Conditional Equivalence

Ming Kawaguchi
University of California, San Diego
mwookawa@cs.ucsd.edu

Shuvendu K. Lahiri
Microsoft Research, Redmond
shuvendu@microsoft.com

Henrique Rebêlo
Federal University of Pernambuco, Brazil
hemr@cin.ufpe.br

A typical software module evolves through many versions over the course of its development. To maintain compatibility with module clients, it is crucial that a module's behavior at its interface does not change in an undesirable manner across versions. The problem of introducing changes which break interface behavior remains one of the most daunting challenges in the maintenance of large software modules.

Static equivalence checking of sequential programs is a useful mechanism to validate semantic equivalence across refactoring changes. However, most changes corresponding to bug fixes and feature additions change the behavior of programs; equivalence checking tools are of limited help in such cases. In this work, we propose the notion of *conditional (partial) equivalence*, a more practical notion of equivalence in which two versions of a program need only be semantically equivalent under a subset of all inputs. We provide a compositional method for checking conditional equivalence and a fix-point procedure parameterized by an abstract domain for synthesizing non-trivial conditions under which two programs are equivalent. Additionally, we propose a method called *differential inlining* to lazily construct summaries of behavioral differences along differential paths interprocedurally, for recursion-free programs. We discuss preliminary experience of a prototype implementation on a set of medium sized C benchmarks.

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

A typical software module evolves through many versions over the course of its development. Each version can be the result of various factors, including feature addition, performance optimizations, refactoring, or specific bug fixes. It is desirable that a module's interface behavior does not change in an undesirable manner across these versions.

Currently, software developers attempt to ensure this by a combination of (a) syntactic program differencing and manual code inspection, and (b) observed behavior on a set of regression tests. However, for large pieces of software, syntactic differencing provides little insight into the effect of a syntactic change on program behavior, and provides little help in code reviews. Hence, regression testing remains the primary safeguard against the introduction of undesirable changes in a software module. These regression tests are carefully crafted by the developers and testers of the module, and can often number in the hundreds of thousands to millions of tests. Usually such tests are system-level tests; running them can be a significant investment in time and computing resources and therefore it may not be cost-effective to execute them on the developer's desktop to validate small code changes. Further, the set of tests is usually designed in reaction to undesirable behaviors observed in the past, and not to maximize absolute coverage; given the astronomical number of paths in a large software module, there is very little confidence in the absolute coverage of test suites. Consequently, the problem of introducing changes that break interface behavior (usually detected months or years later) remains one of the most daunting challenges in the maintenance of large software modules.

Over the last decade, several approaches have been proposed for providing static feedback about behavioral differences with high coverage at compile time. Godlin and Strichman [16] describe a compositonal method for proving the equivalence of two programs. Symbolic execution [23] has been used to prove equivalence [7, 8, 11, 25] for both hardware and software; they can generate symbolic differences [25] that represent behavioral changes for loop and recursion free programs. Both these approaches make heavy use of uninterpreted functions to abstract procedure calls (and complex operations in hardware) that are semantically equivalent across two versions. This abstraction facilitates scalable reasoning and avoids the path explosion problem.

However, software changes corresponding to bug fixes and feature addition introduce changes in behavior. Such changes not only change the behavior of the procedure that undergoes the change, but all the callers upto the public API of a module. In such cases, equivalence checking tools provide little useful information as they would indicate that these procedures are not equivalent. Further, it is unsound to use uninterpreted functions to abstract procedure calls when the behaviors of a procedure differ. Existing work [16] would inline the body of the changed procedure, which might lead to an explosion in the code size.

In this work, we formalize *conditional equivalence checking*, a compositional method for checking that two procedures are equivalent under a condition c. Checking or inferring conditional equivalence can offer assurance about the behaviors that change across versions. It can also help a tester to prune the space of behaviors to cover by new regression tests — preference can be given to inputs under which the two versions may differ.

We provide a method for modularly checking if two versions of a program P are conditionally equivalent under a set of conditions C, one per procedure $f \in P$. We provide a generic framework, parameterized by an abstract domain, for inferring conditions under which two procedures are equivalent. In other words, we provide a constructive method to infer conditional equivalence when equivalence does not hold. An interesting aspect of this framework is the use of theorem provers in an incremental fashion to enumerate intraprocedural paths that differ in two versions. Finally, for pro-

grams where changes do not happen inside loops or recursive procedures, we describe a method called *differential inlining* (partly inspired by the "demand-driven" construction of procedure summaries [1, 3]) for computing differential summaries lazily; differential summaries [20, 25] are summaries of procedures that only show the input and output symbolic states for inputs that result in differential behavior. We describe some challenges presented by programs that manipulate mutable heaps, as well as insights needed to prove (conditional) equivalence with SMT solvers for such programs. We have a prototype implementation of some of the ideas described in this paper and describe our preliminary experience from using the tool on medium sized C examples.

In summary, the paper makes the following contributions:

1. We formalize the problem of *conditional equivalence checking*, a compositonal method for checking equivalence under some condition, in the presence of loops, recursion and mutable heaps. The significance of the result lies in providing a proof rule for checking that two versions of a loop or a recursive procedure are equivalent under some condition; this is something that was not possible previously.
2. We present a generic framework, parameterized by any abstract domain, for inferring non-trivial conditions under which two versions of a program are equivalent. This opens up the possibility of leveraging advances in program analysis for program evolution.
3. We describe a method called differential inlining to construct differential summaries *lazily* describing behavioral changes when loops and recursive procedures are not involved. This has the potential of being more scalable (by inlining only the behaviors that have changed) compared to inlining procedures that cannot be proven equivalent.

The rest of the paper is organized as follows: Section 2 describes an overview of the techniques in an informal manner. Section 3 sets up the programming language and background of modular checking; Section 4 formalizes the techniques in this paper, including challenges required to deal with the heap. Section 5 describes a prototype tool and preliminary experience. Section 6 describes some related work and Section 7 concludes.

2. Overview

In this section, we introduce the main concepts in the paper at an informal level. The examples are expressed in a variant of the BoogiePL language [5]; we formalize the subset used in Section 3. Variables and expressions in the language can be either of a scalar type τ or a map type $[\tau']\tau$ (meaning a map of τ values indexed by values of type τ'). Scalar types τ can be either `ref` (denoting objects), `int` (denoting integers) or `bool` (denoting Booleans). In most examples, the maps model arrays or a field in an object; i.e., Data[x] denotes x.Data in a language like Java.

2.1 Conditional equivalence checking

Figure 1 shows two versions of a procedure `locate`. The example is a BPL encoding of one of the C benchmarks (`replace`) from the SOFTWARE-ARTIFACTS INFRASTRUCTURE REPOSITORY(SIR) [29] benchmarks. The versions correspond to `source` and `v6` in the distribution. Each version corresponds to a seeded change that causes an undesirable behavior difference. Each tail-recursive procedure `locate` in Figure 1 actually models a while loop in the actual C program — we treat all loops as tail-recursive procedures in this paper.

The first version of the procedure searches for an integer `c` in the array `pat` between $[\texttt{offset}+1, \texttt{i}]$. Success is denoted by the return value. The expressions following the `post` statements are procedure postconditions. The comment `//CHANGE` denotes a source line in the second version which differs from the first version. The changes affect the return value only when `pat[offset] = c` and none of the locations in $[\texttt{offset}+1, \texttt{i}]$ contain `c`.

Consider a *partial equivalence* checker that can soundly prove that the two procedures have the same input-output behavior for every input on which both procedures terminate (these notions are formalized in Section 4). Since the behaviors of the two versions differ, such a checker would return a "don't know" to indicate that the procedures may possibly exhibit different behaviors under some inputs. This is not terribly interesting information for a user who would like to understand more about the differences.

In this work, we propose *conditional partial equivalence* checking under a condition c. This can be seen as a refinement (or generalization) of partial equivalence checking as it allows a user to prove equivalence under a subset of all inputs. — of course, it requires the user to propose a condition c under which the equivalence holds. Given two versions of a program P consisting of a set of procedures, and a set of conditions $C = \lambda h \in P.c(h)$ (one for each procedure in P), we define $\text{CONDEQ}(P^1, P^2, C)$ to denote that for each $f \in P$, the two versions f^1 and f^2 are equivalent under the inputs $C(f)$. In this work, we provide a sound modular checker for $\text{CONDEQ}(P^1, P^2, C)$ (Section 4.3).

Initially, let us imagine that the contracts in `post` are not provided. Our checker can prove that the two versions of `locate` are equivalent under the input condition:

$$\texttt{i} < \texttt{offset}$$

This is not very hard to see because under this input, none of the versions make a recursive call and return the same value.

Let us now assume that we have been provided with the contracts in `post` for each version. The contracts have been proven on the individual versions using standard contract checking machinery based on Floyd-Hoare logic [18]. In our case, these postconditions were separately verified using the contract checker Boogie [5]. With these additional contracts, we can weaken the condition under which the two versions can be shown equivalent to the following:

$$\texttt{i} < \texttt{offset} \vee \texttt{pat}[\texttt{offset}] \neq \texttt{c}$$

This illustrates an important point — although the checker is incomplete for $\text{CONDEQ}(P^1, P^2, C)$, it can leverage additional contracts present in the code. The checking (and later the inference) algorithms operate in a generic assume-guarantee framework for program verification. Because of this, they can be strengthened by code contracts that specify program invariants for procedures. These contracts can be generated for a single version of the program by hand, or automatically using an off-the-shelf invariant inference tool.

2.2 Inferring conditional equivalence

Consider the simple example in Figure 2 that shows the effect of a feature addition. The example illustrates a recursive procedure `Process` to evaluate an expression tree rooted at the parameter `x`. We have omitted the bodies for the leaf procedures (e.g. `Add`, `AddU`). The parameter `isUnsigned` is used in the second version to support a new feature (unsigned addition). The second version also contains a simple refactoring (e.g. use of a local variable $\texttt{t}_3$).

For such simple changes, it may be desirable to have techniques that can automatically infer non-trivial conditions under which the two versions are equivalent. This example has been deliberately chosen to illustrate that there are an infinite number of paths through the recursive procedures that indicate a behavioral difference. In this paper, we provide a generic framework that can use an abstract domain and an incremental theorem prover to automatically synthesize such conditions even in

```
//Version 1
bool locate(c:int, pat:[int]int,
            offset:int, i:int, flag:bool)

post (i ≤ offset) ⟹ return = flag;
{
    bool ret := flag;
    if (i > offset) {
          if (c = pat[i]) {
                ret := true;
                return locate(c,pat,offset,offset,ret);
          } else {
                return locate(c,pat,offset,i-1,ret);
          }
     }
     return ret;
}
```

```
//Version 2
bool locate(c:int, pat:[int]int,
            offset:int, i:int, flag:bool)

post (i ≤ offset ∧ pat[offset] ≠ c) ⟹ return = flag;
{
    bool ret := flag;
    if (i >= offset) { //CHANGE
          if (c = pat[i]) {
                ret := true;
                return ret;  //CHANGE
          } else {
                return locate(c,pat,offset,i-1,ret);
          }
     }
     return ret;
}
```

Figure 1. Conditional equivalence checking

the presence of loops and recursion. For this example, we can instantiate the framework (described in Section 4.4) with the abstract domain of predicate abstraction [17] and the set of predicates $\mathcal{P} = \{\texttt{isUnsigned}, \texttt{Op[x]} = 0, \texttt{Op[x]} = 1\}$ chosen from the conditionals in the program. For this abstraction, our technique automatically synthesizes that the two versions are equivalent under the input condition $\neg\texttt{isUnsigned} \vee \texttt{Op[x]} = 0$. We describe the details of the inference in Section 4.4.

2.3 Differential inlining

```
void F(int x, int y){
     B(m[y]);
     D[x] := g;
}
void B(int z){
     if (z > 0) g := 1;
     else       g := 0;
     m[0] := 1;
}
```

```
void F(int x, int y){
     B(m[y]);
     D[x] := g;
}
void B(int z){
     if (z > 0) g := z;
     else       g := 0;
     m[0] := 1;
}
```

Figure 3. Example to illustrate differential inlining.

```
void F(int x, int y){

 if (m[y] > 0)
    g, m := 1,
            U.B_m(m[y], m);
 else
    g, m := U.B_g(m[y], m),
            U.B_m(m[y], m);

 D[x] := g;
}
```

```
void F(int x, int y){

 if (m[y] > 0)
    g, m := m[y],
            U.B_m(m[y], m);
 else
    g, m := U.B_g(m[y], m),
            U.B_m(m[y], m);

 D[x] := g;
}
```

Figure 4. Example after differential inlining.

Finally, let us demonstrate the concepts of differential summaries and differential inlining by the example described in Figure 3. The two versions of the program differ in how the global variable g is updated in procedure B — the two versions have a differring effect on g only for the case $\texttt{z} > 0$.

Unlike the previous examples, the example in this Section is recursion free. For programs without recursion and loops, we provide a bottom up algorithm (Section 4.5) that constructs the differential summary for each procedure and uses the differential summary at a call site. With this algorithm, we obtain the following differential summary $\triangle^{\texttt{B}}$ for procedure B:

COND	st^1	st^2
$\texttt{z} > 0$	$\texttt{g} \mapsto 1$	$\texttt{g} \mapsto \texttt{z}$

A differential summary contains the conditions under which the two versions have different side-effects (COND) and the side effects. The notation $\texttt{x} \mapsto e$ denotes the expression e of the variable x at exit from the procedure. Since both m and D always have the same side effect in either version, it is not present in the differential summary.

Using this differential summary, the calls to B are inlined (as shown in Figure 4). We have introduced uninterpreted functions U.B_g and U.B_m to represent the effect of B on variables g and m respectively. The functions have arity two since the variables read inside B are {z, m}. We call this process differential inlining to indicate that only the differential effects are inlined at the call sites; in all other cases, a common uninterpreted function is used to capture an arbitrary but identical side effect on both versions.

We obtain the following differential summaries for F:

COND	st^1	st^2
$\texttt{m[y]} > 0$	$\texttt{g} \mapsto 1$	$\texttt{g} \mapsto \texttt{m[y]}$
	$\texttt{D} \mapsto \texttt{D[x} := 1\texttt{]}$	$\texttt{g} \mapsto \texttt{D[x} := \texttt{m[y]]}$

3. Programs

In this section, we present a simple programming language to describe deterministic programs containing scalars, and describe how to generate modular verification conditions for an annotated program. We extend the language with maps to model programs with the heap later in Section 4.6.

3.1 Syntax and semantics

Figure 5 shows a simple programming language without loops — loops are modeled as tail-recursive procedures. The language supports scalar variables ($Vars$) and various operations on them. The type of any variable $\texttt{x} \in Vars$ is integer (int). Variables can either be procedure local or global. We denote $G \subseteq Vars$ to be the set of global variables for a program.

Expressions ($Expr$) can be either scalar variables, constants, result of a binary operation in the language (e.g. $+, -$, etc.). Expressions can also be generated by the application of a function symbol U to a list of expressions ($U(e, \ldots, e)$). A $U \in Functions$ represents a function symbol, some of which may have specific interpretations. 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 relational operations (e.g. $\{\leq, =, \geq\}$)

```
[ref]int Op, Val;
[ref]ref A1, A2;

// Version 1
int Process(ref x, bool isUnsigned){
    if(Op[x] = 0) //constant
         return Val[x];
    else {
        int t1 := Process(A1[x], isUnsigned);
        int t2 := Process(A2[x], isUnsigned);
        if (op[x] = 1)
            return Add(t1,t2);
        else
            return Sub(t1,t2);
    }
}
```

```
// Version 2
int Process(ref x, bool isUnsigned){
    if(Op[x] = 0) //constant
         return Val[x];
    else {
        int t1 := Process(A1[x], isUnsigned);
        int t2 := Process(A2[x], isUnsigned);
        int t3 := Op[x];
        if (t3 == 1)
            return isUnsigned? AddU(t1,t2) : Add(t1,t2);
        else
            return Sub(t1,t2);
    }
}
```

Figure 2. Inferring conditional equivalence.

x	$\in$	*Vars*		
U	$\in$	*Functions*		
e	$\in$	*Expr*	::=	$\mathsf{x} \mid c \mid e\ \mathsf{binop}\ e \mid U(e,\ldots,e) \mid \mathsf{old}(e)$
ϕ	$\in$	*Formula*	::=	$\mathsf{true} \mid \mathsf{false} \mid e\ \mathsf{relop}\ e \mid \phi \wedge \phi \mid \neg\phi$
				$\forall u : \mathsf{int}.\ \phi$
s	$\in$	*Stmt*	::=	$\mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \mathsf{x} := e$
				$s; s \mid s\ \diamond\ s \mid \mathsf{x} := \mathsf{call}\ f(e,\ldots,e)$
p	$\in$	*Proc*	::=	$\mathbf{pre}\ \phi_f\ \mathbf{post}\ \phi_f$
				$\mathsf{int}\ f(\mathsf{x}_f : \mathsf{int},\ldots) : \mathsf{ret}_f\ \{\ s\ \}$

Figure 5. A simple programming language without loops. Loops are modeled as tail-recursive procedures.

on *Expr*, Boolean operations ($\{\wedge, \neg\}$), or quantified expressions ($\forall u : \mathsf{int}.\phi$). For any expression (or formula) e, *FV*(e) refers to the variables that appear *free* in e.

The statement skip denotes a no-op. The statement assert ϕ behaves as a skip when the formula ϕ evaluates to true in the current state; else the execution of the program *fails*. 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; $s; t$ denotes the sequential composition of two statements s and t. $s\ \diamond\ t$ denotes a non-deterministic choice to either execute statements in s or t. We require that the $s \diamond t$ statement along with the assume is only used to desugar conditional statements; the statement if (e) $\{s\}$ else $\{t\}$ is modeled as $\{\mathsf{assume}\ e; s\} \diamond \{\mathsf{assume}\ \neg e; t\}$. The restricted use of $s\ \diamond\ t$ ensures that the programs constructed in this language are *deterministic*, i.e. there is a unique execution from a state for a statement $s \in \mathit{Stmt}$.

A procedure $p \in \mathit{Proc}$ has a name f, a set of parameters *Params*(f) of type int, a return variable ret_f of type int, a body s, a precondition $\mathbf{pre}\ \phi_f$ and a postcondition $\mathbf{post}\ \psi_f$. ϕ_f is a formula, such that *FV*(ϕ_f) $\subseteq$ *Params*(f) $\cup$ *G*; ψ is a formula, such that *FV*(ϕ_f) $\subseteq$ *Params*(f) $\cup$ *G* $\cup \{\mathsf{ret}_f\}$; Procedure calls are modeled using the call statement. The procedure call can have a side effect by modifying one of the global variables.

For any procedure f, we denote $RV_f \subseteq$ *Params*(f) $\cup$ *G* to be the variables that can be read during the execution of f. We denote $WV_f \subseteq G \cup \{\mathsf{ret}_f\}$ to be the subset of global variables and the return variable of f that may be written during an execution of f. Both RV_f and WV_f account for globals that are read from or written to in the procedures that can be called from f, in addition to the definition of f.

A state σ of a program at a given program location is a valuation of the variables in scope, including procedure parameters, locals and the globals. We omit the definition of an execution as it is quite standard.

3.2 Verification condition generation

The weakest (liberal) precondition of an assertion $\phi \in \mathit{Formula}$ with respect to a statement $s \in \mathit{Stmt}$ is denoted as $wlp(s, \phi)$. Intuitively, $wlp(s, \phi)$ represents the set of states from which executing s does not fail any assertions in s and moreover if the execution terminates, it does so in a state satisfying ϕ. Figure 6 shows the weakest precondition for the statements in our language, and are fairly standard [4]. We use $\phi[e/\mathsf{x}]$ to denote syntactic substitution of all occurrences of a variable x with e in the formula ϕ.

$$
\begin{array}{lcl}
wlp(\mathsf{skip}, \phi) & = & \phi \\
wlp(\mathsf{assert}\ \psi, \phi) & = & \psi \wedge \phi \\
wlp(\mathsf{assume}\ \psi, \phi) & = & \psi \implies \phi \\
wlp(\mathsf{x} := e, \phi) & = & \phi[e/\mathsf{x}] \\
wlp(s; t, \phi) & = & wlp(s, wlp(t, \phi)) \\
wlp(s\ \diamond\ t, \phi) & = & wlp(s, \phi) \wedge wlp(t, \phi)
\end{array}
$$

Figure 6. Weakest precondition for simple statements without procedure calls.

To perform modular verification of a procedure f, a procedure call call $h(e)$ is desugared by first asserting the preconditions of h, scrambling all the variables that could be modified by h and then assuming the postconditions of h. To scramble a variable, we need a new statement havoc x that introduces non-determinism in the programming language. Instead, for a deterministic program, one can use an uninterpreted function to assign an arbitrary value the variables in the modified set. For simplicity, let $RV_h = \{\mathsf{x}_h, \mathsf{y}\}$ and $WV_h = \{\mathsf{z}, \mathsf{ret}_h\}$; the call $\mathsf{r} := \mathsf{call}\ h(e)$ is desugared as:

$$
\begin{array}{l}
\mathsf{assert}\ \phi_h[e/\mathsf{x}_h]; \\
\mathsf{r}, \mathsf{z} := U.h_{\mathsf{ret}_h}(\mathsf{x}_h, \mathsf{y}),\ U.h_{\mathsf{z}}(\mathsf{x}_h, \mathsf{y}); \\
\mathsf{assume}\ \psi_h[e/\mathsf{x}_h][\mathsf{r}/\mathsf{ret}_h];
\end{array}
$$

We use $\mathsf{x}, \mathsf{y} := e_1, e_2$; to denote a parallel assignment. Finally, to verify a procedure f with specifications:

$$
\begin{array}{l}
\mathbf{pre}\ \phi_f\ \mathbf{post}\ \psi_f \\
\mathsf{int}\ f\ (\mathsf{x}_f : \mathsf{int}, \ldots) : \mathsf{ret}_f\ \{\ s\ \}
\end{array}
$$

the partial correctness is expressed as the formula (called the *verification condition*) $VC(f) \doteq wlp(\mathsf{assume}\ \phi_f;\ s\ ;\ \mathsf{assert}\ \psi_f, \mathsf{true})$.

The negation of this formula is checked for unsatisfiability by a theorem prover; if the theorem prover declares the formula unsatisfiable, the procedure satisfies the partial specification. In our case, we use an SMT solver as a theorem prover to automatically check the unsatisfiability. Modern SMT solvers support efficient reasoning over equality, uninterpreted functions, select-update arrays, arithmetic and restricted quantifiers. The substitution for assignment statements in the wlp transformer may lead to an exponential blowup; this is avoided by introducing auxiliary variables (static single assignment) and checking an equisatisfiable formula whose size is linear in the size of the program [4].

4. Conditional equivalence

4.1 Notation

We first provide notation and assumptions that will be used in the rest of the paper to simplify the presentation.

A base program $P = \{f_1, \ldots, f_k\}$ consists of a set of procedures. We assume that a base program is accompanied by two versions P^1, P^2 such that, for each procedure $f \in P$, there exist versions $f^1 \in P^1$ and $f^2 \in P^2$; this can be achieved by introducing a procedure with an empty body with a non-deterministic return value if some version does not exist in the program source. For the sake of this section, we assume that programs are *i.e.*, closed. That is, if a procedure $f \in P$ calls a procedure g, then $g \in P$. We assume that f^1 and f^2 have the same *signature*, *i.e.*, the same names and types for the parameters and the return value—one can think of f as being an interface for f^1, f^2. We denote RV_f, WV_f to be the union of the variables read from and written to in either of the procedure versions. Similarly, we take the global variables to be the union of the globals in either program version.

Hereafter, for simplicity of exposition, we assume that each procedure f takes a single parameter x_f. Unless otherwise mentioned, g refers to the only global variable in the program.

We use the notation $C = \lambda h.\ \phi(h)$ to be an indexed (by procedures) set of formulas such that $C(f)$ denotes the formula for the procedure f. We extend this notation to refer to an indexed set of expressions, constants, sets of states, *etc*.

4.2 Conditional equivalence

For a procedure f, an input (state) is a valuation to variables in $G \cup \mathit{Params}(f)$. An output (state) of a procedure is a valuation to the variables in $G \cup \{\mathsf{ret}_f\}$. For a state σ and a procedure f, let $\mathit{Exec}(f, \sigma)$ denote the (output) state obtained by executing the body of f starting from an input state σ to f. We use ω to denote a non-terminating execution that diverges.

We now define the notions of *conditional partial equivalence* and *partial equivalence* between two versions of a procedure or program.

DEFINITION 1 (**Partial Equivalence under σ**). *Two procedures f^1 and f^2 with identical signatures are said to be* partially equivalent under an input σ *if either (i) $\mathit{Exec}(f^1, \sigma) = \omega$, or (ii) $\mathit{Exec}(f^2, \sigma) = \omega$, or (iii) $\mathit{Exec}(f^1, \sigma) = \mathit{Exec}(f^2, \sigma)$.*

For a procedure f, a formula $c \in \mathit{Formula}$ such that $\mathit{FV}(c) \subseteq \mathit{Params}(f) \cup G$ is called an *input condition* for f. An input condition denotes a set of input states for a procedure f. For a formula c and a state σ that assigns a valuation to the free variables in c, we use $\langle c \rangle_\sigma$ to denote the evaluation of c.

DEFINITION 2 (**Conditional partial equivalence**). *Two procedures f^1 and f^2 with identical signatures are said to be* conditionally partially equivalent under an input condition *c, denoted as* $\textsc{CondEq}(f^1, f^2, c)$*, if f^1 and f^2 are partially equivalent under each input in $\{\sigma \mid \langle c \rangle_\sigma = \mathsf{true}\}$.*

An indexed set of input conditions $C = \lambda h.\ c(h)$ is one where $C(f)$ is an input condition for $f \in P$. We will often refer to C simply as input conditions.

DEFINITION 3. *Given two versions P^1 and P^2 of a program P and an indexed set of input conditions $C = \lambda h.\ c(h)$ for procedures in P, P^1 and P^2 are conditionally partially equivalent under C, denoted as* $\textsc{CondEq}(P^1, P^2, C)$*, if, for each $f \in P$, f^1 and f^2 are conditionally partially equivalent under $C(f)$.*

When $\textsc{CondEq}(f^1, f^2, \mathsf{true})$, we say that f^1 and f^2 are *partially equivalent* [16]. Similarly, when $\textsc{CondEq}(P^1, P^2, \lambda h.\ \mathsf{true})$, we say that P^1 and P^2 are partially equivalent. Hereafter, we refer to $\lambda h.\mathsf{true}$ as $TRUE$ and use the term equivalence to refer to partial equivalence.

4.3 Checking conditional equivalence

We present a method for proving partial equivalence of two versions P^1 and P^2 of a program under input condition C. Our procedure is an extension of the method for proving equivalence of two versions of a program [16]. It retains the *modularity* of equivalence checking [16], meaning that procedure calls are reasoned about using abstractions of callees, instead of inlining.

We define a procedure $\mathit{CondEqProc}_f^C$ that has the same signature as f and checks for $\textsc{CondEq}(f^1, f^2, C(f))$:

```
void CondEqProc_f^C(x_f : int){
  assume (C(f)(x_f, g));
  g_0 := g;
  inline r_1 := call FlattenNestedCalls(f^1, C, 1)(x_f);
  g_1 := g;
  g := g_0;
  inline r_2 := call FlattenNestedCalls(f^2, C, 2)(x_f);
  g_2 := g;
  assert (g_1 = g_2 ∧ r_1 = r_2);
}
```

Here, the inline annotation on a call statement denotes that the body of the called function, with appropriate substitutions, is placed into the program text at the point of the call. First, let us define $\textsc{FlattenNestedCalls}(f, C, v)$: let $f \in P$ be a procedure, C a set of conditions and $v \in \{1, 2\}$ a version. The result of $\textsc{FlattenNestedCalls}(f, C, v)$ is a new procedure with the same signature as f, except that calls inside f are eliminated; any call $\mathsf{t} := \mathsf{call}\ h(e)$ inside f in version $v \in \{1, 2\}$ is replaced with the following statement block:

```
if (C(h)[e/x_h])
    t, g := U.h_{ret_h}(e, g), U.h_g(e, g);
else
    t, g := U.h^v_{ret_h}(e, g), U.h^v_g(e, g);
```

Here we assume that $WV_h = \{\mathsf{ret}_h, \mathsf{g}\}$ and $RV_h = \{\mathsf{x}_h, \mathsf{g}\}$. The effect of this translation is to replace the procedure call, $\mathsf{call}\ h(e)$, with invocations of uninterpreted functions such that, if $C(h)$ holds at the call site, then $\mathsf{call}\ h(e)$ has the same effect on WV_h in both f^1 and f^2. Notice that when $C(h)$ holds at the call site of h, then we use the same uninterpreted function $U.h_\mathsf{x}(e, \mathsf{g})$ to update a variable x in both versions of f.

The procedure $\mathit{CondEqProc}_f^C$ checks that if f^1 and f^2 are executed starting from the same input where $C(f)$ holds, then the observable outputs are the same. The assume statement restricts the input state to satisfy $C(f)$. Variable g_0 captures the initial value of the globals (g in this case); it is used to restore the global state after executing f^i. Variables g_i (respectively r_i) for $i \in \{1, 2\}$

capture the state of the globals (respectively, return values) after executing f^i. Finally, the assert checks that the observable outputs over WV_f are equivalent. In Figure 1 we illustrated the use of contracts for increasing the precision of checking equivalence. Any contract that has been proven on the individual versions can be safely used as an assumption at the call site of a procedure.

We establish the soundness of our checking procedure with the following theorem.

THEOREM 1. *If the assertion in CondEqProc$_f^C$ holds for each $f \in P$, then* CONDEQ(P^1, P^2, C) *holds.*

Proof sketch: We present a sketch of the proof here:

For each procedure $f \in P$ and a version f^i for $i \in \{1, 2\}$, we define $\widehat{f^i}$ as the following procedure:

```
int f̂ⁱ(x_f : int){
  assume (C(f)(x_f, g));
  inline ret_f := call StructureNestedCalls(fⁱ, C, i)(x_f);
  return ret_f;
}
```

The result of STRUCTURENESTEDCALLS(f^i, C, i) is a new procedure with the same signature as f^i, except that procedure calls of the form t := call $h^i(e)$ in f^i are replaced by the following statement block:

```
if (C(h)[e/x_h])
    t := call ĥⁱ(x_h);
else
    t := call hⁱ(x_h);
```

Notice that all recursive calls in $\widehat{f^i}$ are made under $C(f)$. Hence, we can prove the following about its behavior under C:

LEMMA 1. *For any input $\sigma \in \{\alpha \mid \langle C(f)\rangle_\alpha = \mathsf{true}\}$, Exec$(\widehat{f^i}, \sigma) =$ Exec(f^i, σ)*

Consider the procedure *CondEqProc*$_{\widehat{f}}^{TRUE}$. It is not hard to see that this is nearly identical to *CondEqProc*$_f^C$. In particular, a call to a function h^i in $\widehat{f}$ made under C is effectively replaced by a different uninterpreted function than a call to h^i not made under C. We can show the following lemma:

LEMMA 2. *The assertion in CondEqProc*$_{\widehat{f}}^{TRUE}$ *holds iff the assertion in CondEqProc*$_f^C$ *holds.*

Now, consider the transformed programs

$$\widehat{P^i} = \{\widehat{f^i} \mid f^i \in P^i\} \cup P^1 \cup P^2$$

for the two versions $i \in \{1, 2\}$. That is, each of the two transformed programs contain a complete copy of the procedures in both P^1 and P^2. We then construct a non-standard mapping between the procedures in $\widehat{P^1}$ and $\widehat{P^2}$ such that each $f^i \in \widehat{P^1}$ corresponds to $f^i \in \widehat{P^2}$ for both $i = 1, 2$ and $\widehat{f^1} \in \widehat{P^1}$ corresponds to $\widehat{f^2} \in \widehat{P^2}$.

We leverage the following inference rule for proving partial equivalence [16]:

LEMMA 3. *[16] If the assertion in CondEqProc*$_f^{TRUE}$ *hold for each $f \in P$, then* EQ(P^1, P^2) *holds.*

By Lemma 3, to show that *CondEqProc*$_{\widehat{P^1} \cup \widehat{P^2}}^{TRUE}$ holds, it suffices to show that, for every $f \in P$, the asserts in *CondEqProc*$_{\widehat{f}}^{TRUE}$ hold.

Finally, let us assume that for each $f \in P$, we have proved the assertion in *CondEqProc*$_f^C$. We know by Lemma 1 that for each $f \in P$, it is also true that *CondEqProc*$_{\widehat{f}}^{TRUE}$. Then, by the reasoning above, we have that EQ$(\widehat{P^1}, \widehat{P^2})$. Hence, it follows from Lemma 2 that CONDEQ(P^1, P^2, C).

4.4 Inferring conditions

In this section, we describe a generic framework for synthesizing conditions under which two versions of a program are partially equivalent. At a high level, the algorithm does the following: it starts by assuming that for every $f \in P$, f^1 are f^2 are partially equivalent under all inputs. If it is unable to prove it, then it enumerates the counterexamples (that prevent the proof) to refine (strengthen) the condition under which the two versions are equivalent. This information is used by the callers of f to refine their conditions in turn. This process is repeated until a fix-point is reached and the conditions do not change. Since this process is not guaranteed to terminate in the presence of loops and recursion, we parameterize the framework with an *abstract domain* that allows us to converge at the cost of precision.

An abstract domain [10] $\mathcal{A} : \langle \Sigma_\mathcal{A}, \alpha_\mathcal{A}, \gamma_\mathcal{A}, \sqsubseteq_\mathcal{A}, \sqcup_\mathcal{A}\rangle$ is characterized by the following:

- $\Sigma_\mathcal{A}$ represents the domain of the elements that forms a lattice under $\sqsubseteq_\mathcal{A}$,
- $\alpha_\mathcal{A}$ represents the abstraction function over $\mathit{Formula} \to \Sigma_\mathcal{A}$, where the formula represents a set of concrete states,
- $\gamma_\mathcal{A}$ represents the concretization function over $\Sigma_\mathcal{A} \to \mathit{Formula}$ representing a set of concrete states,
- the pair $(\alpha_\mathcal{A}, \gamma_\mathcal{A})$ forms a *galois connection*, and
- $\sqcup_\mathcal{A}$ represents the function that returns the join of two elements of $\Sigma_\mathcal{A}$.

In addition we also assume that $\alpha_\mathcal{A}(\mathsf{false})$ forms the bottom element $\bot \in \Sigma_\mathcal{A}$, and $\alpha_\mathcal{A}(\mathsf{true})$ form the top element $\top \in \Sigma_\mathcal{A}$. For simplicity, we do not provide a *widening* operator; it can be easily incorporated into the framework.

Algorithm 1 CONSTRCONDEQUIV

Require: Two versions P^1 and P^2 of a program P
Require: An abstract domain $\mathcal{A} : \langle \Sigma_\mathcal{A}, \alpha_\mathcal{A}, \gamma_\mathcal{A}, \sqsubseteq_\mathcal{A}, \sqcup_\mathcal{A}\rangle$
Ensure: An indexed set of conditions C s.t. CONDEQ(P^1, P^2, C)
Ensure: An indexed set of differential summaries $\triangle$

```
1:  WL ← P
2:  D ← λh. α_A(false)
3:  △ ← λh. {}
4:  while ¬ IsEmpty(WL) do
5:      f ← Dequeue(WL)
6:      d, δ ← ComputeNewCondEquiv(f¹, f², D, △)
7:      if ¬(d ⊑_A D(f)) then
8:          D(f) ← D(f) ⊔_A d
9:          WL ← WL ∪ Callers(f)
10:         △^f ← δ
11:     end if
12: end while
13: return λh. ¬γ_A(D(h)), △
```

Let us now describe the algorithm and give the proof sketch of the above theorem. The parts of the algorithms highlighted with text should be ignored until Section 4.5 on differential inlining.

Let us first look at Algorithm 1 CONSTRCONDEQUIV that implements the outer loop of the analysis. It maintains a work-list WL of procedures from P, and the while loop in line 4 iterates until this

Algorithm 2 COMPUTENEWCONDEQUIV

Require: Two versions f^1 and f^2 of a procedure f
Require: An abstract domain $\mathcal{A} : \langle \Sigma_\mathcal{A}, \alpha_\mathcal{A}, \gamma_\mathcal{A}, \sqsubseteq_\mathcal{A}, \sqcup_\mathcal{A} \rangle$
Require: An indexed set of abstract elements D
Require: An indexed set of differential summaries $\triangle$
Ensure: An element of $\Sigma_\mathcal{A}$
Ensure: A differential summary δ for f

1: $d \leftarrow \alpha_\mathcal{A}(\mathsf{false})$
2: $\delta \leftarrow \{\}$
3: $C \leftarrow \lambda h.\ \neg\gamma_\mathcal{A}(D(h))$
4: $\phi \leftarrow \neg VC(\mathit{CondEqProc}_f^C)$
5: $\phi \leftarrow \neg VC(\mathit{CondEqProc}_f^\triangle)$
6: **while** $(cex \leftarrow \text{CheckSat}(\phi)) \neq \mathtt{nil}$ **do**
7: **if** $cex =$ UNKNOWN **then**
8: $\delta \leftarrow \delta \cup \{(\neg\gamma_\mathcal{A}(d), \top, \top)\}$
9: **return** $\alpha_\mathcal{A}(\mathsf{true})$, δ
10: **end if**
11: $(\pi^1, \pi^2) \leftarrow \text{GetPath}(cex)$
12: $(c^1, c^2) \leftarrow (\text{PathCnstr}(\pi^1, f^1), \text{PathCnstr}(\pi^2, f^2))$
13: $c \leftarrow c^1 \wedge c^2$
14: $(st^1, st^2) \leftarrow (\text{SymbExec}(\pi^1), \text{SymbExec}(\pi^2))$
15: $\delta \leftarrow \delta \cup \{(c, st^1, st^2)\}$
16: $d \leftarrow d \sqcup_\mathcal{A} \alpha_\mathcal{A}(c)$
17: $\phi \leftarrow \phi \wedge \neg\gamma_\mathcal{A}(d)$
18: **end while**
19: **return** d, δ

work-list is empty. D is an indexed set of abstract elements from $\Sigma_\mathcal{A}$, one for each procedure in P — $D(f)$ represents the conditions expressed over $\mathcal{A}$ under which the two versions of f are *possibly* not equivalent at any point during the analysis. D is initialized to the bottom element $\alpha_\mathcal{A}(\mathsf{false})$ for each procedure, indicating that we start off assuming the two versions for each procedure are partially equivalent under true. At each stage, a procedure f is removed from WL and COMPUTENEWCONDEQUIV(f^1, f^2, D) is invoked to construct new conditions under which two versions of f may not be equivalent. The line 7 checks if the new set of conditions is already subsumed by $D(f)$; $d \sqsubseteq_\mathcal{A} D(f)$ indicates that no new distinct condition was found and nothing has to be updated. Otherwise, $D(f)$ is updated with the join of $D(f)$ and d (line 8), and consequently the callers of f have to be reanalyzed. The procedure returns $\lambda h.\neg\gamma_\mathcal{A}(D(h))$ as C, since $\gamma_\mathcal{A}(D(h))$ represents the condition under which the two versions of h may differ. Algorithm 2 COMPUTENEWCONDEQUIV is called with two versions of f, along with a snapshot of the current D. Lines 3 creates C from the current D and uses it to construct $\mathit{CondEqProc}_f^C$ in line 4. ϕ is a formula whose satisfiable assignments indicate conditions under which the versions of f may not be partially equivalent. CheckSat(ϕ) is a call to the theorem prover, and returns $\mathtt{nil}$ when ϕ can be shown unsatisfiable; otherwise it returns a counterexample object. The loop terminates when $\mathtt{nil}$ is returned.

A counterexample can be UNKNOWN denoting a timeout, spaceout, or the inability of a prover to produce satisfying assignments. Such a counterexample is interpreted as "don't know" and the algorithm simply returns the abstract element $\alpha_\mathcal{A}(\mathsf{true})$ or $\top$ accordingly. When the counterexample *cex* is not UNKNOWN, we require that the counterexample object encodes the control path in a procedure along which an assertion fails. The details of encoding intraprocedural control flow paths in a verification condition can be found in earlier work on ESC/Java [24]. The encoding uses auxiliary variables to denote the *labels* of the control flow nodes and the theorem prover assigns a value to these labels to indicate the intraprocedural control flow path in a counterexample object. This feature is supported by many modern VC generation tools such as the Boogie verifier in conjunction with theorem provers such as Simplify [13] and Z3 [12].

The procedure GetPath(cex) in line 11 takes the counterexample object and returns the control flow paths π^1 and π^2 in the two versions f^1 and f^2 respectively. Recall that since loops are compiled away as tail-recursive procedures, each procedure only has a bounded number of intraprocedural control flow paths. In our specific case, any π^i for $i \in \{1, 2\}$ starts at entry to f^i, and ends at the return of f^i — this is because the only assertion in $\mathit{CondEqProc}_f^C$ is the final assertion for checking the equality of the globals. Given a control path from entry to exit of $\mathit{CondEqProc}_f^C$, it is easy to split the path into the paths inside f^1 and f^2 respectively.

The procedure PathCnstr(π, f) in line 12 takes a procedure f and a control flow path π in f and returns a conjunction of the tests along the path in terms of the state at the beginning of π. Since π^1 and π^2 start at the entry of each procedure, PathCnstr(π^i, f^i) are both expressions in terms of RV_f. More formally, a path π in a procedure f induces a straight line program $s_\pi \in \mathit{Stmt}$ with only skip (skip), assume (assume), assignment ($:=$) and sequential composition $(s; t)$ statements. PathCnstr(π, f) is defined as $pre(s_\pi, \mathsf{true})$, where pre is defined as follows:

$$
\begin{array}{lcl}
pre(\mathsf{skip}, \phi) & = & \phi \\
pre(\mathsf{assume}\ \psi, \phi) & = & \psi \wedge \phi \\
pre(\mathsf{x} := e, \phi) & = & \phi[e/\mathsf{x}] \\
pre(s; t, \phi) & = & pre(s, pre(t, \phi))
\end{array}
$$

Finally, line 16 updates the set of new conditions under which the two versions may be different. Line 17 constrains ϕ to not consider assignments satisfying $\gamma_\mathcal{A}(d)$ in future iterations. Note that calls to CheckSat(ϕ) in line 6 are always made with more constrained formulas, and therefore exploit the incremental nature of modern SMT solvers. The incremental nature of queries allows the SMT solvers to retain the pruning of the search space from earlier queries for future queries, and can be extremely beneficial in practice.

THEOREM 2. *For two versions P^1 and P^2 of a procedure P, and an abstract domain $\mathcal{A}$, if C is the set of conditions returned by* CONSTRCONDEQUIV$(P^1, P^2, \mathcal{A})$, *then* CONDEQ(P^1, P^2, C) *holds.*

A couple of points:

1. Instead of initializing the WL in Algorithm 1 with P, it is sound to only consider procedures that are transitive callers of procedures that have undergone a syntactic change; the rest of the procedures are trivially semantically equivalent. This is a useful optimization as most changes only affect a very small fraction of procedures in a large module.
2. For any procedure $f \in P$, the formula $C(f)$ computed by CONSTRCONDEQUIV method are expressions over the parameters of f and the global variables. However, they may contain the uninterpreted function symbols $U.h_g$ or $U.h_g^v$ for any (transitive) callee h.
3. Let us consider the case of programs without loops and procedure calls. In such a case, the fixpoint procedure will terminate in at most $|P|$ steps if the Dequeue(WL) respects the call order; i.e., when the Dequeue procedure returns a procedure f, then there are no callees of f present in WL. It is easy to see that the WL monotonically decreases in each step.

4. Although the main use of an abstract domain is to allow ConstrCondEquiv to terminate in the presence of loops and recursion, it can also be useful for programs without loops and recursion. Notice that in line 17, ϕ is updated with $\phi \wedge \neg\gamma_{\mathcal{A}}(d)$ instead of $\phi \wedge \neg\gamma_{\mathcal{A}}(c)$. Since $c \sqsubseteq_{\mathcal{A}} d$ at this line, constraining ϕ with $\neg\gamma_{\mathcal{A}}(d)$ instead of $\neg\gamma_{\mathcal{A}}(c)$ has the effect of converging faster for the loop in line 6. This can be particularly useful when there are large number of paths in a procedure and the change appears along both branches of a conditional statement (i.e. under ψ and $\neg\psi$).

Let us revisit the example from Section 2.2 to show how the algorithm above works. Recall that the abstract domain used is predicate abstraction with the set of predicates $\mathcal{P} = \{\texttt{isUnsigned}, \texttt{Op[x]} = 0, \texttt{Op[x]} = 1\}$.

In our first iteration, we attempt to prove conditional equivalence for the condition `true` and obtain a counterexample for the single program path corresponding to the path condition $\texttt{isUnsigned} \wedge \texttt{Op[x]} \neq 0 \wedge \texttt{Op[x]} = 1$. Since this constraint can be precisely captured by $\mathcal{A}$, d is updated to $\texttt{isUnsigned} \wedge \texttt{Op[x]} \neq 0 \wedge \texttt{Op[x]} = 1$.

In the second iteration, we attempt to prove conditional equivalence for the weaker condition $\neg\gamma_{\mathcal{A}}(d) \doteq \neg\texttt{isUnsigned} \vee \texttt{Op[x]} = 0 \vee \texttt{Op[x]} \neq 1$. This proof attempt fails again, and we obtain a counterexample program paths that correspond to the path condition $\texttt{isUnsigned} \wedge \texttt{Op[x]} \neq 0 \wedge \texttt{Op[A1[x]]} = 1$. The constraint is approximated in the abstract domain as $\texttt{isUnsigned} \wedge \texttt{Op[x]} \neq 0$. Performing a join with d updates d to $\texttt{isUnsigned} \wedge \texttt{Op[x]} \neq 0$.

In the third iteration, we attempt to prove conditional equivalence for the weaker condition $\neg\texttt{isUnsigned} \vee \texttt{Op[x]} = 0$ which succeeds. Without the use of abstraction, there are an infinite number of paths through the recursive procedure that indicate a behavioral difference.

4.5 Differential inlining

With conditional equivalence, we have focused on the question "*when* do the two versions differ?". Another important question is "*how* do the two versions differ?". Person et al. [25] proposed *differential summaries* as a way to symbolically represent the set of input conditions under which two versions of a procedure differ, and the symbolic expression for the output variables. When the procedures affected by a change (either syntactically changed or their transitive callers) do not have any loops or recursion, we extend the algorithm in the previous section to compute differential summaries. The idea is quite simple: first, we generate the differential summaries by generating the symbolic expressions along all pairs of paths that differ in the two versions. Next, we use these summaries at the call sites of a procedure. We formalize this idea and extend Algorithms ConstrCondEquiv and ComputeNewCondEquiv described earlier. The main subtlety of the section is to conservatively deal with Unknown outputs of a theorem prover.

For the rest of the section, we assume the following two preconditions:

1. the set of procedures in P do not have any loops or recursion in either version, and
2. the abstract domain $\mathcal{A}$ is the same as the concrete domain; in other words, $\alpha_{\mathcal{A}}$ and $\gamma_{\mathcal{A}}$ are just the identity functions.

A differential summary $\triangle^f$ for a procedure f is a set of tuples of the form (c, st^1, st^2) where:

- c in an input condition for f, such that $FV(c) \subseteq RV_f$.
- Each st^i is a *symbolic store*, a partial map that stores a symbolic expression st^i_{x} for each $\mathsf{x} \in WV_f$. For each x, $FV(st^i_{\mathsf{x}}) \subseteq RV_f$; in other words the symbolic expressions are purely in terms of the input state of f. Intuitively, st^i_{x} is the effect on x of executing f from an input satisfying c.

The changes required to compute differential summaries are marked by text lines in the two algorithms. The changes in algorithm ConstrCondEquiv are related to initializing, updating and returning $\triangle^f$. Since each procedure is only analyzed once in the while loop in line 4, the value of $\triangle^f$ at exit is really the value computed by ComputeNewCondEquiv in line 10. Notice that we have extended ComputeNewCondEquiv to take an additional input and return an additional value.

Let us now look at ComputeNewCondEquiv extentions. Line 5 uses $\triangle$ instead of C to construct the procedure *CondEqProc*; also note that we discard the value of ϕ computed in line 4 to account for this. Intuitively, $\textit{CondEqProc}^{\triangle}_f$ uses the differential summaries in $\triangle^h$ to replace any calls to h inside procedure f; we describe this formally shortly. The other changes lie in updating δ with (i) a differential summary under $\neg\gamma_{\mathcal{A}}(d)$ with unknown side effects $\top$ for both the versions (line 8) when the theorem prover returns Unknown, and (ii) with the result of symbolic execution along the two paths π^1 and π^2 respectively (line 15). Here $\text{SymbExec}(\pi)$ is a procedure for computing symbolic expressions for variables in WV_f along the control path π; we omit details of this procedure.

We now describe how $\triangle$ is used to construct $\textit{CondEqProc}^{\triangle}_f$. The definition of $\textit{CondEqProc}^{\triangle}_f$ is exactly the same as $\textit{CondEqProc}^{C}_f$, except FlattenNestedCalls(C, f, v) is replaced with calls to FlattenNestedCalls$(\triangle, f, v)$. It replaces any call $\mathsf{r} := \mathsf{call}\ h(e)$ inside f in version v as follows:

- For each $(c, st^1, st^2) \in \triangle^h$ such that $st^i \neq \top$:

  $$\mathsf{if}\ (c[e/\mathsf{x}_h])\ \ \mathsf{r}, \mathsf{g} := st^v_{\mathsf{ret}_h}[e/\mathsf{x}_h], st^v_{\mathsf{g}}[e/\mathsf{x}_h];$$

  to use the actual summary for h under the input c.
- If $(c, \top, \top) \in \triangle^h$:

  $$\mathsf{if}\ (c[e/\mathsf{x}_h])\ \ \mathsf{r}, \mathsf{g} := U.h^v_{\mathsf{ret}_h}(e, \mathsf{g}), U.h^v_{\mathsf{g}}(e, \mathsf{g});$$

  to use completely arbitrary summaries for the two versions, since $\triangle^h$ has unknown summaries $\top$ for the two versions. This is the result of encountering Unknown during the computation of $\triangle^h$ in an earlier iteration.
- Finally, under all other conditions:

  $$\mathsf{else}\ \ \mathsf{r}, \mathsf{g} := U.h_{\mathsf{ret}_h}(e, \mathsf{g}), U.h_{\mathsf{g}}(e, \mathsf{g});$$

  to use the same side-effect for both the versions.

Observe that we only use the actual summaries for h (computed by symbolic execution inside h in line 14) along the conditions where the two versions of h differ. We refer to this process as differential inlining, since it only exposes the details of h along the differential paths.

For any $h \in P$, let

$$\gamma^h \doteq \bigvee \{c \mid (c, \_, \_) \in \triangle^h\}$$

Theorem 3. *For any $h \in P$,* $\text{CondEq}(h^1, h^2, \neg\gamma^h)$ *holds.*

4.6 Dealing with the heap

So far, we have only considered programs containing scalar variables. In this section, we show that the ideas can be carried forward to programs manipulating the heap as well. We also describe some additional challenges that are introduced when dealing with the heap.

First, we model the heap as a collection of *maps*, from integers to integers; this is a fairly standard way to model the heap for Java (ESC/Java [14]), C# (Spec# [6]) or C (HAVOC [9]) programs. These maps model different fields in the program and map addresses to values. A map can be modeled just like a scalar, except that we introduce two special functions $sel \in Functions$ and $upd \in Functions$; $sel(e_1, e_2)$ selects the value of a map value e_1 at index e_2, and $upd(e_1, e_2, e_3)$ returns a new map value by updating a map value e_1 at location e_2 with value e_3. The theory of maps with *sel* and *upd* is supported efficiently by modern SMT solvers. We assume that the type checker for expressions in Figure 5 ensures that map expressions do not participate in arithmetic operations. We introduce the following syntactic sugar for a map variable x in the programming language: $\mathsf{x}[e]$ for $sel(\mathsf{x}, e)$ and $\mathsf{x}[e_1] := e_2$ for $\mathsf{x} := upd(\mathsf{x}, e_1, e_2)$.

We also have to slightly modify the check that compares the global variables for equality in the procedure $CondEqProc_f^C$ described earlier in the section. Consider the two versions of a procedure f below:

```
[int]int x;
int f(){
    x[0] := 1;
    x[1] := 2;
}
```

```
int f(){
    x[1] := 2;
    x[0] := 1;
}
```

The standard theory of *sel* and *upd* for arrays is imprecise to prove that the two versions have the same effect on x. This is because the symbolic expression for x in the two versions are $upd(upd(\mathsf{x}, 0, 1), 1, 2)$ and $upd(upd(\mathsf{x}, 1, 2), 0, 1)$ respectively, which are not equal according to the theory of equality and uninterpreted functions. To circumvent this, we instead check for $\forall u : \mathsf{int}.x[u] = y[u]$, when trying to check the equality of two map values x and y in the $CondEqProc_f^C$ procedure. Although the theory of arrays can be extended with the rule for *extensionality* [31] to deal with the above shortcoming, it can compromise the efficiency of SMT solvers and are not supported by default.

Programs manipulating pointers (in C) or references (in Java) also support dynamic heap allocation. This can be a challenge for verifying equivalence as allocators are sources of non-determinism.

```
ref x; [ref]int a;
int f(){
    x := new();
    a[x] := 1;
}
```

```
int f(){
    x := new();
    a[x] := 1;
}
```

The above example may be difficult to verify because each invocation of `new` returns an arbitrary reference.

Instead, we augment our language to allow (deterministic but arbitrary) allocation and deallocation of objects using statements $\mathsf{x} := \mathsf{new}$ and $\mathsf{free}(\mathsf{x})$ respectively. This is modeled by introducing a global map $\mathsf{alloc} : \mathsf{int} \rightarrow \{0, 1\}$ to reflect the allocation status of a pointer. $\mathsf{x} := \mathsf{new}$ is desugared as:

$$\mathsf{x} := New(\mathsf{alloc}); \mathsf{assume}\ \mathsf{alloc}[\mathsf{x}] = 0; \mathsf{alloc}[\mathsf{x}] := 1;$$

where New is an uninterpreted function. Similarly, $\mathsf{free}(\mathsf{x})$ is desugared as:

$$\mathsf{assert}\ \mathsf{alloc}[\mathsf{x}] = 1; \mathsf{alloc}[\mathsf{x}] := 0;$$

This allows us to show the partial equivalence of programs that can allocate and delete objects. The use of New allows us to model allocation as a deterministic statement.

Of course, our checks can be too strong when the allocation patterns differ in the two versions. It would be useful to develop a theory of array isomorphism in the theorem prover, that may improve the precision of the approach. Although some ideas for comparing heaps are presented at the level of C programs [16], we are currently not aware of such theories in the SMT solvers for efficiently reasoning about the heap.

5. Evaluation

5.1 Implementation

We have prototyped some of the techniques described in this paper in a tool SYMDIFF, which works on BoogiePL programs. The BoogiePL language is very close in spirit to the language described in Section 3. The tool takes two versions of a program, along with a mapping between the procedures, arguments and globals of the two versions. This mapping is automatically generated for closely related programs. We leverage the verification condition generation in Boogie along with the Z3 [12] SMTsolver for checking verification conditions and enumerating intraprocedural counterexamples incrementally.

We have implemented the conditional equivalence checking (Section 4.3) and the differential inlining (Section 4.5). We have not implemented the abstract fix-point algorithm for inferring conditions using an abstract domain — note that this is not a limitation for changes outside loop and recursive procedures. The conditional equivalence checker additionally requires the user to specify the conditions for equivalence; the input program can also contain other contracts that can be checked on the individual versions using Boogie. We currently manually transform loops into tail-recursive procedures for the checking, but this will be automated soon. For differential inlining (that requires loop and recursion-free programs) loops and recursive calls are unrolled a bounded number of times. By default, the tool only analyzes procedures that have undergone a change and their transitive callers. The remaining procedures calls are replaced with uninterpreted functions. For the differential inlining, the tool generates a differential summary for each procedure analyzed. In addition, it also highlights the intraprocedural control paths that exhibit different behaviors on the same input.

A number of tools perform semantics-preserving compilation of programs written in popular programming languages such as C, Java, C# etc. into BoogiePL programs. For example, Spec# [6] translates C# programs, and HAVOC [9] translates C programs into BoogiePL. Therefore, SYMDIFF is effectively a language-agnostic tool that can work with multiple languages. At present, we have integrated HAVOC to translate C program to BoogiePL. The translation provides accurate modeling of the heap, pointer arithmetic and the types present in the program. Specifically, there is a distinct map for each of the word-type fields and each pointer type (e.g. `int *` or `struct A *`) in the program. Each pointer or word-type value is modeled as an integer and various operations on pointers are compiled as operations on integers. Details of the translation can be found in the HAVOC paper [9]. The highlighting of counterexample paths has been extended to highlight source C programs, but the differential summaries are currently generated in terms of the BoogiePL program.

5.2 Experience

We describe some preliminary experience with the tool, and challenges ahead to make it more generally applicable. We have applied the tool to various benchmarks in the SIRbenchmark suites [29], including the 7 benchmarks distributed as part of the `siemens` suite and the `space` program. The `siemens` benchmarks vary in size from 200-700 LOC, the `space` benchmark has around 10 KLOC. In addition, the tool has been applied to bug fixes in two Windows kernel modules ranging upto 20 KLOC. However, since most of the changes we have seen only affect around 5-6 procedures in a module, the absolute size of the module is not indicative. The encouraging part of the tool is that it can scale to procedures several hun-

```
//Version 1
token get_token(tp)
token_stream tp;
{
   int i=0,j; int id=0;
   char ch,ch1[2];
    ch1[0]='0'; ch1[1]='0';
    /* initial the buffer    */
    for (j=0;j<=2 /*80*/;j++)
        { buffer[j]='0';}
    ch1[0]='0'; ch1[1]='0';
    ch=get_char(tp);
    /* strip all blanks until meet characters */
    while(ch==' '||ch=='n')
       {
        ch=get_char(tp);
       }
    buffer[i]=ch;
    if(is_eof_token(buffer)==TRUE)
       return(buffer);
    f(is_spec_symbol(buffer)==TRUE)
       return(buffer);
    if(ch =='"')
        id=1;  /* prepare for string */
    if(ch ==59)
       id=2;     /* prepare for comment */
    ch=get_char(tp);

    /* until meet the end character */
    while (is_token_end(id,ch) == FALSE)
    {
        i++; buffer[i]=ch; ch=get_char(tp);
    }
    /* hold the end charcater           */
    ch1[0]=ch;
    if(is_eof_token(ch1)==TRUE)
        { ch=unget_char(ch,tp);
          if(ch==EOF)unget_error(tp);
          return(buffer);
       }
    if(is_spec_symbol(ch1)==TRUE)
       { ch=unget_char(ch,tp);
         if(ch==EOF)unget_error(tp);
         return(buffer);
       }
    if(id==1)
      { i++; buffer[i]=ch;
        return(buffer);
      }
    if(id==0 && ch==59)

      { ch=unget_char(ch,tp);
        if(ch==EOF)unget_error(tp);
        return(buffer);
      }
  return(buffer);
}
```

```
//Version 2
token get_token(tp)
token_stream tp;
{
   int i=0,j; int id=0;
   char ch,ch1[2];
    ch1[0]='0'; ch1[1]='0';
    /* initial the buffer    */
    for (j=0;j<=2 /*80*/;j++)
        { buffer[j]='0';}
    ch1[0]='0'; ch1[1]='0';
    ch=get_char(tp);
    /* strip all blanks until meet characters */
    while(ch==' '||ch=='n')
       {
        ch=get_char(tp);
       }
    buffer[i]=ch;
    if(is_eof_token(buffer)==TRUE)
       return(buffer);
    f(is_spec_symbol(buffer)==TRUE)
       return(buffer);
    if(ch =='"')
        id=1;  /* prepare for string */
    if(ch ==59)
       id=2;     /* prepare for comment */
    ch=get_char(tp);

    /* until meet the end character */
    while (is_token_end(id,ch) == FALSE)
    {
        i++; buffer[i]=ch; ch=get_char(tp);
    }
    /* hold the end character           */
    ch1[0]=ch;
    if(is_eof_token(ch1)==TRUE)
        { ch=unget_char(ch,tp);
          if(ch==EOF)unget_error(tp);
          //CHANGE
       }
    if(is_spec_symbol(ch1)==TRUE)
       { ch=unget_char(ch,tp);
         if(ch==EOF)unget_error(tp);
         return(buffer);
       }
    if(id==1)
      { i++; buffer[i]=ch;
        return(buffer);
      }
    if(id==0 && ch==59)

      { ch=unget_char(ch,tp);
        if(ch==EOF)unget_error(tp);
        return(buffer);
      }
  return(buffer);
}
```

Figure 7. Example for `print_tokens2`

dred lines large (e.g. in Windows) containing several hundred (and sometimes thousands) of intraprocedural paths in seconds and find semantic differences introduced by the bug-fixes. In most cases, we see between 2-10 intraprocedural paths in the procedure that has undergone simple bug-fixes (between 1-4 lines) and all the paths in the callers. Our initial hope was that the use of differential inlining will reduce the number of paths in callers by removing infeasible paths that do not exhibit the changes, but we have not found such cases in these examples.

Figure 7 shows the output of our tool when comparing two versions (`source` and `v3`) in the `print_tokens2` benchmark in the `siemens` suite. The highlighted regions demonstrate intraprocedural paths along with the procedure `get_token` has two different side effects. A couple of oddities regarding the trace before we explain the example: since loops are unrolled (in this case a bound of 4 was specified), the body of a loop is highlighted when the loop is executed one or more times. Ihis happens with the constant `for` loops in line 9. The `for` loop also highlights a well-known bane for loop unrolling (the loop that executes 80 times) — we had to manually decrease the bound to see interesting behaviors for this procedure. Apart from the `for` loop, the two other `while` loops are executed 0 times to illustrate the difference. The line marked with `//CHANGE` is the line that has been removed from the second version of the program. We do not expect the reader to understand the code to appreciate the difference. For this procedure, the only side effect is through the return variable `buffer`. The interesting part of the difference is to understand that various conditions inside the procedure are correlated — abstracting the common parts of the code with an uninterpreted function may result in imprecision [25], at least in legacy codebases. First, observe that there are quite a few control paths that return even before reaching the changed location. Secondly, for the two versions to have different side effect on `buffer`, both the lines 39 and 47 have to be executed. However the condition guarding this line (`id==1`) depends on an guarded assignment to `id` in line 24. Given the highly constrained nature of the example, it is quite easy for regression tests to miss this behavioral difference and for a developer to be sure of the change by code inspection. In this case, the differential summary (deciphered from the BoogiePL level) provides quick feedback regarding the effect, indicating that the value of `buffer` in the two versions have values

$$[\,'0', '0', \ldots]$$

and

$$[\,'0',\ U.unget\_char^{\texttt{ret}}(U.get\_char^{\texttt{ret}}(\texttt{tp}), \texttt{tp}), \ldots]$$

where $U.f^{\texttt{ret}}$ is the uninterpreted function to model the effect on the return value of a procedure `f`.

In general, we have found the intraprocedual path to be useful for understanding the conditions that trigger behavior differences and the differential summaries to understand the actual effect. Currently, the differential summaries are bloated due to the large arity of the uninterpreted functions used to model the effect of a procedure. The arity of these functions are determined by the RV_f for the procedure f; our conservative analysis of this set make the arities very large (around 30). Although this has not been an issue with SMT solvers, inspecting nested function applications with arity greater than 3 can be cumbersome. We often use "…" to hide the arguments of such functions for display.

6. Related work

The techniques in this paper build on two well-explored lines of previous work.

First, our work utilizes compositional reasoning of modular contract checking [18] and, more recently, its application for program equivalence checking [16]. Contract based checkers enable the proof of programs with recursive procedures and loops by allowing the user to annotate them with contracts when required. On the other hand, there has been a series of work on the use of symbolic execution [23] and the use of uninterpreted functions for checking equivalence of hardware [7] and more recently, software designs [8, 11, 25]. The focus of these methods is to allow completely automatic checking, at the cost of decreased coverage when changes are present inside loops and recursive procedure calls. The use of symbolic execution also allows the construction of differential summaries that symbolically encode the differences [20, 25]. Naive symbolic execution of all the paths may lead to exponential path-explosion problem, and various approximations [11] or summarization [15] may have to be done to circumvent it. We discuss our relationship to these methods in more details below.

In this work we describe a compositional method for checking conditional equivalence, which we believe to be a more practical use of equivalence checking for many programs changes. Our work generalizes the earlier work for checking partial equivalence [16] compositionally. Although, the work on regression verification [16] allows a user to assert the equality of variables under some condition, the conditions are not used to decompose the proof of equivalence. In other words, it cannot prove that two versions of a loop or a recursive procedure are equivalent under a condition stronger than `true`. We have also provided a framework for inferring the conditions (more general than `false`) under which the two versions are equivalent. One of our contribution is to allow this framework (for recursion free cases) to also generate differential summaries as counterexamples to the equivalence checking problem. We believe that the combination of methods presented in this paper improves the applicability of modular contract checkers for providing feedback on program changes. We also believe that conditional equivalence checking can benefit from other optimizations to break mutually recursive procedures [16].

For loop-free and recursion-free programs, the methods based on symbolic execution [8, 11, 25] can provide a complete method to check for equivalence, if the path explosion can be avoided by replacing callees with the same uninterpreted function. In the presence of changes, these techniques may become less scalable if the use of an uninterpreted function is prevented with the slightest of change. Conditional equivalence allows the usage of uninterpreted functions even when procedures are not equivalent under all inputs. We believe this can potentially aid methods based on symbolic execution as well. On the other hand, demand-driven symbolic execution [1] may be used to refine the contracts of a procedure (e.g. example in Figure 1). An empirical evaluation of these techniques on a common set of benchmarks will constitute an interesting future work to understand the strengths of each of these techniques.

Our method of differential inlining also has interesting resemblance with previous works (outside the domain of equivalence checking) in demand-driven summary generation [1, 3]. These techniques lazily refine the procedure summary starting from a completely uninterpreted summary of a procedure, based on counterexamples to runtime assertions [3] or need for generating test cases in dynamic symbolic execution [1]. However, in our case we do not *refine* an uninterpreted function; our goal is to identify a subset of inputs under which the two procedures can be replaced by an uninterpreted function. Similarly, the use of a generic abstract domain and the theorem prover for inferring non-trivial conditions for equivalence has resemblance with earlier work on generating precise abstract transformers [28].

In addition to the use of theorem provers, Jackson and Ladd [20] present a method based on tracking the dependencies between the input and the output variables to check for semantic differences. Although the method can be quite efficient, it is not complete — there could be semantic differences even when the dependencies

are identical. We believe this was the first paper that motivated a static approach to providing semantic differences for software, and have inspired our work on differential summaries. Kim and Notkin [22] present a *rule-based* approach to summarizing large, systematic code changes by correlating them and then relating them according to language semantics. While this work excels at concisely summarizing large, systematic textual changes, it is not intended to capture behavioral differences across code changes. In addition to these works, several syntactic approaches have been proposed to deal with characterizing differences (see [2] for a complete list).

In addition to the static techniques, a variety of dynamic techniques exist to capture behavioral changes [19, 26, 27]. The dynamic approaches require a set of regression test cases in addition to the two versions of the program. Some of these approaches [19, 26] use symbolic execution for better debugging of regression failures. Finally, there is a rich history of techniques related to *regression test selection* (see [30] for details), regression test generation (see [32] for details) and identifying code clones ([21] for details) that are orthogonal to this paper.

7. Conclusions

In this work, we provide a set of techniques for checking and inferring conditional equivalences between closely related versions of a program, in the presence of loops, recursion and the heap. There are several open questions that need to be addressed to make such tools useful to a developer. The choice of an abstract domain for inferring conditional equivalence will likely depend on the nature of the application. We will need demand-driven refinement of the uninterpreted functions used to abstract procedures, when equivalence cannot be established for the expected conditions. Most importantly, more elaborate case studies in the hands of a developer will provide insights into the utility of such a tool in the development life cycle.

Acknowledgement

We are grateful to Ofer Strichman for his feedback on an earlier draft of this work.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, LNCS 4963, pages 367–381, 2008.

[2] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Autom. Softw. Eng.*, 14(1), 2007.

[3] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Computer Aided Verification (CAV '07)*, LNCS 4590, pages 366–378, 2007.

[4] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.

[5] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO '05)*, LNCS 4111, pages 364–387, 2005.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, pages 49–69, 2005.

[7] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1): 1–41, January 2001.

[8] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC*, pages 368–371, 2003.

[9] 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.

[10] 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.

[11] D. W. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. P. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*, 34(1):61–91, 2006.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, pages 337–340, 2008.

[13] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.

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

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

[17] 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.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[19] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *PLDI*, pages 453–464, 2009.

[20] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*, pages 243–252, 1994.

[21] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, pages 81–92. ACM, 2009.

[22] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[24] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3), 2005.

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

[26] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/SIGSOFT FSE*, pages 33–42, 2009.

[27] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, pages 432–448, 2004.

[28] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation, (VMCAI)*, LNCS 2937, pages 252–266, 2004.

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

[30] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, pages 97–106, 2002.

[31] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *LICS '01*, pages 29–37. IEEE Computer Society, June 2001.

[32] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided path exploration for regression test generation. In *ICSE Companion*, pages 311–314. IEEE, 2009.