Verifying Relative Safety, Accuracy, and Termination for Program Approximations

Shaobo He[1], Shuvendu K. Lahiri[2], and Zvonimir Rakamarić[1]

[1] University of Utah, Salt Lake City, UT, USA
[2] Microsoft Research, Redmond, WA, USA

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, developers must ensure that the leveraged approximations do not introduce significant, intolerable divergence from the reference implementation, as specified by several established robustness criteria. In this work, we show the application of automated differential verification towards verifying relative safety, accuracy, and termination criteria for a class of program approximations. We use mutual summaries to express *relative* specifications for approximations, and SMT-based invariant inference to automate the verification of such specifications. We perform a detailed feasibility study showing promise 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 and runtime) of an application through controlled (small) deviations from the intended results [12, 30, 28, 31]. These studies illustrate a large class of applications (e.g., machine learning, web search, multimedia, sensor data processing) that can tolerate small approximations without significantly compromising quality. Low-level approximation mechanisms include, for example, approximating digital logic elements, arithmetic, or sensor readings; high-level mechanisms include approximating loop computations, generating multiple approximate candidate implementations, or leveraging neural networks.

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 type systems [31], to static analyses [6], and interactive theorem provers [5] to study the effects of approximations while also providing various correctness guarantees. While these techniques have significantly increased the potential to employ approximate computing in practice, a drawback is that they often either lack the required level of precision or degree of automation.

In this work, we describe the application of SMT-based (Satisfiability Modulo Theories [2]) automated *differential program verifiers* [3, 11] for specifying and verifying properties of approximations. Such verifiers (e.g., SymDiff [14, 15]) leverage SMT

```
var src:[int]int, srcLen:int;
var dst:[int]int, dstLen:int;

procedure Strcpy() {
  var i:int;
  i := 0;

  while(src[i] != 0) {
    assert i<srcLen && i<dstLen;
    dst[i] := src[i];
    i := i + 1;
  }
  dst[i] := 0;
}
```

```
procedure StrcpyApprox() {
  var i:int; var j:int;
  i := 0; j := 0;

  while(src[i] != 0) {
    assert i<srcLen && j<dstLen;
    dst[j] := src[i];
    i := i + 1; j := j + 1;
    if (src[i] == 0) { break; }
    i := i + 1;
  }
  dst[j] := 0;
}
```

Fig. 1. Approximating string copy.

solvers to check assertions and semi-automatically infer intermediate program invariants over a pair of programs. We describe three broad classes of approximation robustness criteria that are amenable to SMT-based automated checking: *relative safety*, *relative accuracy*, and *relative termination*. Relative safety criteria ensure that approximations preserve a set of generic (program agnostic) properties. For example, *relative assertion safety* [5, 15] ensures that the approximation does not introduce any new assertion failures over the base program (e.g., it is desirable to ensure that an approximation does not introduce an array out of bound access). Similarly, *relative control flow safety* ensures that the approximation does not influence the control flow of a program [31]. Relative accuracy criteria specify the acceptable difference between precise and approximate outputs for specific approximations [5]. In addition to these established criteria, we propose the concept of relative termination [11, 8] as another important (program-agnostic) criterion for ensuring robustness of approximations. Intuitively, relative termination ensures that the approximation (such as loop perforation) does not change a terminating execution to a non-terminating one. We illustrate these on a few concrete examples next.

1.1 Motivating Examples

Relative Assertion Safety Fig. 1 describes two implementations of a string copy procedure: `Strcpy` is the precise version and `StrcpyApprox` is the approximate one. The approximate version implements a variant of *loop perforation* (a well-known approximation technique [19]) that only copies every other element from `src` to `dst`. The changes are highlighted using the underlined statements. The original program scans the `src` array until a designated end marker (`0` in this example) is encountered, and copies the elements to the `dst` array. The approximation introduces a fresh index variable `j` for indexing `dst` and increments `i` twice every iteration (unless the loop exit condition is true).

The memory safety of the program is ensured by a set of implicit assertions that guard for out-of-bound access of the arrays (e.g., **assert** `i < srcLen` before the access `src[i]`) — we only show a subset of assertions in the example. The bounds `srcLen` and `dstLen` are additional parameters to represent the bounds of the arrays. It is not

```
var str:[int]int, x:int, y:int;
procedure ReplaceChar() {
  call Helper(0);
}
procedure Helper(i:int) {
  var tmp:int;
  if (str[i] != 0) {
    tmp := str[i];
    havoc tmp;
    str[i] := tmp==x ? y : tmp;
    call Helper(i+1);
  }
}
```

Fig. 2. Replacing a character in a string.

hard to see that the base program `Strcpy` satisfies memory safety under some non-trivial preconditions. For example, a caller needs to ensure that `src` contains `0` within its bounds, and that the `dst` array has enough capacity to copy `src`. In addition, the client needs to ensure that the value of `srcLen` (resp. `dstLen`) is within the runtime bounds of the `src` (resp. `dst`) array — such bounds are not readily available for low-level languages such as C. In other words, the proof of (absolute) array bound safety of `Strcpy` requires access to additional runtime state for bounds, non-trivial preconditions, and loop invariants for the loop.

On the contrary, it is relatively simple to establish that the approximate version `StrcpyApprox` is *relative assertion safe* with respect to `Strcpy`. We provide an *almost* automatic proof using a differential verifier,[3] without access to additional runtime states or preconditions (§ 5). The intuition is that the approximation `StrcpyApprox` does not access any *additional* indices that could not be accessed in `Strcpy`. At the same time, the complexity of the example (loop exit condition depends on array content) and approximation (introducing a **break** statement) makes it difficult for any existing static-analysis-based approaches (e.g., [19, 31]) to ensure the safety of the approximation.

Relative Termination Just like preserving assertions, preserving terminating executions is an important criteria for almost any approximation. In other words, if an input leads to a terminating execution on the precise program, one needs to ensure that the approximation does introduce a non-terminating behavior. Consider again procedure `StrcpyApprox` from Fig. 1, and let us assume unbounded integers (`i` and `j`) and unbounded arrays (`src` and `dst`). Let us also assume that assertion failure does not terminate the program. In such a case, the base version `Strcpy` only terminates for those inputs where `src` has `0` as its element — other inputs may cause non-termination. It is desirable to ensure that `StrcpyApprox` at least terminates on all such inputs. For example, if the line i := i + 1 is (mistakenly) replaced with i := i - 1, the verifier should reject the approximation.

Similar to the proof of (absolute) assertion safety for `Strcpy`, a proof of (absolute) termination would require (i) a non-trivial existentially quantified precondition about the existence of `0` and (ii) a ranking function relating `i` with the first index containing `0`, among other ingredients. We show that we are able to avoid these complexities by reasoning about relative termination [11], instead of establishing each program terminates in isolation.

Control Flow Safety The program in Fig. 2 replaces a given character `x` with `y` in a character array `str`. The procedure `Helper` iterates over indices of the array until the

[3] We required the user to provide a simple additional predicate and unroll the first loop once.

termination character (`\0` in this case) is reached. Consider the approximation of the variable `tmp` indicated by the underlined statement — this models a case where the variable `tmp` is stored in an *unreliable memory* that may trade off cost for accuracy [21]. Approximating statements that impact control flow often leads to serious problems in unacceptably high corruptions in output data and program crashes. Hence, preservation of control flow has been identified as a natural and useful relaxed specification for approximations [31]. Since `tmp` flows into `str` that controls the conditional, a standard dataflow-based analysis would mark the approximation as unsafe.

However, observe that the fragment of the array that stores the value in `tmp` in fact never participates in the conditional. Our approach leverages differential verification to check for control flow safety, which allows for precise analysis (§3.2). Interestingly, we formalize the concept that an approximation does not affect control as a pair of incomparable relative properties: (i) a relative safety property that all pairs of terminating executions follow same control flow sequence (§ 3.2), and (ii) a relative termination property that the sets of terminating executions are identical in the two programs.

1.2 Our Approach

In this paper, we perform a *feasibility study* of using a differential verifier (§2) for expressing and verifying various relative specifications related to approximations (§3). We are the first to propose and demonstrate the idea of *relative termination* to the problem of verifying approximations. We leverage and extend the SymDiff infrastructure [14, 15, 11] to express and verify these specifications. We describe some of the extensions needed to improve the automation for the benchmarks we considered (§4). Overall, our verifier requires less than 1 manually supplied predicate on average to verify the safety of the approximations (§5). This is due to the fact that most proofs require relatively simple 2-program relational properties, as opposed to complex program-specific invariants. Our results give us confidence to apply the prototype on original source code written in languages such as C and Java, to serve as an independent validator for approximations introduced by approximate compilers (i.e., translation validation [20] for approximate compilers such as ACCEPT [31]).

2 Background

2.1 Programs

A program $P \in \mathit{Programs}$ consists of a set of procedures in *Procedures* and a set of global variables. Each procedure $p \in \mathit{Procedures}$ contains a list of input and output parameters, local variables, and a *body*. A body for a procedure p is an acyclic control flow graph with a set of nodes Nodes_p and $\mathit{Edges}_p \subseteq \mathit{Nodes}_p \times \mathit{Nodes}_p$, with an entry node $n^e_p \in \mathit{Nodes}_p$ and an exit node $n^x_p \in \mathit{Nodes}_p$. Each node $n \in \mathit{Nodes}_p$ in the control flow graph contains one of the following statements in *Stmts*:

$$s,t \in \mathit{Stmts} ::= \textbf{skip} \mid \textbf{assume}\ e \mid \textbf{assert}\ e \mid \textbf{havoc}\ x \mid \\ x := e \mid \textbf{call}\ x_1,\ldots x_k := q(e_1,\ldots,e_n)$$

where x, x_i represent program variables and $e, e_i \in \mathit{Exprs}$ are *expressions*. The precise set of types of variables and the expression language are left unspecified. Types include Booleans and integers, while expressions are built up using constants, interpreted (e.g.

arithmetic and relational operations) or uninterpreted functions. Arrays are modeled using interpreted functions *select* and *update* from the logical theory of arrays [2].

We only sketch the semantics for the statements here — the semantics of programs is built up using semantics of statements over control flow graphs and is fairly standard [1]. A *state* $\sigma \in \Sigma$ is an assignment of values to variables in scope. To model assertions, we introduce a ghost Boolean global variable OK, and model **assert** e as an assignment $OK := OK \wedge e$. A state $\sigma \in \Sigma$ for which OK evaluates to **false** under σ is termed as an *error state*. Each statement $s \in Stmts$ defines a transition relation $\|s\| \subseteq \Sigma \times \Sigma$, where **skip** represents the identity relation and $(\sigma, \sigma) \in \|\textbf{assume } e\|$ if σ evaluates the Boolean expression e to **true**. Moreover, $(\sigma, \sigma') \in \|x := e\|$ if σ' is obtained by updating the value of variable x with the valuation of e in σ. Similarly, $(\sigma, \sigma') \in \|\textbf{havoc } x\|$ if σ' and σ agree on the value of all variables except x. The semantics of a call statement is standard — it pushes the caller state on a *call stack*, executes the callee q with values of e_i as inputs, and upon termination pops the call stack and updates x_i variables with values of outputs of q. We denote a node n containing a call to q as a callsite of q. Conditional statements are encoded using **assume** and **skip** statements on the control flow graph [1]; loops are encoded using tail-recursive procedures.

An *execution* is a sequence $\langle (n_0, \sigma_0), \ldots, (n_i, \sigma_i), \ldots \rangle$ where either (i) $(n_i, n_{i+1}) \in Edges_p$ (for some p) and $(\sigma_i, \sigma_{i+1}) \in \|s_i\|$ where s_i is a non-call statement at n_i, or (ii) n_i is a callsite of q, n_{i+1} equals n_q^e (the entry node of q), and σ_{i+1} is the input state of q obtained from the caller state σ_i, or (iii) n_i is n_q^x (the exit node of q), n_{i+1} is the unique successor of the corresponding callsite of q, and σ_{i+1} is the caller state (after the call) obtained from the output state σ_i. For each procedure p, we define its input-output transition relation $\mathscr{T}_p$ as the set of pairs (σ, σ') such that there is an execution of p starting in input state σ (with an empty call stack) and terminating in output state σ' (with an empty call stack). For the rest of the paper, we assume that we are given two versions $P_1, P_2 \in Programs$ of a program with disjoint set of procedures and globals. We distinguish components of the two versions using subscripts 1 and 2 respectively.

2.2 Mutual Summary Specifications

Given two procedures $p_1 \in P_1$ and $p_2 \in P_2$, we define a *2-program input-output expression* as an expression over inputs and outputs of p_1 and p_2. The inputs can refer to the input parameters and globals (within an $\textbf{old}(e)$ subexpression where the construct **old** evaluates the subexpression at procedure entry), and outputs can refer to the output parameters and globals. For example, if g_i refers to global variables, x_i (resp. y_i) refers to input (resp. output) parameters of a pair of procedures p_1, p_2, the expression $\neg(\textbf{old}(g_1 \leq g_2) \wedge x_1 \leq x_2 \wedge g_1 + y_1 > g_2 + y_2)$ is a 2-program input-output expression relating inputs and outputs of p_1 and p_2. Given such a 2-program input-output expression e, and two pairs of input-output states $(\sigma_1, \sigma_1') \in \mathscr{T}_{p_1}$ and $(\sigma_2, \sigma_2') \in \mathscr{T}_{p_2}$, the value of e is obtained by evaluating the inputs (resp. outputs) of f_i under σ_i (resp. σ_i').

Definition 1. (Mutual Summary [11]). *Given two procedures $p_1 \in P_1$ and $p_2 \in P_2$, a 2-program input-output Boolean expression e is a* mutual summary *for p_1, p_2 if the value of e evaluates to* **true** *for every pair of input-output states in $\mathscr{T}_{p_1} \times \mathscr{T}_{p_2}$.*

We use mutual summaries to express relative safety and accuracy specifications over two programs. Intuitively, a mutual summary is a summary (or postcondition) for the product procedure over the pair of procedures p_1, p_2.

2.3 Relative Termination Specifications

Given two procedures $p_1 \in P_1$ and $p_2 \in P_2$, we define a *2-program input expression* as an expression over inputs of p_1 and p_2. Such expressions do not contain $\mathbf{old}(e)$ since they may only refer to the input globals. The expression $(g_1 \leq g_2 \wedge x_1 = x_2)$ is an example of a 2-program input expression relating inputs of two procedures.

Definition 2. (Relative Termination Conditions [11]). *Given two procedures $p_1 \in P_1$ and $p_2 \in P_2$, a 2-program input Boolean expression e is a* relative termination condition *for p_1, p_2 if for each pair of input states σ_1, σ_2 of p_1, p_2 that evaluates e to* **true***, if σ_1 has at least one terminating execution for p_1, then so does σ_2 for p_2.*

Note that for inputs satisfying the relative termination condition, the procedure p_2 terminates at least as often as the procedure p_1. This is helpful for specifying intermediate relationships between recursive procedure pairs when p_2 terminates in fewer iterations than p_1 under the same input.

3 Preserving Safety, Accuracy, and Termination

In this section, we first show that mutual summary specifications can be used to capture both relative safety (assertion §3.1 and control flow §3.2) and relative accuracy (§3.3) for approximations. Finally, we describe the use of relative termination specifications for describing approximations (§3.4).

3.1 Preserving Assertion Safety

Recall from §1.1 that we informally describe relative assertion safety as a robustness criterion that assertions in approximate programs should fail less often than their counterparts in precise programs. We formalize this as follows:

> A procedure $p_2 \in P_2$ has a differential error with respect to a procedure $p_1 \in P_1$ if there exists a common input state σ such that $(\sigma, \sigma_1) \in \mathscr{T}_{p_1}$ and σ_1 is not an error state, and there exists $(\sigma, \sigma_2) \in \mathscr{T}_{p_2}$ such that σ_2 is an error state. *Relative assertion safety* of p_2 with respect to p_1 holds if there are no differential errors in p_2 with respect to p_1.

Recall that assertions are desugared using a ghost variable OK (§2.1). Relative assertion safety is then encoded as the following mutual summary specification for p_1 and p_2: $(\mathbf{old}(\bigwedge_{x \in X} x_1 = x_2)) \Rightarrow (OK_1 \Rightarrow OK_2)$, where X denotes the set of input parameters and globals of p — each variable $x \in X$ is named x_1 (resp. x_2) in program P_1 (resp. P_2).

3.2 Preserving Control Flow Safety

Preserving control flow safety has been identified as an important robustness criterion for approximations (§1.1). Next, we show that we can use mutual summaries to capture that the approximation does not affect control flow (modulo termination). We first define an automatic program instrumentation for tracking control flow. Let a *basic block* 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 $cflow := trackCF(cflow, blockID)$,

```
function RelaxedEq(x:int, y:int) returns (bool) {
  (x <= 10 && x == y) || (x > 10 && y >= 10 && x >= y)
}

procedure Swish(max_r:int, N:int) returns (num_r:int) {
  var old_max_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. 3. Swish++ open-source search engine example.

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 $p_1 \in P_1$ and $p_2 \in P_2$ be the two versions of a procedure p in the original and the approximate program. We denote with X the set of input parameters and globals of p — each variable $x \in X$ is named x_1 (resp. x_2) in program P_1 (resp. P_2). Then the mutual summary $(\mathbf{old}(\bigwedge_{x \in X} x_1 = x_2)) \Rightarrow (cflow_1 = cflow_2)$ states that if the two procedures start out in the same state, the values of the *cflow* variables are equal on termination. If p_1 and p_2 satisfy this mutual summary specification, then the following holds:

> For any pair of executions $(\sigma, \sigma_1) \in \mathscr{T}_{p_1}$ and $(\sigma, \sigma_2) \in \mathscr{T}_{p_2}$ starting at the same input state σ, the sequences of basic blocks in the two executions are identical.

Note that the specification only ensures that every pair of terminating executions from σ follow the same control flow. It does not preclude p_2 to not terminate on the input state σ. We address this issue using relative termination specifications that further ensure that (for deterministic programs) if p_1 terminates on σ, then so does p_2.

3.3 Preserving Accuracy

The accuracy criterion ensures that approximations do not cause unacceptable divergence of outputs between two program versions. For example, a write operation to approximate memory may introduce a small error into the written value [21]. Such errors can be amplified by a program (e.g., through multiplication by a large constant), and lead to significant and unintended output difference between the original and approximate program. Hence, the accuracy criterion is used to capture the acceptable quantitative gap between precise and approximate outputs. Mutual summaries naturally express such specifications by relating the inputs and outputs of a procedure pair.

Fig. 3 gives the *Swish++* open-source search engine example taken from a recent approximate computing work by Carbin et al. [5]. It takes as input a threshold for the maximum number of results to display `max_r` and the total number of search results `N`, and returns the actual number of results to display `num_r` bounded by `max_r` and `N`. The approximation nondeterministically changes the threshold to a possibly smaller number, without suppressing the top 10 results. This allows the search engine to trade-off the number of search results to display under heavy server load, since users are typically interested in the top few results. The predicate *RelaxedEq* denotes the relationship between the original and the approximate value. We express and prove the accuracy criterion (akin to *acceptability property* [5, 24]) as the mutual summary $\mathbf{old}(max\_r_1 = max\_r_2 \wedge N_1 = N_2) \Rightarrow RelaxedEq(num\_r_1, num\_r_2)$.

3.4 Preserving Termination

We use relative termination conditions (§2.3) to specify that the approximate program terminates at least as often as the base program, and we note the following. The relative termination conditions for a procedure pair may not always be simple equalities over input states. For the pair of `Helper` procedures in Fig. 2, the relative termination condition satisfied by the two versions is $i_1 = i_2 \wedge (\forall j :: j \geq i_1 \Rightarrow src_1[j] = src_2[j])$, since the recursive calls may not preserve the segment of the array before i. In the presence of a **havoc** statement in p_2 (Fig. 2), the specification only guarantees that p_2 has at least one terminating execution on a common input to p_1. To address this, we perform a standard trick of modeling a **havoc** x statement as a read from a global stream of unconstrained values [14]. This can be done using a global a array and a counter c into the array and replacing **havoc** x with $x := a[c++]$. With this, the array becomes a part of the input and the *internal* non-determinism is converted into an input non-determinism. For the transformed program the relative termination specification ensures that none of the terminating executions in p_1 fails to terminate in p_2.

4 Verifying Relative Specifications

In this section, we describe how we leverage and extend SymDiff [14, 15, 11], a differential verifier for procedural programs that employs SMT-based checking and automatic invariant inference. Although SymDiff already provided many building blocks, we extended it to improve the automation of checking mutual summaries and relative termination conditions. Previously, to verify the relative specifications on the (top-level) entry procedures, the user had to fully annotate all intermediate mutual summaries and relative specification conditions for every pairs of procedures [11]; SymDiff only provided a verifier for fully annotated pairs of procedures. We improve the automation in three main directions:

1. We leverage a product program construction for procedural programs that allows inferring relative specifications using off-the-shelf invariant inference tools [15]. This product construction was already present in SymDiff but was customized for checking a specific form of relative specifications (namely, relative assertion safety).
2. We use inferred preconditions for the product program as candidate relative termination conditions for intermediate procedure pairs.
3. We augment the specific invariant inference scheme used in SymDiff over the product program to allow for the user to supply additional predicates.

We informally elaborate on these ideas next. The details of the product construction [15] and checking relative termination conditions [11] are beyond the scope of this paper.

4.1 Procedural Product Programs

We recollect a particular product construction for procedural programs as implemented in SymDiff [15]. The product construction is novel in several ways. First, it can handle procedures (including recursion) in P_1 and P_2 unlike most other product constructions that are intraprocedural [3]. Second, the product program can be fed to any off-the-shelf invariant inference engine to infer mutual summaries over P_1 and P_2.

Given P_1 and P_2, the product program $P_{1\times2}$ consists of procedures in P_1, P_2 and a set of product procedures described below. The set of globals of $P_{1\times2}$ is the disjoint union

of globals of P_1 and P_2. For a pair of procedures $p_1 \in P_1$ and $p_2 \in P_2$, we introduce a product procedure $p_{1\times2}$ whose input (resp. output) parameters are the disjoint union of input (resp. output) parameters of p_1 and p_2. The body of $p_{1\times2}$ is a sequential composition of bodies of p_1 and p_2 followed by a series of *replay* blocks. We informally sketch these replay blocks using an example. Let q_1 be a call within p_1 body and q_2 be a call within p_2 body. For any path in $p_{1\times2}$ where q_1 and q_2 are both executed with inputs i_1, i_2 and produce outputs o_1, o_2 (where both inputs and outputs include global mutable state), we constrain (o_1, o_2) to be the output of executing $q_{1\times2}$ over inputs (i_1, i_2) in the product program. To perform the replay, each call site in p_1 and p_2 is instrumented to record the inputs and outputs, and global state is set/reset in the replay code.

The resultant product program (which is just another program in *Programs*) has the following property (we are the first to formalize this connection):

> For any product procedure $p_{1\times2} \in P_{1\times2}$, if a 2-program 2-state expression e is satisfied by every $(\sigma_{1\times2}, \sigma'_{1\times2}) \in \mathscr{T}_{p_{1\times2}}$, then e is a mutual summary specification for (p_1, p_2).

In other words, if an expression e (over the two program states) is a valid summary (or postcondition) for $p_{1\times2}$, it is a valid mutual summary for the pair of procedures p_1 and p_2. This provides a sound rule for proving mutual summaries over P_1 and P_2: we can express a mutual summary over p_1 and p_2 (e.g., any of the specifications in §3) as a specification over the product procedure $p_{1\times2}$, and verify $P_{1\times2}$ using any off-the-shelf program verifier.

4.2 Invariant Inference

To verify a mutual summary, we annotate the resultant product program $P_{1\times2}$ with a summary of the top-level procedures, and let a program verifier infer intermediate specifications (preconditions and postconditions of intermediate $q_{1\times2}$ procedures). It was noted in earlier work that most specifications on product procedures tend to be relational or 2-program (e.g., $i_1 \leq i_2$), which requires exploiting the structural similarity between P_1 and P_2. Running an invariant inference engine as is (e.g., Duality [17]) results in generation of one-program-specific invariants and fails to infer 2-program specifications. Therefore, SymDiff exploits the mapping between parameters and globals to add candidate relational predicates such as $i_1 \bowtie i_2$, where $\bowtie \in \{\leq, \geq, <, >, \Leftarrow, \Rightarrow, =\}$, for copies of a variable i in two programs. Relational specifications can be generated by composing these predicates using predicate abstraction [10] or Houdini [9]. The advantage of Houdini (that only infers subsets of these predicates) is that it is typically fast and predictable, and has been shown to scale to very large programs [34]. We leverage the Houdini-based 2-program specification inference in SymDiff, but we also added a facility for a user to augment the set of automatically generated predicates. Our study shows that such a mechanism was useful in several cases to provide domain-specific guesses for the required predicates.

4.3 Inferring Relative Termination Conditions

The product program $P_{1\times2}$ is not suitable for proving termination related properties as it is meant for proving relative safety properties (on pairs of terminating executions). We therefore fall back to the technique proposed for checking relative termination conditions [11]. We briefly sketch the technique before highlighting the inference extension we have implemented.

Given P_1 and P_2, we construct a product program $P_{1\otimes 2}$ by creating product procedures $p_{1\otimes 2}$ for two versions of a procedure p. Let us assume that we have a relative termination condition $RT_{p_{1\otimes 2}}$ for the procedure $p_{1\otimes 2}$. Recall that $RT_{p_{1\otimes 2}}$ is an expression over inputs of p_1 and p_2 (§2.3). For each procedure p (in either version), we create an uninterpreted relation R_p containing all the input-output state pairs of p (i.e., over-approximates $\mathscr{T}_p$). We add a background axiom encoding the assumption that if there exists $(\sigma_1, \sigma_1') \in R_{p_1}$ and $(\sigma_1, \sigma_2) \in RT_{p_{1\otimes 2}}$, then there exists σ_2' such that $(\sigma_2, \sigma_2') \in R_{p_2}$:

$$\forall \sigma_1, \sigma_1', \sigma_2 :: \left(R_{p_1}(\sigma_1, \sigma_1') \wedge RT_{p_{1\otimes 2}}(\sigma_1, \sigma_2)\right) \Rightarrow (\exists \sigma_2' :: R_{p_2}(\sigma_2, \sigma_2')).$$

Each procedure $p_{1\otimes 2}$ starts by assuming the relative termination condition, followed by the body of p_1 and p_2, all composed sequentially. Before any call (to say q_2) inside p_2's body, we add the assertion **assert** $\exists \sigma_2' :: R_{q_2}(\sigma_2, \sigma_2')$, where σ_2 is the state of the input to the call to q_2 and σ_2' is the output state of q_2. Intuitively, such an assertion before every call (which is the only way to avoid termination in the absence of loops) ensures that a call to q_2 must be preceded by a call to q_1 in the path inside $p_{1\otimes 2}$ — in other words, q_2 is called less often than q_1 on any execution. If all such assertions hold for the given $RT_{q_{1\otimes 2}}$ for all procedures $q \in P$, then the relative termination of the entry level procedures is established.

Although the relative termination condition for the top-level procedures is often simple (equality of the input states), intermediate procedures may only satisfy weaker relationships. For example, sometimes a relationship such as $i_1 \leq i_2$ holds for a loop index i to indicate that the second procedure terminates earlier. Also, recall the non-trivial specification for the intermediate *Helper* procedure in §3.4 where only segments of arrays are equal. Clearly, manually specifying all the *RT* can be quite cumbersome in the presence of multiple procedures.

We leverage the product program $P_{1\times 2}$ used earlier to heuristically guess possible *RT* expressions. We have observed that the inferred preconditions to a product procedure $p_{1\times 2}$ often represent sound relationships between inputs of p_1 and p_2 in any execution. One can, however, construct examples where the inferred precondition is not sound for relationship between inputs to p_1 and p_2 — e.g., due to non-termination or fewer call-sites of a procedure in the new version. We heuristically install a precondition to $p_{1\times 2}$ (from $P_{1\times 2}$) as $RT_{p_{1\otimes 2}}$ (in $P_{1\otimes 2}$) and try verifying $P_{1\otimes 2}$. If verification succeeds, we have established the relative termination property. In the case study, we show that this heuristic suffices for all but one of our benchmarks.

5 Case Study

In this section, we describe our feasibility study of using differential program verification techniques for automatic verification of several classes of program approximations.

Benchmarks Table 1 lists our benchmarks and presents the results of verifying them using our framework. We used the following benchmarks in our experiments:

- Case studies taken from previous work by Carbin et al. [5]: *LU Decomposition*, *Water*, and *Swish++*. We provide the same guarantees as this previous work, and in addition we prove relative termination for a modified version of *Swish++*.
- Array and string operations: *Replace Character*, *Array Operations*, *Array Search*, *String Hash*, *String Copy*, *Selection Sort*, and *Bubble Sort*.

Table 1. Experimental results. LOC is the number of lines of Boogie code in approximate programs; Criterion is the verified property; #Preds is the number of predicates automatically generated by SymDiff; #Man is the number of manually provided predicates; Time is the total runtime in seconds, including inference.

Benchmark	LOC	Criterion	#Preds	#Man	Time(s)
Cube Root	7	Relative Termination	12	0	6.5
Loop Perforation	11	Relative Termination	10	0	4.8
Gradient Descent	17	Relative Termination	22	0	6.4
String Hash	19	Assertion Safety	25	0	7.8
		Relative Termination	19	0	4.9
Swish++	22	Accuracy	14	2	6.5
		Relative Termination	14	0	4.8
Water	27	Assertion Safety	32	0	5.8
Pointer Perforation	28	Relative Termination	26	0	5.1
Replace Character	31	Assertion Safety	15	0	7.7
		Control Flow Safety	15	0	7.9
		Termination	5	0	5.1
String Copy	32	Assertion Safety	20	2	7.7
		Relative Termination	14	0	6.5
LU Decomposition	33	Accuracy	32	2	5.7
Array Search	33	Relative Termination	30	0	7.1
Array Operations	43	Control Flow Safety	44	0	8.2
Sobel	49	Relative Termination	190	1	5.3
Selection Sort	57	Control Flow Safety	81	0	8.5
ReadCell	60	Assertion Safety	37	1	14.0
		Control Flow Safety	37	1	14.0
Bubble Sort	67	Control Flow Safety	59	0	8.2
JPEG Quantization	96	Accuracy	19	3	6.3

- Loop approximation examples: *Cube Root*, *Gradient Descent*, *Loop Perforation*, and *Pointer Perforation*.
- Image processing programs taken from the ACCEPT benchmark suite [29]: *ReadCell* (extracts information from the header of an image file), *Sobel* (implements a Sobel image filter), and *JPEG Quantization* (quantization stage of a JPEG encoder).

We only prove important criteria for every benchmark since some either do not hold or are trivial to prove. All experiments were performed on a 2.3 GHz Intel i7-3610QM machine with 8GB RAM and running Microsoft Windows. Our extensions to SymDiff and benchmarks are available at `http://symdiff.codeplex.com` (rt-feature branch).

Discussion As experimental results show, we successfully used our approach to verify a variety of approximation robustness criteria. Verification of most benchmarks terminates in under one minute, which indicates that our technique has potential to scale to larger examples. Only two manual steps were occasionally needed to complete the proof. First, in several benchmarks we had to unroll once tail-recursive procedures extracted from loops (e.g., *String Copy*, *String Hash*). Second, we had to provide additional predicates for the benchmarks with non-zero #Man field in Table 1. The need for manual predicates can be broken down into roughly two categories: (i) simple non-relational predicates such as $j_2 \leq i_2$ (e.g., *String Copy*), and (ii) non-trivial relational

predicates that require arithmetic such as `RelaxedEq` (e.g., *Swish++* in Fig. 3, *LU*). These predicates are mainly used for proving domain-specific relative accuracy properties, and reusing the predicate `RelaxedEq` often suffices for the proof. Our study shows that our inference techniques successfully generated most of the required specifications automatically, indicating that most relative specifications do not heavily depend on complex program-specific invariants.

5.1 Experience

We describe next in more detail our experience verifying some of the listed benchmarks.

Replace Character and Sorting Recall the *Replace Character* example from Fig. 2, where we wish to verify that the approximation maintains control flow safety. The main challenge of this verification task is to capture the fact that control flow depends on only a fragment of the array, which is identical in the two programs. We capture this property by defining a quantified predicate template $\mathit{ArrayEqAfter}(\mathit{str}_1, \mathit{str}_2, i_1) \doteq \forall j : \textbf{int} :: j \geq i_1 \Rightarrow \mathit{str}_1[j] = \mathit{str}_2[j]$, which is then automatically instantiated using our inference engine. The proof of control flow safety for the selection sort example shown in Fig. 4 also leverages this predicate. The selection sort algorithm sorts an array by pushing the maximum element of the $[c \ldots n-1]$ subarray to the position c after every iteration. Once an element has been pushed to the front, it does not play a part in determining future control flow behavior Therefore, approximating such end elements does not influence the control flow of the algorithm. In addition to selection sort, we also verified control flow safety for a version of bubble sort containing a similar approximation. Unlike selection sort where the leftmost index is approximated, the approximation in bubble sort requires introducing an additional instruction to havoc the rightmost array element of each iteration. A similar predicate *ArrayEqBefore*, specifying that the two arrays are equal before some index, captures that fact that the subarray before each iteration is precise and thus facilitates the proof.

JPEG Quantization Fig. 6 shows the source code of a JPEG encoder quantization stage taken from the ACCEPT benchmark suite [29]. Each element in `data` gets its quantized value stored in `Temp` by multiplying it with the corresponding element in `QuantTable`, and dividing the result by 2^{15} after adding 2^{14} to it. This application is suitable for an approximation that allocates `data` in approximate memory since the error `e` introduced to the stored value (denoted by the predicate `RelaxedEq`) is masked or reduced after division by 2^{15}. The approximation is introduced using the underlined statements, and the following mutual summary expresses the desired relative accuracy specification:

$$\textbf{old}(\mathit{data}_1 = \mathit{data}_2) \Rightarrow (\forall\, i : \textbf{int} :: (i \geq 0 \wedge i \leq 63) \Rightarrow \mathit{RelaxedEq}(\mathit{Temp}_1[i], \mathit{Temp}_2[i], 2))$$

The most involved manually provided predicate $\mathit{RelaxedAfter}(\mathit{Temp}_1, \mathit{Temp}_2, i)$ is similar to *ArrayEqAfter*. It is based on the observation that after each iteration of the loop, all corresponding elements of the arrays Temp_1 and Temp_2 after index i should satisfy *RelaxedEq* with the error bound of 2.

String Examples To prove relative assertion safety for the example from Fig. 1, we had to manually unroll the loop in `Strcpy` once and provide two atomic predicates. Such loop unrolling helps SymDiff to infer the equality between i_1 and i_2, which indicates

that the *src* arrays are accessed in the same way and thus implies relative assertion safety. The manual predicates needed for this example relate indices of array *dst*, and have the form $j_2 \leq i_2$. With these predicates, relative assertion safety is established for array *dst* since dst_2 is accessed less often than dst_1. In addition, we proved relative termination of `StrcpyApprox` with respect to `Strcpy`. This required a simple relative termination condition automatically inferred by SymDiff, $src_1 = src_2 \wedge i_1 = i_2$, since we unrolled the loop in *Strcpy* once. Such bounded loop unrolling often facilitates the verification of relative termination since it allows for the proof to be discharged using a simpler relative termination condition.

Simple Cube Root Calculation We implemented a benchmark that calculates the integer approximation `r` of the cube root of `x` by performing a simple iterative search guarded with the nonlinear condition `r*r*r<=x`. We further approximate this computation by performing loop perforation, which speeds up the search at the expense of losing precision, and potentially leads to non-termination. Automatically proving program termination is especially hard when loop conditions contain nonlinear arithmetic, which complicates generation of adequate ranking functions. We easily proved relative termination of this benchmark using the simple relative termination condition $r_1 \leq r_2$ that is automatically inferred.

6 Related Work

A number of complementary approaches have been recently proposed to reason about approximations. These approaches can be roughly categorized (with overlaps) into (i) language based, (ii) static analysis, and (iii) dynamic approaches. Language based approaches propose language constructs and annotations to make approximations explicit in a program [31, 5]. EnerJ [31] introduces approximate types and ensures that such values do not impact precise computations, including conditional statements. ACCEPT [29] automatically searches for code regions that can be approximated based on type annotation and static compiler analysis pass. Our work can be used to improve the precision of the type-based analysis, as shown in §1.1.

Carbin et al. [5] develop a special-purpose language and constructs for introducing approximations and relaxed specifications (based on *relational Hoare logic* [3]), and prove correctness of transformations using the general purpose Coq theorem prover [7]. Each proof for their three benchmarks required roughly 330 lines of proof scripts according to the authors. We provide the same guarantees for these three benchmarks almost completely automatically (see § 5), thereby showing that mutual summaries and SMT-based verification can significantly improve the automation for most transformations covered by this approach.

Rely [6] is a programming language that allows users to verify probabilistic quantitative reliability guarantees under the impact of unreliable hardware on overall behavior of a program with an associated static analysis. Chisel [18] is a synthesis framework that generates optimal programs for execution on approximate hardware that satisfy given accuracy and reliability specifications. Unlike our approach, Chisel can only establish relative specifications for syntactically equivalent program versions. Moreover, Chisel ensures control flow equivalence using a simple dependence analysis, which is less precise than our approach as illustrated in §1.1. On the other hand, Chisel can reason about

probabilities, which our approach currently does not support. ExPAX [22] 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 approximation level for each operation in the set so that energy consumption is minimized and reliability constraints are satisfied. DECAF [4] combines static type inference, dynamic tracking, and runtime check to give probabilistic guarantee on the quality of approximate programs.

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 [32] and LLFI [33] approximate instructions at runtime. Though these techniques achieve high levels of accuracy, they provide no formal coverage guarantees, unlike our approach. Offline dynamic analysis techniques provide information on dataflow and correlation difference (e.g., [25, 26]). 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 statistical methods [27], the reasoning is only for a subset of all the possible executions of the program.

Finally, our work is related to previous works on translation validation [23, 20] that validate equivalence-preserving intraprocedural compiler transformations, using lockstep symbolic execution and SMT solvers. However, mutual summaries and the product construction allows for richer relaxed specifications other than equivalence, interprocedural reasoning [11], and leveraging off-the-shelf verifiers and inference engines.

7 Conclusions and Future Work

In this paper, we have described the application of automated SMT-based differential verification for providing formal guarantees of approximations. The structural similarity between original and approximate programs are leveraged to automate most intermediate relative specifications. Our extensions to SymDiff allowed us to verify a variety of criteria that ensure robustness of approximate programs, including relative control flow safety, assertion safety, accuracy, and termination. We are also first to propose relative termination as an important robustness criterion. Our feasibility study shows that the techniques we developed can be effectively used to automatically prove program approximations. We are currently working on automating predicate generation, using more expressive inference engines such as interpolants [16] and indexed predicate abstraction [13] to infer remaining specifications.

References

1. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO (2006)
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: SMT (2010)
3. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL (2004)
4. Boston, B., Sampson, A., Grossman, D., Ceze, L.: Probability type inference for flexible approximate programming. In: OOPSLA (2015)
5. Carbin, M., Kim, D., Misailovic, S., Rinard, M.C.: Proving acceptability properties of relaxed nondeterministic approximate programs. In: PLDI (2012)

6. Carbin, M., Misailovic, S., Rinard, M.C.: Verifying quantitative reliability for programs that execute on unreliable hardware. In: OOPSLA (2013)
7. The Coq proof assistant. `http://coq.inria.fr`
8. Elenbogen, D., Katz, S., Strichman, O.: Proving mutual termination. Formal Methods in System Design 47(2), 204–229 (2015)
9. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME (2001)
10. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV (1997)
11. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Towards modularly comparing programs using automated theorem provers. In: CADE (2013)
12. Kugler, L.: Is "good enough" computing good enough? Commun. ACM 58(5), 12–14 (2015)
13. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Log. 9(1) (2007)
14. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SymDiff: A language-agnostic semantic diff tool for imperative programs. In: CAV (2012)
15. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: ESEC/FSE (2013)
16. McMillan, K.L.: An interpolating theorem prover. In: TACAS (2004)
17. McMillan, K.L.: Lazy annotation revisited. In: CAV (2014)
18. Misailovic, S., Carbin, M., Achour, S., Qi, Z., Rinard, M.C.: Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. SIGPLAN Not. 49(10), 309–328 (2014)
19. Misailovic, S., Sidiroglou, S., Hoffmann, H., Rinard, M.: Quality of service profiling. In: ICSE (2010)
20. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000)
21. Nelson, J., Sampson, A., Ceze, L.: Dense approximate storage in phase-change memory. In: Ideas and Perspectives session at ASPLOS (2001)
22. Park, J., Ni, K., Zhang, X., Esmaeilzadeh, H., Naik, M.: Expectation-oriented framework for automating approximate programming. In: WACAS (2014)
23. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: (TACAS) (1998)
24. Rinard, M.: Acceptability-oriented computing. In: OOPSLA (2003)
25. 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
26. Ringenburg, M.F., Sampson, A., Ceze, L., Grossman, D.: Profiling and autotuning for energy-aware approximate programming. In: WACAS (2014)
27. Roy, P., Ray, R., Wang, C., Wong, W.F.: ASAC: Automatic sensitivity analysis for approximate computing. In: LCTES (2014)
28. Sampson, A.: Hardware and Software for Approximate Computing. Ph.D. thesis, University of Washington (2015)
29. Sampson, A., Baixo, A., Ransford, B., Moreau, T., Yip, J., Ceze, L., Oskin, M.: ACCEPT: A programmer-guided compiler framework for practical approximate computing. Tech. Rep. UW-CSE-15-01-01, University of Washington
30. Sampson, A., Bornholt, J., Ceze, L.: Hardware-software co-design: Not just a cliché. In: SNAPL (2015)
31. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: EnerJ: Approximate data types for safe and general low-power computation. In: PLDI (2011)
32. Sharma, V.C., Haran, A., Rakamarić, Z., Gopalakrishnan, G.: Towards formal approaches to system resilience. In: PRDC (2013)
33. Thomas, A., Pattabiraman, K.: LLFI: An intermediate code level fault injector for soft computing applications. In: SELSE (2013)
34. Vanegue, J., Lahiri, S.K.: Towards practical reactive security audit using extended static checkers. In: S&P (2013)

A Benchmarks

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

procedure SelectionSort() {
  var c:int, position:int, temp:int;
  c := 0; position := 0; temp := 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. 4. Selection sort.

```
procedure CubeRoot(x:int)
    returns(r:int) {
  r := 1;
  while (r*r*r <= x) r := r+1;
}
```

```
procedure CubeRootApprox(x:int)
    returns(r:int) {
  r := 1;
  while (r*r*r <= x) r := r+2;
}
```

Fig. 5. Simple cube root calculation.

```
function RelaxedEq(x:int, y:int, e: int) returns (bool) {
  x <= y + e && y <= x + e
}

function RelaxedEqAll(x: [int]int, y: [int]int, e: int)
    returns(bool) {
  (forall j: int:: (j>=0 && j<= 63) ==> RelaxedEq(x[j], y[j], e))
}

function RelaxedAfter(x: [int]int, y: [int]int, e:int, i: int)
    returns (bool) {
  (forall j: int:: (j>i && j<= 63) ==> RelaxedEq(x[j], y[j], e))
}

function ShortInt(x:int) returns (bool) {
  x <= 32767 && x >= -32768
}

const QuantTable:[int]int;
var Temp:[int]int;

procedure quantization(data: [int]int)
modifies Temp;
requires (forall j: int :: ShortInt(data[j]));
{
  var i : int; var value : int;
  var data_old: [int]int;

  data_old := data; havoc data; assume RelaxedEqAll(data, data_old, 16);
  i := 63;
  while(i >= 0) {
    value := data[i] * QuantTable[i];
    value := sdiv((value + 16384), 32768);
    Temp[i] := value;
    i := i - 1;
  }
}
```

Fig. 6. JPEG quantization.