Memoized Symbolic Execution

Guowei Yang
University of Texas at Austin

Corina S. Păsăreanu
Carnegie Mellon/NASA Ames

Sarfraz Khurshid
University of Texas at Austin

***Abstract*—This paper introduces *memoized symbolic execution* (*Memoise*), a novel 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 embodiment 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.**

***Keywords*-symbolic execution; trie; constraint solving;**

I. Introduction

Forward symbolic execution [13], [7], [9], [21], [18], [6] is a powerful technique that is gaining popularity for systematic exploration of program behaviors. The technique is conceptually simple: represent program paths (of interest) as formulas that symbolically represent the state updates and branches along the paths, and use the formulas as a basis of analyzing the program by utilizing constraint solving technology that allows reasoning about the formulas. However, in practice, the technique can be costly to apply due to its inherent high time 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], [24]. 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 [22], regression analysis [16], and continuous testing [26]. 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 novel approach that addresses both the 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 global *state* of a symbolic execution run. Specifically, Memoise introduces an efficient trie [8], [27] data structure for a compact representation of the symbolic paths generated during a symbolic execution run. 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. Moreover, computations based on the trie from the last symbolic execution run can *guide* future runs, e.g., to discard branch sequences deemed no longer to be of interest.

We developed a prototype tool for memoized symbolic execution of Java programs. The implementation uses the Symbolic PathFinder tool [18], part of the Java PathFinder open-source framework. Experiments with Memoise show its benefits in standard scenarios of using symbolic execution, such as, with iterative deepening of exploration depth, to perform regression analysis, or to enhance coverage. We believe the approach introduced by Memoise also holds much promise in optimizing a variety of other novel analyses based on symbolic execution (as discussed in Section IV).

The main contributions of this paper are:

- **Memoized symbolic execution**. We introduce the concept of storing the state of a run of symbolic execution and utilizing and updating it during the next run.
- **Trie data structure for symbolic execution**. Memoise presents a novel application domain for a well-known data structure.
- **Applications**. We discuss how a suite of applications of symbolic execution likely benefit from the approach introduced by Memoise.
- **Demonstration**. Experiments using our Memoise prototype 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*.

```
1 public 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

II. Background

Symbolic execution [13], [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 Sym_1, Sym_2 and Sym_3, respectively. Program variables are assigned expressions in terms of these symbolic inputs; e.g., after executing statement 4, `d` becomes $Sym_2 - Sym_1$. 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 PCs.

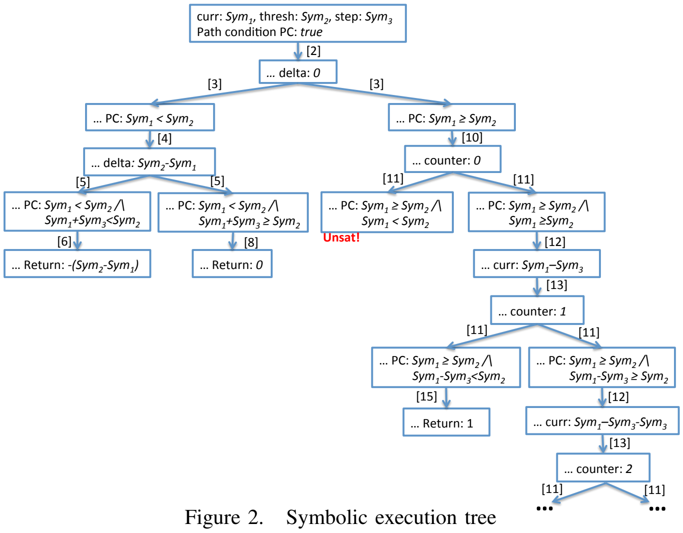
curr: Sym_1, thresh: Sym_2, step: Sym_3
Path condition PC: true
[2]
... delta: 0
[3]
[3]
... PC: $Sym_1 < Sym_2$
... PC: $Sym_1 \geq Sym_2$
[4]
[10]
... delta: Sym_2-Sym_1
... counter: 0
[5]
[5]
[11]
[11]
... PC: $Sym_1 < Sym_2$ /\ Sym_1+Sym_3<Sym_2
... PC: $Sym_1 < Sym_2$ /\ Sym_1+$Sym_3 \geq Sym_2$
... PC: $Sym_1 \geq Sym_2$ /\ $Sym_1 < Sym_2$
Unsat!
... PC: $Sym_1 \geq Sym_2$ /\ $Sym_1 \geq Sym_2$
[6]
[8]
[12]
... Return: -(Sym_2-Sym_1)
... Return: 0
... curr: Sym_1–Sym_3
[13]
... counter: 1
[11]
[11]
... PC: $Sym_1 \geq Sym_2$ /\ Sym_1-Sym_3<Sym_2
... PC: $Sym_1 \geq Sym_2$ /\ Sym_1-$Sym_3 \geq Sym_2$
[15]
[12]
... Return: 1
... curr: Sym_1–Sym_3-Sym_3
[13]
... counter: 2
[11]
[11]
...
...

Figure 2. Symbolic execution tree

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.

III. Memoized Symbolic Execution

A. Overview

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 $\langle p, b\rangle$ based on the following two observations.

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.

B. Symbolic Execution Trie

Memoise uses an efficient trie [8], [27] based data structure for representing the symbolic execution tree, i.e. the global state of a symbolic execution run. A trie (prefix tree) is an ordered tree that enables efficient re**trie**val of the information stored in it.

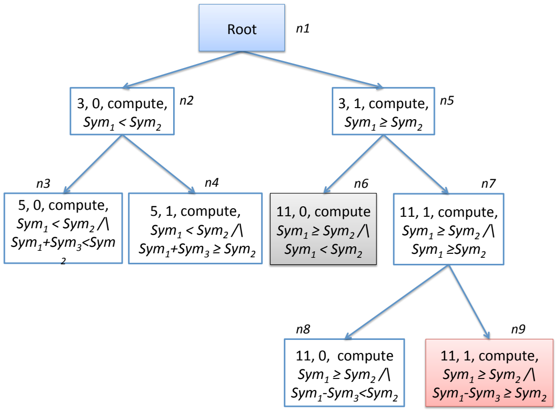
Root
n1
3, 0, compute, $Sym_1 < Sym_2$
n2
3, 1, compute, $Sym_1 \geq Sym_2$
n5
n3
5, 0, compute, $Sym_1 < Sym_2 \wedge Sym_1+Sym_3<Sym_2$
n4
5, 1, compute, $Sym_1 < Sym_2 \wedge Sym_1+Sym_3 \geq Sym_2$
n6
11, 0, compute $Sym_1 \geq Sym_2 \wedge Sym_1 < Sym_2$
n7
11, 1, compute, $Sym_1 \geq Sym_2 \wedge Sym_1 \geq Sym_2$
n8
11, 0, compute $Sym_1 \geq Sym_2 \wedge Sym_1-Sym_3<Sym_2$
n9
11, 1, compute, $Sym_1 \geq Sym_2 \wedge Sym_1-Sym_3 \geq Sym_2$

Figure 3. Example trie

The *symbolic execution trie* compactly represents the symbolic execution tree generated during symbolic execution and can be maintained and re-used on-the-fly, either when the program is checked with a greater bound or the program undergoes changes. The benefits of using *symbolic execution 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.

1) Build: The symbolic execution trie is built on-the-fly during symbolic execution. Whenever a conditional instruction is symbolically executed, a trie node is created, recording the location of the symbolic conditional, i.e., method and the bytecode offset, the choice taken by the execution, and the corresponding path constraint. Therefore, the symbolic execution trie compactly represents the symbolic execution tree, and captures important information generated during symbolic execution. Figure 3 shows the trie for our running example (for depth bound 3). There are three types of trie nodes:

- **Unsatisfiable**: nodes that have an unsatisfiable path condition (e.g., $n6$ in Figure 3).
- **Boundary**: nodes that are trie leaves, due to depth bounded symbolic execution (e.g., $n9$ in Figure 3).
- **Regular**: all the other nodes.

2) Maintenance: The trie needs to be updated according to the changes in the corresponding symbolic execution tree. We consider two scenarios in which the symbolic execution tree changes: (1) the program is changed, and the symbolic execution of the program generates a different symbolic execution tree; (2) the program is not changed but it is analyzed using different execution depth bound, and the symbolic execution tree is trimmed or extended due to the change of the bound. We describe how the trie is maintained in both of the two scenarios.

First, when the program is changed, an impact analysis is applied to find the parts of the trie that need to be updated. Since only the execution of the program change would lead to a change in the trie, we check impacted nodes, which are the trie nodes whose corresponding program execution may directly lead to the execution of the change, and only the sub-parts rooted at the impacted nodes need to be updated. The trie path prefixes from the root to those impacted nodes should remain the same. One extreme is that the change is unreachable and not executed during symbolic execution, and thus there are no impacted nodes in the trie, in which case there is no need to update the trie at all.

Second, when the program is not changed, but the program is symbolically executed using different bounds. It is common for users to increase the bound to check for more program behaviors. In such case, the trie needs to be extended to reflect the new behaviors. However, the behaviors within the bound remain the same as before and thus the old trie forms a sub-structure of the new trie and the boundary nodes are extended with parts which encode the new program behavior.

3) Size: 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. Since the trie need to be stored to and loaded from disk so that it can be used across different runs of symbolic execution, the trie for big exploration space may be too big, and the storing and loading of the trie may take much time. However, the trie could be compressed depending on the applications of the trie. Furthermore, we expect the cost of building and maintaining the trie to be amortized during multiple successive applications of symbolic execution.

C. Memoise

In "classical" symbolic execution, the symbolic execution results for a system cannot be reused in a subsequent analysis of that system. Users have to re-run symbolic execution from the very beginning to generate results, even when system does not have any change.

Memoized symbolic execution enables efficient re-execution based on the results cached in the trie structure. For re-execution, Memoise first analyzes the trie to mark the paths which need to be re-executed - all nodes on the paths from the root to the "candidate" nodes are marked as "enabled". When performing iterative deepening, the boundary nodes are candidate nodes; and when performing regression analysis, the impacted nodes are candidate nodes. 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 a disabled node is encountered, the traversal backtracks and at the same time requests the symbolic execution to backtrack as well. When a candidate node is encountered, constraint solving is turned on. The part of the trie rooted at the candidate node is then built when new states are explored, using traditional symbolic execution. Constraint solving is turned off again when the traversal backtracks from a candidate node.

D. 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 the unsatisfiable nodes in a trie would be always unsatisfiable nodes if their path conditions do not change. The second assumption maintains the correspondence of nodes across tries for different runs of symbolic execution, i.e., the executions of a path that is not impacted by a change or specified execution bound would result in nodes with the same position in the tries.

IV. Enabled Applications

We envision many applications that can be optimized using memoized symbolic execution. We describe in detail here 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 other, less standard, applications that could benefit from Memoise.

A. Three Representative Applications

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 symbolic execution tree in the trie structure defined in the previous section. Then, we select the paths that end in a boundary node and, guided by the trie, we execute them up to the next depth bound. During re-execution we turn off constraint solving for the portion of the path that has been already explored in the previous iteration, as 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 $n1 \to n5 \to n7 \to n9$ to re-execute since only $n9$ is a boundary 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.

2) Heuristics-guided Symbolic Execution: The iterative-deepening approach described above has been 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 execution of specific statements, and intuitively the more those specific statements are executed, the more likely the uncovered part would be covered. Therefore, we can count how many times those 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.

3) Regression Symbolic Execution (RSE): In regression symbolic execution (RSE), program differences are utilized to make symbolic execution more efficient on the subsequent program version. The results generated by RSE should be complete, i.e., they should be the same as the results generated by regular symbolic execution.

Memoise enables RSE by only allowing the paths impacted by the change to be re-executed. Moreover, 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 used backward reachability to find the first symbolic conditional branch on that path, and the trie nodes corresponding to the branch that 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. $n3$ 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.

B. Other Applications

1) Continuous Testing: Memoise can enable exciting new applications such as "continuous testing" [20]. 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 [20], 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 RSE, a differential analysis could monitor for program changes and determine the impacted branches 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.

2) Load Balancing for Parallel Symbolic Execution: Parallel techniques have shown promise in addressing the scalability issues of symbolic execution [24]. 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 [24] 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 re-distributing the computation during the parallel exploration, based on the previously cached results.

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, even when running out of resources, Memoise returns compact information about the partially explored symbolic state space. The symbolic execution trie stores the constraints that provide a logical characterization of the "context" in which the method has been analyzed. Such "pre-conditions" can be used for a modular verification. If a "caller" method m_1 satisfies the pre-conditions of partially analyzed method m_2, we know that there will be no errors exhibited (assuming m_2 did not exhibit any errors during the partial run). The trie structure can be further mined for additional 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 guided by the trie paths that end in boundary nodes.

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

V. Implementation and Experiments

A. Implementation

We use Symbolic PathFinder (SPF) [17], [15], 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.

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 symbolic execution, and different 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, the fully qualified method name, and the path constraint is collected in runtime. When the search depth bound is hit, the current trie node at that point is marked as *boundary*.

We have both implemented the reachability and counter heuristics for guiding the symbolic execution towards increasing the code coverage. 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 the 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.

B. Example Programs

1) Loops: Looping programs pose particular challenges to symbolic execution and handling them efficiently is an active area of research. We investigate here how the proposed techniques based on Memoise can help with dealing loops. The example in Figure 4 has been used in previous work on loop analysis [10] where test inputs are generated to exercise the two statements at line 6 and line 13.

```
1  public 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 4. Example program with one loop

```
1  public static 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 5. Example program with two loops

The example in Figure 5 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.

2) BankAccount: The bank account example shown in Figure 6 has been used in previous work [11] 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

```
public class BankAccount {
  private int balance;
  private int numberOfWithdrawals;
  public void deposit(int amount) {
    if (amount > 0)
      balance = balance + amount;
  }
  public void withdraw(int amount) {
    if (amount > balance) {
      printError();
      return;
    }
    if (numberOfWithdrawals >= 5) {
      assert false;
      printError();
      return;
    }
    balance = balance - amount;
    numberOfWithdrawals++;
}
```

Figure 6. A bank account example

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.

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 [19], [12]. 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 LOC.

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 LOC. We have used this example before in the context of regression analysis [16].

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).

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 MathWorks[6]. 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 [23].

C. Experimental Results

1) Iterative Deepening: We conducted several groups of experiments. In each group, we increase 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 with both depth A and depth B. We collected time and states explored results which are reported by SPF for regular symbolic execution and iterative deepening at both of the two iterations. We also collected the number of calls to the underlying constraint solver.

In Table I-(a), we can see that the symbolic execution with trie building took 10 to 20 seconds more than the regular symbolic execution; the iterative deepening approach explored fewer states, and had much fewer solver calls, but took almost the same time compared with regular symbolic execution. Note that the space is very big, and much time was spent on storing and loading the trie. When this part of time is excluded, the iterative deepening approach at depth 25 and 30 took only 26 seconds and 17 seconds respectively, which is significantly less than regular symbolic execution.

In Table I-(b), we find that symbolic execution with trie building and regular symbolic execution took almost the same time, and the time for building the trie is almost negligible. However, the iterative deepening approach achieved great reductions in terms of the state space explored, the number of solver calls, and time. Especially for the last group where the depth is increased from 34 to 35, the reduction is more than an order of magnitude.

In Table I-(c), similar to (b), the time cost for building the trie is little. Interestingly, the iterative deepening made half calls to the solver, and took half time as regular symbolic execution, but almost explored the same number of states. Moreover, the time difference between iterative deepening approach and regular symbolic execution is almost the time used by regular symbolic execution at the smaller depth. The reason for this is that constraint solving for this example is quite expensive, and the constraint solving during executing the paths up to the smaller depth is turned off in symbolic execution with iterative deepening at the larger depth.

We find that symbolic execution with trie building and regular symbolic execution took almost the same time, and the time for building the trie is almost negligible. However, the iterative deepening approach achieved great reductions in terms of the state space explored, the number of solver

Depth		Time for A (mm:ss)		States for B		Time for B (mm:ss)		#Solver calls for B	
A	B	Regular	Build	Regular	Iterative	Regular	Iterative	Regular	Iterative
24	25	00:34	00:43	349272	252952	00:38	00:46	335358	77312
29	30	01:01	01:20	644184	171784	01:02	00:54	629758	32256

(a) WBS Example

Depth		Time for A (mm:ss)		States for B		Time for B (mm:ss)		#Solver calls for B	
A	B	Regular	Build	Regular	Iterative	Regular	Iterative	Regular	Iterative
24	25	01:17	01:28	17103	16756	02:40	01:35	12252	2942
29	30	02:54	02:48	33273	15250	02:50	01:24	25684	1540
34	35	03:01	02:48	35359	1476	03:38	00:20	27636	18

(b) MerArbiter Example

Depth		Time for A (mm:ss)		States for B		Time for B (mm:ss)		#Solver calls for B	
A	B	Regular	Build	Regular	Iterative	Regular	Iterative	Regular	Iterative
9	10	03:12	03:15	674	647	04:15	02:01	591	243
11	12	13:48	14:02	2243	2113	26:13	12:12	2160	966

(b) Apollo Example

Table I
ITERATIVE DEEPENING RESULTS

calls, and time. Especially for the last group where the depth is increased from 34 to 35, the reduction is more than an order of magnitude.

Since building the trie only monitors the search and builds the data structure, instead of changing the behavior of the search engine or 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.

2) Regression Symbolic Execution (RSE): We performed experiments on *TCAS* and *MerArbiter* to evaluate the effectiveness of regression symbolic execution based on memoized symbolic execution. The trie is built when symbolic execution is performed of the original program version, and then reused for the run of symbolic execution of a new program version.

We collected the number of states explored, time cost, and the number of solver calls for both regular symbolic execution and regression symbolic execution on the changed program version only, because the cost for building the trie is similar to what we have for iterative deepening approach.

For *TCAS*, we randomly selected three mutant versions v6, v25, and v30 from the SIR repository [3]. Version v6 has an operator change from "<" to "<=", v25 has an operator change, and v30 has an return value change. Since there were no previous versions 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`.

Version	States		Time (mm:ss)		#Solver calls	
	Regular	RSE	Regular	RSE	Regular	RSE
v6	2688	2186	04:48	03:15	2566	1696
v25	2688	2658	04:40	03:46	2566	2016
v30	752	722	01:08	01:06	686	632

(a) TCAS Example

Version	States		Time (mm:ss)		#Solver calls	
	Regular	RSE	Regular	RSE	Regular	RSE
v1	17718	6567	01:45	00:39	12596	2072
v2	17103	43	01:47	00:05	12252	2

(b) MerArbiter Example

Table II
REGRESSION SYMBOLIC EXECUTION RESULTS

In Table II, we can see that for *TCAS*, the reduction of states explored is not much. Compared with regular symbolic execution, RSE took about 1.5 minutes less than regular symbolic execution on version v6, about one minute less on version v25, but about the same time on version v30. The reduction in the number of solver calls varies. There is about one third reduction on v6, but almost no reduction on v30.

For *MerArbiter*, the reduction is significant. On version v1, RSE 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 2, the reduction achieved by RSE is even more significant. The differences in both time, states explored, and number of solver calls are several orders of magnitude.

3) Heuristics-guided Symbolic Execution: For the *BankAccount* example shown in Figure 6, 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 one minute and 46 seconds, and made 8190 solver calls. The two heuristics were applied based on the trie collected at a smaller depth, which is termed as base depth. We picked 20, 25, 30, and 35 as the base depth. In Table III-(a), we can see that, for tries built at both depth 20 and 25, the reachability heuristic selected half of the paths ended with boundary nodes as candidates to execute, explored half of the state space, took about 50 seconds less than regular symbolic execution, and made about four thousand 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 III-(b), with counter heuristic applied, the number of candidate path is 1 or 2 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 regular symbolic execution, but the time reduction is similar to what was achieved by 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 reachability heuristic or 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 4 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 5 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 III-(c). At depth 40, the heuristic had no effect 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 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 with depth bound 25 for the run with depth bound 30. In Table III-(d), we find that the savings achieved by the reachability heuristic are significant. We checked the byte-code 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.

Base Depth	Candidates	States	Time (mm:ss)	#Solver calls
20	32/64	8253	00:56	4032
25	128/256	8445	00:55	3840
30	0/512	-	-	-
35	0/2048	-	-	-

(a) Reachability Heuristic for BankAccount

Base Depth	Candidates	States	Time (mm:ss)	#Solver calls
20	1/64	277	00:50	127
25	1/256	93	00:50	31
30	2/512	93	00:39	29
35	2/2048	53	00:47	5

(b) Counter Heuristic for BankAccount

Depth	States		Time (mm:ss)		#Solver calls	
	Regular	HR	Regular	HR	Regular	HR
40	1484	1484	00:04	00:04	1482	1482
60	3416	3416	00:03	00:03	3414	1932
80	6114	538	00:04	00:03	6114	420
100	9616	578	00:06	00:03	9614	420
120	13610	312	00:09	00:03	13608	114

(c) Reachability Heuristic for example with two loops

Depth	States		Time (mm:ss)		#Solver calls	
	Regular	HR	Regular	HR	Regular	HR
25→30	33273	2071	03:15	00:20	25684	932

(d) Reachability Heuristic for MerArbiter

Table III
HEURISTICS GUIDED SYMBOLIC EXECUTION RESULTS

D. 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 as specific underlying constraint solver, (3) the selection of examples used in the experiments, (4) the specific depths picked for symbolic execution, and (5) 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.

E. Discussion

For program WBS, the results of iterative deepening show that loading and storing trie could be costly, especially when the trie is growing big. However, since the paths ended with regular nodes and unsatisfiable nodes are not useful for the next iteration of symbolic execution, they can be pruned when re-using the trie, and even more there is no need for storing them. Thus, an incomplete trie with only paths ended with boundary nodes can be used specifically for iterative deepening approach. The main benefit is space reduction and reduction of the time for storing and loading the trie, but the reduction in states and solver calls should remain the same.

The savings of using regression symbolic execution depends on the location of the change, and may vary quite a lot between different kinds of changes. It is supported by our results of regression symbolic execution.

In previous work, we have developed directed incremental symbolic execution (DiSE) [16] 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. RSE 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 RSE 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 RSE. First, DiSE only generates affected path conditions, while RSE generates 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 RSE is dynamic, based on the trie, thus RSE is more precise. For example, when the change is in an un-covered code, RSE 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 a regular symbolic execution; RSE explores the unchanged path prefix more efficiently by turning off the constraint solving.

We implemented several 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, it is not reflected in savings of time. The reason is that constraint solving for those programs is very cheap. We believe that for programs with complex constraints, such as *Apollo*, one would gain more benefits from using Memoise.

VI. 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 [16], parallel symbolic execution [24], 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. [14], [28]. 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 path constraints, turning-off constraint solving etc. Furthermore, the approaches [14], [28] target a particular application, namely regression analysis, while Memoise can enable multiple applications.

VII. Conclusions

We presented memoized symbolic execution (Memoise), a new approach for more 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 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. The results cached by Memoise can be further used to guide the analysis in successive runs.

Experiments using our prototype embodiment 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. In the future, we plan to investigate in more detail and to implement the other applications that we outlined here for Memoise. We further plan to perform more experiments to fully assess the merits of the presented techniques in practice.

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 *Proceedings 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, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[10] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, 2011. ACM.

[11] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 297–306, September 2008.

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

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

[14] 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.

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

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

[17] C. Păsăreanu, 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*, pages 15–25, 2008.

[18] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE*, pages 179–180. ACM, 2010.

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

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

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

[22] 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.

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

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

[25] A. Taleghani and J. M. Atlee. Search-carrying code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 367–376, New York, NY, USA, 2010. ACM.

[26] 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.

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

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