Pruning Unaffected Code in Equivalence Checking of Program Versions

Sahar Badihi*, Yi Li†, and Julia Rubin*
*The University of British Columbia, Canada, †Nanyang Technological University, Singapore

***Abstract*—Symbolic execution is a powerful static analysis technique which can be used for program equivalence checking. Yet, due to complex programming constructs, such as loops and non-linear arithmetic, scaling symbolic execution to real systems remains a challenge. In this paper, we discuss two of the most prominent approaches for dealing with the scalability issue when considering equivalence of two subsequent versions of a program. These approaches leverage the fact that program versions are largely similar and prune parts of the code that are not affecting the equivalence of the versions. We discuss advantages and disadvantages of these approaches and outline a solution that combines their advantages while mitigating the disadvantages.**

I. Introduction

Equivalence checking establishes whether two versions of a program have identical behavior and is used in a variety of tasks, such as determining the correctness of compiler optimizations or code refactoring [1]. Functional equivalence – establishing that two versions of a program produce the same output for any identical input – is the most popular form of equivalence used in practice [2].

Symbolic execution – a static program analysis technique that uses symbolic rather than actual values as program inputs, explores all possible program execution paths, and computes the execution result of each path in terms of symbolic inputs – is often used to implement equivalence checking. The core idea behind the use of symbolic execution for functional equivalence checking is to compute a program's *symbolic summary*, which captures the *path conditions* and *effects* of each path [2]. For example, the method *log* in Figure 1a has three parameters: *currentTime*, *t*, and *length*, two local variables: *val*, and *temp* and a global variable *old*, which are represented by six respective symbolic variables: *CT*, *T*, *L*, *V*, *TMP*, and *O*. The method has several execution paths; the condition of the path at Lines 4–6 is (CT-T<100) and the effect is (Ret=O+1).

The symbolic summary of each path is a conjunction of formulas representing its path condition and effects at the end of the path; the symbolic summary of a method is simply a disjunction of all its path summaries. Comparing the symbolic summaries of two methods can help determine their equivalence or find a counterexample to demonstrate non-equivalence. More specifically, the summaries of two methods S_1 and S_2 are combined into a formula $(\neg(S_1 \iff S_2))$ and given to a constraint solver, typically, SAT or SMT. If the formula holds, there is an assignment that makes the two methods different, i.e., an *equivalence counterexample*. Otherwise, there is no such assignment and thus the methods are equivalent.

```
1
2  old = 1;
3  log(currentTime, t, length){
4    if(currentTime - t < 100){
5      val = 1 + old;
6      return val; }
7    else {
8      for(i = 0; i < length; i++)
9          val = val + i;
10     temp = old;
11     old = val;
12     return val + temp; }
13 }
```

(a) Version 1

```
base = 1;
old = 1;
log(currentTime, t, length){
  if(currentTime - t < 100){
    val = base + old;
    return val; }
  else {
    for(i = 0; i < length; i++)
        val = val + i;
    temp = old;
    old = val;
    return val + temp; }
}
```

(b) Version 2

```
base = 1;
old = 1;
log(currentTime, t, length){
  if(currentTime - t < 100){
    val = base + old;
    return val; }
  else {
    for(i = 0; i < length; i++)
        val = val + i;
    temp = old;
    old = val;
    return val + base; }
}
```

(c) Version 3

Fig. 1: Example program.

Limitations of symbolic execution are well-known: unbounded loops and recursion lead to a large number of paths and the exact number of iterations is difficult/impossible to determine statically. Thus, most techniques rely on a user-specified *bound* for the number of iterations, e.g., two or five. For the *log* method in Figure 1a, unrolling the *for* loop at Lines 8–9 with the bound of five produces five execution paths, each with its corresponding condition and effect.

Bounded equivalence checking is not as accurate as full equivalence checking as it might miss feasible behaviors, e.g., in the sixth execution of the loop. Complex expressions in summaries, such as non-linear integer arithmetic, also hinder equivalence checking as they lead to expressions in summaries that are intractable for modern constraint solvers.

To deal with these problems, a variety of techniques leverage information about changed code parts to simplify the produced summaries [2], [3], [4], [5]. We discuss these techniques and outline their limitations in Section II. We propose an approach to mitigate some of these limitations in Section III.

II. Analysis of Existing Solutions

Differential symbolic execution (DSE). Person et al. [2] is one of the first to use symbolic execution for program equivalence checking. DSE uses *uninterpreted functions*, i.e., functions that are free to take any value, to encode code blocks that are unchanged between two methods, to reduce the scope of the analysis.

For example, consider the two versions of the *log* method in Figures 1a and 1b, V_1 and V_2. The change between the versions is the replacement of the "magic number" 1 at Line 5 of V_1 with the global variable *base* introduced at Line 1 of V_2. In the "naïve" approach to constructing symbolic summaries of these two methods, one needs to bound the loop (Lines 8–9) to a user-specified depth, say five, producing summaries that contain six clauses each: one for the *if* part of the *log* method and five for the *else* part.

Yet, DSE identifies the common code block between the two versions of these methods (Lines 8–11 in Figures 1a and 1b),

collects all variables that are defined or changed in the block, e.g., val, $temp$, and old, and represents each as an uninterpreted functions which accepts as inputs all variables that are used in the block, e.g., $length$, val, and old. That is, the common code in this example is represented by three uninterpreted functions, $UF_{val}(L,V,O)$, $UF_{temp}(L,V,O)$, and $UF_{old}(L,V,O)$. The return statements (Line 12 in both figures), even if common, are not converted to uninterpreted functions as they capture the effect of the entire path and are required for the summary. The equivalence checking formula produced by DSE then consists of only two paths for the method log in each V_1 and V_2:

$$\neg\Big(\big((CT-T<100 \wedge O==1 \wedge \mathbf{Ret}=\mathbf{1}+\mathbf{O}\big)\vee$$
$$\big(CT-T\geqslant 100 \wedge IP(L,V,O) \wedge Ret=UF_{val}(L,V,O)+UF_{temp}(L,V,O)\big)\big) \iff$$
$$\big((CT-T<100 \wedge B==1 \wedge O==1 \wedge \mathbf{Ret}=\mathbf{B}+\mathbf{O}\big)\vee$$
$$\big(CT-T\geqslant 100 \wedge IP(L,V,O) \wedge Ret=UF_{val}(L,V,O)+UF_{temp}(L,V,O)\big)\big)\Big)$$

Even in the presence of uninterpreted functions, this formula can be solved by an SMT solver because the solver uses the equality logic of uninterpreted functions, i.e., it assumes that given the same inputs, instances of the same function always return the same value. In this case, the formula evaluates to false, meaning that the two methods are equivalent. The verification result produced by DSE is thus "stronger" and more accurate than that of the "naïve" checker with loop bounding.

However, when comparing V_2 to the next revision of the program, V_3 in Figure 1c, where the return value at Line 12 was modified, DSE will consider Lines 1–2, 5, and 8–11 common. The variables used at these line will be represented by uninterpreted functions, resulting in the summary specified below:

$$\neg\Big(\big((CT-T<100 \wedge IP(B,O) \wedge Ret=UF_{val}(B,O)\big)\vee$$
$$\big(CT-T\geqslant 100 \wedge IP(L,V,O) \wedge \mathbf{Ret}=\mathbf{UF_{val}}(\mathbf{L},\mathbf{V},\mathbf{O})+\mathbf{UF_{temp}}(\mathbf{L},\mathbf{V},\mathbf{O})\big)\big) \iff$$
$$\big((CT-T<100 \wedge IP(B,O) \wedge Ret=UF_{val}(B,O)\big)\vee$$
$$\big(CT-T\geqslant 100 \wedge IP(L,V,O) \wedge \mathbf{Ret}=\mathbf{UF_{val}}(\mathbf{L},\mathbf{V},\mathbf{O})+\mathbf{UF_{base}}()\big)\big)\Big)$$

The first path in both versions will be judged equivalent due to equality of uninterpreted function $UF_{val}(B,O)$. To solve the second path, one needs to determine whether $UF_{temp}(L,V,O)$ can be equal to $UF_{base}()$, which requires full symbolic summaries of these functions.

IMPacted Summaries (IMP-S). Instead of identifying common code blocks, Bakes et al. [3] propose a technique for pruning program paths that are not impacted by the changed code. It performs backward and forward program slicing to identify all such statements, e.g., statements at Lines 8–13 in Figure 1b. It then prunes all clauses of the full equivalence checking formula that do not contain impacted statements. The authors prove that such pruning grantees correct equivalence checking result for any given bound.

For Figures 1a and 1b, the impacted statements are at Lines 1–2 and 4–6 only. Thus, the final equivalence formula given below, again, evaluates to false, proving that the methods are equivalent without performing any loop bounding.

$$\neg\big((CT-T<100 \wedge O==1 \wedge \mathbf{Ret}=\mathbf{1}+\mathbf{O}) \iff$$
$$(CT-T<100 \wedge B==1 \wedge O==1 \wedge \mathbf{Ret}=\mathbf{B}+\mathbf{O})\big)$$

However, when comparing the methods in Figures 1b and 1c, the statement at Line 9, inside the *for* loop, is impacted. IMP-S thus cannot prove equivalence without bounding the loop, as the "naïve" approach.

Both **ModDiff** [4] and **CLEVER** [5] perform path pruning similar to IMP-S, scaling the analysis to work in an inter-procedural manner, either bottom-up or top-down. They exhibit behaviors similar to IMP-S for both examples in Figure 1.

III. Proposed Idea

We observe that while IMP-S considers the statement in the *for* loop of Figure 1c (Line 9) as impacted, it is not *affecting the equivalence of the methods* as this statement is (a) common and (b) has the exact same effect on symbolic summaries in both versions of the method. As such, the value computed by this statement can be represented by an uninterpreted function without hindering the "solvability" of the final equivalence checking formula. As discussed in Section II, that is not true for all the common code, as representing values computed at Lines 1 and 10 with uninterpreted functions will lead to an unsolvable summary.

The main idea proposed in this paper is thus to identify and represent with uninterpreted functions only *common code that is not affecting the summaries*. To this end, first, all impacted statements are calculated. Then, we identify all impacted statements that are common between the versions. This set of statements is further divided into two: *affected* and *unaffected*. The unaffected statements can be represented by uninterpreted functions, improving the scalability and applicability of the equivalence checking process for cases when *impacted, common, unaffected statements contain loops and non-linear arithmetic*.

For the example in Figures 1b and 1c, such procedure will produce the summary specified below, allowing us to establish equivalence without bounding loops, which none of the existing approaches can do.

$$\neg\big((CT-T\geqslant 100 \wedge O==1 \wedge \mathbf{Ret}=\mathbf{UF_{val}}(\mathbf{L},\mathbf{V},\mathbf{O})+\mathbf{O}) \iff$$
$$(CT-T\geqslant 100 \wedge B==1 \wedge \mathbf{Ret}=\mathbf{UF_{val}}(\mathbf{L},\mathbf{V},\mathbf{O})+\mathbf{B})\big)$$

At the time of writing, we implemented a naïve process for separating affected and unaffected statements: it first abstracts all common impacted statements and then iterative refines them when an SMT solver requires such a refinement. We showed that the approach works on a number of examples. As future work, we intend to explore more robust approaches for making such distinction a priori, without reverting to an SMT solver. We look forward to discussing this topic at the FMCAD Student Forum.

References

[1] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic Program Alignment for Equivalence Checking," in *Proc. of PLDI'19*.

[2] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential Symbolic Execution," in *Proc. of FSE'08*.

[3] J. Backes, S. Person, N. Rungta, and O. Tkachuk, "Regression Verification Using Impact Summaries," in *Proc. of SPIN Workshop on Model Checking of Software'13*.

[4] A. Trostanetski, O. Grumberg, and D. Kroening, "Modular Demand-driven Analysis of Semantic Difference for Program Versions," in *Proc. of SAS'17*.

[5] F. Mora, Y. L. Li, J. Rubin, and M. Chechik, "Client-specific Equivalence Checking," in *Proc. of ASE'18*.