4.2.2 Load Balancing for Parallel Execution

Parallel techniques have shown promise in addressing the scalability issues of symbolic execution [25]. The trie structure obtained

from a "shallow" memoized symbolic execution can be used to obtain information for building balanced partitions of the symbolic execution tree. The obtained static partitions can then be distributed for further "deeper" parallel symbolic execution on different machines. This would be more efficient than a previous parallel execution approach [25] that uses a set of disjoint pre-conditions for static partitioning; these additional pre-conditions contribute extra constraints that may slow down the analysis significantly. The trie can be further used to perform dynamic load balancing, by redistributing the computation during the parallel exploration, based on the previously cached results.

#### 4.2.3 Partial Symbolic Execution

When "classical" symbolic execution runs out of resources (time or memory), the significant computation performed by symbolic execution is typically lost. In contrast Memoise returns compact information about the partially explored symbolic state space. The trie structure obtained from the partial run can be mined for information that may be useful to the user such as path feasibility and unreachable code. As expected, Memoise also enables a form of incremental partial symbolic execution, i.e. next time one can restart symbolic execution from where it ended, guided by the trie paths that end in boundary nodes.

#### 4.2.4 Component Certification

Component-based software engineering enables rapid development of systems through the assembly of pre-existing components. Before using an acquired component one must *certify* that the component is safe and performs as advertised. This is particularly important for third-part components that come from untrusted sources. Memoise enables program certification, by reducing it to checking the provided trie. Thus, the trie acts as the "program certificate", and program certification is then the efficient memoized symbolic re-execution. Since in re-execution, constraints solving is turned off, program certification could be done on different platforms; this is a significant benefit, since it is often the case that many constraint solvers are platform specific.

The approach is similar to "search-carrying code" [26], which uses explicit-state model checking for certification. Symbolic execution may be better suited for certification, since it can analyze programs that are "open" (i.e. have un-specified inputs), which is typical for components, while explicit-state model checking analyzes closed systems.

## 5. EVALUATION

In this section, we present the experiments we have conducted to evaluate our proposed approach, with respect to the time and memory cost incurred by building, storing and retrieving the trie and the savings that can be achieved with Memoise. We have implemented memoized symbolic execution and the three representative applications described in the previous section, and we performed the experiments on several non-trivial example programs, the largest example having 4697 lines of code. We begin with a description of our implementation.

### 5.1 Implementation

We use Symbolic PathFinder (SPF) [18, 19], an open source symbolic execution tool for Java bytecode. SPF is part of the Java PathFinder verification tool-set [2] which includes JPF-core, an explicit-state software model checker, and several extension projects, one of them being SPF. JPF-core implements an extensible custom Java Virtual Machine (VM), state storage and backtracking capabilities, different search strategies, as well as *listeners* for monitoring and influencing the search. By default, JPF-core executes the program concretely based on the standard semantics of the Java. SPF replaces the concrete execution semantics with a non-standard symbolic interpretation of bytecodes.

```
1 void testLoop1(int x) {
2   int c=0 , p=0 ;
3   while(true) {
4     if(x<=0) break;
5     if(c==50) {
6       System.out.println("abort1");
7       assert false; // error 1
8     }
9     c=c+1;
10    p=p+c;
11    x=x-1;
12  }
13  if(c==30) {
14    System.out.println("abort2");
15    assert false; // error 2
16  }
17 }
```

**Figure 8: Example program with one loop**

Symbolic execution of conditional instructions is performed by generating a non-deterministic choice using a *PC choice generator*. Each choice is associated with a path constraint encoding the condition or its negation respectively. The path constraints are checked for satisfiability using off-the-shelf decision procedures or constraint solvers. If the path constraint is satisfiable, the search continues; otherwise, the search backtracks.

We have implemented the procedures for: building the trie, iterative deepening, regression analysis, and the two guided heuristics for increasing the coverage during symbolic execution. All the procedures are implemented as JPF *listeners*. When building the trie, JPF's search events such as "state advanced" and "state backtracked" are monitored, so that whenever a conditional instruction bytecode is symbolically executed a trie node is created as a child of the current trie node, and the current trie node is updated while the search is advanced or backtracked correspondingly. Information including the conditional instruction bytecode offset, the choice taken by execution, and the fully qualified method name, is collected at runtime and stored in the trie. The approach can be easily extended to store other kind of information that might be relevant for an analysis, such as the constraint associated with the condition in the code. When the search depth bound is hit, the current trie node at that point is marked as *boundary*. The saving/loading of tries is implemented using the Java Serialization API, which stores Object state to a file in disk.

To facilitate regression analysis and heuristic search, we implemented a custom control flow analysis, and the mapping between control flow graphs and the trie is maintained, so that the analyses can be conducted. The analysis traces back from the change to the root in each control flow graph path, to find the nearest branch of a symbolic conditional. The branch has an offset and a choice, which are used to map to the trie to find the impacted trie nodes. The parts of trie rooted at those impacted trie nodes should be rebuilt.

JPF/SPF are publicly available. We have put the code for our implementation and the experiments in JPF-memoise, a sub-project of JPF, and we plan to release it publicly soon (the code for the case studies is already available at:
https://hostdb.ece.utexas.edu/∼gyang/memoise).

### 5.2 Example Programs

#### 5.2.1 Loops

Looping programs pose particular challenges to symbolic execution and handling them efficiently is an active area of research.

```
1 void testLoop2(int x, int y) {
2   int c=0 , p=0 ;
3   while(true) {
4     if(x<=0){
5       break;
6     }
7     if(c==100) {
8       System.out.println("abort1");
9       assert false; // error 1
10    }
11    c=c+1;
12    x=x-1;
13  }
14
15  while(true) {
16    if(y<=0){
17      break;
18    }
19    if(p==50 && c==2) {
20      System.out.println("abort2");
21      assert false; // error 2
22    }
23    p=p+1;
24    y=y-1;
25  }
26 }
```

**Figure 9: Example program with two loops**

```
1 public class BankAccount {
2   private int balance;
3   private int numberOfWithdrawals;
4   public void deposit(int amount) {
5     if (amount > 0)
6       balance = balance + amount;
7   }
8   public void withdraw(int amount) {
9     if (amount > balance) {
10      printError();
11      return;
12    }
13    if (numberOfWithdrawals >= 5) {
14      assert false;
15      printError();
16      return;
17    }
18    balance = balance - amount;
19    numberOfWithdrawals++;
20 }
```

**Figure 10: A bank account example**

We investigate here how the proposed techniques based on Memoise can help with dealing loops. The example in Figure 8 has been used in previous work on loop analysis [11] where test inputs are generated to exercise the two statements at line 6 and line 13.

The example in Figure 9 is similar, except that the part after the first loop is another loop, instead of a simple conditional statement. Note that in both cases, symbolic execution results in an unbounded execution tree.

### 5.2.2 *BankAccount*

The bank account example shown in Figure 10 has been used in previous work [13] to illustrate method sequence generation using symbolic execution and evolutionary testing. The example implements a bank account service. In the `BankAccount` class, the `deposit` method is used to deposit money in the account. The `withdraw` method is used to withdraw money from the account. In `withdraw`, if the amount to be withdrawn is greater than the account balance,an error message is printed and the method exits. If the number of withdrawals (`numberOfWithdrawals`) completed so far is greater than or equal to a fixed quantity (5) an error message is again printed and the method exits; otherwise, the withdrawal amount is dispensed, and both `balance` and `numberOfWithdrawals` are updated.

### 5.2.3 *WBS*

Wheel Brake System (WBS) is a synchronous reactive component from the automotive domain. This method determines how much braking pressure to apply based on the environment. The Java model is based on a Simulink model derived from the WBS case example found in ARP 4761 [20, 14]. The Simulink model was translated to C using tools developed at Rockwell Collins and manually translated to Java. It consists of one class and 231 lines of code.

### 5.2.4 *TCAS*

Traffic Anti-Collision Avoidance System (TCAS) is a system to avoid air collisions. Its code in C together with 41 mutants are available at SIR repository [3]. We manually converted the code to Java. The Java version has 143 lines of code.

### 5.2.5 *MerArbiter*

MerArbiter models a component of the flight software for NASA JPL's Mars Exploration Rovers (MER). The analyzed software consists of a Resource Arbiter and several user components. Each user serves one specific application, such as imaging, controlling the robot arm, communicating with earth, and driving. The arbiter module moderates access to several shared resources. It prevents potential conflicts between resource requests coming from different users and it enforces priorities. For example, it does not make sense to start a communication session with Earth while the rover is driving.

MerArbiter has been modeled in Simulink/Stateflow and it was automatically translated into Java using the Polyglot framework [5]. The configuration for our analysis involved two users and five resources. The example has 268 classes, 553 methods, 4697 lines of code (including the Java Polyglot execution framework).

### 5.2.6 *Apollo*

The Apollo Lunar Autopilot is a Simulink model that was automatically translated to Java. The translated Java code has 2.6 KLOC in 54 classes. The Simulink model was created by an engineer working on the Apollo Lunar Module digital autopilot design team. The goal was to study how the model could have been designed in Simulink, if it had been available in 1961. The model is available from MathWorks6. It contains both Simulink blocks and Stateflow diagrams and makes use of complex Math functions (e.g. Math.sqrt). The code has been analyzed before using Symbolic PathFinder with the Coral solver [24].

## 5.3 Experimental Results

### 5.3.1 *Iterative Deepening*

We conducted several groups of experiments. In each group, we increased the depth from A to B. At depth A we built the trie while at depth B, we re-used and updated the trie. We also conducted regular symbolic execution as implemented in SPF at both depth A and depth B. Table 1 shows the results for our experiments on *WBS*, *MerArbiter*, and *Apollo*. This table shows the time and memory (Mem) results for regular symbolic execution and for Memoise. It also shows the number of states, the number of constraint solver calls and the size of Trie that is saved during Memoise at depth B. Reg represents regular symbolic execution while ID represents Memoise for iterative deepening. ID-c and ID-p respectively represent Memoise with compression and without compression.

For symbolic execution at depth A, we find that the time cost of Memoise is greater than regular symbolic execution for *WBS*, while for the other two examples, the time cost of running the two tech-

**Table 1: Iterative Deepening Results**

| Depth | | Sym Exe at Depth A | | | | Sym Exe at Depth B | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Time (ss) | | Mem (MB) | | States | | #Solver calls | | Time (ss) | | | Mem (MB) | | | Trie (MB) | |
| A | B | Reg | ID | Reg | ID | Reg | ID | Reg | ID | Reg | ID-p | ID-c | Reg | ID-p | ID-c | ID-p | ID-c |
| 24 | 25 | 16 | 21 | 305 | 419 | 349272 | 252952 | 335358 | 77312 | 20 | 24 | 22 | 242 | 474 | 474 | 18.8 | 13.4 |
| 29 | 30 | 34 | 60 | 246 | 260 | 644184 | 171784 | 629758 | 32256 | 37 | 36 | 27 | 214 | 500 | 486 | 35.4 | 9.3 |

(a) WBS Example

| Depth | | Sym Exe at Depth A | | | | Sym Exe at Depth B | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Time (ss) | | Mem (MB) | | States | | #Solver calls | | Time (ss) | | | Mem (MB) | | | Trie (MB) | |
| A | B | Reg | ID | Reg | ID | Reg | ID | Reg | ID | Reg | ID-p | ID-c | Reg | ID-p | ID-c | ID-p | ID-c |
| 24 | 25 | 35 | 38 | 304 | 367 | 17103 | 16756 | 12252 | 2942 | 47 | 46 | 45 | 413 | 263 | 395 | 0.9 | 0.8 |
| 29 | 30 | 86 | 87 | 419 | 333 | 33273 | 15250 | 25684 | 1540 | 92 | 45 | 45 | 413 | 345 | 263 | 2.0 | 0.9 |
| 34 | 35 | 96 | 97 | 419 | 345 | 35359 | 1476 | 27636 | 18 | 102 | 9 | 9 | 292 | 404 | 243 | 2.1 | 0.1 |

(b) MerArbiter Example

| Depth | | Sym Exe at Depth A | | | | Sym Exe at Depth B | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Time (ss) | | Mem (MB) | | States | | #Solver calls | | Time (ss) | | | Mem (MB) | | | Trie (MB) | |
| A | B | Reg | ID | Reg | ID | Reg | ID | Reg | ID | Reg | ID-p | ID-c | Reg | ID-p | ID-c | ID-p | ID-c |
| 9 | 10 | 127 | 125 | 425 | 352 | 674 | 647 | 255 | 121 | 195 | 76 | 77 | 421 | 343 | 296 | 0.03 | 0.03 |
| 11 | 12 | 538 | 549 | 413 | 414 | 2243 | 2113 | 2160 | 966 | 1033 | 490 | 483 | 390 | 316 | 420 | 0.12 | 0.11 |

(c) Apollo Example

niques is almost the same. We examined the cost distributions of Memoise for *WBS*, and found that it took 5 seconds and 11 seconds to save to disk the trie built at depth 24 and 29 respectively. Thus, the time cost for building a trie itself is not much. While Memoise costs more memory for four groups of iterations, it cost even less memory for other three groups of iterations. The memory cost results for symbolic execution at depth B show the similar trend. To understand this observation, we made several runs of symbolic execution with the same configuration, and found that the memory cost reported by SPF vary a lot. We conjectured that this depends on how the underlying garbage collection works, and comparison of the memory cost shown in the table is not very meaningful. On the other hand, in terms of space cost, a comparison based on number of states is more meaningful.

Since building the trie only monitors the search and builds the data structure, without changing the behavior of the search engine or the underlying constraint solver, it should not influence the number of states and number of solver calls. We didn't report the states and the number of solver calls for A in the table, but we examined the results on each group and the results were the same as expected.

For symbolic execution at depth B, we found that for most groups of iterations Memoise explored fewer states, made fewer solver calls, and correspondingly took less time. Especially for the last group in *MerArbiter* example where the depth is increased from 34 to 35, the reduction is more than an order of magnitude. For *WBS*, although the reduction of states and solver calls is still significant, the reduction of time is however not much, and Memoise without compression even took more time than regular symbolic execution. Again, we found loading and saving the trie took 12 seconds and 18 seconds at depth 25 and 30 respectively. Therefore, the reduction of time that was spent on state-space exploration and constraint solving is still significant. Moreover, for both *WBS* and *MerArbiter*, the reduction for the number of states, solver calls, and time appears to get more significant when the depth goes deeper for *WBS* and *MerArbiter*. The reason could be that when the state space of a program is searched deeper, normally more paths get complete or infeasible, and thus can be pruned in the next iteration.

Interestingly, for *Apollo* example, although Memoise explored almost the same number of states as regular symbolic execution, it achieved more than 2X speedup. The key contributing factor for this speedup is the reduction of solver calls.

The table also shows that for *WBS* and *MerArbiter*, compression makes the trie size smaller, while for *Apollo*, it is not the case. From this observation, we conjectured that the symbolic execution tree for *Apollo* is quite balanced and few parts can get removed by compression. This is supported by the fact that little reduction is achieved in terms of the number of states explored. It is interesting to find that Memoise with compression outperforms Memoise without compression in terms of time cost only in *WBS*. We note here that compression enables better memory usage, and thus when the trie size is big and saving/loading of trie is time consuming, the performance in terms of time cost could also benefit from compression.

### 5.3.2 Regression Analysis

We performed experiments on *TCAS* and *MerArbiter* to evaluate the effectiveness of regression analysis based on Memoise. The trie is built when symbolic execution is performed on the original program version, and then reused for the run of symbolic execution on a new program version.

For *TCAS*, we randomly selected three mutant versions v6, v25, and v30 from the SIR repository [3]. Compared to the original version v0, version v6 has an operator change from "<" to "<=", v25 has an operator change, and v30 has a return value change. Since there were no version histories for *MerArbiter*, we randomly picked two methods, and manually introduced the changes. Version v1 has a change to the return value in the method `guard` of class `Transition300`, and version v2 has an operator change from "==" to "! =" in the method `guard` of class `Transition186`.

Table 2 gives the results for memoized regression analysis. It shows the number of states explored, the number of solver calls, the time and memory cost for both regular symbolic execution (Reg) and Memoise (RA) on the new program version. Furthermore, it shows the trie size stored after the Memoise is performed on the new version, with compression (RA-c) and without compression (RA-p) respectively, and the cost for merging in cases where compression is conducted.

The memoized regression analysis is based on the trie built during the initialization run of Memoise on original program version v0. As expected, Memoise and regular symbolic execution on version v0 yield the same number of states and same number of constraint solver calls. For *TCAS*, Memoise took 146 seconds and 203 MB of memory, compared to 149 seconds and 223 MB from regular symbolic execution. For *MerArbiter*, Memoise took 47 seconds and 401 MB of memory, compared to 49 seconds and 344 MB from regular symbolic execution.

Across the three *TCAS* versions, the performance gain from using Memoise varies a lot. We find that for version v6, Memoise

**Table 2: Regression Analysis Results**

| Version | States | | #Solver calls | | Time (ss) | | | Mem (MB) | | | Trie (MB) | | Merging (ss) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Reg | RA | Reg | RA | Reg | RA-p | RA-c | Reg | RA-p | RA-c | RA-p | RA-c | |
| v6 | 2688 | 2186 | 2566 | 1696 | 166 | 119 | 118 | 147 | 211 | 147 | 0.16 | 0.13 | 0.43 |
| v25 | 2688 | 2658 | 2566 | 2016 | 145 | 67 | 68 | 137 | 214 | 229 | 0.16 | 0.11 | 0.38 |
| v30 | 752 | 722 | 686 | 632 | 40 | 40 | 40 | 136 | 216 | 138 | 0.04 | 0.04 | 0.34 |

(a) TCAS Example

| Version | States | | #Solver calls | | Time (ss) | | | Mem (MB) | | | Trie (MB) | | Merging (ss) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Reg | RA | Reg | RA | Reg | RA-p | RA-c | Reg | RA-p | RA-c | RA-p | RA-c | |
| v1 | 17718 | 6395 | 12596 | 2072 | 55 | 20 | 20 | 292 | 416 | 329 | 0.93 | 0.33 | 0.92 |
| v2 | 17103 | 43 | 12252 | 2 | 51 | 3 | 3 | 380 | 218 | 217 | 0.91 | 0.01 | 0.77 |

(b) MerArbiter Example

**Table 3: Heuristics-Guided Symbolic Execution Results**

| Scenario | Candidates | States | #Solver calls | Time (ss) | Mem (MB) |
|---|---|---|---|---|---|
| 20→37 | 32/64 | 8253 | 4032 | 28 | 149 |
| 25→37 | 128/256 | 8445 | 3840 | 29 | 216 |
| 30→37 | 0/512 | - | - | - | - |
| 35→37 | 0/2048 | - | - | - | - |

(a) Reachability Heuristic for BankAccount

| Scenario | Candidates | States | #Solver calls | Time (ss) | Mem (MB) |
|---|---|---|---|---|---|
| 20→37 | 1/64 | 277 | 127 | 25 | 153 |
| 25→37 | 1/256 | 93 | 31 | 26 | 153 |
| 30→37 | 2/512 | 93 | 29 | 27 | 121 |
| 35→37 | 2/2048 | 53 | 5 | 24 | 121 |

(b) Counter Heuristic for BankAccount

| Scenario | Candidates | States | | #Solver calls | | Time (ss) | | Mem (MB) | |
|---|---|---|---|---|---|---|---|---|---|
| | | Reg | HR | Reg | HR | Reg | HR | Reg | HR |
| 40→60 | 76/76 | 3416 | 3416 | 3414 | 1932 | 1 | 2 | 215 | 150 |
| 60→80 | 1/114 | 6116 | 538 | 6114 | 420 | 2 | 1 | 342 | 121 |
| 80→100 | 1/154 | 9616 | 578 | 9614 | 420 | 3 | 1 | 291 | 151 |
| 100→120 | 1/194 | 13610 | 312 | 13608 | 114 | 4 | 1 | 342 | 216 |

(c) Reachability Heuristic for Example with Two Loops

| Scenario | Candidates | States | | #Solver calls | | Time (ss) | | Mem (MB) | |
|---|---|---|---|---|---|---|---|---|---|
| | | Reg | HR | Reg | HR | Reg | HR | Reg | HR |
| 25→30 | 101/3580 | 33273 | 2071 | 25684 | 932 | 106 | 11 | 319 | 294 |

(d) Reachability Heuristic for MerArbiter

explored about one fifth less states, made about 1000 less solver calls, and took about one third less time than regular symbolic execution. However, Memoise explored almost the same number of states as regular symbolic execution for versions v25 and v30. Interestingly Memoise achieved 2X speedup for version v25 while it had no speedup for version v30. The reason for this is that fewer number of solver calls were made for version v25.

For *MerArbiter*, the reduction is significant for both versions. On version v1, memoized regression analysis explored about one third of the number of states, took less than one third of the time compared with regular symbolic execution, and made about one sixth number of calls to constraint solver. On version v2, the reduction achieved by memoized regression analysis is even more significant. The differences in both the number of states explored, and the number of solver calls are in several orders of magnitude.

We find that compression reduces the size of the trie significantly for the *MerArbiter* versions, while for the *TCAS* versions it does not. This indicates that most parts of the trie were impacted by changes made in the *TCAS* versions. The merging time is not much according to the results in the table.

### 5.3.3 Heuristics-Guided Symbolic Execution

For the *BankAccount* example shown in Figure 10, a symbolic driver which symbolically selects the method deposit or withdraw and symbolically picks the amount to be withdrawn and to be deposited, is used to generate sequences of methods to cover the program. We note that the statements at lines 14, 15 and 16 are hard to cover. 37 is the smallest depth bound at which symbolic execution can cover the three statements.

Without using heuristics, the regular symbolic execution with depth bound 37 explored 16381 states, took 30 seconds, and made 8190 solver calls. The two heuristics were applied based on the trie collected at a smaller depth. We picked 20, 25, 30, and 35 to run the initialization run of Memoise to build the trie.

In Table 3-(a), we can see that, for tries built at both depth 20 and 25, the reachability heuristic selected half of the trie paths ended with boundary nodes as candidates to execute. Compared with regular symbolic execution, it explored about half of the state space, took about one or two seconds less of time, and made about half of the number of solver calls. However, for the two tries built at depth 30 and 35, no path was taken as a candidate, and the heuristic is just not applicable.

In Table 3-(b), with counter heuristic applied, the number of candidate paths is one or two for the tries built at the four different depths. Moreover, the number of states explored and the number of solver calls are much less than for regular symbolic execution, but the time reduction is similar to what was achieved by the reachability heuristic. It is conjectured that most time is spent on solving some specific hard-to-solve constraints, and the conjecture seems supported by the last row in the table. Note that for all cases where either the reachability heuristic or the counter heuristic is applicable, the hard-to-cover part was covered.

We have also analyzed the two loop examples using the reachability heuristics. `Error 1` in Figure 8 is very difficult to uncover; we considered both `error 1` and `error 2` as our coverage targets. The reachability heuristic can not help with this example, since no matter what the depth bound is, the symbolic execution tree has only one boundary node, resulting in no pruning. On the other hand, the loop example shown in Figure 9 contains a more realistic scenario, with two loops, where each loop has an error to cover, and the first error is harder to cover. We ran symbolic execution with the depth iteratively deepened, from 40 to 120, each time the depth is increased by 20. The results are shown in Table 3-(c). At depth 40, the heuristic is not applicable since there was no trie available; while at depth 60, the heuristic applied on the trie built at depth 40 only reduced calls to the constraint solver since both targets `error 1` and `error 2` were reachable from all boundary nodes of the trie. However, since symbolic execution

at depth 60 covered `error 2`, leaving `error 1` as the only target, we used reachability heuristic to guide the symbolic execution to cover `error 1`. The `error 1` was not covered until the depth was 120. We find that the heuristic (HR) explored much less number of states and made less solver calls as well. The time difference is not significant since the constraint solving and space exploration took just few seconds.

For the *MerArbiter* example, we used the reachability heuristic with the class modeling the arbiter as target. We used the trie collected while running symbolic execution at depth 25 for the run at depth 30. In Table 3-(d), we find that the savings achieved by the reachability heuristic (HR) are significant. We checked the bytecode coverage for both regular symbolic execution and the reachability heuristic guided symbolic execution: 0.91 for heuristic guided vs. 0.93 for regular symbolic execution at depth 30. Although the regular symbolic execution covered a little more which is reasonable considering that a lot more effort was spent in regular symbolic execution, the reachability heuristic does help improve the coverage of the target class at less cost.

We also collected the number of states explored, time cost, and the number of solver calls for all initialization runs of Memoise in this study. As expected, the initialization run of Memoise explored the same number of states, made the same number of solver calls, and spent almost the same time as regular symbolic execution.

## 5.4 Threats to Validity

The primary threats to *external validity* for our experiments include (1) the use of SPF where our approach and the enabled analyses were implemented, (2) the use of specific underlying constraint solver, (3) the selection of examples used in the experiments, (4) the specific depths picked for symbolic execution, and (5) the mutants selected or created. Implementing our approach and enabled analyses in another framework or using another constraint solver/decision procedure could produce different results. Some of the examples selected for our experiments are small, but they are used by recent research work to show some limitation of current symbolic execution and they can serve as good example for illustrating the effectiveness of our approach. The different depth specified may produce different results. We controlled this by using several groups with different depths.

The primary threat to *internal validity* of our experiments is the possible faults in the implementation of our approach and analyses and also in SPF. We controlled for this threat by testing the implementation on examples that we can manually verify.

With respect to threats to *construct validity*, the metrics we selected to evaluate the cost reduction achieved by memoized symbolic execution and its enabled analyses are commonly used to measure the cost of symbolic execution.

## 5.5 Discussion

The savings of using Memoise for regression analysis depend on the location of the change, and may vary quite a lot between different kinds of changes. This observation is supported by our results for regression analysis. In previous work, we have developed directed incremental symbolic execution (DiSE) [17] for regression analysis. DiSE uses static analysis to determine the differences between two program versions and uses this information to guide the execution of symbolic paths towards exercising that difference. Regression analysis using trie may not always be as good as DiSE, the reason being that DiSE analyzes affected conditionals, and explores the branches in the symbolic execution tree only for them. For unaffected conditionals, it just explores "one" feasible representative branch. In this sense, DiSE covers all affected branches, but not affected paths. However, our memoized regression analysis implementation considers all affected paths, and thus often times the savings are not as much as what DiSE achieves.

However, there are several advantages of using memoized regression analysis. First, DiSE only generates affected path conditions, while memoized regression analysis generates a trie which represents all paths. If a user wants a complete test suite using DiSE, he/she needs to check what path conditions get obsolete, which is not clear how to do. Second, DiSE is based on static analysis, using control and data flow analysis in CFG; while memoized regression analysis is dynamic, based on the trie, thus it is more precise. For example, when the change is in an un-covered code, memoized regression analysis does not need to explore the state space at all, while DiSE still needs to explore the part affected by the change. Third, for an affected path DiSE performs regular symbolic execution; memoized regression analysis explores the unchanged path prefix more efficiently by turning off the constraint solving.

We implemented two simple heuristics enabled by Memoise for experiments; there could be however more effective heuristics based on Memoise. We leave it for future work. Note that we only turn off the constraint solving when re-executing a path. More significant savings could be achieved, e.g. by saving JPF state and restarting from there. This is left again for future work.

For most experiments, we find that the savings in terms of number of solver calls is significant. However, this is not always reflected in savings of time. The reason is that constraint solving for some of the analyzed programs is cheap. We believe that for programs with complex constraints, such as *Apollo*, one would gain more benefits from using Memoise.

# 6. RELATED WORK

There are many recent works that use symbolic execution to perform some of the applications that we discussed in this paper, such as regression analysis [17], parallel symbolic execution [25], etc. We have already discussed the relationship between some of these works and ours throughout the paper.

The main contribution of our work is the concept of memoized symbolic execution, which turns out to enable a multitude of applications. In this respect, Memoise is most related to recent works on incremental and regression model checking, e.g. [16, 29]. Those approaches save the state space graph from one exploration and examine this graph to determine whether a certain execution is needed during the next exploration (after a program change). The work is done in the context of explicit state model checking and therefore is not concerned with the specific details of symbolic execution, such as storing choices for symbolic execution, turning-off constraint solving etc. Furthermore, the approaches [16, 29] target a particular application, namely regression analysis, while Memoise can enable multiple applications. Our first heuristic that uses reachability information to guide the symbolic execution of a program (Section 4) is similar to the one presented in Yang's previous work [29], however, as already mentioned, that work was done in the context of explicit state model checking, not symbolic execution.

The use of trie has an effect similar to caching of constraints in symbolic execution as performed by the KLEE tool [6]. KLEE achieves orders of magnitude speed-up because there are often many redundant constraints during symbolic path exploration. Indeed in our approach, constraint solving is turned off during re-execution (see Section 3) so we expect speedups similar to KLEE. However, KLEE caches solver calls and enables speedup for one run, whereas Memoise enables reuse and speedup across multiple runs (e.g. Apollo, Table 1).

Recent work on must summaries [12] enables a form of compositional symbolic execution [9] but does not consider regression or incremental analysis across different program versions. In contrast, memoized symbolic execution enables regression analysis, as well as incremental iterative deepening and application of heuristics. Moreover, our trie structure can provide a representation for must summaries.

## 7. CONCLUSIONS

We presented memoized symbolic execution (Memoise), a new approach for the efficient application of forward symbolic execution, that leverages the results cached from previous analysis runs to improve the analysis in the current run. Memoise uses a trie-based data structure to store the choices taken during a symbolic execution run and further maintains the trie during successive runs. Memoise reduces the analysis cost by using the trie to quickly guide the search for previously explored paths (with the constraint solving turned off) and by pruning the paths that are not relevant for the current run. The results cached by Memoise can be further used to heuristically guide the search for new paths in successive runs.

Experiments using our prototype implementation of Memoise demonstrate its potential in enabling more efficient symbolic execution in the context of three typical applications: iterative deepening, regression analysis, and heuristics-guided symbolic execution. Although our trie representation is lightweight, we would like to investigate large tree-based data structures such as the AVL-trees used in relational databases to store the trie for the memoized symbolic execution of very large programs. We further plan to investigate compositional techniques for increased scalability and to evaluate the other applications that we outlined here for Memoise.

Symbolic PathFinder (SPF) handles non-determinism by using JPF's choice generators. Therefore, the same mechanism for storing and re-playing the choices that is described in this paper can be used to handle non-determinism in memoized symbolic execution; we would need to make a modification to our JPF listener to keep track of the additional choices.

## 8. ACKNOWLEDGMENTS

This work was supported by NSF Grant CCF-0845628, AFOSR Grant FA9550-09-1-0351, and Google Summer of Code 2011.

## 9. REFERENCES

[1] SMT-COMP 2011. http://www.smtcomp.org/2011/.

[2] Java PathFinder Tool-set. http://babelfish.arc.nasa.gov/trac/jpf.

[3] SIR Repository. http://sir.unl.edu.

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

[5] D. Balasubramanian, C. S. Pasareanu, M. W. Whalen, G. Karsai, and M. R. Lowry. Polyglot: modeling and analysis for multiple statechart formalisms. In *ISSTA*, pages 45–55, 2011.

[6] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[8] E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, September 1960.

[9] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[11] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, pages 23–33, 2011.

[12] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

[13] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE*, pages 297–306, 2008.

[14] A. Joshi and M. P. E. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, pages 122–135, September 2005.

[15] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[16] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *ICSE*, pages 291–300, 2008.

[17] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

[18] C. S. Păsăreanu, P. C. Mehlitz, D. H. 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*, pages 15–25, 2008.

[19] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[20] SAE-ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[21] D. Saff and M. D. Ernst. Continuous testing in eclipse. In *ICSE*, pages 668–669, 2005.

[22] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[23] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[24] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. Coral: solving complex constraints for Symbolic PathFinder. In *NFM*, pages 359–374, 2011.

[25] M. Staats and C. S. Pasareanu. Parallel symbolic execution for structural test generation. In *ISSTA*, pages 183–194, 2010.

[26] A. Taleghani and J. M. Atlee. Search-carrying code. In *ASE*, pages 367–376, 2010.

[27] M. W. Whalen, P. Godefroid, L. Mariani, A. Polini, N. Tillmann, and W. Visser. Fite: future integrated testing environment. In *FoSER*, pages 401–406, 2010.

[28] D. E. Willard. New trie data structures which support very fast search operations. *J. Comput. Syst. Sci.*, 28:379–394, July 1984.

[29] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *ICSM*, pages 115–124, 2009.