Mutual Summaries: Unifying Program Comparison Techniques

Chris Hawblitzel[1], Ming Kawaguchi[2], Shuvendu K. Lahiri[1], and Henrique Rebêlo[3]

[1] Microsoft Research, Redmond, WA, USA
[2] University of California, San Diego
[3] Federal University of Pernambuco, Brazil

Abstract. In this paper, we formalize *mutual summaries* as a contract mechanism for comparing two programs, and provide a method for checking such contracts modularly. We show that mutual summary checking generalizes equivalence checking, conditional equivalence checking and translation validation. More interestingly, it enables comparing programs where the changes are *interprocedural*. We have prototyped the ideas in SymDiff, a Boogie based language-independent infrastructure for comparing programs.

1 Introduction

The ability to compare two programs statically has applications in various domains. Comparing successive versions of a program for behavioral equivalence across various refactoring and ensuring that bug fixes and feature additions do not introduce compatibility issues, is crucial to ensure smooth upgrades [3, 7, 4]. Comparing different versions of a program obtained after various compiler transformations (*translation validation*) is useful to ensure that the compiler does not change the semantics of the source program [8, 6]. There are two enablers for program comparison compared to the more general problem of (single) program verification. First, one of the two programs serves as an implicit specification for the other program. Second, exploiting simple and automated abstractions for similar parts of the program can lead to greater automation and scalability.

In spite of the recent progress in program comparison, there are several unsatisfactory issues. There appears to be a lack of uniform basis to formalize these techniques — techniques for comparing programs range from the use of verification condition generation [3, 4] to checking *simulation relations* [6]. Most techniques focus on *intraprocedural* transformations of loops [8, 6] and do not extend to interprocedural transformations. The ones that focus on interprocedural transformations use coarse abstractions of procedures as uninterpreted functions [3, 4] and are not extensible.

In this paper, we formalize *mutual summaries* as a uniform basis for comparing two programs in a programming language without loops and jumps (but containing recursive procedures). Mutual summaries generalize single program contracts such as preconditions and postconditions by relating the summaries of two procedures, usually from two different programs. We provide a sound method for checking mutual summaries modularly. We then describe sound program transformations that can transform unstructured programs (containing loops and jumps) to the language above. We demonstrate that our framework is general enough to capture many existing techniques for comparing programs, including those based on checking simulation relations [6]. Our framework currently lacks the automation provided for specific forms of equivalence checking (e.g. identifying procedures to inline while checking programs with mutually recursive procedures [3], or automatically synthesizing a class of simulation relations for compiler transformations [6]). On the other hand, we show examples of comparing two programs with interprocedural changes for monotonic behavior (§3) and conditional equivalence (§4.2) that are beyond the capabilities of earlier works.

Mutual summaries are currently being incorporated into SymDiff [4], a language-independent framework for comparing programs based on Boogie [1]. Programs in various source languages (C, .NET, x86) are analyzed by translating them to Boogie (e.g. we use Havoc [5] to translate C programs into Boogie). SymDiff provides functionalities to report differences as program traces at the source files along with values of program expressions. As a future direction, we hope to extend the framework to formalize frameworks for comparing classes of concurrent programs [10].

2 Programs

In this section, we present a simple programming language that supports recursive procedures, but does not support loops and unstructured jumps. We will show a way to transform programs with unstructured jumps (including loops) into the language presented in this section (§4.3).

2.1 Syntax and semantics

Figure 1 shows the syntax of the programming and the assertion language. The language supports variables (*Vars*) and various operations on them. The type of any variable $\mathsf{x} \in \mathit{Vars}$ is integer (int). Variables can either be procedure local or global. We denote $G \subseteq \mathit{Vars}$ to be the set of global variables for a program.

Expressions (*Expr*) can be either 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 ($U(e, \ldots, e)$). 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. $\{\leq, =, \geq\}$) on *Expr*, Boolean operations ($\{\wedge, \neg\}$), or quantified expressions ($\forall u : \mathsf{int}.\phi$). Formulas can also be the result of applying a relation R to a list of expressions. A $R \in \mathit{Relations}$ represents a relation symbol, some

$$
\begin{array}{lll}
\mathsf{x} \in \mathit{Vars} & & \\
R \in \mathit{Relations} & & \\
U \in \mathit{Functions} & & \\
e \in \mathit{Expr} & ::= & \mathsf{x} \mid c \mid e \ \mathsf{binop}\ e \mid U(e, \ldots, e) \mid \mathsf{old}(e) \\
\phi \in \mathit{Formula} & ::= & \mathsf{true} \mid \mathsf{false} \mid e \ \mathsf{relop}\ e \mid \phi \wedge \phi \mid \neg\phi \\
 & & R(e, \ldots, e) \mid \forall u : \mathsf{int}.\ \phi \\
s \in \mathit{Stmt} & ::= & \mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \mathsf{x} := e \mid \mathsf{havoc}\ \mathsf{x} \\
 & & s; s \mid s \ \diamond\ s \mid \mathsf{x} := \mathsf{call}\ f(e, \ldots, e) \\
p \in \mathit{Proc} & ::= & \mathsf{pre}\ \phi_f\ \mathsf{post}\ \psi_f \\
 & & \mathsf{int}\ f(\mathsf{x}_f : \mathsf{int}, \ldots) : \mathsf{ret}_f\ \{\ s\ \}
\end{array}
$$

Fig. 1. A simple programming language.

of which may have specific interpretations. For any expression (or formula) e, $FV(e)$ refers to the variables that appear *free* in e.

A map 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.

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; havoc x scrambles the value of a variable x to an arbitrary value. $s; t$ 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. The $s \ \diamond\ t$ statement along with the assume can be used to model conditional statements; the statement if (e) $\{s\}$ else $\{t\}$ is a syntactic sugar for $\{\mathsf{assume}\ e; s\} \ \diamond\ \{\mathsf{assume}\ \neg e; t\}$.

A procedure $p \in \mathit{Proc}$ has a name f, a set of parameters $\mathit{Params}(f)$ of type int, a return variable ret_f of type int, a body s. Procedure calls are denoted using the call statement. The procedure call can have a side effect by modifying one of the global variables. Procedures can be annotated with *contracts*: a precondition pre ϕ_f and a postcondition post ψ_f. ϕ_f is a formula, such that $FV(\phi_f) \subseteq \mathit{Params}(f) \cup G$; ψ_f is a formula, such that $FV(\psi_f) \subseteq \mathit{Params}(f) \cup G \cup \{\mathsf{ret}_f\}$. We refer to preconditions and postconditions as *single program* contracts in the rest of the paper, to distinguish from contracts relating two programs.

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.

For a program annotated with contracts, there are standard methods of transforming them into logical formulas (often referred to as verification conditions) [1]. A procedure call is first replaced by $\mathsf{assert}\ \phi_f; \mathsf{havoc}\ \mathsf{g}; \mathsf{assume}\ \psi_f$, and the resulting call-free fragment is translated into a formula by variants of weakest-precondition transformer [2]. If the resulting formula is valid (usually

```
int g1;

pre x1 >= 0
void Foo1(int x1){
     if (x1 < 100){
        g1 = g1 + x1;
        Foo1(x1 + 1);
     }
}
```

```
int g2;

pre x2 >= 0
void Foo2(int x2){
     if (x2 < 100){
        g2 = g2 + 2*x2;
        Foo2(x2 + 1);
     }
}
```

Fig. 2. Example demonstrating mutual summaries.

checked by a Satisfiability Modulo Theories (SMT) solver [9]), then the program satisfies its contracts.

3 Mutual summaries

3.1 Definition

A *program* P consists of a set of procedures $\{f_1, \ldots, f_k\}$. For the sake of this paper, we assume that programs are *closed*; i.e., if a procedure $f_1 \in P$ calls a procedure f_2, then $f_2 \in P$.

We use the notation $C = \lambda f^1.f^2.\ \phi(f^1, f^2)$ to be an indexed (by a pair of procedures) set of formulas such that $C(f^1, f^2)$ denotes the formula for the pair (f^1, f^2). We extend this notation to refer to an indexed set of expressions, constants, sets of states, *etc.* 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 a program that can be read and written to by any procedure.

Definition 1 (Mutual summaries). *For any pair of procedures $f^1 \in P^1$ and $f^2 \in P^2$, a formula $C(f^1, f^2)$ is a mutual summary, if the signature of $C(f^1, f^2)$ only refers to variables that are in scope at exit from either f^1 or f^2. This includes the parameters, the globals, the return variables for both f^1 and f^2; in addition, $C(f^1, f^2)$ can refer to the value of the globals at entry to a procedure using the* $\mathsf{old}(.)$ *notation.*

For an execution of a procedure f, the *summary* of the execution is a relationship between the pre and the post states of the execution. Given two programs P^1 and P^2, an indexed set of mutual summaries $C : P^1 \times P^2 \rightarrow \mathit{Formula}$, we define the *mutual summary checking* problem as follows:

> For any pair of procedures (f^1, f^2), the summaries of any pair of executions of f^1 and f^2 should satisfy $C(f^1, f^2)$.

We expect C to be sparse; i.e., it will be defined for a few pair of procedures and will be true for most pairs.

Example 1. Consider the two programs in Figure 2. Consider the following mutual summary $C(\mathsf{Foo1}, \mathsf{Foo2})$ for this pair of procedures:

$$(\mathsf{x1} = \mathsf{x2} \wedge \mathsf{old}(\mathsf{g1}) \leq \mathsf{old}(\mathsf{g2})) \implies (\mathsf{g1} \leq \mathsf{g2})$$

The summary says that if the procedures Foo1 and Foo2 are executed in a state where the respective parameters are equal, and the global g1 is less than or equal to g2 (the old(.) used to denote the state at entry to the procedures), then the resulting state (if both procedures terminate) will satisfy $\mathsf{g1} \leq \mathsf{g2}$.

Instead of specifying the preconditions, we could have alternately strengthened the antecedent in the mutual summary with $\mathsf{x1} \geq 0$. We chose to use the precondition to demonstrate the use of single program contracts in checking mutual summaries.

3.2 Checking mutual summaries

In this section, we describe a modular method for checking a program pair (P^1, P^2) annotated with a set of mutual summaries C. This consists of a method for *guaranteeing* that a mutual summary $C(f^1, f^2)$ holds and a method for *assuming* the mutual summaries of smaller executions while modularly checking the mutual summaries.

First, we define the following augmented procedure $\textsc{MutualCheck}\langle f^1, f^2\rangle$ that defines how to check a mutual summary $C(f^1, f^2)$:

```
void $\textsc{MutualCheck}\langle f^1, f^2\rangle(\mathsf{x}_1 : \mathsf{int}, \mathsf{x}_2 : \mathsf{int})\{$
    inline $\mathsf{r}_1 :=$ call $f^1(\mathsf{x}_1)$;
    inline $\mathsf{r}_2 :=$ call $f^2(\mathsf{x}_2)$;
    assert $C(f^1, f^2)(\mathsf{x}_1, \mathsf{x}_2, \mathsf{old}(\mathsf{g}^1), \mathsf{old}(\mathsf{g}^2), \mathsf{r}_1, \mathsf{r}_2, \mathsf{g}^1, \mathsf{g}^2)$;
$\}$
```

The only new thing to be explained is the "inline" keyword used for inlining a procedure. Consider a procedure f in a program P with a precondition ϕ, and a postcondition ψ, and body f_{body}. When we use inline at a call site of f, the call is replaced by a assume $\phi; f_{body}$, with appropriate substitutions for the parameters of f.

Second, we define an uninterpreted summary predicate for each procedure f, and add it as a "free" postcondition for f. The "free" postconditions of a procedure are unchecked postconditions that are only assumed at call sites, but never asserted.

```
free post $R_f(\mathsf{x}, \mathsf{old}(\mathsf{g}), \mathsf{ret}, \mathsf{g})$
modifies g
int $f(\mathsf{x} : \mathsf{int}) : \mathsf{ret}$;
```

For the cases when the global g is not modified in the procedure f, one can remove the modifies clause on g and add a postcondition post $\mathsf{g} = \mathsf{old}(\mathsf{g})$.

Finally, we relate the uninterpreted summary predicates of a pair of procedures using an axiom[1] (ignore the expressions inside $\{.\}$ for now):

[1] An axiom in Boogie does not refer to any program state.

$$\begin{array}{l}\mathsf{axiom}(\\ \forall x_1, x_2, g_1, g_2, r_1, r_2, g_1', g_2' : \{R_{f^1}(x_1, g_1, r_1, g_1'), R_{f^2}(x_2, g_2, r_2, g_2')\}\\ \quad R_{f^1}(x_1, g_1, r_1, g_1') \wedge R_{f^2}(x_2, g_2, r_2, g_2') \implies\\ \quad C(f^1, f^2)[g_1/\mathsf{old}(g_1), g_2/\mathsf{old}(g_2), g_1'/g_1, g_2'/g_2]\\ );\end{array}$$

The axiom states that if we encounter a pair of terminating procedure calls f^1 and f^2, then we can assume the mutual summary $C(f^1, f^2)$ holds for the summaries of these two nested callees, while proving the mutual summaries of the callers. Observe that this is analogous to modular (single) program verification, where we assume the postconditions of callees while checking the contracts in the caller.

Theorem 1. *If P^1 and P^2 satisfy their single program contracts, and the contracts on each of the* MutualCheck$\langle f^1, f^2\rangle$ *procedures hold modularly for every pair of procedures $(f^1, f^2) \in P^1 \times P^2$, then the two programs satisfy the mutual summary specifications C.*

One may wonder whether the use of the quantified axiom above adds complexity to the resulting verification conditions, in particular when the contracts are expressed in a ground (quantifier-free) fragment. However, the presence of the *triggers* in the axiom (a list of expressions inside $\{.\}$, containing all the bound variables) ensure that the axioms only get instantiated at a bounded number of times. The trigger above instructs the SMT solver to instantiate this quantifier once for every pair of procedure calls of (f^1, f^2) in the verification condition, which is statically bounded.

4 Applications of mutual summaries

4.1 Equivalence checking

In various equivalence checking applications [3, 7], one checks if two procedures have identical input output behavior. For any two procedures (f^1, f^2) that are expected to be equal, the formula $C(f^1, f^2)$ can be automatically generated to be:

$$(\mathsf{x}_{f^1} = \mathsf{x}_{f^2} \wedge \mathsf{old}(\mathsf{g}_1) = \mathsf{old}(\mathsf{g}_2)) \implies \mathsf{ret}_{f^1} = \mathsf{ret}_{f^2} \wedge \mathsf{g}_1 = \mathsf{g}_2$$

There is usually an optimization for the purpose of equivalence checking: instead of using two relations for the summary of the two equivalent procedures, one can use an identical uninterpreted function for both the procedures. Using an identical uninterpreted function implicitly ensures the axiom above due to the *congruence* rule of functions, which ensures that the same input yields the same output.

On the other hand, use of an uninterpreted function restricts these analysis to deterministic procedures. The generality provided by using uninterpreted relations for a summary along with mutual summaries allows greater flexibility, without affecting the automation or efficiency needed for checking equivalence in the above cases.

```
int f1(int x1){
   if (Op[x1] == 0)
     return Val[x1];
   else if (Op[x1] == 1)
     return f1(A1[x1]) + f1(A2[x1]);
   else if (Op[x1] == 2)
     return f1(A1[x1]) - f1(A2[x1]);
   else
     return 0;
}
```

```
int f2(int x2, bool isU){
   if (Op[x2] == 0)
     return Val[x2];
   else if (Op[x2] == 1){
     if (isU)
        return  uAdd(f2(A1[x2], true),
                     f2(A2[x2], true));
     else
        return f2(A1[x2], false) +
               f2(A2[x2], false);
   }
   else if (Op[x2] == 2){
     if (isU)
       return uSub(f2(A1[x2], true),
                   f2(A2[x2], true));
     else
       return f2(A1[x2], false) -
              f2(A2[x2], false);
   }
   else
     return 0;
}
```

Fig. 3. Example for conditional equivalence.

4.2 Conditional equivalence checking

Bug fixes and feature additions result in two versions of a program that are behaviorally equivalent under a set of inputs. In *conditional equivalence* [4], the notion of equivalence can be extended to prove behavioral equivalence under a set of conditions (e.g. when the feature is turned off, or for non-buggy inputs). The key idea is to use identical uninterpreted functions for a procedure summary for inputs that would make the procedures behave the same; and use different uninterpreted functions for other inputs. We show how mutual summaries can be used for showing conditional equivalence, and more importantly helps address one of the problematic issues of using identical uninterpreted functions.

Figure 3 contains two versions of a procedure f (denoted as $\mathsf{f1}$ and $\mathsf{f2}$ respectively) that recursively evaluates an expression rooted at the argument x. The new version differs in functionality when an additional argument isU is provided that indicates "unsigned" arithmetic instead of the signed arithmetic represented by $\{+,-\}$.

The goal for conditional equivalence is to show that two versions of f are identical when isU is false. The following mutual summary $C(\mathsf{f1}, \mathsf{f2})$ can be used to ensure such a fact:

$$(\mathsf{x1} = \mathsf{x2} \wedge \neg \mathsf{isU}) \implies \mathsf{ret}_{\mathsf{f1}} = \mathsf{ret}_{\mathsf{f2}}$$

Earlier work [4] of using identical uninterpreted function for the two versions was problematic because of an additional parameter in the second version. Using an uninterpreted function of arity one would be unsound, as the return value of `f2` depends on `isU` parameter. On the other hand, using an uninterpreted function of arity two would require an user to intervene to provide the second argument when modeling `f1`'s summary.

4.3 Translation validation

Translation validation [8, 6] is a special case of equivalence checking, where the goal is to verify that a transformed program is equivalent to the original program. This is useful for verifying correct compilation of a program by a compiler. For example, a translation validator might compare a source program in a high-level language with the assembly language program generated by a compiler from the source program. Alternately, a translation validator might compare the a compiler's intermediate representation of the source program before and after each compiler phase.

This section describes how to use SymDiff to perform translation validation, focusing on the validation of compiler loop optimizations. The validation consists of three steps:

1. First, for each of the two versions of the program, loops (or, more generally, unstructured gotos) are transformed into recursive procedures, using user-provided or compiler-provided labels to guide the transformation.
2. Second, for each of the two versions of the program, calls to these recursive procedures may be inlined zero or more times to express the effect of loop optimizations such as loop unrolling.
3. Finally, mutual summaries are used to express the relation between the two versions of the program after loop extraction and inlining.

Figure 4 shows an example of three versions of a program: `Foo1` is the original program (a simple while loop), `Foo2` is an optimized version of `Foo1` (a do-while loop, with strength reduction applied to `v` and `t` hoisted from the loop), and `Foo3` is a heavily optimized version of `Foo1` (an unrolled do-while loop). The first two versions are taken from Necula [6], which describes a technique that validates the equivalence of these two versions, while the third version demonstrates loop unrolling, an optimization not handled by Necula [6]. In this example code, `g` and `n` are global constants, `a` is a global variable, and other variables are local variables. For clarity, we have replaced some of the memory references in the original example code with ordinary variable names. This simplification omits some reasoning about aliasing, but doesn't otherwise alter the nature of the example.

Loops and unstructured control SymDiff expects loops to be translated into recursive procedures. Compiler internal languages not only have loops, but may have unstructured (non-reducible) control flows built from arbitrary labels

```
int Foo1() {
  i := 0;
       //FUNCTION
  While1://LABEL
  if(i < n) {
    t := g;
    u := t * i;
    v := u + 3;
    a[i] := v;
    i := i + 1;
    goto While1;
  }
  return i;
}
```

```
int Foo2() {
  i := 0;
  if (n > 0) {
    t := g;
    v := 3;
    do2:
      a[i] := v;
      i := i + 1;
      v := v + t;
             //FUNCTION
      While2://LABEL
      if (i < n)
        goto do2;
  }
  return i;
}
```

```
int Foo3() {
  i := 0;
  v := 3;
  if (n > 1) {
    t := g;
    do3:
      a[i] := v;
      v := v + t;
      a[i + 1] := v;
      v := v + t;
      i := i + 2;
             //FUNCTION
      While3://LABEL
      if (i + 1 < n)
        goto do3;
  }
  if (i < n) {
    a[i] := v;
    i := i + 1;
  }
  return i;
}
```

Fig. 4. Translation validation example

and goto statements. To translate arbitrary unstructured control flow graphs into recursive procedures, we require the program to declare certain labels as special *function labels*. Each function label is used to generate one (possibly recursive) procedure, and each goto to that function label becomes a tail-recursive call to the procedure generated for that function label. To ensure that the resulting program has no loops, we require that every cycle in the original program pass through at least one function label.

In Figure 4, the labels `While1`, `While2`, and `While3` are marked as function labels, while the other labels (`do2` and `do3`) are ordinary local labels. In addition, the beginning of each procedure is implicitly marked with a function label. In this example, we have placed the function labels so that they match as closely as possible between the three versions of the program: in each version, the `While` function label appears directly before the loop condition. While this careful placement is not strictly necessary for performing the verification, it makes it easier to write the mutual summary formula that relates the states of the programs at the function labels.

Given a program where every cycle passes through a function label, the following simple algorithm transforms this program into a set of mutually recursive procedures. First, each function label becomes a procedure, whose parameters are all local variables and procedure parameters in scope. Second, the body for each procedure is the collection of statements reachable from that procedure's function label via paths that do not pass through a function label. Finally, each goto statement to a function label (or implicit fall-through to a function label) becomes a tail-recursive call to the procedure for that function label. Notice that in the second step, the same statements might be included separately in different procedures, if those statements are reachable from different function labels. In the worst case, each statement could be included in each generated procedure, so the worst-case size of the resulting program is the product of the original program size and the number of function labels.

Validating optimized code Figure 5 shows the result of translating Figure 4 into recursive procedures. For example, the new procedure `Foo1` consists of only the statements that are reachable from the beginning of the original `Foo1` procedure without passing through the function label `While1`. Some statements are duplicated between procedures; for example, `do2` is reachable from both the beginning of `Foo2` and from `While2`, and thus appears in both the new `Foo2` procedure and the new `While2` procedure. To avoid confusion between immutable procedure parameters and mutable local variables, we have introduced new local variables (e.g. `i1'`) for each procedure parameter that is modified by the procedure body.

We now describe how to use SymDiff to validate the code in Figure 5, specifically the equivalence of `Foo1`, `Foo2`, and `Foo3` and the equivalence of `While1`, `While2`, and `While3`.

To validate `While1` with `While2`, we need only relate the state of these procedures as follows:

$C(\mathsf{While1}, \mathsf{While2}) =$

$(\mathsf{i1} = \mathsf{i2} \wedge \mathsf{g} = \mathsf{t2} \wedge 3 + \mathsf{g} * \mathsf{i1} = \mathsf{v2} \wedge \mathsf{old}(\mathsf{a1}) = \mathsf{old}(\mathsf{a2})) \implies (\mathsf{r1} = \mathsf{r2} \wedge \mathsf{a1} = \mathsf{a2})$

Given this, SymDiff automatically checks that `While1` with `While2` are equivalent.

In contrast to the closely related `While1` and `While2`, Figure 5's `Foo1` and `Foo2` don't look similar at first glance. Nevertheless, we can make them comparable by inlining a copy of `While1` into `Foo1`:

```
int Foo1() {
  i1 := 0;
  i1' := i1;
  if(i1' < n) {
    t1 := g;
    u1 := t1 * i1';
    v1 := u1 + 3;
    a1[i1'] := v1;
    i1' := i1' + 1;
    return While1(i1');
  }
  return i1';
}
```

```
int Foo1() {
  i1 := 0;
  return While1(i1);
}


int While1(int i1) {
  i1' := i1;
  if(i1' < n) {
    t1 := g;
    u1 := t1 * i1';
    v1 := u1 + 3;
    a1[i1'] := v1;
    i1' := i1' + 1;
    return While1(i1');
  }
  return i1';
}
```

```
int Foo2() {
  i2 := 0;
  if (n > 0) {
    t2 := g;
    v2 := 3;
    a2[i2] := v2;
    i2 := i2 + 1;
    v2 := v2 + t2;
    return While2(
      i2, t2, v2);
  }
  return i2;
}


int While2(int i2,
           int t2,
           int v2) {
  i2' := i2;
  v2' := v2;
  if (i2' < n) {
    a2[i2'] := v2';
    i2' := i2' + 1;
    v2' := v2' + t2;
    return While2(
      i2', t2, v2');
  }
  return i2';
}
```

```
int Foo3() {
  i3 := 0;
  v3 := 3;
  if (n > 1) {
    t3 := g;
    a3[i3] := v3;
    v3 := v3 + t3;
    a3[i3 + 1] := v3;
    v3 := v3 + t3;
    i3 := i3 + 2;
    return While3(
      i3, t3, v3);
  }
  if (i3 < n) {
    a3[i3] := v3;
    i3 := i3 + 1;
  }
  return i3;
}


int While3(int i3,
           int t3,
           int v3) {
  i3' := i3;
  v3' := v3;
  if (i3' + 1 < n) {
    a3[i3'] := v3';
    v3' := v3' + t3;
    a3[i3 + 1] := v3';
    v3' := v3' + t3;
    i3' := i3' + 2;
    return While3(
      i3', t3, v3');
  }
  if (i3' < n) {
    a3[i3'] := v3';
    i3' := i3' + 1;
  }
  return i3';
}
```

Fig. 5. Translation validation example

After this inlining, the following mutual summary suffices to check the equivalence of `Foo1` and `Foo2`:

$$C(\mathsf{Foo1}, \mathsf{Foo2}) = (\mathsf{old(a1)} = \mathsf{old(a2)}) \implies (\mathsf{r1} = \mathsf{r2} \wedge \mathsf{a1} = \mathsf{a2})$$

(Currently, we rely on the user or compiler to specify the mutual summaries and to specify which procedure calls should be inlined. However, SymDiff completes the remaining equivalence checking automatically.)

The verification of `While2` to `While3` and `Foo2` to `Foo3` is similar: simply inline `While2` once into `Foo2` and inline `While2` once into itself, so that `While2` and `Foo2` express the same degree of loop unrolling that `While3` and `Foo3` express. (Note that the number of times inlining is performed will depend on the degree of loop unrolling.) Given this inlining and the following definitions, SymDiff automatically checks the equivalence of `While2` and `While3`, and `Foo2` and `Foo3`:

$C(\mathsf{Foo2}, \mathsf{Foo3}) = (\mathsf{old(a2)} = \mathsf{old(a3)}) \implies (\mathsf{r2} = \mathsf{r3} \wedge \mathsf{a2} = \mathsf{a3})$

$C(\mathsf{While2}, \mathsf{While3}) =$

$(\mathsf{i2} = \mathsf{i3} \wedge \mathsf{t2} = \mathsf{t3} \wedge \mathsf{v2} = \mathsf{v3} \wedge \mathsf{old(a2)} = \mathsf{old(a3)}) \implies (\mathsf{r2} = \mathsf{r3} \wedge \mathsf{a2} = \mathsf{a3})$

In summary, we can use SymDiff to check programs with loops for equivalence, even if the control flow is unstructured (non-reducible). In fact, in each case above, the SymDiff tool was able to check equivalence in just a few seconds. Nevertheless, this process is not entirely automated, since it relies on three pieces of information: the location of function labels, the desired inlining, and the definitions of mutual summaries. In the case of compiler validation, however, this information can be produced by the compiler, or, for common compiler optimizations, it might be automatically inferred, using techniques described by Necula [6].

In addition to the examples above, we have been able to encode the translation validation proofs of many common compiler optimizations using mutual summaries [**?**]. For example, it is possible to validate instances of software pipelining, loop peeling and restricted cases of loop fusion and loop fission. On the other hand, optimizations such as loop reversal, loop interchange that may change the order of updates to an array cannot be handled solely based on mutual summaries. We are currently investigating encoding the Permute rule [**?**] using mutual summaries; the rule has been needed to validate such transformations.

5 Conclusion

In this paper, we introduced mutual summaries as a mechanism for comparing programs and provided a method for checking them modularly using modern SMT solvers. Mutual summaries are general enough to encode many existing equivalence checking proofs, including those based on checking simulation relations for translation validation. More interestingly, it enables comparing programs with interprocedural changes. We are currently working on extending the technique to handle more complex interprocedural program transformations, and automating the generation of mutual summaries.

References

1. 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.
2. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
3. B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.
4. M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, 2010.
5. S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. In *Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.
6. 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.
7. S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.
8. 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.
9. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.
10. S. F. Siegel and T. K. Zirkel. Collective assertions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '11)*, LNCS 6538, pages 387–402, 2011.