Differential assertion checking

S. K. Lahiri
Microsoft Research
Redmond, WA, USA

K. L. McMillan
Microsoft Research
Redmond, WA, USA

R. Sharma
Stanford University
CA, USA

C. Hawblitzel
Microsoft Research
Redmond, WA, USA

ABSTRACT

Previous versions of a program can be a powerful enabler for program analysis by defining new relative specifications and making the results of current program analysis more relevant. In this paper, we describe the approach of *differential assertion checking* (DAC) for comparing versions of a program with respect to a set of assertions. DAC provides a natural way to write relative specifications over two programs. We introduce a novel modular approach to DAC by reducing it to single program checking that can be accomplished by any program verifier. In particular, we leverage automatic invariant generation to synthesize relative specifications for pairs of loops and procedures. We provide a preliminary evaluation of a prototype implementation within the SymDiff tool along two directions (a) soundly verifying bug fixes in the presence of loops and (b) providing a knob for suppressing alarms when checking a new version of a program.

1. INTRODUCTION

There are several factors limiting the adoption of static analysis tools in the hands of developers. For static assertion checking, these include the need to define an assertion (or specification) to check, to provide environment specifications and to provide auxiliary invariants for loops and procedures. Although many auxiliary invariants can be synthesized automatically by invariant generation methods, the undecidable nature (or the high practical complexity) of assertion checking precludes complete automation for a general class of user-supplied assertions.

It has often been proposed that utilizing previous versions of an evolving program can significantly reduce the cost of program analysis [21]. Such approaches run in two primary directions. First, in the presence of program refactoring, two versions can be checked for semantic equivalence to ensure the correctness of the transformation [22, 9, 16]. Second, verification can be performed *incrementally*, for example by carrying over invariants that are unaffected by the syntactic changes [26]. Although these techniques are useful in their own right, they are applicable in very limited contexts. First, most software changes (including some called refactoring) induce some behavioral change. Equivalence checking is too strong for such cases. Moreover, incremental verification can only be performed when the previous version does not have any false warnings — unfortunately, this is too strong a requirement for the usage of static analysis tools. Such false warnings have to be either removed by manually specifying additional invariants, or the tool has to resort to ad-hoc heuristics to suppress a class of warnings. The former seriously undermines the productivity gained from the use of static analysis, whereas the latter leads to brittle tools that may suppress true bugs.

In this paper, we propose another direction for exploiting previous versions of a program as an implicit specification, which appears to opens up an interesting space for trading off soundness for cost required to apply an assertion checker. Our observation is simple:

> We can often prove *relative correctness* between two similar programs with respect to a set of assertions statically with significantly lower cost than ensuring absolute correctness.

Given a program P with a set of assertions A, one traditionally asks whether there is an environment for P in which one of the assertions in A fails. One can instead ask a relative version of this question: given two versions P and P' containing a set of assertions A: does there exist an environment in which P passes but P' fails? We formalize this idea as the problem of *differential assertion checking* (DAC): checking two versions of a program with respect to a set of assertions. Although this provides a weaker guarantee of correctness of P', it closely corresponds to an interesting class of bugs (*regressions*) that are often most relevant to a developer and have a good chance of getting fixed. Moreover, we argue that DAC has several desirable traits, when checking absolute correctness is rife with false alarms:

1. DAC allows for writing natural relative specifications without a lot of modeling (additional ghost variables) to express the properties.

2. DAC can be used to show that bug fixes do not cause additional regressions for a set of assertions.

3. Exploiting the structural similarity of programs P and P' allows simple relative specifications to answer the relative questions.

An idea similar to DAC was earlier proposed in the context of filtering false alarms for concurrent programs [14] (we discuss subtle differences in Section 3). At a high level, one can see this work as applying the idea towards evolving programs and extending the idea to deal with unbounded loops and recursion ([14] was restricted to bounded programs).

1.1 Motivating example

```
void StringCopy.1(
      wchar_t *dst,
      wchar_t *src,
      int size)
{
      wchar_t *dtmp = dst,
              *stmp = src;
      int i;
      for (i = 0;
           *stmp &&
            i < size - 1;
           i++)
         *dtmp++ = *stmp++;
      *dtmp = 0;
}
```

```
void StringCopy.2(
      wchar_t *dst,
      wchar_t *src,
      int size)
{
      wchar_t *dtmp = dst,
              *stmp = src;
      int i;
      for (i = 0;
           i < size - 1 &&
           *stmp;
           i++)
         *dtmp++ = *stmp++;
      *dtmp = 0;
}
```

Figure 1: Motivating example (in C): two versions of StringCopy (Figure 1 [12]).

Consider the two versions of the procedure `StringCopy` described in Figure 1. The version `StringCopy.2` is a procedure for copying the contents of a `char` buffer `src` into `dst`, described in an earlier work [12]. Let us first ignore `StringCopy.1`, which is a buggy version of `StringCopy.2`. In this paper, we adopt the convention that procedures on the left side of the figures corrspond to buggy versions and those on the right correspond to correct versions. Let us illustrate the complexities of verifying the memory safety of `StringCopy.2` in isolation.

1. To specify memory safety, one needs to define the *bounds* of a buffer for C programs (unlike Java or C#). This can be accomplished by adding a ghost variable `Bound` that maps each allocated pointer (such as `dst`) to a non-negative integer. One possible way to specify the memory safety is to precede any dereference `*e` with the assertion assert $\texttt{Bound(e)} > 0$.
2. One needs a *precondition* that the bounds of `dst` and `src` have some relationship with `size`, and the two buffers are disjoint.
3. Finally, one needs to write a loop invariant to record that `dtmp` always points inside the buffer pointed by `dst`, among other things.

Even for such a simple procedure, specifying and verifying the memory safety can be quite complex if the user is left to define the assertions, environment conditions and intermediate invariants.

Now we define relative memory safety of `StringCopy.2` with respect to `StringCopy.1`. First, observe that the difference in the two versions lies in the loop exit condition where the conjunction (`&&`) is applied in reverse order — this gives different behaviors due to the short-circuit semantics of `&&`. We want to check that `StringCopy.2` accesses only the memory locations which `StringCopy.1` accesses for any input. We can define and check relative memory safety in a generic fashion as follows:

1. Define an uninterpreted predicate *Valid* that maps each pointer to a Boolean value. Each dereference `*e` is preceded by assert $\mathit{Valid}(e)$.
2. Let `ok.i` be a global Boolean variable for `StringCopy.i` procedure that is true if no assertion has failed. We replace assert ϕ by code that sets `ok.i` to false if ϕ is false. We say `StringCopy.2` is correct $r$elative to `StringCopy.1` if, when both start in the same state (parameters and the heap) and both terminate, if the former terminates in a state where `ok.1` is true, then the latter also terminates in a state satisfying `ok.2`.
3. Assuming the two loops are automatically extracted as tail-recursive procedures (§ A) `loop.1` and `loop.2` respectively, we show how to construct a composed procedure for the two loops and attach a simple relative specification on the composed procedure.

```
pre    stmp.1 == stmp.2 &&
       dtmp.1 == dtmp.2 &&
       Mem_char.1 == Mem_char.2 &&
       i.1 == i.2 &&
       size.1 == size.2 &&
       ok.1 <==> ok.2

post   ok.1 ==> ok.2 &&
       dtmp.1 == dtmp.2
proc MS_loop.1_loop.2(dst.1, ..., dst.2, ...);
```

Here pre refers to a precondition and post refers to a postcondition, and `Mem_char.i` refers a global array that models the state of the heap. Moreover, we show how such a specification can be inferred using the techniques in this paper.

Note that we did not require any precondition about the inputs to the program, nor any correlation about the bounds nor any relationship with null-terminated buffers. This checking succeeds and we have proven that `StringCopy.2` has a memory footprint no larger than `StringCopy.1`. On the other hand, if one were to check the relative correctness of `StringCopy.1` with respect to `StringCopy.2` under the relative memory safety specification, one would get a counterexample where `size` equals 0 and pointer `src` does not satisfy `Valid`. This counter-example captures the seeded bug: an address that `StringCopy.1` dereferences but `StringCopy.2` does not.

1.2 Overview

In the rest of the paper, using the background developed in § 2, we formalize the notion of differential assertion checking (DAC) (§ 3), and illustrate its use for defining relative specifications (§ 3.1). We provide an algorithm for checking DAC modularly by transforming the relative correctness problem into verifying assertions over a single composed program (§ 4). This allows us to leverage any off-the-shelf program verifier and invariant generation tools to check the relative correctness problem. We demonstrate a simple scheme

$$
\begin{array}{lllll}
\mathsf{x} & \in & \mathit{Vars} & & \\
R & \in & \mathit{Relations} & & \\
U & \in & \mathit{Functions} & & \\
e & \in & \mathit{Expr} & ::= & \mathsf{x} \mid c \mid U(e,\ldots,e) \mid \mathsf{old}(e) \\
\phi & \in & \mathit{Formula} & ::= & \mathsf{true} \mid \mathsf{false} \mid e \;\mathsf{relop}\; e \mid \phi \wedge \phi \mid \\
 & & & & \neg\phi \mid R(e,\ldots,e) \mid \ldots \\
s & \in & \mathit{Stmt} & ::= & \mathsf{skip} \mid \mathsf{assert}\;\phi \mid \mathsf{assume}\;\phi \mid \mathsf{x} := e \mid \\
 & & & & \mathsf{havoc}\;\mathsf{x} \mid s;s \mid \mathsf{x} := \mathsf{call}\; f(e,\ldots,e) \\
c & \in & \mathit{CFStmt} & ::= & L: \mid \mathsf{goto}\; L_1,\ldots,L_n \mid \mathsf{return} \\
f & \in & \mathit{Body} & ::= & c \mid s;f \mid f;f \\
p & \in & \mathit{Proc} & ::= & \mathsf{int}\; f(\mathsf{x}_f : \mathsf{int},\ldots) : \mathsf{r}_f \;\{\; f_{body} \;\}
\end{array}
$$

Figure 2: A simple programming language. The set of goto statements do not form any cycles in the control flow graph.

based on HOUDINI [8] that suffices for a class of programs (§ 5.1). We have created a prototype implementation of our method inside SYMDIFF [16], a semantic differencing tool. We evaluate the tool along two different directions. First, we use DAC to soundly verify that the version after a bug fix is relatively correct with respect to the buggy version (§ 6.1). Second, we show that DAC can provide a systematic knob for suppressing alarms when analyzing a new version of a program (§ 6.2). Together, the experiments indicate the potential of DAC to be a generic framework to exploit previous versions of a program.

2. BACKGROUND

Figure 2 describes a simple programming language (a subset of the BOOGIE [2] programming language) with recursive procedures and an assertion language. We assume that loops are already desugared into this language (we describe a method in § A). The language supports variables (*Vars*) and various operations on them. Expressions (*Expr*) can be variables, constants, or the result of applying a (possibly interpreted) function U to a list of expressions. The expression $\mathsf{old}(e)$ refers to the value of e at the entry to a procedure. *Formula* represents Boolean valued expressions and can be the result of (interpreted or uninterpreted) relational operations on *Expr*, Boolean operations ($\{\wedge, \neg\}$), or possibly quantified expressions ($\forall u : \mathsf{int}.\phi$). Note that the programming language is fairly expressive and can be used to model arrays. An array can be modeled in this language, by introducing two special functions $\mathit{sel} \in \mathit{Functions}$ and $\mathit{upd} \in \mathit{Functions}$; $\mathit{sel}(e_1, e_2)$ selects the value of a map value e_1 at index e_2, and $\mathit{upd}(e_1, e_2, e_3)$ returns a new map value by updating a map value e_1 at location e_2 with value e_3.

A state of a program at a given program location is a valuation of the variables in scope (procedure parameters, locals and global variables) and a program counter pc that indicates the next statement to be executed. A program consists of a set of *basic blocks*, where each basic block consists of a statement $s \in \mathit{Stmt}$ terminated with a control flow statement *CFStmt* (goto or return statement). A goto statement goto $L_1, \ldots, L_n$ non-deterministically sets the pc to any one of the n labels. We restrict the use of goto statements to not form any cycles in the control flow graph. The statement skip denotes a no-op. The statement assert ϕ is used to statically check that the formula ϕ holds; assert ϕ has no effect on the dynamic state. The statement assume ϕ behaves as a skip when the formula ϕ evaluates to true in the current state; else the execution of the program is *blocked*. The assignment statement is standard, havoc x scrambles the value of a variable x to an arbitrary value, and $s; t$ denotes the sequential composition of two statements s and t. Conditional statements are modeled by using the goto statement and assume statements. Procedure calls are denoted using the call statement, and can have a side effect by modifying one of the global variables.

Let Σ be the set of all states for a program. For any procedure $p \in \mathit{Proc}$, we assume a transition relation $\mathcal{T}_p \subseteq \Sigma \times \Sigma$ that characterizes the input-output relation of the procedure p. In other words, two states $(\sigma, \sigma') \in \mathcal{T}_p$ if there is an execution of the procedure p starting at σ and ending in σ'. The transition relations can be defined inductively on the structure of the program and is fairly standard for our simple language [2].

There are a host of tools for modeling most high level languages (such as C, C#, Java) in this language (such as SPEC# [4], HAVOC [5]). We only note that such translations use the arrays to model the heap (e.g. an array per field in Java) where the arrays are indexed by objects or pointers. We defer further discussion of the translations to these earlier works.

3. DIFFERENTIAL ASSERTION CHECKING

In this section, we formalize our approach of differential assertion checking (DAC). The basic concept of DAC appears in a previous work in the context of filtering false alarms in verification of concurrent programs using sequential executions [14]. However, it was described in a simpler setting where loops were unrolled and procedures were inlined a bounded number of times.

Before proceeding, we establish a few notations that we follow in the paper unless explicitly stated otherwise. First, we assume that any assertion assert ϕ is replaced by the assignment $\mathsf{ok} := \mathsf{ok} \wedge \phi$ to a global ok variable. Second, given that we are considering two versions P_1 and P_2 of a program, we suffix the names of procedures, globals (including ok) and parameters with the version number. Third, we label a state σ as *failing* if ok variable is false in σ. Finally, we assume a one-one (not necessarily onto) mapping between the globals, procedures, and their parameters between the two versions; we often equate states from two versions when we really mean that the two states assign the same value to the mapped variables of the two states.

Definition 1. (*Differential assertion checking*) Given two procedures p_1 and p_2, p_2 has a differential error with respect to p_1 (denoted as $DAC(p_2, p_1)$) if there exists an input state σ such that (1) there exists a state σ'_1 such that $(\sigma, \sigma'_1) \in \mathcal{T}_{p_1}$ and σ'_1 is non-failing, and (2) there exists a state σ'_2 such that $(\sigma, \sigma'_2) \in \mathcal{T}_{p_2}$ and σ'_2 is failing.

We define a procedure p_2 to be *relatively correct with respect to* p_1 if $DAC(p_2, p_1)$ does not hold.

The above definition differs from the definition of *differential error* ($\mathit{DiffErr}(p_2, p_1)$) [14] in a subtle way. The difference lies in whether we insist the input σ to be non-failing for *every* execution in p_1 (in $\mathit{DiffErr}(p_2, p_1)$) as opposed to be failing on some execution in p_1 (in $DAC(p_2, p_1)$). We provide a simple example that distinguishes the two views. For this example, $DAC(\mathtt{p2}, \mathtt{p1})$ holds as there is a state (empty)

```
proc p1() {
    havoc x;
    if (x) assert false;
}
```

```
proc p2() {
    assert false;
}
```

from which p1 succeeds (when the internal variable x is assigned false) and p2 fails. However, $\mathit{DiffErr}(\mathtt{p2}, \mathtt{p1})$ does not hold because there is no input state from which all executions are non-failing for p1. It is easy to observe that if $\mathit{DiffErr}(p_2, p_1)$ holds then $DAC(p_2, p_1)$ holds, but not otherwise.

The definition of *DiffErr* was motivated by comparing concurrent interleaved executions with their sequential counterparts. We adopt the slightly modified definition for DAC to several reasons. First, the check for $DAC(p_2, p_1)$ can be encoded very naturally using single program verifiers:

```
assume i1 == i2 && g1 == g2;
call p1(i1); call p2(i2);
assert (ok.1 ==> ok.2);
```

where we use i and g to denote parameters and globals. On the other hand, the *DiffErr* check is more complicated because checking it is undecidable even for bounded programs without quantifiers. This added complexity is not needed for comparing similar versions of a program; we have found that internal non-deterministic choices are less common. Whenever non-determinism is present (say reading chars using scanf), the choices can be aligned on the two sides to return the same arbitrary sequence of choices in the two programs (see [16]). In such a modeling, the non-deterministic choices become reads from an input array, thereby making the array part of the input of the two programs.

3.1 Relative specifications

Recall that writing meaningful specifications often require access to a host of ghost state that is not present explicitly as part of the program state (§ 1.1). In addition to checking existing assertions in the two versions differentially, DAC also facilitates writing relative specifications using the same syntax of single program assertions. Instead of defining the buffer overrun checks on the two programs and checking them differentially, it often helps to pose questions such as: are there inputs for which P_2 accesses buffer regions that are not accessed by P_1? Such specifications can be written by introducing an uninterpreted predicate *Valid* and adding an assertion before accessing any pointer p: assert $\mathit{Valid}(p)$. Such a specification will be useless for checking a single program (every pointer dereference might be flagged as a warning), but will naturally provide a relative specification. Moreover, such a specification can be strengthened using semantics of the particular property that is desired. For example, when checking for non-null pointer dereferences, one can constrain the predicate by adding an axiom:

$$\mathsf{axiom}(\forall x : \mathsf{int} :: x \neq 0 \Rightarrow \mathit{Valid}(p))$$

Similarly, while checking for buffer overflows, one can add an axiom:

$$\mathsf{axiom}(\forall x : \mathsf{int}, y : \mathsf{int} :: x \leq y \Rightarrow \mathit{Valid}(y) \Rightarrow \mathit{Valid}(x))$$

This will allow the DAC to not show a warning when the program P_2 accesses an index that is smaller than an index accessed by P_1. This is specially useful when the entire history of indices accessed by P_1 is not stored (especially while doing a modular proof of DAC (§ 4) that only records an abstraction of the history of accesses on the two programs). Finally, one can even capture properties such as equivalence of two procedures (modulo termination). For a procedure $p \in P$, let o be the set of out parameters and g be the set of globals modified by p. If we assert $\mathit{ValidEQ}(o, g)$ (for an uninterpreted predicate *ValidEQ*) on the post-state of p and then perform DAC on two versions p_1 and p_2, then the relative specification is correct if and only if the two programs are equivalent.

4. MODULAR DAC

In the previous section, we defined the problem of $DAC(p_2, p_1)$ for a pair of procedures p_1 and p_2. In this section, we provide a mechanism to check for $DAC(p_2, p_1)$ (or rather verify that p_2 is relatively correct with respect to p_1) in a procedure modular manner. In other words, we will verify the relative correctness without inlining the callers inside a procedure, but rather using some specifications. We provide a program transformation technique that compiles the relative correctness check of two programs P_1 and P_2 into a single composed program, which can be analyzed by an off-the-shelf program verifier. In particular, the transformation allows us to leverage existing invariant inference mechanisms for single programs for inferring relative specifications. The transformations is not specific to the problem of differential assertion checking, and is applicable whenever there is a need to compare two programs.

4.1 Composed program

```
proc f1(x1): r1
modifies g1
{
  s1;
L1:
  w1 := call h1(e1);
  t1
}
```

```
proc f2(x2): r2
modifies g2
{
  s2;
L2:
  w2 := call h2(e2);
  t2
}
```

Given two programs P_1 and P_2 each containing a set of procedures, and a one-one mapping between procedures, let us consider two particular mapped procedures $\mathtt{f1} \in P_1$ and $\mathtt{f2} \in P_2$. We have specified the modified set of globals for each procedure using modifies keyword. For ease of exposition, we have assumed that the read set of a procedure is a superset of the set of modified variables.

Figure 3 describes a composed procedure MS_f1_f2 that is constructed for each pair of mapped procedures. First, note that the signature (parameters, modifies sets) of the procedure is a disjoint union of the signatures of the individual procedures. The body of MS_f1_f2 consists of sequential composition of the bodies of f1 and f2, in addition to some extra instrumentation. Since loops are already extracted as tail-recursive procedures, the body of any procedure contains no loops.

The instrumentations consist of two parts. The first part consists of storing the input and the output state at each call site. The second part consists of constraining the outputs of pairs of call sites (from different programs) to be the result

```
proc MS_f1_f2(x1,x2) returns (r1,r2)
modifies g1, g2
{
  // initialize call witness variables
  b_l1, b_l2, ... := false, false, ...;

  [[s1;]]
L1:
  i_l1, gi_l1 := e1, g1; //store inputs
  call w1 := h1(e1);
  b_l1 := true; //set call witness
  o_l1, go_l1 := w1, g1; //store outputs

  [[t1;]]

  [[s2;]]
L2:
  i_l2, gi_l2 := e2, g2; //store inputs
  call w2 := h2(e2);
  b_l2 := true; //set call witness
  o_l2, go_l2 := w2, g2; //store outputs

  [[t2;]]

  //one block for each pair of call sites
  //for a pair of mapped procedures
  ....
  if (b_l1 && b_l2) { //for (L1,L2) pair
    //store the globals
    st_g1, st_g2 := g1, g2;

    g1, g2 := gi_l1, gi_l2;
    call k1, k2 := MS_h1_h2(i_l1, i_l2);
    assume (k1 == o_l1 && g1 == go_l1);
    assume (k2 == o_l2 && g2 == go_l2);

    //restore globals
    g1, g2 := st_g1, st_g2;
  }
  ...

  return;
}
```

Figure 3: Composed procedure for `f1` and `f2`.

of executing the corresponding composed procedure over the input states at the two call sites. This allows us to infer facts about pairs of procedure calls and to apply them in context.

We describe each of the steps in detail with respect to a pair of call sites from `f1` and `f2` respectively. At a given call site (say for label `L1`), we store the arguments and the input value of global variables into local variables (`i_l1` and `gi_l1`) respectively. Since `f1` only modifies globals from `g1`, it suffices to store this subset of globals. Similarly, we record the returned value and the globals after return into local variables (`o_l1` and `go_l1`) respectively. Each call site also has a local Boolean *witness* variable (`b_l1`) that is initialized to `false` and set to `true` after the call has returned. The figure shows the transformation of the two particular call sites; other call sites in the remainder of the procedures are similarly instrumented (indicated by the double brackets in "[[si;]]").

After the instrumentation of the bodies of the two procedures, we add a conditional block for each pair of mapped call sites. The blocks are guarded by the Boolean witness variables for the call sites; these blocks are executed only when the corresponding call sites were encountered in an execution and both returned. Each block first stores the values of the globals into local `st_gi` variables. Next, it calls the composed procedure `MS_h1_h2` (this time for the pair of callees), with the calling contexts restored from the `gi_li` variables, passing stored arguments `i_li` as inputs to the composed procedure. The return values (returns and globals) are constrained to be the recorded values from after the two calls, using the `assume` statements. Finally, the globals are restored from the `st_gi` variables, erasing the effect of the call.

We use the notation $\sigma_1 \oplus \sigma_2$ to denote a composed state consisting of a state from the two programs with disjoint signatures.

THEOREM 1. *For two programs P_1 and P_2 and two procedure $p_1 \in P_1$ and $p_2 \in P_2$, $(\sigma_1, \sigma_1') \in \mathcal{T}_{p_1}$ and $(\sigma_2, \sigma_2') \in \mathcal{T}_{p_2}$ if and only if $(\sigma_1 \oplus \sigma_2, \sigma_1' \oplus \sigma_2') \in \mathcal{T}_{MS\_p_1\_p_2}$.*

PROOF. We only sketch the main ideas here. The first part of $MS\_p_1\_p_2$ has the effect of executing p_1 and p_2 in parallel, recording the pre- and post-states of the procedure calls in ghost variables. The second part always has a terminating execution and has no effect. That is, by induction on recursion depth, we can assume the theorem for the call to $MS\_h_1\_h_2$. This guarantees a behavior for which the subsequent `assume` statements are true. Moreover the program's global state is restored. Thus the net effect of $MS\_p_1\_p_2$ is simply to execute p_1 and p_2. □

Theorem 1 illustrates that the transformation performed is not just limited to performing differential assertion checking, but provides a general method to exploit similarity between procedures in program proving. The main power of the transformation comes from providing the additional composed procedures over which one can write specifications towards the proof of a final specification (like DAC). An invariant inference engine now has the extra flexibility to infer invariants about the composed procedures in addition to the procedures in P_1 and P_2.

Consider two versions of `Foo` where the second version accesses fewer indices in the array `a`.

```
var a.1:[int]int;
const MAX: int;
proc Foo.1() {
  var i: int, t:int;
  i := 0;
  while (i <= MAX) {
    assert Valid(i);
    t := a.1[i];
    i := i + 1;
  }
}
```

```
var a.2:[int]int;
const MAX: int;
proc Foo.2() {
  var i: int, t:int;
  i := 0;
  while (i < MAX) {
    assert Valid(i);
    t := a.2[i];
    i := i + 1;
  }
}
```

Let us assume that the loops are extracted into procedures `Loop.1` and `Loop.2` respectively. Our approach will generate the following composed method `MS_Loop.1_Loop.2`. The relative specification (using the keyword `post`) says that if the values of `i` and `ok` are equal at the start of a loop execution, then `Loop.2` fails less often than `Loop.1`. This is an inductive specification, and also sufficient to prove the DAC property for the outer procedures `Foo.1` and `Foo.2`.

```
proc Loop.1(i.1, t.1)
    returns (i.1', t.1');
modifies ok.1
```

```
proc Loop.1(i.2, t.2)
    returns (i.2', t.2');
modifies ok.2
```

```
post (i.1 == i.2 && old(ok.1) <==> old(ok.2))
            ==> (ok.1 ==> ok.2)
proc MS_Loop.1_Loop.2(i.1, t.1, i.2, t.2)
        returns (i.1', t.1', i.2', t.2');
modifies ok.1, ok.2
```

The example also illustrates one other important aspect. The specifications of composed procedures typically have the above form, but are not entirely trivial to obtain. If we had included the equality `t.1 == t.2` alongside `i.1 == i.2` our specification would have been too weak, since `t` is not initialized on entry to the loops. Mutual specifications are often mostly independent of the actual effect of procedures (a great advantage) but may not be the trivial equality over all the state variables in scope.

4.1.1 Relative vs. absolute specifications

On the other hand, let us consider the complexity of the specifications *w*ithout the composed procedure. To prove the DAC property on the two versions of `Foo`, one will need to provide the following precondition for `Foo.1`:

```
pre forall j :: 0 <= j && j <= MAX ==> Valid(j)
```

Informally, this provides the weakest precondition of `Foo.1` to ensure that the procedure does not fail. To prove the DAC property, we will need a loop invariant on `Loop.2` procedure:

```
pre 0 <= i.2 && i.2 <= MAX
```

Although this is another way to prove the DAC property, it demonstrates that one may require program specific (possibly quantified) invariants (since it talks about `MAX`) that may become arbitrarily complex to specify and more difficult to infer. On the other hand, the relative specification used for proving the DAC property using the composed procedure can be fairly easy to guess as it may depend little on details of the actual procedures.

5. INFERRING RELATIVE CONTRACTS

Since the composed procedures have the same syntax as the underlying procedures in P_i programs, we can use any invariant inference technique that can be used to generate invariants for P_i programs. In particular, we can use ideas based on abstraction interpretation [6], abstraction-refinement based [15] predicate abstraction techniques [10], and interpolants [19]. However, any invariant synthesis technique is necessarily incomplete and might either be limited by the underlying domain or may diverge trying to find the inductive invariant. Therefore, it is wise to inject some domain knowledge while looking for invariants for proving differential properties like DAC.

In general, there are two forms of contracts for a composed procedure such as $MS\_f_1\_f_2$ in Figure 3. The precondition of such a procedure would be a predicate over the parameters and globals (i_1, i_2, g_1, g_2), and the postcondition would be predicate over the input and output parameters and globals $(i_1, i_2, \mathsf{old}(g_1), \mathsf{old}(g_2), r_1, r_2, g_1, g_2)$ — we assume that the read sets are also included in g_i globals. Further, many natural two-state postconditions have the form $\phi(i_1, i_2, \mathsf{old}(g_1), \mathsf{old}(g_2)) \Rightarrow \psi(r_1, r_2, g_1, g_2)$. Finally, each of ϕ and ψ usually relate mapped variables (whenever such a mapping can be easily obtained by matching names or types) from the two programs using relations such as equality, inequality and Boolean implications.

5.1 Conjunctive relative specifications

We describe a simple scheme for synthesizing a subset of above specifications, namely conjunctive relative specifications. For each composed procedure we automatically generate a set of *candidate preconditions* and *candidate postconditions* and use the HOUDINI algorithm [8] to infer a subset of these that are inductive for the program and proves the specification. HOUDINI performs a greatest fix-point computation starting with the set of all candidate contracts as live (preconditions and postconditions) and kills a candidate when it cannot be proved modularly assuming the other live candidates. The process is repeated until either no candidate can be removed, or the desired specification can no longer be proved. In the former case, a sufficient inductive invariant has been synthesized for the specification; the latter case indicates either the property does not hold or the set of candidates is insufficient. For BOOGIE programs, one can use an efficient implementation of HOUDINI algorithm using the `/contractInfer` switch in BOOGIE [28].

Now, we describe the set of candidates that are automatically generated for each composed procedure such as $MS\_f_1\_f_2$ in Figure 3. For simplicity, we also assume that each program P_i has a single entry procedure (say p_i^0) that is not called from within P_i and all procedures in P_i have a body. For each f_i ($i \in \{1, 2\}$), let us denote I_i as in-parameters, M_i as the *ref* set of globals, R_i as the out-parameters and G_i as the mod set of globals. For each procedure other than the entry procedure, we first define the sets V_i as $I_i \cup M_i$ (for preconditions) and $R_i \cup G_i$ (for postconditions). For any pair of mapped variables $v_1 \in V_1$ and $v_2 \in V_2$, we add the following expressions as either preconditions or postconditions: (i) $\{v_1 \Rightarrow v_2, v_2 \Rightarrow v_1\}$ for Booleans, (ii) $\{v_1 \leq v_2, v_2 \leq v_1\}$ for integers and (iii) $\{v_1 = v_2\}$ otherwise. Given these candidates, HOUDINI algorithm generates the strongest inductive conjunctive invariant (if any) over these candidates that can prove the DAC specification.

6. EVALUATION

In this section, we describe an implementation and evaluation of DAC inside SYMDIFF [16]. SYMDIFF is an infrastructure for leveraging program verification techniques for comparing programs. The tool is agnostic to source languages (C, Java, C#, x86) as it operates on the BOOGIE intermediate verification language. It currently has a front-end for C programs (using the HAVOC [5] tool) that we use for our experiments. Internally, SYMDIFF leverages the efficient verification condition generation in BOOGIE [3] along with the Z3 [7] theorem prover to verify loop-free and call-free fragments. The implementation of DAC consists of around 800 lines of C# code and mainly performs the following program transformations: (i) introduces an `ok` variable and rewrites the assertions present in a program, (ii) generates the composed procedures (Figure 3), (iii) adds the DAC specification for the entry procedures, and (iv) generates the candidate contracts for the composed procedures (§ 5.1). In addition, for each procedure p, it adds a postcondition $\texttt{ok} \Rightarrow \mathsf{old}(\texttt{ok})$ — this captures the semantics that the ok variable can only transition from true to false.

In the next two subsections, we describe our experience with applying DAC towards two directions. First, we evaluate the inference of relative specifications for verifying bug fixes for a set of small C examples with unbounded loops (§ 6.1). Next, we evaluate the effectiveness of DAC as a mechanism for filtering alarms for evolving programs compared to checking assertions on a single program (§ 6.2).

6.1 Verifying bug fixes

Table 1 describes the result of performing DAC on a set of C examples (except iter which is a hand written BOOGIE example). Each example contains between one and three procedures with at least one loop. The first two examples are already described in this paper, iter in Section 4 and strcpy in Figure 1. The rest of the examples are drawn from the VERISEC suite containing "snippets of open source programs which contained buffer overflow vulnerabilities, as well as the corresponding patched versions." [29]. For each of these benchmarks, we add an assertion assert $Valid(p)$ before any dereference to a pointer expression p. This includes array accesses where $a[i]$ is treated as $*(a + n * i)$ for an array whose base type occupies n bytes. Performing DAC checks that the corrected version is dereferencing only the memory locations which the buggy version does and the bug fix has not inadvertently increased the memory footprint.

Example	# Glbs	# Cands	# Infrd
iter	2	13	6
strcpy	19	29	28
apache-1	23	88	72
madwifi-1	36	187	59
madwifi-2	30	141	117
sendmail-1	20	77	49
sendmail-2	24	65	56

Table 1: Bug fix verification results. "Glbs" denotes globals in the BOOGIE translation of each program, "Cands" denotes candidate preconditions or postconditions, "Infrd" denotes the subset of "Cands" that were inferred by HOUDINI.

The examples in the VERISEC suite range from around 20 to 50 lines of C code (see Figure 4 for the sendmail-1 example). Table 1 indicates that the number of global variables is non-trivial in each example (except iter which is a manually encoded BOOGIE program). These globals (generated by HAVOC [5]) model various aspects of C semantics including maps for each pointer types and fields, allocation status of pointers, and deterministic sequence of values returned by functions such as nondet_int (Figure 4). Finding the right relative specifications can be extremely time consuming given the sizes of product programs. Therefore, the inference is quite invaluable in discovering the relative invariants needed to prove the DAC property, even for these small C examples. Only one example (apache-1) required an additional (absolute) specification not generated by our tool — it specifies that a loop index variable never decreases. For rest of the benchmarks, we were able to automatically infer contracts which were sufficient to prove that the memory footprint of the correct version was no larger than the footprint of the buggy version.

The pair of procedures for sendmail-1 in Figure 4 illustrates a couple of challenges for differential reasoning. First, note that the fix resets the counter fb to 0 under some condition. Therefore, the values of fb on the two programs will get out of sync after fb reaches MAXLINE, since the buggy program will continue to increment fb. Hence the precondition of the composed procedure for the loops only satisfies the specification $\texttt{fb.2} \leq \texttt{fb.1}$. Second, if *Valid* is completely unconstrained, one may not be able to prove the DAC property modularly without using quantifiers in the invariants to record the history of accesses in the first loop. Instead, we constrain *Valid* by the axiom $\forall x, y :: x \leq y \wedge Valid(y) \Rightarrow Valid(x)$ (§ 3.1), allowing the simple relative specifications to prove the DAC property.

```
int main (void)
{
  ...
  fb = 0;
  while ((c1 = nondet_int ())
          != EOF) {
    /* BAD */
    fbuf[fb] = c1;
    fb++;
  }
  /* force out partial
     last line */
  if (fb > 0) {
    /* BAD */
    fbuf[fb] = EOS;
  }
  return 0;
}
```

```
int main (void)
{
  ...
  fb = 0;
  while ((c1 = nondet_int ())
          != EOF) {
    /* OK */
    fbuf[fb] = c1;
    fb++;
    if (fb >= MAXLINE)
      fb = 0;
  }
  /* force out partial
     last line */
  if (fb > 0) {
    /* OK */
    fbuf[fb] = EOS;
  }
  return 0;
}
```

Figure 4: Example of modular bug fix verification (sendmail-1). The "BAD" and "OK" denote buggy and fixed buffer accesses respectively.

In addition to the examples, a vast majority of the remaining examples from the VERISEC suite contain fixes of two categories. In the first case, occurrences of strcpy(char *dest, char *src) was replaced by strncpy(char *dest, char *src, size_t size). Since these are library procedures, we were tempted to write a relative specification that says that whenever the input states (the dest and src pointers and the heap array for char *) to the two procedures are identical and strcpy does not fail, then strncpy does not fail. However, this is not sound as strncpy can access more indices than strcpy in the dest array to fill up the indices up to size with null. If we specify the actual preconditions of the two procedures using ghost states to store the sizes of buffers (similar to § 1.1), all of the fixed versions can be proved directly without the need for any loop invariants or relative reasoning. In other words, DAC does not bring any value in verifying these bug fixes. For the other cases, the fixes correspond to invoking strncpy with a smaller value of the size argument in the fixed version. Although we have not verified these examples, we believe a relative specification of the following form on these library procedures will allow us to verify the fixes without additional ghost states.

```
post    (dest.1 == dest.2 && src.1 == src.2   &&
          old(Mem_char.1 == Mem_char.2) &&
          size.1 <= size.2 && old(ok.1 <==> ok.2))
             ==>
              ok.1 ==> ok.2
proc MS_strncpy.1_strncpy.2(dest.1, src.1, size.1,
                            dest.2, src.2, size.2);
```

Hence, we have demonstrated that DAC can be used for verification of bug fixes. Starting from buggy and correct versions of programs from a standard buffer overflow benchmark, DAC automatically infers relative contracts and proves that the bug fix does not introduce dereferences of new locations; hence, eliminating the possibility of a regression.

6.2 Filtering warnings

In this section, we evaluate the trade offs of differential reasoning as a mechanism for filtering warnings from a program verifier for evolving programs. When a single program is analyzed for some specification (say memory safety) by a verifier, for some programs, invariably there is a flood of warnings. Many such warnings are false alarms due to the limitations of static checking. A developer in such a situation will need some knobs which can lead him to warnings of interest. In evolving software projects, a user is often less concerned with warnings that were present in the earlier releases.

In this section, we perform two case studies for exploring such knobs: with benchmarks from Software-artifact Infrastructure Repository [25] and Windows device drivers [30]. For this section, we check the DAC property with respect to the absence of null dereference errors. Each dereference of a pointer p is preceded with an assertion about $\mathit{Valid}(p)$. Unlike the previous section, we however do not solely focus on changes that correspond to introduction or bug fixes for this class of assertions.

For the purpose of this section, we have done several restrictions and simplifications. First, the loops present in any procedure is unrolled two times. This is done to separate the benefits of DAC from the precision gain obtained by using an invariant inference engine. Second, we only consider one candidate postcondition for the composed procedure where the mapped procedures are semantically equal. This is the default summary considered by SymDiff for performing equivalence checking. In other words, the summary of the composed procedure $MS\_p_1\_p_2$ is limited to either the procedure equivalence or the trivial summary `true`.

We instantiate the framework with five configurations: (i) `single`: each procedure in P_2 (without taking P_1 into account) affected by the change is checked modularly without any preconditions and callee postconditions. This is the default behavior of the static analysis performed by Havoc. (ii) `sound`: when analyzing P_1 and P_2 differentially, we use the candidate summaries described above for the callees. (iii) `unsound`: we assume that callees do not modify the `ok` variables. This amounts to unsoundly assuming that callees do not fail even when called from different states in P_2 compared to P_1. (iv) `bogus`: we unsoundly assume that callees are equivalent including the effect on the `ok` variables. (v) `nonmodular`: we check DAC non-modularly by inlining callees and do not use any specifications. We have designed the different options to compare modular DAC (represented closest by `sound`) with (a) non-differential reasoning (`single`), (b) non-modular DAC (`nonmodular`), (c) effect of increasing unsoundness (`unsound` and `bogus`), which in turn restricts the adversarial environments a static analysis can consider while analyzing a procedure, on a large class of examples. Note that the degree of unsoundness increases in going from `sound` to `unsound` to `bogus`. These modular analyses try to find a single input for an internal procedure for which P_1 does not fail, but P_2 does. On the other hand, `nonmodular` performs an analysis assuming equal inputs only for the entry procedures and not for the internal procedures. As expected, our experiments demonstrate that `sound` $\supseteq$ `unsound` $\supseteq$ `bogus` and `sound` $\supseteq$ `nonmodular` in terms of versions that have warnings. We have also observed that the runtime of `nonmodular` is often 10-100 times more expensive compared to the modular approaches.

Table 2 describes the results on the siemens and space suite of C benchmarks, available from the Software-artifact Infrastructure Repository [25]. Each program in this suite has several versions (the column `versions`) that correspond to injecting various bugs encountered during the development of these benchmarks. However, these bugs are usually functional bugs (changing some conditional or mutating an arithmetic operation) that often do not manifest in null dereference errors. As can be seen from the table, the number of warnings (110 versions out of a total of 127 versions in 848 procedures) arising while checking null dereference absolutely (`single`) can be quite high, even when focusing on the procedures impacted by the change. In comparison, number of warnings progressively decreases with the use of `sound`, `unsound`, and `bogus` options. The `nonmodular` represents the true set of DAC errors; however, inlining does not scale to large programs such as space. Out of these warnings, we have confirmed that 2 warnings in `schedule` are true null-dereference bugs caused by the change. We also notice that `unsound` and `bogus` options are very similar in nature, except that more procedures can fail in `unsound` (e.g. `schedule2`).

Figure 5 (from `schedule2`) shows an example where performing differential reasoning allowed suppressing a warning generated by `single`. On analyzing just a single procedure, every dereference of `job` is flagged as a warning, since the input value of `job` can be `null`. DAC is able to show relative correctness: the second program does not dereference a null pointer if the first one does not.

For the same example, the difference in `unsound` and `bogus` can be seen by looking at the caller `upgrade_prio` of `get_process` (that is syntactically unchanged) (either version in Figure 6). `unsound` flags a warning because it expects `get_process` to return different values in P_1 and P_2 for the `job` variable after its call to `get_process` (since the two versions of `get_process` are not equal), whereas `bogus` does not.

Finally, Figure 6 shows an example where a false warning was caused due to a missing specification of a callee (again from `schedule2`): whenever `get_process` returns a positive value in the `status` variable, the variable `job` is initialized to a non-null value. Hence, even with strong unsound assumptions made by `bogus` for the callees, modular DAC can still cause false warnings due to missing summaries.

Table 3 shows the result of comparing two versions of sample device drivers in the Windows Device Driver Kit (WinDDK) [30]. The drivers for Windows Vista were considered as P_1 and the drivers for Windows 7 were consid-

Name	single	sound	unsound	bogus	nonmodular	versions	LOC	#procs
PrintTokens	5 (6)	5 (6)	0 (0)	0 (0)	0	5	565	18
PrintTokens2	6 (6)	3 (3)	0 (0)	0 (0)	0	10	508	19
Replace	32 (103)	10 (44)	4 (4)	4 (4)	2	32	562	21
Schedule	9 (17)	6 (14)	3 (3)	3 (3)	3	9	410	18
Schedule2	8 (16)	5 (36)	3 (7)	3 (3)	3	10	306	17
TotInfo	12 (12)	6 (8)	2 (2)	2 (2)	2	23	405	7
Space	38 (688)	15 (179)	10 (101)	10 (10)	MO	38	9128	136
Total	110 (848)	50 (290)	22(117)	22 (22)	10+	127	11884	236

Table 2: `Name` is the name of the benchmark; `version` is the number of different versions analyzed. `LOC` is lines of code and `#procs` is the number of procedures in each program. The numbers $x(y)$ mean that x versions and y procedures show warnings. "MO" is a out-of-memory exception.

Name	Diff	SymDiff	single	sound	unsound	bogus	nonmodular	LOC	#procs
firefly	1	1	1	1	1	1	1	634	7
moufilter	4	2	0	0	0	0	0	504	6
pciide	4	0	1	0	0	0	0	182	5
sfloppy	14	6	11	1	1	1	2	3404	20
diskperf	4	4	4	3	2	2	2	2319	24
event	1	1	0	0	0	0	1	555	5
cancel	3	1	0	1	0	0	0	476	5
Total	31	15	16	6	4	4	6	8074	72

Table 3: `Name` is the name of the benchmark; `Diff` is the number of procedures syntactically modified between Vista and Win7 `SymDiff` is the number of procedures for which the summary is true for the composed procedure. `LOC` is lines of code and `#procs` is the number of procedures in Win7 driver.

```
int
get_process ( prio ,  ratio ,  job )
      int  prio ;
      float  ratio ;
      struct  process  ** job ;
{
      ...
      if ( ratio  < 0.0 ||
           ratio  > 1.0)
           return (BADRATIO);
      ...
      *job  = *next;
      if (*job)  {
           ...
           return(TRUE);
      }
      else  return(FALSE);
}
```

```
int
get_process ( prio ,  ratio ,  job )
      int  prio ;
      float  ratio ;
      struct  process  ** job ;
{
      ...
      if ( ratio  < 0.0 ||
           ratio  >= 1.0)
           return (BADRATIO);
      ...
      *job  = *next;
      if (*job)  {
           ...
           return(TRUE);
      }
      else  return(FALSE);
}
```

Figure 5: Difference between "Single" and "Sound" on `schedule2`. The line in italics shows the change.

```
int  upgrade_prio ( prio ,  ratio )
      int  prio ;
      float  ratio ;
{
      int  status ;
      struct  process  * job ;
      if ( prio  < 1 ||
           prio  > MAXLOPRIO)
              return(BADPRIO);
      if (( status  =
              get_process ( prio ,
              ratio ,  &job)) <= 0)
           return( status );
      job-> priority   = prio  + 1;
      ...
}
```

```
int  upgrade_prio ( prio ,  ratio )
      int  prio ;
      float  ratio ;
{
      int  status ;
      struct  process  * job ;



      if (( status  =
              get_process ( prio ,
              ratio ,  &job)) <= 0)
           return( status );
      job-> priority   = prio  + 1;
      ...
}
```

Figure 6: Imprecision in `bogus`

ered as P_2. The first column shows the name of the driver. The second column shows the number of procedures which were syntactically modified in going from Vista WinDDK to Win7 WinDDK for the same driver. The third column shows the number of functions which SymDiff failed to prove equivalent. Again the results are expected: the number of alarms are more for absolute correctness (`single`) than relative correctness. The sound strategy raises more alarms than the unsound and bogus strategies.

The experiments illustrate the potential of modular DAC towards providing a set of systematic knobs to narrow down the set of warnings resulting due to the program modification.

7. RELATED WORK

The idea of performing relative correctness checking is certainly not new; it goes back at least to checking simulation between two designs (usually at different levels of abstraction) using refinement mappings [1]. In contrast, DAC specifications are not necessarily refinement checks; the assertions present in a given program can be used to induce the relative specification. Moreover we do not require the specification of a refinement mapping given that the two programs under consideration are at the same level of abstraction. The most popular form of relative specifications for programs is equivalence checking. Such specifications come up most naturally while performing and checking compiler optimizations [27, 23, 20]. Translation validation [23, 20] is a technique that compares the source program and the compiled program for equivalence. Recent approaches including regression verification [9], differential symbolic execution (DSE) [22] and SymDiff [16] have used equivalence checking for comparing different versions of a program, using a very constrained form of intermediate relative specification (namely, equivalence). In contrast to these works on equivalence checking, DAC can be used to check more general relative specifications for evolving programs. Although DSE provides differential summaries (for loop-free and recursion-free procedures) for arbitrary program changes, it does not provide a decision problem that DAC provides. In our experience with SymDiff, separating intended changes from unintended ones is the hardest problem when displaying differences to a user; DAC provides an intuitive specification whose violations are expected to be interesting for a user. Moreover, the DAC specifications need not be very program specific and can talk about relative specifications (such as using the *Valid* predicate for checking memory safety differentially) that are fairly abstract and thus applicable to most programs. Finally, unlike previous approaches we provide a mechanism to leverage any off-the-shelf program verifier and invariant inference engine to check these relative specifications. The idea of comparing two programs with respect to assertions present has been suggested in previous works [17, 14, 24], but they do not provide a mechanism to specify or generate intermediate relative specifications, especially for loops. Gu et al. [11] investigate the *completeness* of a bug fix with distance-bounded weakest precondition, but cannot provide any soundness guarantees in the presence of unbounded loops and recursion. Mutual summaries [13] provide a mechanism for writing relative specifications by using axioms on pairs of procedures. These mutual summaries can be seen as postconditions on the composed procedures. However, the approach cannot leverage any off-the-shelf program verifiers and invariant inference engines to discover relative specifications.

8. CONCLUSION

In this work, we have described DAC as a mechanism for trading off cost for guarantees obtained while verifying evolving programs. We have reduced checking DAC to analysis of a single program that can utilize any program verification and invariant inference tool. We have provided an implementation of a simple scheme for automating the inference. We are currently integrating other tools based on interpolants [19] to generate relative specifications when the current scheme does not suffice.

APPENDIX

A. LOOPS

In this section, we describe how loops are transformed into tail-recursive procedures. [1] Although extracting loops as tail-recursive procedures is fairly standard, our approach differs from previous approaches [18] by avoiding non-determinism in modeling the extracted procedure. This is important when comparing two programs; internal non-determinism makes program comparison difficult [14]. Our approch requires that the control flow graph is *reducible*, i.e., there is only one entry point for a loop. This assumption is true for almost any program generated from high-level languages such as C and Java.

We illustrate our approach informally using the example below where we use goto statements to model various control flow constructs present in high-level languages:

```
L0:
  s1;
  goto L0;   //continue
  s2;
  goto L1; //break/jmp/return
  s3;
  goto L0;   //loopback
L1:
  s4;
```

The loop is replaced by the following code fragment, where we use "[[s]]" to denote transforming any loops recursively inside a statement s.

```
L0: i' := call L0_loop(i);
   [[s1;]]
  assume false; //goto L0;
   [[s2;]]
  goto L1; //break/jmp/return
   [[s3;]]
  assume false; //goto L0;
L1:
   [[s4;]]
```

Here `i` represents the non-global variables in scope. In addition to the call to the tail recursive procedure `L0_loop`, the interesting aspect is the duplication of the last iteration of the loop body after the recursive call. The purpose of this is to handle goto statements that jump out of the loop (such as `goto L1`) [18]. The body of the tail recursive procedure transforms jumps to the loop head as tail-recursive calls. The main change to make the extracted procedure deterministic is to replace the jumps outside the loop by a statement that restores the state of the return and globals to the initial state.

```
proc L0_loop(i): i' {
  i' := i;
   [[s1;]]
  i' := call L0_loop(i'); //tail-recursive call
  return;
   [[s2;]]
  i' := i; g := old(g); return; //restore state
   [[s3;]]
  i' := call L0_loop(i'); //tail-recursive call
  return;
}
```

[1] The exact Boogie options to be specified are `/printInstrumented /extractLoops /deterministicExtractLoops`.

B. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO '05*, LNCS 4111, pages 364–387, 2005.

[3] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87, 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS '05*, LNCS 3362, pages 49–69, 2005.

[5] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, pages 302–314, 2009.

[6] P. Cousot and R. Cousot. Abstract interpretation : A Unified Lattice Model for the Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL '77)*. ACM Press, 1977.

[7] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, pages 337–340, 2008.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe (FME '01)*, 2001.

[9] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.

[10] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 72–83, June 1997.

[11] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *International Conference on Software Engineering (ICSE'10)*, pages 55–64. ACM, 2010.

[12] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *International Conference on Software Engineering (ICSE'06)*, pages 232–241. ACM, 2006.

[13] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Mutual summaries and relative termination. Technical Report MSR-TR-2011-112, Microsoft Research, 2011.

[14] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *Principles of Programming Languages (POPL'12)*, pages 19–30. ACM, 2012.

[15] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.

[16] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification (CAV'12)*, volume LNCS 7358, pages 712–717, 2012.

[17] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: opportunities, applications, and challenges. In *Workshop on Future of Software Engineering Research (FoSER'10)*, pages 201–204. ACM, 2010.

[18] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *Computer Aided Verification (CAV'12)*, volume LNCS 7358, pages 427–443, 2012.

[19] K. L. McMillan. An interpolating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume LNCS 2988, pages 16–30, 2004.

[20] G. C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, pages 83–94, 2000.

[21] D. Notkin. Longitudinal program analysis. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE '02)*, page 1. ACM, 2002.

[22] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.

[23] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*, pages 151–166, 1998.

[24] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Computer Aided Verification (CAV'11)*, volume LNCS 6806, pages 669–685, 2011.

[25] Software-artifact Infrastructure Repository. Available at `http://sir.unl.edu/portal/index.html`.

[26] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Computer Aided Verification (CAV'94)*, volume LNCS 818, pages 351–363, 1994.

[27] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Principles of Programming Languages (POPL '09)*, pages 264–276, 2009.

[28] The Boogie Verifier. Available at `http://boogie.codeplex.com`.

[29] Verisec Suite. Available at `http://se.cs.toronto.edu/index.php/Verisec_Suite`.

[30] Microsoft Windows Driver Kit (WDK). `http://www.microsoft.com/whdc/devtools/ddk/default.mspx`.