Regression Verification using Impact Summaries

John Backes[1], Suzette Person[2], Neha Rungta[3], and Oksana Tkachuk[3]

[1] University of Minnesota
back0145@umn.edu
[2] NASA Langley Research Center
suzette.person@nasa.gov
[3] NASA Ames Research Center
neha.s.rungta,oksana.tkachuk@nasa.gov

Abstract. Regression verification techniques are used to prove equivalence of closely related program versions. Existing regression verification techniques leverage the similarities between program versions to help improve analysis scalability by using abstraction and decomposition techniques. These techniques are sound but not complete. In this work, we propose an alternative technique to improve scalability of regression verification that leverages change impact information to partition program execution behaviors. Program behaviors in each version are partitioned into (a) behaviors impacted by the changes and (b) behaviors not impacted (unimpacted) by the changes. Our approach uses a combination of static analysis and symbolic execution to generate summaries of program behaviors impacted by the differences. We show in this work that checking equivalence of behaviors in two program versions reduces to checking equivalence of just the impacted behaviors. We prove that our approach is both sound and complete for sequential programs, with respect to the depth bound of symbolic execution; furthermore, our approach can be used with existing approaches to better leverage the similarities between program versions and improve analysis scalability. We evaluate our technique on a set of sequential C artifacts and present preliminary results.

1 Introduction

Various reduction, abstraction, and compositional techniques have been developed to help scale software verification techniques to industrial-sized systems. Although such techniques have greatly increased the size and complexity of systems that can be checked, analysis of large software systems remains costly. Regression analysis techniques, e.g., regression testing [16], regression model checking [22], and regression verification [19], restrict the scope of the analysis by leveraging the differences between program versions. These techniques are based on the idea that if code is checked early in development, then subsequent versions can be checked against a prior (checked) version, leveraging the results of the previous analysis to reduce analysis cost of the current version.

Regression verification addresses the problem of proving equivalence of closely related program versions [19]. These techniques compare two programs with a

large degree of syntactic similarity to prove that portions of one program version are equivalent to the other. Regression verification can be used for guaranteeing backward compatibility, and for showing behavioral equivalence in programs with syntactic differences, e.g., when a program is refactored to improve its performance, maintainability, or readability.

Existing regression verification techniques leverage similarities between program versions by using abstraction and decomposition techniques to improve scalability of the analysis [10, 12, 19]. The abstractions and decomposition in the these techniques, e.g., summaries of unchanged code [12] or semantically equivalent methods [19], compute an over-approximation of the program behaviors. The equivalence checking results of these techniques are sound but not complete—they may characterize programs as not functionally equivalent when, in fact, they are equivalent.

In this work we describe a novel approach that leverages the impact of the differences between two programs for scaling regression verification. We partition program behaviors of each version into (a) behaviors impacted by the changes and (b) behaviors not impacted (unimpacted) by the changes. Only the impacted program behaviors are used during equivalence checking. We then prove that checking equivalence of the impacted program behaviors is equivalent to checking equivalence of all program behaviors for a given depth bound. In this work we use symbolic execution to generate the program behaviors and leverage control- and data-dependence information to facilitate the partitioning of program behaviors. The impacted program behaviors are termed as *impact summaries*. The dependence analyses that facilitate the generation of the impact summaries, we believe, could be used in conjunction with other abstraction and decomposition based approaches, [10, 12], as a complementary reduction technique. An evaluation of our regression verification technique shows that our approach is capable of leveraging similarities between program versions to reduce the size of the queries and the time required to check for logical equivalence.

The main contributions of this work are:

- A regression verification technique to generate impact summaries that can be checked for functional equivalence using an off-the-shelf decision procedure.
- A proof that our approach is sound and complete with respect to the depth bound of symbolic execution.
- An implementation of our technique using the LLVM compiler infrastructure, the KLEE Symbolic Virtual Machine [4], and a variety of Satisfiability Modulo Theory (SMT) solvers, e.g., STP [7] and Z3 [6].
- An empirical evaluation on a set of C artifacts which shows that the use of impact summaries can reduce the cost of regression verification.

2 Motivation and Background

2.1 Checking Functional Equivalence

In this work, we focus on functional equivalence [12]. Two programs, P_0 and P_1, are functionally equivalent iff for all possible input values to the programs, they

```
1: int func(unsigned int val) {
2:   if((val & 0x03) == 0) { //divisible by 4
3:      val = val + 4; // change to val = val + 2;
4:      return mod2(val)
5:  } else return 0;
6: }
7: int mod2(unsigned int x) {
8:   return ((x & 0x01) == 0); // divisible by 2
9: }
```

Fig. 1. Program behavior is unchanged when the constant value in line 3 is even.

both produce the same output, i.e., they return the same value and result in the same global state. In general, proving functional equivalence is undecidable, so we prove functional equivalence with respect to a user-specified depth-bound for loops and recursive functions. Note that this notion of equivalence is similar to the k-equivalence defined in [19].

Equivalence checking techniques that use uninterpreted functions as a mechanism for abstraction and decomposition [10, 12, 19] produce sound but not complete results. The example in Figure 1 demonstrates how the use of uninterpreted functions can lead to false negatives. The input to methods `func` and `mod2` is an unsigned integer. If the input to `func`, *val*, is divisible by four, then in version V_0 of `func`, four is added to *val* and method `mod2` is invoked with the updated variable, *val*. Next, `mod2` returns true if its input, x, is divisible by two; otherwise it returns false. Suppose, a change is made to line 3 in V_1 of `fun` and two is added to *val* in lieu of four. Both versions of `func` are functionally equivalent, i.e., for all possible inputs to `func`, the output is the same in both versions.

Symdiff is a technique which uses uninterpreted functions during equivalence checking [10]. It modularly checks equivalence of each pair of procedures in two versions of the program. To check the equivalence of the `func` method, it replaces the call to `mod2` at line 4 with an uninterpreted function. The inputs to the uninterpreted function are parameters and global values read by the method. In V_0 of `func` the uninterpreted function for the call to `mod2` is $f\_mod2(val + 4)$ while in V_1 it is $f\_mod2(val + 2)$. The procedures are then transformed to a single logical formula whose validity is checked using verification condition generation. Symdiff will report V_0 and V_1 of `func` as not equivalent due to the different input values to the uninterpreted function: f_mod2. The use of uninterpreted functions results in an over-approximation because equality logic with uninterpreted functions (EUF) relies on functional congruence (consistency) —a conservative approach to judging functional equivalence which assumes that instances of the same function return the same value if given equal arguments [9]. Other equivalence checking techniques that rely on uninterpreted functions will report similar false negatives.

1: int a, b;
2: `void test(int` x, `int` y){
3: **if**$(x > 0)$ $a = a + 1$; **else** $a = a + 2$; //*change* $x <= 0$
4: **if**$(y > 0)$ $b = b + 1$; **else** $b = b + 2$;
5: }

Fig. 2. An example where equivalence cannot be naively checked using DiSE.

2.2 Symbolic Execution

Symbolic execution uses symbolic values in lieu of concrete values for program inputs and builds a path condition for each execution path it explores. A path condition contains (a) a conjunction of constraints over the symbolic input values and constants such that they represent the semantics of the statements executed on a given path p and (b) the conjunction of constraints that represent the effects of executing p—the return value and the final global state. The disjunction of all the path conditions generated during symbolic execution is a symbolic summary of the program behaviors. Version V_0 of the `test` method in Figure 2 has two integer inputs x and y whose values determine the updates made to the global variables a and b. There are four path conditions for V_0 generated by symbolic execution:

1. $x > 0 \wedge y > 0 \wedge a_0 = a + 1 \wedge b_0 = b + 1$
2. $\neg(x > 0) \wedge y > 0 \wedge a_1 = a + 2 \wedge b_0 = b + 1$
3. $x > 0 \wedge \neg(y > 0) \wedge a_0 = a + 1 \wedge b_1 = b + 2$
4. $\neg(x > 0) \wedge \neg(y > 0) \wedge a_1 = a + 2 \wedge b_1 = b + 2$.

Each path condition has constraints on the inputs x and y that lead to the update of global variables a and b. The variables a_0, a_1, b_0, and b_1 are temporary variables that represent the final assignments to global variables a and b.

2.3 Change Impact Analysis

The DiSE framework, in our previous work, implements a symbolic execution based change impact analysis for a given software maintenance task [13, 17]. DiSE uses the results of static change impact analyses to direct symbolic execution toward the parts of the code that may be impacted by the changes. The output of DiSE is a set of impacted path conditions, i.e., path conditions along program locations impacted by differences in programs.

The inputs to DiSE are two program versions and a target client analysis. DiSE first computes a syntactic diff of the program versions to identify locations in the source code that are modified. Then DiSE uses program slicing-based techniques to detect impacted program locations, i.e., locations that have control- and data-dependencies on the modified program locations. The set of impacted program locations is used to direct symbolic execution to explore execution paths

containing impacted locations. In the parts of the program composed of locations not impacted by the change, DiSE explores a subset of the feasible paths through that section.

The dependence analyses and pruning within the DiSE framework are configurable based on the needs of the client analysis. To illustrate how DiSE computes path conditions for generating test inputs to cover impacted branch statements, consider the example in Figure 2. Suppose a change is made to line 3 where the condition $x > 0$ in V_0 of `test` is changed to $x <= 0$ in V_1. Due to this change, the conditional statement and assignments to global variable a on line 3 are marked as impacted in both versions. The goal of the symbolic execution in DiSE is to generate path conditions that cover both *true* and *false* branches of the conditional branch statement, $x <= 0$, and explore *any one* of the branches of the conditional branch statement, $y > 0$. The path conditions for program version, V_0, that may be generated by DiSE are:

1. $x > 0 \wedge y > 0 \wedge a_0 = a + 1 \wedge b_0 = b + 1$
2. $\neg(x > 0) \wedge y > 0 \wedge a_1 = a + 2 \wedge b_0 = b + 1$;

Here both branches of the $x \leq 0$ are explored while the *true* branch of the $y > 0$ is explored. Similarly the path conditions for version V_1 that may be generated by DiSE are:

1. $x \leq 0 \wedge \neg(y > 0) \wedge a_0 = a + 1 \wedge b_1 = b + 2$
2. $\neg(x \leq 0) \wedge \neg(y > 0) \wedge a_1 = a + 2 \wedge b_1 = b + 2$.

In version, V_1 both branches of $x \leq 0$ are still explored but the *false* branch of the $y > 0$ is explored. Note this is because DiSE does not enforce a specific branch to be explored for an unimpacted conditional statement. These path conditions can be solved to generate test inputs that drive execution along the paths that contain impacted locations.

The path conditions generated for regression testing, related to impacted branch coverage, in the DiSE framework under-approximate the program behaviors. As shown above the constraints on the variable y in the path conditions generated by DiSE shown above are different in V_0 from those generated in V_1—the path conditions represent different under-approximations of the program behaviors. This under-approximation does not allow the path conditions to be used for equivalence checking. Furthermore the dependence analysis is also tailored to suit the needs of the client analyses. The client analyses that are currently supported in DiSE are related to regression testing (test inputs to satisfy different coverage criteria) and improving DARWIN based delta debugging.

In this work we add support for performing equivalence checking within the DiSE framework. For this we define a set of static change impact rules that allow us to precisely characterize the program statements as impacted or unimpacted such that checking equivalence of behaviors of two programs reduces to the problem of checking equivalence of the behaviors encoded by the impacted statements.

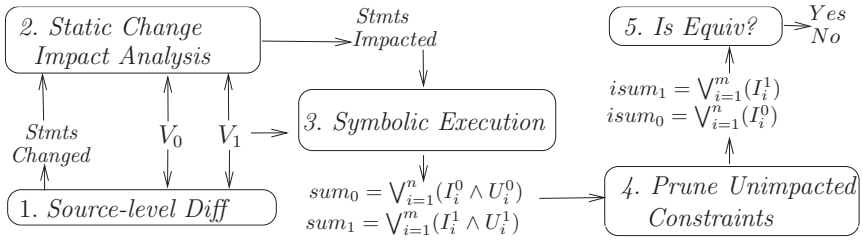

Fig. 3. Overview of regression verification using impact summaries.

3 Regression Verification using Impact Summaries

An overview of our regression verification technique is shown in Figure 3. Steps 1–3 in Figure 3 represent a static change impact analysis that is performed on V_0 and V_1. The change impact analysis marks the program statements that are impacted by the differences between V_0 and V_1. The outputs from Step 3 are the program behavior summaries (full summaries) for program versions V_0 and V_1. Each symbolic summary consists of the path conditions representing the program execution behaviors.

In order to facilitate the characterization of the program behaviors as impacted or unimpacted, we first define a mechanism to distinguish between different behaviors encoded within a given path condition. For the example shown in Figure 2 each path condition encodes two program behaviors; the first program behavior is related to the input variable, x, and global variable, a; while the second program behavior is related to the input variable y and global variable b. We can make this distinction because the operations on variables x and a are completely disjoint from the operations on variables y and b. The constraints on x and a represent one set of program behaviors for the example in Figure 2 while the constraints on y and b represent another set of behaviors. Based on this distinction a path condition can contain *num* behaviors such that the set of constraints encoding each behavior are completely disjoint from the constraints encoding the other behaviors.

In this work, we partition the constraints in each path condition generated by the change impact analysis as either *impacted* or *unimpacted*. An impacted (unimpacted) constraint I_i (U_i) is a constraint that is added to the path condition as a result of executing an impacted (unimpacted) program statement during symbolic execution. The conjunction of the impacted constraints, I_i, in a path condition represents impacted program behaviors, while the conjunction of the unimpacted constraints, U_i in a path condition, represents unimpacted program behaviors.

(1) if $S_i \in \mathbb{I}$ and S_j is *control dependent* on S_i then $\mathbb{I} \cup \{S_j\}$
(2) if $S_i \in \mathbb{I}$ and S_j *uses* (reads) the value of a variable *defined* (written) at S_i then $\mathbb{I} \cup \{S_j\}$

(3) if $S_i \in \mathbb{I}$ an S_i is *control dependent* on S_j then $\mathbb{I} \cup \{S_j\}$
(4) if $S_i \in \mathbb{I}$ and S_j defines (writes) a variable whose value is *used* (read) at S_i then $\mathbb{I} \cup \{S_j\}$

Table 1. Control and data dependence rules for computing impacted statements.

Definition 1. *A **full summary** is a disjunction of the impacted constraints I_i and the unimpacted constraints U_i for a program with n paths: $sum = \bigvee_{i=1}^{n}(I_i \wedge U_i)$.*

For example, the full summary for V_0 containing n paths is given by $sum_0 = \bigvee_{i=1}^{n}(I_i^0 \wedge U_i^0)$. The full summaries are post-processed in Step 4, as shown in Figure 3 to remove the unimpacted constraints and generate *impact summaries*.

Definition 2. *An **impact summary** consists of a disjunction of the impacted constraints I_i for a program with n paths: $isum = \bigvee_{i=1}^{n}(I_i)$.*

The resulting impact summaries are then checked for functional equivalence [12] in Step 5, by using an off-the-shelf SMT solver, e.g., STP [7] or Z3 [6] to check for logical equivalence. In Section 4 we prove that the full summaries for two programs are functionally equivalent iff their impact summaries are functionally equivalent. Formally, we demonstrate that for a program V_0 with n paths and a program V_1 with m paths, Formula 1 is a tautology.

$$[(\bigvee_{i=1}^{n} I_i^0) \leftrightarrow (\bigvee_{i=1}^{m} I_i^1)] \leftrightarrow [\bigvee_{i=1}^{n}(I_i^0 \wedge U_i^0) \leftrightarrow \bigvee_{i=1}^{m}(I_i^1 \wedge U_i^1)] \quad (1)$$

3.1 Computing Impacted Program Statements and Behaviors

In this section we present the set of rules that are necessary to conservatively compute, for sequential programs, the set of program statements that may be *impacted* by added or deleted program statements. We then briefly discuss how the set of impacted statements can be used to compute impacted program behaviors. The static analysis in this work uses standard control- and data-flow analysis to compute the set of impacted statements. The rules for the forward and backward flow analysis are shown in Table 1. Given the conservative nature of the analysis, it may mark certain unimpacted statements as impacted. The analysis, however, is guaranteed to find all impacted statements. We present a high-level description of how the rules are applied in the steps below:

Step 1 A source-level syntactic diff is performed to generate the change sets for the related program versions V_0 and V_1. The change set for V_0 is $\mathbb{C}_0$. It contains the set of statements in V_0 that are removed in V_1. The change set for V_1 is $\mathbb{C}_1$ which contains statements in V_1 that are added with respect to V_0. Note

that all edited statements can be treated as removed in one version and added in another.

Step 2 The impact set for program version V_0 is initialized with statements in the change set of V_0: $\mathbb{I}_0 := \mathbb{C}_0$.

Step 3 To account for forward control- and data-flow dependence, rules (1) and (2) in Table 1 are iteratively applied to $\mathbb{I}_0$ until they reach a fixpoint.

Step 4 The impact set for program version V_1 is initialized to the change set of V_1: $\mathbb{I}_1 := \mathbb{C}_1$.

Step 5 For all statements in the impact set of V_0, $\forall S_i \in \mathbb{I}_0$, if there exists a corresponding statement $S'_i \in V_1$ such that $S_i \sim S'_i$—then it is added to the impact set of V_1, $\mathbb{I}_1 := \mathbb{I}_1 \cup \{S'_i\}$. This step is performed to account for the impact of the statements *removed* in V_0.

Step 6 To compute the impact of the changes using forward control- and data-flow dependences, rules (1) and (2) in Table 1 are iteratively applied to $\mathbb{I}_1$ until a fixpoint is reached. Rule (3) is applied once to $\mathbb{I}_1$ to account for backward control-flow dependence. Finally, Rule (4) is applied to $\mathbb{I}_1$ transitively to compute the reaching definitions.

Step 7 Statements from the impact set of V_1 are mapped to the impact set of V_0: $\forall S_i \in \mathbb{I}_1$, if there exists a corresponding statement in $S'_i \in V_0$, $S_i \sim S'_i$—then it is added to the impact set of V_0, $\mathbb{I}_0 := \mathbb{I}_0 \cup \{S'_i\}$. This step accounts for the impact of statements *added* to V_1.

The constraints generated by symbolic execution at impacted program statements on path i are added to the impact summary, I_i while the unimpacted constraints are added to U_i. We can check functional equivalence of two programs using their impact summaries.

The static analysis rules presented in this section compute the set of impacted program statements within a method, i.e., the analysis is intraprocedural. In [17] we present an interprocedural change impact analysis. The algorithm in [17] statically computes the impacted program statements (impact set) for all the methods disregarding the flow of impact through different method invocations. During symbolic execution these impact sets are then dynamically refined based on the calling context, propagating the impact of changes between methods through method arguments, global variables and method return values. Due to space limitations we present only the intraprocedural version of the impact analysis in this paper. Our empirical evaluation of regression verification, however, is performed using the interprocedural version of the algorithm. Next we present an example to illustrate how impact summaries are computed for an interprocedural program.

3.2 Example

Figure 4 shows two versions of the C function `Init_Data` that invoke the same function `Set_Struct` (shown in Figure 4 (c)). Note that even though the analysis is performed on the single static assignment form of the program, to enable better readability we describe it in terms of the source. The `Init_Data` function first initializes two arrays, `Data0` and `Data1`, and the pointer to a data structure,

```
1: #define Len0 512
2: #define Len1 512
3: int Data0[Len0], Data1[Len1];
4: struct_A* StructA;
5: int Init_Data(int capacity)
6:   for(int i = 0; i < capacity∧i < Len0; i++)
7:     Data0[i] = 0;
8:   for(int i = 0; i < capacity∧i < Len1; i++)
9:     Data1[i] = 0;
10:  StructA = Set_Struct(StructA)
11:  if(capacity > Len0)
12:    return 0;
13:  if(capacity > Len1)
14:    return 0;
15:  return 1;
```

(a) V_0

```
1: #define Len0 512
2: int Data0[Len0], Data1[Len0];
3: struct_A* StructA;
4: int Init_Data(int capacity)
5:   for(int i = 0; i < capacity∧i < Len0; i++)
6:     Data0[i] = 0;
7:     Data1[i] = 0;
8:   StructA = Set_Struct(StructA)
9:   if(capacity > Len0)
10:    return 0;
11:  return 1;
```

(b) V_1

```
1: struct_A * Set_Struct(struct_A * st)
2:   if(st == NULL)
3:     return newStructA();
4:   else
5:     return ClearContents(st);
```

(c) the `Set_Struct` function

Fig. 4. Two related versions of `Init_Data` that are functionally equivalent.

`StructA`. Then, if the value of `capacity` is greater than the constant length defined for arrays `Data0` or `Data1`, the function returns zero; otherwise, it returns one. V_1 is a refactored version of V_0. In V_1, a single value specifies the length of both arrays, `Data0` and `Data1`. The refactoring also moves the initialization of `Data1` into the upper `for` loop. The two versions of `Init_Data` in Figure 4 are functionally equivalent; given same value of `capacity`, both implementations produce the same output, i.e., return the same value, and `Data0`, `Data1`, and `StructA` will point to the same initialized memory[4].

The edits to the refactored program version in Figure 4 are related to statements that access and edit the array `Data1` and the constant `Len1`. These edits, however, do not impact the program statements that reference the data structure `StructA` and `Data0`. First, let us consider the accesses to `StructA` (via `st` in function `Set_Struct`); these are completely disjoint from the operations related to `Data1` and `Len1`. Hence, the program behaviors related to the operations on `st` in this context are not impacted by the changes. The constraints related to `StructA` and `st` generated at line 10 in V_0 and line 8 in V_1 of function `Init_Data` and at lines $2-5$ in function `Set_Struct` are unimpacted and can safely be discarded from the full summaries before checking equivalence. Now, consider the accesses to `Data0` and its interactions with accesses to `Data1`. Although the assignments to both `Data0` and `Data1` are control dependent on the for loop at line 6, in the context of V_0, the assignment to `Data0` is not impacted by the changes. Consequently, the constraints on `Data0` at line 7 can also be discarded before checking equivalence. Moreover, functional equivalence of V_0

[4] We make no claims about the initialized memory's location (the value of the pointers), only the contents of the memory.

$(i_0 = 0) \wedge\ (i_0 < capacity) \wedge (i_0 < 512) \wedge (Data0[i_0] = 0)\ \wedge (Data1[i_0] = 0) \wedge$
$(i_1 = 1) \wedge\ (i_1 < capacity) \wedge (i_1 < 512) \wedge (Data0[i_1] = 0)\ \wedge (Data1[i_1] = 0) \wedge$
$\ldots$
$(i_{511} = 511) \wedge\ (i_{511} < capacity) \wedge (i_{511} < 512) \wedge (Data0[i_{511}] = 0)\ \wedge (Data1[i_{511}] = 0) \wedge$
$st = 0 \wedge st = objRef \wedge$
$StructA = st \wedge capacity <= 512 \wedge ret = 1$

(a)

$(i_0 = 0) \wedge\ (i_0 < capacity) \wedge (i_0 < 512) \wedge (Data1[i_0] = 0) \wedge$
$(i_1 = 1) \wedge\ (i_1 < capacity) \wedge (i_1 < 512) \wedge (Data1[i_1] = 0) \wedge$
$\ldots$
$(i_{511} = 511) \wedge\ (i_{511} < capacity) \wedge (i_{511} < 512) \wedge (Data1[i_{511}] = 0) \wedge$
$capacity <= 512 \wedge ret = 1$

(b)

Fig. 5. (a) A conjunction of an unimpacted and impacted constraints along path i in V_1: $I_i^1 \bigwedge U_i^1$. (b) An impacted constraint along path i in V_1: I_i^1.

and V_1 in Figure 4 can be proven using impact summaries that do not contain constraints over `Data0`, `StructA`, or `st`.

The arrays `Data0` and `Data1`, the pointer to `StructA`, and the input variable `capacity` are defined as symbolic in this example. In Figure 5(a) we show a summary for the path in program V_1 shown in Figure 4(b) that contains both impacted and unimpacted constraints. There are 512 iterations of the `for` loop that are encoded in the path using the loop index i, and there are constraints over `StructA`, `st`, and `capacity` as well. In contrast, Figure 5(b) contains only the set of impacted constraints from the same path. From this example, we can see that discarding unimpacted constraints can dramatically reduce the size of the summaries used in regression verification.

4 Correctness Proofs

In this section we compare two program versions V_0 and V_1. We eventually show that the equivalence of their respective summaries, sum_0 and sum_1, can be implied by proving the equivalence of $isum_0$ and $isum_1$. Likewise, we show that if $isum_0$ and $isum_1$ are not equivalent, then sum_0 and sum_1 are not equivalent.

To simplify the presentation of our work, we discuss the correctness of the equivalence checking using the intraprocedural change impact analysis. The same correctness argument holds for an interprocedural analysis that dynamically tracks the flow of impact through method parameters and global variables. The change impact analysis described in Section 3 is conservative for sequential programs; it adds every statement that *may* be impacted by a change to the impact sets $\mathbb{I}_0$ and $\mathbb{I}_1$. We argue that the statements that are considered unimpacted by the analysis are not relevant to a proof of equivalence of the program versions.

Lemma 1. *Given closely related program versions V_0 and V_1, if a program statement is common to both versions, then it is either impacted in both versions or unimpacted in both versions.*

Proof. This follows from Steps 5 and 7 of the static impact analysis shown in Table 1 (Section 3). Step 5 assigns $\mathbb{I}_1$ to be equal to $\mathbb{I}_0$ after performing the data- and control-flow analysis on V_0 (except for statements that are removed in V_1 or added in V_0). Then Step 7 adds statements from $\mathbb{I}_1$ to $\mathbb{I}_0$ after performing the data-flow, control-flow, backward control-flow, and reaching definition analysis on V_1 (except for statements added to V_1 or removed from V_0). Therefore, the only statements that differ between $\mathbb{I}_0$ and $\mathbb{I}_1$ are those that have been added or removed.

Next we argue that for every path i in V_0, there exists a path j in V_1 such that i and j contain the same set of unimpacted statements and, similarly, for every path j in V_1, there exists a path i in V_0 such that i and j contain the same set of unimpacted statements.

Lemma 2. *Given closely related program versions V_0 and V_1, for every path $(I_i^0 \wedge U_i^0)$ there exists a path $(I_j^1 \wedge U_j^1)$ such that $U_i^0 \equiv U_j^1$. Likewise, for every path $(I_j^1 \wedge U_j^1)$ there exists a path $(I_i^0 \wedge U_i^0)$ such that $U_j^1 \equiv U_i^0$*

Proof. By contradiction. Assume there is some path containing a certain sequence of unimpacted instructions in one program version but not the other. This implies that the result of some conditional statement S_c differs between the two versions and that the set of unimpacted instructions is control dependent on S_c. Clearly the predicate in S_c uses the result of an impacted write statement or S_c is control dependent on another impacted conditional statement. According to Rules (1) – (4) in Table 1, S_c is impacted. Furthermore, because the unimpacted statements are control dependent on S_c, they are also impacted.

Corollary 1. *The set of* unique *unimpacted constraints in V_0 is the same as the set of* unique *unimpacted constraints in V_1. This implies Formula 2*

$$(\bigvee_{i=1}^{n} U_i^0) \leftrightarrow (\bigvee_{i=1}^{m} U_i^1) \tag{2}$$

As defined in Section 3, a program's symbolic summary consists of the disjunction of the constraints along each possible execution path in the program. Each path consists of a set of impacted and unimpacted constraints. In Theorem 1 we show that the unimpacted and impacted constraints can be effectively *de-coupled* from each other in a program's summary.

Theorem 1. *Given a program version V_0 with n paths, Formula 3 is valid.*

$$\bigvee_{i=1}^{n} (I_i^0 \wedge U_i^0) \leftrightarrow [(\bigvee_{i=1}^{n} I_i^0) \wedge (\bigvee_{i=1}^{n} U_i^0)] \tag{3}$$

Proof. See extended technical report for this proof [1].

In Theorem 2 we consider the overlap between the space of assignments to program variables that satisfy impacted constraints and the space of assignments to program variables that satisfy unimpacted constraints. Specifically, we claim that for some path in a program summary, if there is some concrete assignment to the program variables that satisfies the impacted constraints, then there is a concrete assignment to the remaining variables (those only present in the unimpacted constraints) that satisfies the unimpacted constraints.

Theorem 2. *Consider a program version V_0 with n paths and a closely related program version V_1 with m paths. Let $u_1, u_2, \ldots u_k$ be program variables present in the unimpacted statements of V_0 (V_1). Let AU be the set of possible concrete assignments to these variables. Let AI_0 (AI_1) be the set of possible concrete assignments to all other variables in V_0 (V_1). For any assignment $x \in AI_0$ ($x \in AI_1$) that satisfies the impacted constraints, there exists an assignment $y \in AU$ that satisfies the unimpacted constraints. Formally, Formulas 4 and 5 are valid.*

$$\forall_{x \in AI_0} \exists_{y \in AU} (I_i^0[x] \rightarrow U_i^0[y]) \tag{4}$$

$$\forall_{x \in AI_1} \exists_{y \in AU} (I_i^1[x] \rightarrow U_i^1[y]) \tag{5}$$

Proof. Rule (4) in Table 1 dictates that the statements defining the value of every variable used in an impacted statement are also impacted. Accordingly, the variables that are common to the impacted and unimpacted statements are *not constrained* by the unimpacted statements. I.e., the result of an unimpacted statement cannot affect the result of an impacted statement. Therefore, if it is possible to satisfy the constraints of I_i^0 (I_j^1), then it is possible to satisfy the constraints of U_i^0 (U_j^1).

Now we show that the impact summaries for two programs versions V_0 and V_1 are equivalent if and only if the summaries for V_0 and V_1 are equivalent. We use the result of Theorem 1 to prove the forward direction (if the impact summaries are equivalent, then the summaries are equivalent). Then we use the result of Theorem 2 to prove the reverse direction (if the summaries are equivalent, then the impact summaries are equivalent).

Theorem 3. *Given program version V_0 with n paths and a closely related program version V_1 with m paths. $isum_0$ and $isum_1$ are equivalent if and only if sum_0 and sum_1 are equivalent. This is formally stated in Formula 1 and is also shown below.*

$$[(\bigvee_{i=1}^{n} I_i^0) \leftrightarrow (\bigvee_{i=1}^{m} I_i^1)] \leftrightarrow [\bigvee_{i=1}^{n} (I_i^0 \wedge U_i^0) \leftrightarrow \bigvee_{i=1}^{m} (I_i^1 \wedge U_i^1)]$$

Proof. ($\Rightarrow$)We begin by assuming Formula 6 is valid

$$(\bigvee_{i=1}^{n} I_i^0) \leftrightarrow (\bigvee_{i=1}^{m} I_i^1) \tag{6}$$

Conjoining the term representing the disjunction of unimpacted constraints of V_0 to the left and right side of Formula 6 yields Formula 7.

$$(\bigvee_{i=1}^{n} I_i^0) \wedge (\bigvee_{i=1}^{n} U_i^0) \leftrightarrow (\bigvee_{i=1}^{m} I_i^1) \wedge (\bigvee_{i=1}^{n} U_i^0) \tag{7}$$

Applying Formula 2 yields Formula 8.

$$(\bigvee_{i=1}^{n} I_i^0) \wedge (\bigvee_{i=1}^{n} U_i^0) \leftrightarrow (\bigvee_{i=1}^{m} I_i^1) \wedge (\bigvee_{i=1}^{m} U_i^1) \tag{8}$$

Applying Formula 3 yields Formula 9.

$$\bigvee_{i=1}^{n} (I_i^0 \wedge U_i^0) \leftrightarrow \bigvee_{i=1}^{m} (I_i^1 \wedge U_i^1) \tag{9}$$

This proves the forward direction, i.e., $(isum_0 \leftrightarrow isum_1) \rightarrow (sum_0 \leftrightarrow sum_1)$. The latter half of the proof, $(sum_0 \leftrightarrow sum_1) \rightarrow (isum_0 \leftrightarrow isum_1)$, is more complex than the first half and is available in the technical report [1].

5 Evaluation

To empirically evaluate the regression verification technique described in this work, we implemented a DiSE framework, Proteus, for analyzing C programs. Note that the earlier DiSE framework implementation was an extension of the Java PathFinder, [21], toolkit to analyze Java programs [13, 17]. A large number of safety critical systems are developed in C; Proteus was developed at NASA to assist in the analysis of these systems.

In Proteus, we use the GNU DiffUtils[5] to compute the initial change set containing the actual source level differences between program versions. The static analysis is implemented as a customized LLVM optimization pass [11]. The output of the static analysis is the set of impacted program statements. The partitioning of constraints during symbolic execution is implemented as an extension to the KLEE symbolic execution engine [4]. As an optimization for discarding unimpacted constraints, we employ the directed search in the DiSE algorithm to prune execution of paths that *differ only* in unimpacted constraints [13, 17]. The final post-processing of the symbolic summaries is performed using a custom application that iterates over the impacted path conditions, removing constraints that are not impacted by the differences. We use the Z3 constraint solver to check for logical equivalence of impact summaries [6].

[5] http://www.gnu.org/software/diffutils

Example	Versions	Equiv	Paths		Constraints			Time Symbc (s)		Time Solver (s)		
			Full	iDiSE	Full	iDiSE	iSum	Full	iDiSE	Full	iDiSE	iSum
Init_Data	V0V1	yes	400	400	103400	103400	82800	51.87	50.67	1.94	1.94	0.76
tcas1	V0V1	yes	118	12	4748	524	332	1.62	0.60	0.09	0.04	0.04
	V1V2	yes	118	118	4772	4772	3956	1.64	1.92	0.09	0.09	0.06
	V2V3	yes	118	118	4796	4796	2908	1.62	1.91	0.08	0.08	0.05
tcas2	V0V1	no	150	12	6052	520	328	2.21	0.63	0.12	0.06	0.05
replace1	V0V1	yes	18	8	98	68	48	0.31	0.25	0.01	0.03	0.03
	V1V2	yes	18	10	98	98	78	0.31	0.32	0.01	0.04	0.04
	V2V3	no	18	2	98	8	4	0.31	0.18	0.01	0.03	0.03
replace2	V1V2	yes	604	604	23736	23736	20980	1.14	1.35	0.11	0.11	0.10
wbs1	V0V1	yes	336	190	13416	11478	9158	1.18	1.63	0.10	0.10	0.08
	V1V2	yes	336	336	13416	13416	10784	1.25	1.42	0.10	0.10	0.09
	V2V3	yes	336	190	13416	11478	10784	1.19	1.34	0.11	0.09	0.08
wbs2	V0V1	no	336	134	13388	5601	4551	1.18	0.83	0.11	0.06	0.06
cornell1	V0V1	yes	10	8	62	48	24	0.10	0.11	0.03	0.03	0.03
cornell2	V0V1	yes	18	10	1864	810	663	0.27	0.29	0.01	0.01	0.01
kernel1	V0V1	yes	-	4	-	282	226	-	21.09	-	218	200
	V1V2	yes	-	4	-	282	226	-	21.32	-	211	208
kernel2	V0V1	yes	4	2	130	114	88	1.56	1.92	0.20	0.13	0.04
kernel3	V0V1	no	4	2	118	58	48	0.67	0.78	0.19	0.12	0.12

Table 2. Equivalence Checking Results

We present the results for the different versions of the six artifacts in Table 2. The details of the artifacts and their versions are described in further detail in the technical report [1]. The experiments are run on a 64-bit Linux machine, with a 2.4GHz processor, and 64GB memory. The `Example` column lists the name of the artifact and the `Versions` column lists the version numbers of the artifacts compared. The `Equiv` column shows whether the versions are equivalent or not. The results contain data from three different configurations: (1) `Full` symbolic execution explores all paths, (2) `iDiSE` prunes paths that only differ in unimpacted constraints (iDiSE refers to the interprcoedural extension of the DiSE framework as defined in [17]), and (3) `iSum` represents the final impact summaries. The `Paths` column lists the number of paths, the `Constraints` column presents the number of constraints in the summaries, and `Time Symbc` column lists the time in seconds. The time reported for iDiSE includes the time to perform the static analysis and incremental symbolic execution. Finally, the `Time Solver` column lists the time taken by Z3 to solve the equivalence queries generated by full symbolic execution, iDiSE, and iSum. The rows marked with '-' indicate that the analysis does not finish within the time bound of one hour.

Overall, the results in Table 2 indicate that reducing the size of the queries reduces the time to check equivalence. In the `tcas2` example, full symbolic execution generates 150 paths while iDiSE only generates 12 paths and we can see corresponding reductions in the number of constraints and time taken to check equivalence. The iDiSE overhead for the set of artifacts is quite small,

Example	Modular (s)	Non-modular (s)	Example	Modular (s)	Non-modular (s)
tcas1V0V1	12.9	17.4	tcas1V2V3	13.6	15
tcas1V1V2	13.6	15	tcas2V0V1	14.3	18.2
wbs1V0V1	13.8	13.8	wbs1V2V3	13.7	14.1
wbs1V0V2	13.8	13.8	wbs2V0V1	14.6	14.4
replace2 V1V2	31.9	29:53.2			

Table 3. Evaulation of artifacts using SymDiff

and the total analysis time (Symbc + Solver) can be considerably less for iDiSE combined with constraint pruning over full symbolic execution. In the two versions of the `kernel1` example, full symbolic execution is unable to complete the analysis within the time bound of one hour, while only four paths are generated by iDiSE. There is a loop in `kernel1` that does not contain any impacted statements; iDiSE is able to ignore paths through the loop and quickly generate the impact summaries. For this example, we can see how leveraging program similarities can dramatically improve the performance of regression verification. Although the time taken for equivalence checking for the other examples is relatively small – just a few seconds – the artifacts themselves are relatively small. We believe that the reductions will be applicable to larger examples as well. For the `replace` example, the solver time for the summaries without pruning is much faster than those with pruning. The tool we used to translate the CVC formula generated by KLEE into SMTLIB format (to be interpreted by Z3) parsed the CVC query into a trivial SMTLIB query for these examples. It is unclear to us why this occurred with the full summaries but not the impact summaries.

Limitations The regression verification technique presented in this work currently supports checking equivalence between two sequential programs without exceptional flow. The equivalence checking reports generated by Proteus are sound and complete for programs that do not have runtime errors or make calls to unsupported libraries. For examples that have runtime errors or make calls to unsupported libraries, the tool reports warnings and continues execution; the equivalence result are reported as inconclusive in the presence of such warnings. The sound and complete reasoning about the equivalence is with respect to a loop bound. It is possible to leverage automatic loop invariant generation and loop summarization techniques in the context of symbolic execution to reason about equivalent programs without a depth bound.

6 Discussion

Revisiting Table 2, the data in the `Full` columns can be considered representative of results in UC-KLEE [15]. The results demonstrate that UC-KLEE can benefit from using our reduction techniques, when analyzing related program versions.

In order to evaluate how other tools perform equivalence checking, we ran SymDiff [10]. We set up the experiments for SymDiff and ran them on a Windows 7 machine with a 1.8 GHz processor and 6 GB of RAM. We experimented with

two SymDiff configurations (a) modular, where the methods are summarized as uninterpreted functions, and (b) non-modular, where the invocations to the different methods are inlined. The non-modular approach is sound and complete with respect to a depth-bound as well. The `kernel` and the `cornell` examples contain constructs that are not currently supported by the C front-end in the current version of SymDiff, so we report on experiments for the rest of the examples. Table 3 shows the total wall clock time in seconds. In the modular approach, SymDiff does not report any false negatives for the examples shown in Table 3. We used a loop bound of four for the `replace` example, the same as the one used in Table 2. We also used the flag in Symdiff to analyze only callers and callees that are reachable from the changed methods to ensure that the set of methods analyzed by SymDiff and Proteus is the same. SymDiff runs on a Windows platform while Proteus runs on a Unix-based platform; we had to run the experiments on different machines and it is not possible to make empirical comparative claims between the two in terms of time. Furthermore, SymDiff and Proteus encode the program behaviors differently, therefore, it is not possible to compare the approaches in terms of the size of the generated formulas. SymDiff does not use any slicing techniques based on change impact analysis, and we believe that it can be beneficial to add such a reduction technique to SymDiff.

Abstract Syntax Tree To calculate the precise initial change sets we can use standard algorithms to match Abstract Syntax Trees (ASTs), [14], and discard differences due to variable renaming and simple re-ordering before we perform the data and control flow analysis. The syntatic differences based on the ASTs are more precise compared to those generated by the GNU DiffUtils. We have support for the AST based syntatic diff in the Java implementation of the DiSE framework and we are currently working on adding it to Proteus.

Static Encoding vs. Bounded Unrolled Program Encoding The correctness of Eq. (1) does not rely on any specific encoding of constraints. We choose, however, to encode the program behaviors generated by symbolic execution (bounded unrolled programs) as constraints rather than use a static encoding for the constraints because (a) the static constraints on heap and array operations are often harder to solve than those generated by symbolic execution and (b) scalable static slicing techniques for interprocedural programs often ignore calling context and are imprecise; we leverage work in [17] to dynamically compute impact information for interprocedural programs.

7 Related Work

Several techniques have been developed for checking equivalence. Differential Symbolic Execution (DSE) uses uninterpreted functions to represent unchanged blocks of code [12]. SymDiff [10] summarizes methods as uninterpreted functions, and uses verification conditions to summarize observable behavioral differences. Regression verification techniques by Strichman et al. [8, 19] use the Context-Bounded Model Checker (CBMC) to check equivalence of closely related C programs. It establishes partial equivalence of functions using a bottom-up de-

composition algorithm. Another approach [18] performs an increment upgrade checking in a bottom-up manner similar to regression verification, using function summaries computed by means of Craig interpolation. These techniques are sound but not complete. Techniques from [18] are used in the PINCETTE project [5]. To curb over approximations, the PINCETTE project also employs dynamic techniques (e.g., concolic testing) to generate regression tests for system upgrades. There is also ongoing work to support program slicing based on the program differences in CBMC.

Similar to our work, UC-KLEE [15] is built on top of KLEE. UC-KLEE is designed to run two functions under test with the same input values and check if they produce the same outputs. As an optimization, UC-KLEE is able to skip unchanged instructions. However, it neither produces nor leverages the impacted behavior information. Partition-based regression verification, [3], computes partitions on-the-fly using concolic execution and dynamic slicing techniques. Each partition contains behaviors generated from a subset of the input space common to two program versions. The goal of the technique is to find test cases that depict semantic differences rather than prove equivalence.

Approaches that cache or reuse constraints to speed up performance (e.g., Green [20]) are orthogonal to our reduction technique. Such techniques are complementary to this work and can be leveraged to achieve higher reduction factors.

8 Conclusions & Future Work

In this work on regression verification we leverage control- and data-flow information to partition the program behavior summaries as either impacted or unimpacted based on the differences between two program versions. We then prove that the impacted constraints of two closely related programs are functionally equivalent iff their entire program behavior summarizations are functionally equivalent. An empirical evaluation on a set of sequential C artifacts shows that reducing the size of the summaries helps reduce the cost of equivalence checking.

In future work, we plan to study the effects of other more compact program summarization encoding schemes such as large-block encoding [2] in combination with the work proposed here. Another avenue of future work is to develop an abstraction-refinement technique using uninterpreted functions to abstract large parts of the program as done in [12, 19], but, use the information about the impacted parts of the code to refine the abstraction when required. We believe such techniques can further improve checking equivalence of large programs.

Acknowlegements We thank Shuvendu Lahiri at Microsoft Research for his help with SymDiff.

References

1. J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries Extended version available online. http://ti.arc.nasa.gov/profile/nrungta/pubs/, 2013.

2. D. Beyer, A. Cimatti, A. Griggio, M. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, Nov. 2009.
3. M. Boehme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *ICSE*, 2013.
4. 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.
5. H. Chockler, G. Denaro, M. Ling, G. Fedyukovich, A. E. J. Hyvrinen, L. Mariani, A. Muhammad, M. Oriol, A. Rajan, O. Sery, N. Sharygina, and M. Tautschnig. Pincette validating changes and upgrades in networked software. In *CSMR*, 2013.
6. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
7. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.
8. B. Godlin and O. Strichman. Regression verification. In *DAC*, 2009.
9. D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2008.
10. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: a language-agnostic semantic diff tool for imperative programs. In *CAV*, pages 712–717, 2012.
11. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.
12. S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.
13. S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.
14. S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *ICSM*, pages 188–197, 2004.
15. D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, pages 669–685, 2011.
16. G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, pages 173–210, 1997.
17. N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *ICSM*, 2012.
18. O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *FMCAD*, UK, 2012.
19. O. Strichman and B. Godlin. *Regression Verification - A Practical Way to Verify Programs*. Springer-Verlag, Berlin, Heidelberg, 2008.
20. W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *SIGSOFT FSE*, page 58, 2012.
21. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *ASE*, 10(2):203–232, 2003.
22. G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *ICSM*, pages 115–124, 2009.