Automated Differential Program Verification for Approximate Computing[*]

Ganesh Gopalakrishnan[1], Arvind Haran[1], Shuvendu K. Lahiri[2], and Zvonimir Rakamarić[1]

[1] School of Computing, University of Utah, UT, USA
{ganesh,haran,zvonimir}@cs.utah.edu
[2] Microsoft Research, Redmond, WA, USA
shuvendu@microsoft.com

Abstract. Approximate computing is an emerging area for trading off the accuracy of an application for improved performance, lower energy costs, and tolerance to unreliable hardware. However, care has to be taken to ensure that the approximations do not cause significant divergence from the reference implementation. Previous research has proposed various metrics to guarantee several relaxed notions of safety for the design and verification of such approximate applications. However, current approximation verification approaches often lack in either precision or automation. On one end of the spectrum, type-based approaches lack precision, while on the other, proofs in interactive theorem provers require significant manual effort.

In this work, we apply *automated differential program verification* (as implemented in SymDiff) for reasoning about approximations. We show that mutual summaries naturally express many relaxed specifications for approximations, and SMT-based checking and invariant inference can substantially automate the verification of such specifications. We demonstrate that the framework significantly improves automation compared to previous work on using Coq, and improves precision when compared to path-insensitive analysis. Our results indicate the feasibility of applying automated verification to the domain of approximate computing in a cost-effective manner.

1 Introduction

Continuous improvements in per-transistor speed and energy efficiency are fading, while we face increasingly important concerns of power and energy consumption, along with ambitious performance goals. The emerging area of *approximate computing* aims at lowering the computational effort (e.g., energy) of an application through controlled (small) deviations from the intended results. These deviations may be deliberately introduced by developers (e.g., selective use of lower energy ALUs or memories) or incidental by letting a non-hardened piece of electronics operate in a harsh environment (e.g., drones) and thus being subject

[*] Partially supported by NSF award CCF 1255776 and SRC contract 2013-TJ-2426.

to “bit flips.” Specific low-level mechanisms to reduce energy consumption and handle faults include approximating digital logic elements [9] or arithmetic [17], and exploiting hardware variability [16]. High-level mechanisms include program directives [28,21], approximating loop computations [43], generating multiple candidate implementations [1,19,45], and frameworks for automating approximate programming [33,14]. There are also approaches to recover from hardware faults through software-level recovery methods [23], and techniques to verify the quantitative reliability of programs [8]. In addition, many of these studies also show that large classes of applications can in fact tolerate small approximations (e.g., machine learning, web search, multimedia, sensor data processing).

There is a growing need to develop *formal* and *automated* techniques that allow approximate computing trade-offs to be explored by developers. Prior research has ranged from the use of types [40], static reliability analysis [8] or interactive theorem provers [7] to study the effects of approximations while also providing formal guarantees. While these techniques have significantly increased the potential to employ approximate computing in practice, a drawback is that they either lack the required level of precision or degree of automation. More importantly, these works do not harness the continuous advances in Satisfiability Modulo Theories (SMT) [4] based automatic software verification [2,29]. SMT-based approaches have the potential of providing a good balance of precision and scalability, without sacrificing automation, at least for a large class of programs written in imperative languages such as C/C++, Java, or C#.

In this paper, we apply *automated differential program verification* [26,18] (implemented in SymDiff [25]) towards the problem of *logical*[3] reasoning about program approximations. Previous work has shown that structural similarity of closely-related programs can be exploited to perform automated verification of relative safety for assertions [26]. Although formalisms based on *Relational Hoare Logic* [5] have been around for reasoning about relational properties, such verifiers are mostly based on interactive theorem provers (e.g., Coq [11]). This precludes leveraging automatic verification condition generation [3], SMT-based checking, and invariant inference. In this work, we unify two ideas in SymDiff to harness the power of SMT solvers towards differential verification. First, we use the concept of *mutual summaries* for specifying relational (two-program) properties related to approximation [18]. Second, we use a novel product program construction for *differential assertion checking* that permits procedural programs, and allows leveraging off-the-shelf program verifiers and invariant inference engines [26]. We describe how the construction can be used to check mutual summary specifications as well. The framework enables inference of simple relational invariants using a form of predicate abstraction in a scalable manner.

We have applied SymDiff towards two approximate computing case studies. First, we illustrate the modeling, specification, and proof of several acceptability conditions for approximate transformations studied by Carbin et al. [7]. They developed a domain-specific language for specifying approximations and

[3] We distinguish from approaches that provide provide probabilistic guarantees regarding approximations [8].

acceptability conditions, and performed the verification of several examples using interactive theorem prover Coq. These examples cover approximations due to truncating loops, unreliable memory, and relative memory safety [26]. Overall, their proofs for three examples required around 955 lines of Coq proof script — this makes it difficult to scale the effort to larger programs or hundreds of such programs. In contrast, our verification in SymDiff requires less than 10 lines of specifications. Second, we apply SymDiff towards formalizing a precise specification for ensuring that an approximation does not impact control flow of a program. Approximations that impact control flow often lead to undesirable behavior, such as non-termination or crashes. Unlike type-based approaches [40], we illustrate that SymDiff can provide verification of this property with desired precision and little overhead. We show examples where the need to track dynamic segments of arrays are crucial to enable precise reasoning about impact on control flow. We also analyze several small C programs to show the *potential* of path-sensitive analysis for improving the set of statements that can be safely approximated without impacting control flow.[4]

In the rest of the paper, we first describe the differential program verification in SymDiff; this includes combining the specification mechanism [18] and product program construction [26], originally developed separately (§ 2). We then describe our two case studies: applying SymDiff towards checking various approximation acceptability conditions [7] (§ 3), and towards checking impact of approximation on control flow (§ 4). We cover related work in the end (§ 5).

2 Differential Program Verification

In this section, we cover recent works on differential program verification [18,26] to verify (relational) properties over two programs, as implemented in the SymDiff tool [25]. We first describe *mutual summaries* [18] as a specification mechanism for relational properties (§ 2.2). Then, we introduce a method for modularly checking mutual summary specifications based on a product program transformation [26] (§ 2.3). Although the transformation was proposed for *differential assertion checking*, we show that the construction can be used to check more general mutual summary specifications. These mechanisms are well-suited for reasoning about programs with multiple (recursive) procedures. More importantly, the technique allows for leveraging any off-the-shelf invariant inference engine to infer intermediate specifications required to prove the desired specification (§ 2.4). We start by formalizing the language in the next section.

2.1 Simple Programming Language

Fig. 1 defines the syntax of our simple programming language, which is a subset of the Boogie language [3]. The language supports integers **int**, arrays [**int**]**int**,

[4] Examples and the tool binary are available at `http://1drv.ms/1ACV34H`; SymDiff source code is available at `http://symdiff.codeplex.com`.

$$\begin{aligned}
\mathit{Type} &::= \ \textbf{int} \ \mid \ [\textbf{int}]\textbf{int} \ \mid \ \textbf{bool} \\
\mathit{Program} &::= \ (\textbf{var}\ \mathit{Id} : \mathit{Type};)^{*}\ \mathit{Procedure}^{+} \\
\mathit{Procedure} &::= \ \textbf{procedure}\ \mathit{Id}((\mathit{Id} : \mathit{Type},)^{*})\ \mathit{Returns}^{?}\ \mathit{Spec}^{*}\ \{\mathit{Body}\} \\
\mathit{Spec} &::= \ \textbf{requires}\ \mathit{Expr}; \ \mid \ \textbf{ensures}\ \mathit{Expr}; \ \mid \ \textbf{modifies}\ (\mathit{Id},)^{*}; \\
\mathit{Returns} &::= \ \textbf{returns}\ ((\mathit{Id} : \mathit{Type},)^{*}) \\
\mathit{Body} &::= \ (\textbf{var}\ \mathit{Id} : \mathit{Type};)^{*}\ \mathit{Stmt} \\
\mathit{Stmt} &::= \ \mathit{Id} := \mathit{Expr} \ \mid \ \mathit{Id}[\mathit{Expr}] := \mathit{Expr} \ \mid \ \textbf{if}\ (\mathit{Expr})\ \ \mathit{Stmt}\ \ \textbf{else}\ \ \mathit{Stmt} \\
&\quad \mid \ \mathit{Stmt};\ \mathit{Stmt} \ \mid \ \textbf{havoc}\ \mathit{Id} \ \mid \ \textbf{call}\ (\mathit{Id},)^{*} := \mathit{Id}((\mathit{Id},)^{*}) \ \mid \ \textbf{return} \\
&\quad \mid \ \textbf{assume}\ \ \mathit{Expr} \ \mid \ \textbf{assert}\ \ \mathit{Expr}
\end{aligned}$$

Fig. 1: Simple programming language. *Id* and *Expr* have the usual meaning.

and booleans **bool**. A program consists of a set of global variables and a set of one or more procedures. A procedure has zero or more input parameters and output variables. The **requires** and **ensures** clauses specify preconditions and postconditions/summaries, respectively; the **modifies** clause specifies the globals that may be modified in a procedure. A procedure body contains local variable declarations and a sequence of statements *Stmt*. Loops are assumed to be already automatically extracted into deterministic tail-recursive procedures [26].

Most statements, including assignments (scalar and array), conditionals, and sequential composition, are standard. Statement **havoc** x sets variable x to an arbitrary value, while the **call** statement denotes a procedure invocation. Informally, the **assert** *Expr* (resp., **assume** *Expr*) *fails* (resp., *blocks*) execution when *Expr* evaluates to **false** in a state; otherwise, it acts as a skip. The expression language of *Expr* is left unspecified, but includes standard integer-valued arithmetic expressions (e.g., $x+y$) and Boolean-valued expressions (e.g., $x \leq y$). In addition, the construct **old**(*Expr*) can be used to evaluate an expression at entry to a procedure.

We informally sketch the semantics for the language here; more formal details can be found in earlier works [3]. A state σ of a program at a program location is a type-consistent valuation of variables in scope at the location, or the error state $Error$. Let Σ be the set of all states for a program. For any procedure p, we assume a transition relation $\mathcal{T}_p \subseteq \Sigma \times \Sigma$ that characterizes the input-output relation of the procedure p. In other words, two states $(\sigma, \sigma') \in \mathcal{T}_p$ if there is an execution of the procedure p starting at σ and ending in σ'. The transition relations can be defined inductively on the structure of the program, which is fairly standard for our simple language. For any state σ and an expression e, $\langle e \rangle_\sigma$ evaluates e in the state σ.

2.2 Specification: Mutual Summaries

Consider a program P and procedure p belonging to P. A *summary* specification S_p is a Boolean-valued expression over the input (parameters and globals) and

```
var g:int; // global
procedure F(x:int)
modifies g;
{
  if (x < 100) {
    g := g + x;
    call F(x+1);
  }
}
procedure Main()
modifies g;
{ call F(0); }
```

```
var g:int; // global
procedure F(x:int)
modifies g;
{
  if (x < 100) {
    g := g + 2*x;
    call F(x+1);
  }
}
procedure Main()
modifies g;
{ call F(0); }
```

Fig. 2: Two versions of a program with a change in `F`.

output (returns and globals) variables of p that specifies a constraint on the transition relation $\mathcal{T}_p$ of the procedure. More formally, a well-formed summary expression S_p induces a relation $\lfloor S_p \rfloor = \{(\sigma, \sigma') \mid \langle S_p \rangle_{\sigma,\sigma'} = \textbf{true}\}$. A procedure p *satisfies a summary* S_p if $\mathcal{T}_p \subseteq \lfloor S_p \rfloor$ — all terminating executions of p satisfy S_p. Consider either of the two `F` procedures in Fig. 2, and the expression $\mathtt{x} \geq 0 \Rightarrow (\mathtt{g} \geq \textbf{old}(\mathtt{g}))$. The expression is a well-formed summary specification for `F` since it only refers to the input state (`x` and $\textbf{old}(\mathtt{g})$) and output state (`g`) of `F`. Both procedures satisfy this summary because none of the terminating executions of either `F` decrease `g` when starting from a state where the parameter `x` is non-negative.

Now consider two procedures p, q. An expression $M_{p,q}$ is a well-formed *mutual summary* specification if it is an expression over the input and output variables of p and q. Such a specification represents the relation $\lfloor M_{p,q} \rfloor = \{(\sigma_p, \sigma'_p, \sigma_q, \sigma'_q) \mid (\sigma_p, \sigma'_p) \in \mathcal{T}_p, (\sigma_q, \sigma'_q) \in \mathcal{T}_q, \langle M_{p,q} \rangle_{\sigma_p,\sigma'_p,\sigma_q,\sigma'_q} = \textbf{true}\}$. A procedure pair (p, q) *satisfies a mutual summary* $M_{p,q}$ if $\mathcal{T}_p \times \mathcal{T}_q \subseteq \lfloor M_{p,q} \rfloor$.

Consider the two programs in Fig. 2, with a change in the procedure `F`, and the following mutual summary for `Main`:

$$\textbf{old}(\mathtt{v1.g} = \mathtt{v2.g}) \Rightarrow \mathtt{v1.g} \leq \mathtt{v2.g}.$$

The summary relates the pre- and post-states of the two versions (prefixed with `v1.` and `v2.` respectively) of the program. It is not difficult to see that the procedure pair (`v1.Main`, `v2.Main`) satisfies this mutual summary specification, since the argument `x` to `F` is always non-negative in executions starting from `Main`. In the next section, we describe how to specify and modularly verify such mutual summary specifications for a pair of programs.

2.3 Modular Checking of Mutual Summaries

We describe the modular checking of mutual summaries (using the construction in previous work [26]) with the aid of the running example in Fig. 2. We first sketch the product program construction for the running example, and later describe how to add specifications to the product program.

Product Programs For a program P, let us overload P to also represent the set of procedures in P. Consider two programs P_1, P_2, and a mapping relation $\beta \subseteq P_1 \times P_2$ that maps procedures from two versions. A default value of β is a one-to-one mapping between identically named procedures from the two programs, but this can be changed by the user. For our running example $P_1 = \{\mathtt{v1.F}, \mathtt{v1.Main}\}$ and $P_2 = \{\mathtt{v2.F}, \mathtt{v2.Main}\}$, and we consider the default mapping $\beta = \{(\mathtt{v1.Main}, \mathtt{v2.Main}), (\mathtt{v1.F}, \mathtt{v2.F})\}$.

Given such P_1, P_2 and β, we construct a product program $P_{1\times2}$ with the following properties:

- The set of globals in $P_{1\times2}$ is the disjoint union of globals in P_1 and P_2. The globals are prefixed with `v1.` and `v2.` respectively to avoid name clashes.
- The set of procedures in $P_{1\times2}$ consists of the disjoint union of procedures from P_1 and P_2 (signature suitably prefixed) along with a set of *product* procedures. For each $(p, q) \in \beta$, there is a procedure MS_p_q whose input and output parameters are concatenations of the parameter lists from p and q.

For the running example, $P_{1\times2}$ consists of globals $\{\mathtt{v1.g}, \mathtt{v2.g}\}$ and procedures $\{\mathtt{v1.F}, \mathtt{v1.Main}, \mathtt{v2.F}, \mathtt{v2.Main}, \mathtt{MS\_v1.F\_v2.F}, \mathtt{MS\_v1.Main\_v2.Main}\}$. Fig. 3 shows the details of the `MS_v1.F_v2.F` procedure. We briefly sketch the important components of this construction:

- Line 6 initializes a list of *call witness* variables, one per call-site within `v1.F` and `v2.F` respectively.
- Lines 8–14 inline the body of `v1.F`, whose statements are underlined. Each procedure call (e.g., line 11) is instrumented so that the inputs and outputs are recorded in local variables and the witness variable for the call-site is set.
- Lines 16–22 do the same for `v2.F`.
- Lines 24–31 are the most interesting part. First, we test using the witness variables if a pair of callees $(\mathtt{v1.F}, \mathtt{v2.F})$ has been executed. If so, we call the joint procedure for the callee-pair `MS_v1.F_v2.f` with the stored arguments and globals. The recursive call to `MS_v1.F_v2.f` in line 27 results from the recursive calls to `F` in the two versions. The assume statements after the call constrain the earlier output values of the two callees. Finally, the globals are restored back to the state before the recursive call to `MS_v1.F_v2.f`.

Let $\sigma_1 \oplus \sigma_2$ denote a composed state consisting of states from the two programs with disjoint signatures. The following theorem relates MS_p_q with the procedures p and q.

Theorem 1 ([26]). *For two procedures $p \in P_1$ and $p_2 \in P_2$, $(\sigma_1, \sigma_1') \in \mathcal{T}_p$ and $(\sigma_2, \sigma_2') \in \mathcal{T}_p$ if and only if $(\sigma_1 \oplus \sigma_2, \sigma_1' \oplus \sigma_2') \in \mathcal{T}_{\mathtt{MS}\_p\_q}$.*

```
procedure MS_v1.F_v2.F(v1.x:int, v2.x:int)
modifies v1.g, v2.g;
requires v1.x >= 0; // intermediate contract (manual)
{
  // initialize call witness variables
  v1.b_1, v2.b_1 := false, false;
  // v1
  if (v1.x < 100) {
    v1.g := v1.g + v1.x;
    v1.i_1, v1.gi_1 := v1.x + 1, v1.g; // store inputs
    call v1.F(v1.x + 1);
    v1.b_1 := true;     // record call
    v1.go_1 := v1.g;    // store outputs
  }
  // v2
  if (v2.x < 100) {
    v2.g := v2.g + 2*v2.x;
    v2.i_1, v2.gi_1 := v2.x + 1, v2.g; // store inputs
    call v2.F(v2.x + 1);
    v2.b_1 := true;     // record call
    v2.go_1 := v2.g;    // store outputs
  }
  // constrain calls
  if (v1.b_1 && v2.b_1) { // for pair of calls
    v1.st_g, v2.st_g := v1.g, v2.g; // store globals
    v1.g, v2.g := v1.gi_1, v2.gi_1;
    call MS_v1.F_v2.F(v1.i_1, v2.i_1);
    assume (v1.g == v1.go_1);  // constrain outputs
    assume (v2.g == v2.go_2);  // constrain outputs
    v1.g, v2.g := v1.st_g, v2.st_g; // restore globals
  }
  return;
}

procedure MS_v1.Main_v2.Main()
modifies v1.g, v2.g;
requires v1.g == v2.g; // globals start equal (automatic)
ensures  v1.g <= v2.g; // mutual postcondition (manual)
{
...
}
```

Fig. 3: Composed program for example in Figure 2. Underlined statements correspond to constituent procedures.

Adding Specifications Theorem 1 allows us to write summary specifications on the product program to capture mutual summary specifications over P_1 and P_2. Since the signature (inputs and outputs) of a product procedure $\texttt{MS\_}p\texttt{\_}q$ is the disjoint union of the signatures of p and q, a well-formed summary expression $S_{\texttt{MS\_}p\texttt{\_}q}$ is a well-formed mutual summary expression for the procedure pair (p, q). Hence, verifying summary specifications on the product program allows us to verify mutual summary specifications over the two programs.

Theorem 2. *Consider a product procedure* $\texttt{MS\_}p\texttt{\_}q \in P_{1\times 2}$ *and a summary specification* $S_{\texttt{MS\_}p\texttt{\_}q}$*. Moreover, let* $M_{p,q}$ *be a mutual summary specification such that* $\lfloor M_{p,q} \rfloor = \lfloor S_{\texttt{MS\_}p\texttt{\_}q} \rfloor$*. If* $\mathcal{T}_{\texttt{MS\_}p\texttt{\_}q} \subseteq \lfloor S_{\texttt{MS\_}p\texttt{\_}q} \rfloor$*, then* $\mathcal{T}_p \times \mathcal{T}_q \subseteq \lfloor M_{p,q} \rfloor$*.*

Recall the mutual summary specification for the pair of `Main` procedures. This can be expressed as a summary for `MS_v1.Main_v2.Main` procedure as a postcondition (**ensures** clause):

$$\textbf{ensures}(\textbf{old}(\texttt{v1.g} = \texttt{v2.g}) \Rightarrow \texttt{v1.g} \leq \texttt{v2.g}).$$

Fig. 3 shows the above specification; however, we have broken up the specification into a **requires** (a precondition constraining the state at entry) and **ensures** clause to simplify the specification. SymDiff automatically inserts the equalities in the **requires** for entry procedures, and the user only has to specify the **ensures** clause in this case.

The program $P_{1\times 2}$ along with the desired specification can be handed off to any off-the-shelf (single) program verifier such as Boogie to attempt the verification. The verifier can leverage advances in SMT solvers to perform reliable and predictable verification. If verification succeeds, then we can establish the mutual summary for the pair of procedures.

2.4 Invariant Inference

Currently, SymDiff performs an automatic inference of simple relative specifications by searching for preconditions and postconditions of the form $\texttt{v1.x} \bowtie \texttt{v2.x}$, where $\bowtie \in \{=, \leq, \geq, <, >, \Rightarrow\}$. It leverages the implementation of the Houdini [15] (monomial predicate abstraction) inference technique available in Boogie [27]. This allows SymDiff to communicate domain-specific (mutual specifications) to the invariant inference engine. Using this inference technique, we are able to infer many intermediate invariants for many realistic examples, as we discuss later in the paper. As invariant inference in Boogie matures (e.g. to incorporate interpolants [29]), the relative specification inference will also mature.

Houdini allows us to automatically infer several intermediate specifications, including inferring that $\texttt{v1.g} \leq \texttt{v2.g}$ is both a precondition and postcondition for `MS_v1.F_v2.F`. In addition, we required $\texttt{v1.x} \geq 0$ as a non-relational (single program) precondition for `MS_v1.F_v2.F` to capture that both `v1.F` and `v2.F` are always invoked with a non-negative parameter (see Fig. 3). We hope to automatically infer such facts in the future once simple abstract domains such as intervals [12] can be leveraged.

```
function RelaxedEq(x:int, y:int) returns (bool) {
  (x <= 10 && x == y) || (x > 10 && y >= 10)
}
procedure swish(max_r:int, N:int) returns (num_r:int) {
  old_max_r := max_r; havoc max_r; assume RelaxedEq(old_max_r, max_r);
  num_r := 0;
  while (num_r < max_r && num_r < N) num_r := num_r + 1;
  return;
}
```

Fig. 4: Swish++ example. Underlined statements introduce the approximation.

3 Case Study: Acceptability of Approximate Programs

In this section, we illustrate the application of differential verification towards three examples from a recent work by Carbin et al. [7]. The authors developed a special-purpose language to specify the transformations and reason about the relaxed specifications. They used the general purpose Coq theorem prover [11] to discharge proof obligations; each proof required roughly 300 lines of proof scripts according to the authors. By leveraging existing program verifiers and SMT solvers, we show that we can obtain the proofs almost completely automatically. In all cases, the final verification takes less than a second.

3.1 Dynamic Knobs

Fig. 4 gives the example from an open-source search engine *Swish++*. The approximation (referred to as Dynamic Knobs) allows the search engine to trade-off the number of search results to display to the user under heavy server load. The approximation is justified as users are typically interested in the top few results, and care more about the performance of displaying the search results. The program `swish` takes as input (a) a threshold for the maximum number of results to display `max_r`, and (b) the total number of search results `N`. It returns the number `num_r` denoting the actual number of results to display, which has to be bounded by `max_r` and `N`.

Approximation The underlined statements denote the approximation that nondeterministically changes the threshold to a possibly smaller number, without suppressing the top few (10 in this case) results. The predicate `RelaxedEq` denotes the relationship between the original and the approximate value — the important part is that approximate value has to be at least 10 when the original value exceeds 10.

Relaxed Specification The relaxed specification (akin to *acceptability property* [7,36]) can be expressed as a mutual summary over the original and approximate version (prefixed with `v1.` and `v2.` respectively) of `swish` as follows:

$$\textbf{old}(\texttt{v1.max\_r} = \texttt{v2.max\_r} \wedge \texttt{v1.N} = \texttt{v2.N}) \Rightarrow \texttt{v1.RelaxedEq}(\texttt{v1.num\_r}, \texttt{v2.num\_r}).$$

```
function A(i:int, j:int) returns (int);
const e:int; axiom e >= 0;
function RelaxedEq(x:int, y:int) returns (bool) {
  x <= y + e && y <= x + e
}
procedure lu(j:int, N:int, max0:int) returns (max:int, p:int) {
  i := j+1; max := max0;
  while (i < N) {
    a := A(i, j);
    old_a := a; havoc a; assume RelaxedEq(old_a,a);
    if (a > max) { max := a; p := i; }
    i := i + 1;
  }
  return;
}
```

Fig. 5: LU decomposition. Underlined statements introduce the approximation.

As described earlier in §2.3, the user only has to specify the postcondition

$$\textbf{ensures}\ \texttt{v1.RelaxedEq}(\texttt{v1.num\_r}, \texttt{v2.num\_r})$$

for MS_v1.swish_v2.swish, since the antecedent (with equalities) is already present for the top-level procedures.

Verification We required four additional intermediate specifications (beyond the relaxed specification) for the proof. Recall that loops are automatically extracted as tail-recursive procedures in SymDiff. The additional intermediate specifications are (a) the relational expression v1.RelaxedEq(v1.num_r, v2.num_r) as both **requires** and **ensures** for the product of the two loop-extracted procedures, and (b) (non-relational) postcondition that each loop-extracted procedure does not decrease the num_r variables across execution of the loop. All the remaining invariants are automatically inferred. In comparison, the Coq proof comprised of 330 lines of proof script.

3.2 Approximate Memory and Data Type

Fig. 5 gives a portion of the *LU Decomposition* algorithm implemented in SciMark2 benchmark suite [41]. The algorithm computes the index of the pivot row p for a column j, where the pivot row contains the maximum value among all rows in the column. It returns the index p of the pivot in addition to the value of the maximum element in column j.

Approximation The underlined statements model the introduction of an error value e if the matrix is stored in approximate memory [32]. As before, the predicate RelaxedEq denotes the relationship between the original and approximate value read from the memory; in this case, they are bounded by a non-negative constant e.

Relaxed Specification Similar to Swish++, the relaxed specification for the pair of `lu` procedures is specified by the postcondition on `MS_v1.lu_v2.lu`:

$$\textbf{ensures}\ \texttt{RelaxedEq}(\texttt{v1.max}, \texttt{v2.max}).$$

Verification Similar to the previous example, the only additional intermediate specifications are the expression `v1.RelaxedEq(v1.max, v2.max)` as both **requires** and **ensures** for the product of the two loop-extracted procedures. Remaining invariants are automatically inferred. In comparison, the Coq proof comprised of 315 lines of proof script.

3.3 Statistical Automatic Parallelization

```
1  var FF, RS:[int]int;
2  var K:int;
3  function exp(int) returns (int);
4
5  procedure water(len_FF:int,
6  len_RS:int, N:int, gCUT2:int) {
7    K := 1;
8    havoc RS; // approximation
9
10   while (K < N) {
11     assert (K < len_FF);
12     assert (K < len_RS);
13     if (RS[K] < gCUT2) {
14       // assert (K < len_FF);
15       FF[K] := exp(RS[K]);
16     }
17     K := K + 1;
18   }
19 }
```

Fig. 6: Water example.

Fig. 6 gives an example from a parallelization of the Water computation [6]. In the loop, the result of `RS[K]` is compared with a cutoff `gCUT2`, and then another array `FF` is updated at index `K`. The bounds of the two arrays are provided in the `len_RS` and `len_FF` variables.

Approximation To maximize performance, the parallelization eliminates locks, which can result in race conditions for the array `RS`. This is modeled by **havoc**-ing the entire array `RS`.

Relaxed Specification The assertions model memory safety, and ensure that the program accesses the two arrays within bounds. The relaxed specification has to ensure *relative memory safety* — that the assertions in the approximate version do not fail more often than the original version.

We exploit the formalization of *differential assertion checking* [26], which automatically replaces an assertion **assert** ϕ with an update to a global variable $\texttt{OK} := \texttt{OK} \wedge \phi$, and inserts a mutual postcondition $\texttt{v1.OK} \Rightarrow \texttt{v2.OK}$ when starting from equal states. Hence, we did not have to make any changes to define the relative specification.

Although it is desirable to check the assertion in line 14 relatively, the approximate version is not relatively correct with this assertion (also mentioned by Carbin et al. [7]). We instead prove the weaker assertion in line 11 that essentially expresses that `len_FF` is not correlated with the value in array `RS`.

Verification We verified this example completely automatically with SymDiff — all intermediate invariants are automatically inferred. In comparison, the Coq proof comprised of 310 lines of proof script.

4 Case Study: Control Flow Equivalence

```
var arr:[int]int;
var n:int; var x:int; var y:int;
procedure ReplaceChar() {
  call Helper(0);
}
procedure Helper(i:int) {
  var tmp:int;
  if (i < n && arr[i] != -1) {
    tmp := arr[i];
    havoc tmp;
    arr[i] := tmp == x ? y : tmp;
    call Helper(i+1);
  }
}
```

Fig. 7: Replacing a character in a string.

Approximating statements that impact control flow often leads to serious problems in guaranteeing program termination, unacceptably high corruptions in output data, and program crashes. Preservation of control flow has been identified as a natural and useful relaxed specification for approximations [40]. One can obtain a conservative estimate of the set of statements that do not affect control flow by performing type-based analysis [40], dataflow analysis [35], or slicing [20]. Although mostly automatic (type-based analyses typically require user-provided type information), these approaches are conservative and cannot exploit detailed program semantics.

Consider the program in Fig. 7 that replaces a given character `x` with `y` in a character array `arr`. The procedure `Helper` iterates over indices of the array until the bound `n` or the termination character (-1 in this case) is reached. Let us consider the approximation of the variable `tmp` indicated by the underlined statement. Since `tmp` flows into `arr` which controls the conditional, most mentioned conservative analysis would mark the approximation as unsafe. However, observe that the indices that store the value in `tmp` never participate in the conditional. Therefore, any analysis that cannot track dynamic segments of an array will result in a false alarm. Similarly, lack of path-sensitivity can also result in such imprecisions (see § A.1).

4.1 Modeling Control Flow

In this section, we describe a precise and automated (although not push-button) approach to ensure that an approximation does not impact control flow by leveraging differential program verification. We achieve this by performing an automatic program instrumentation (described below) and then leveraging the differential verifier as described earlier in § 2.

Let us define a *basic block* to be the maximal sequence of statements that do not contain any conditional statements. We also assume that each such basic block has a unique identifier associated with it. To track the sequence of basic blocks visited along any execution, we augment the state of a program by introducing an integer-valued global variable `cflow`. Then, we instrument every basic block of the program with a statement of the form `cflow` := `trackCF`(`cflow`, `blockID`), where `trackCF` is an uninterpreted function defined as `trackCF(int, int) returns int` and `blockID` is the unique integer identifier of the current basic block.

Let `v1.p` and `v2.p` be the two versions of a procedure `p` in the original and the approximate program. Consider the following mutual summary for the product procedure `MS_v1.p_v2.p` (assuming the inputs including `cflow` start out equal):

$$\textbf{ensures}\ \texttt{v1.cflow} == \texttt{v2.cflow}.$$

If the product program satisfies this mutual specification, then the injected approximations do not change the control flow of the program. More formally, if `MS_v1.p_v2.p` satisfies this specification, then the following holds:

> For any pair of executions $(\sigma_1, \sigma_2) \in \mathcal{T}_{v1.p}$ and $(\sigma_1, \sigma_3) \in \mathcal{T}_{v2.p}$ starting at the same input state σ_1, the sequences of basic blocks in the two executions are identical.

Hence, we translate the problem of determining if a set of approximations impacts control flow to the problem of verifying a mutual summary on the product program. Note that the formalism currently does not detect non-termination introduced in the approximation; we plan to leverage the *relative termination* specifications in future work [18].

We have verified that the approximation in Fig. 7 does not impact control flow. In addition to the mutual specification on the procedures, we needed the following mutual precondition on `MS_v1.Helper_v2.Helper`:

$$\textbf{requires}\ (\forall j : \textbf{int} :: j \geq \texttt{v1.i} \Rightarrow \texttt{v1.arr}[j] == \texttt{v2.arr}[j]).$$

The specification captures the fact that two versions of `Helper` are invoked with identical array fragments, namely $\texttt{arr}[\texttt{i}, \ldots, \texttt{n}]$. For any value of `i` on entry to `Helper` in either versions, executions only inspect this fragment of the array to determine branch conditions. Thus, with little user effort, SymDiff is successfully able to (soundly) verify that the statement can be safely approximated.

4.2 C Benchmarks

We have also performed preliminary experiments to determine the *feasibility* of using differential analysis to infer the set of approximations that do not impact control flow. The main objective is to compare our differential analysis with a more traditional *taint* analysis [13]. The taint analysis for Boogie programs (also implemented in SymDiff) checks if a statement lies in the slice of any of the conditionals using interprocedural dataflow analysis. To develop an automated differential analysis (Diff-Inline), we inline procedure calls (up to a small bound, say 10) before checking the mutual summary specifications — this leads to an unsound analysis in the presence of loops and recursion. In essence, the taint analysis and Diff-Inline provide respectively a lower and upper bound on the number of statements that can be safely approximated without impacting control flow. We have chosen a set of 9 realistic C programs including sorting,

Table 1: Experimental results for control flow equivalence. LOC is the number of lines of code; #P is the number of procedures; #Locs. is the number of program locations that could potentially be approximated; #Inline is the chosen inlining bound; #Approx. is the reported number of locations that can be approximated (i.e., those that do not affect control flow when approximated); Time is cumulative runtime in minutes.

Benchmark	LOC	#P	#Locs.	Diff-Inline			Taint	
				#Inline	#Approx.	Time	#Approx.	Time
Insertion Sort	24	2	13	10	1	1.3	1	1.1
Bubble Sort	25	2	13	10	1	1.6	1	0.8
Selection Sort	30	2	15	10	2	1.8	1	1.9
Brightness Correction	21	1	8	10	4	1.4	4	0.4
Arithmetic Mean Filter	27	1	13	10	5	1.3	5	2.5
Centroid Computation	55	3	30	10	14	8.3	14	3.3
Matrix Multiplication	38	3	17	16	7	5.4	7	2.9
Linked List Operations	76	5	40	6	7	55.4	2	0.8
Array Map Operations	78	7	35	10	12	115.4	3	2.5

image processing [30], data structure implementations, and operations on matrices. Table 1 summarizes our benchmarks. We first translate them into Boogie programs using the HAVOC verifier [10]. In our experiments, an approximation is modeled as just a **havoc** statement (of the appropriate variable) introduced at every program location of interest (i.e., variable assignments). We then establish control flow equivalence for every such approximation in turn, and we report total cumulative runtimes.

Table 1 presents the results of our experiments. Overall, benchmarks from the domain of image processing are most amenable to approximations that do not affect control flow, since most computations are local to a pixel neighborhood. As expected, the inlining-based approach scales poorly compared to the modular taint analysis. However, it is encouraging to see that the differential analysis improves the precision on three benchmarks. Among these, we have studied the `SelectionSort` example in detail, and have applied the sound differential verification on it (see § A.2). The imprecision of taint analysis is caused by the same reason (unable to distinguish segments of an array) we encountered in the `ReplaceChar` example from Fig. 7. Verification of the mutual summary specification required 7 more intermediate invariants, mainly due to the presence of nested loops. The example illustrates the robustness of our differential approach to verify more complex examples, albeit with a little more user effort.

5 Related Work

A number of complementary approaches have been recently proposed to reason about approximations. These approaches can be roughly categorized (with overlaps) into (a) language based, (b) static analysis, and (c) dynamic approaches.

Language based approaches propose language constructs and annotations to make approximations explicit in a program [40,7]. EnerJ [40] introduces approximate types and ensures that such values do not impact precise computations, including conditional statements. Our work can be used to improve the precision of the type-based analysis, as demonstrated in § 4. Carbin et al. [7] describe language constructs for introducing approximations and relaxed specifications, and prove correctness of transformations using Coq. We show that mutual summaries and SMT-based verification can significantly improve the automation for most transformations covered by this approach.

Rely [8] performs static quantitative reliability analysis to provide probabilistic guarantees on the impact of approximations on overall behavior of a program. We believe that SymDiff can be augmented with this framework to create an automated framework for improving precision using relative invariants. ExPAX [33] is a framework that generates a set of safe-to-approximate operations based on a dataflow taint analysis. It develops an algorithm to compute the level of approximation for each operation in the set so that energy consumption is minimized and reliability constraints are satisfied.

Among dynamic approaches, fault injection at the source or intermediate representation level has been used to profile the sensitivity of output quality to approximations. Fault injectors such as KULFI [42], LLFI [44], and PDSFIS [22] approximate instructions at runtime. Though these techniques achieve high levels of accuracy, they provide no formal coverage guarantees, unlike our technique. Offline dynamic analysis techniques provide information on dataflow and correlation difference (e.g., [37,38]). The former may be imprecise as it is based on static dataflow analysis, while the latter again does not provide formal guarantees. Although there are optimizations for selective instruction perturbation, such as the statistical methods [39], the reasoning is only for a subset of all the possible executions of the program.

Finally, our work is closely related to previous works on translation validation [34,31] that validate equivalence-preserving intraprocedural compiler transformations, using lock-step symbolic execution and SMT solvers. However, mutual summaries and the product construction allows for richer relaxed specifications other than equivalence, interprocedural reasoning [18], and leveraging off-the-shelf program verifiers and inference engines.

6 Conclusions

In this paper, we have described the application of automated differential verification for providing formal guarantees of approximations. The structural similarity between original and approximate programs are leveraged to automate most intermediate relative specifications. We are currently working on exploiting more powerful invariant inference engines such as interpolants [29] and indexed predicate abstraction [24] to infer remaining specifications. We would also like to leverage the concept of relative termination [18] to improve on the partial correctness guarantees of mutual summaries.

References

1. Baek, W., Chilimbi, T.M.: Green: A framework for supporting energy-conscious programming using controlled approximation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 198–209 (2010)
2. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 203–213 (2001)
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects (FMCO). pp. 364–387 (2006)
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: International Workshop on Satisfiability Modulo Theories (SMT) (2010)
5. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 14–25 (2004)
6. Blume, W., Eigenmann, R.: Performance analysis of parallelizing compilers on the perfect benchmarks programs. IEEE Trans. Parallel Distrib. Syst. 3(6), 643–656 (Nov 1992)
7. Carbin, M., Kim, D., Misailovic, S., Rinard, M.C.: Proving acceptability properties of relaxed nondeterministic approximate programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 169–180 (2012)
8. Carbin, M., Misailovic, S., Rinard, M.C.: Verifying quantitative reliability for programs that execute on unreliable hardware. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). pp. 33–52 (2013)
9. Chakrapani, L.N., George, J., Marr, B., Akgul, B.E.S., Palem, K.V.: Probabilistic design: A survey of probabilistic CMOS technology and future directions for terascale IC design. In: International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC). pp. 101–118 (2006)
10. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 19–33 (2007)
11. The Coq proof assistant. `http://coq.inria.fr`
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL). pp. 238–252 (1977)
13. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (May 1976)
14. Esmaeilzadeh, H., Sampson, A., Ceze, L., Burger, D.: Neural acceleration for general-purpose approximate programs. Commun. ACM 58(1), 105–115 (Dec 2014)
15. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: International Symposium of Formal Methods Europe (FME). pp. 500–517 (2001)
16. Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N., Gupta, R.K., Kumar, R., Mitra, S., Nicolau, A., Rosing, T.S., Srivastava, M.B., Swanson, S., Sylvester, D.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. on CAD of Integrated Circuits and Systems 32(1), 8–23 (2013)

17. Han, J., Orshansky, M.: Approximate computing: An emerging paradigm for energy-efficient design. In: IEEE European Test Symposium (ETS). pp. 1–6 (2013)
18. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Towards modularly comparing programs using automated theorem provers. In: International Conference on Automated Deduction (CADE). pp. 282–299. Springer (2013)
19. Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., Rinard, M.: Dynamic knobs for responsive power-aware computing. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 199–212 (2011)
20. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12(1), 26–60 (1990)
21. Hukerikar, S., Diniz, P., Lucas, R.: A programming model for resilience in extreme scale computing. In: IEEE Dependable Systems and Networks Workshops (DSN-W). pp. 1–6 (Jun 2012)
22. Jin, A., Jiang, J., Hu, J., Lou, J.: A pin-based dynamic software fault injection system. In: Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for. pp. 2160–2167. IEEE (2008)
23. de Kruijf, M., Nomura, S., Sankaralingam, K.: Relax: An architectural framework for software recovery of hardware faults. In: ACM SIGARCH Computer Architecture News. vol. 38, pp. 497–508. ACM (2010)
24. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Log. 9(1) (2007)
25. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SymDiff: A language-agnostic semantic diff tool for imperative programs. In: International Conference on Computer Aided Verification (CAV). pp. 712–717 (2012)
26. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 345–355 (2013)
27. Lahiri, S.K., Qadeer, S., Galeotti, J.P., Voung, J.W., Wies, T.: Intra-module inference. In: International Conference on Computer Aided Verification (CAV). pp. 493–508 (2009)
28. Liu, S., Pattabiraman, K., Moscibroda, T., Zorn, B.G.: Flikker: Saving DRAM refresh-power through critical data partitioning. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 213–224 (2011)
29. McMillan, K.L.: An interpolating theorem prover. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 16–30 (2004)
30. Myler, H.R., Weeks, A.R.: The pocket handbook of image processing algorithms in C. Prentice Hall Press (2009)
31. Necula, G.C.: Translation validation for an optimizing compiler. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 83–94 (2000)
32. Nelson, J., Sampson, A., Ceze, L.: Dense approximate storage in phase-change memory. In: Ideas and Perspectives session at ASPLOS (2001)
33. Park, J., Ni, K., Zhang, X., Esmaeilzadeh, H., Naik, M.: Expectation-oriented framework for automating approximate programming. In: Workshop on Approximate Computing Across the System Stack (WACAS) (2014)

34. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 151–166 (1998)
35. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 49–61 (1995)
36. Rinard, M.: Acceptability-oriented computing. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 221–239 (2003)
37. Ringenburg, M.F., Sampson, A., Ackerman, I., Ceze, L., Grossman, D.: Dynamic analysis of approximate program quality. Tech. Rep. UW-CSE-14-03-01, University of Washington
38. Ringenburg, M.F., Sampson, A., Ceze, L., Grossman, D.: Profiling and autotuning for energy-aware approximate programming. In: Workshop on Approximate Computing Across the System Stack (WACAS) (2014)
39. Roy, P., Ray, R., Wang, C., Wong, W.F.: ASAC: Automatic sensitivity analysis for approximate computing. In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES). pp. 95–104 (2014)
40. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: EnerJ: Approximate data types for safe and general low-power computation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 164–174 (2011)
41. SciMark 2.0. `http://math.nist.gov/scimark2`
42. Sharma, V.C., Haran, A., Rakamarić, Z., Gopalakrishnan, G.: Towards formal approaches to system resilience. In: IEEE Pacific Rim International Symposium on Dependable Computing (PRDC). pp. 41–50 (2013)
43. Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., Rinard, M.C.: Managing performance vs. accuracy trade-offs with loop perforation. In: Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 124–134 (2011)
44. Thomas, A., Pattabiraman, K.: LLFI: An intermediate code level fault injector for soft computing applications. In: Workshop on Silicon Errors in Logic System Effects (SELSE) (2013)
45. Zhu, Z.A., Misailovic, S., Kelner, J.A., Rinard, M.: Randomized accuracy-aware program transformations for efficient approximate computations. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 441–454 (2012)

A Additional Control Flow Equivalence Examples

A.1 Example Requiring Path-Sensitivity

In the example in Fig. 8, the approximation does not affect control flow since the sign of x is preserved, and the control flow only depends on the sign. We had to specify the following mutual precondition for the loop-extracted procedure:

$$\mathtt{v1.x} > 0 \Leftrightarrow \mathtt{v2.x} > 0.$$

A.2 Selection Sort

For proving the selection sort example in Fig. 9, the following 7 manual invariants are required:

- Two non-relational invariants to indicate that loops in Find do not decrease d variables.
- Two non-relational invariants to indicate that loops in Find do not decrease position when the initial value of position is bound by d.
- Three relational invariants (for the two loops and Find) to indicate that values of position, andcflow only depend on a sub-fragment of the array.

```
var x:int;
var y:int;
var n:int;

procedure foo() returns (r:int) {
  r := 0;
  y := x + 3;
  havoc y; assume y != 0;
  y := y * y;
  x := x * y;

  while (0 < n && n < 1000) {
    if (x > 0) {
      n := n + 1;
      r := r + n;
    } else {
      n := n - 1;
      r := r - n;
    }
  }
  return;
}
```

Fig. 8: Control flow equivalence example requiring path-sensitivity. Underlined statements introduce the approximation.

```
var array:[int]int;
var n:int;

procedure SelectionSort() {
  var c:int, position:int, temp:int;
  position := 0;
  temp := 0;
  c := 0;

  while (c < (n - 1)) {
    call position := Find(c);
    if (position != c) {
      temp := array[position];
      array[position] := array[c];
      havoc temp;
      array[c] := temp;
    }
    c := c + 1;
  }
}

procedure Find(c:int) returns (position:int) {
  var d:int;
  position := c;
  d := c + 1;
  while (d < n) {
    if (array[position] > array[d]) {
      position := d;
    }
    d := d + 1;
  }
}
```

Fig. 9: Selection sort control flow equivalence example. Underlined statements introduce the approximation.