Noname manuscript No.
(will be inserted by the editor)

Verifying Relative Safety, Accuracy, and Termination for Program Approximations

Shaobo He · Shuvendu K. Lahiri · Zvonimir Rakamarić

the date of receipt and acceptance should be inserted later

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 [21,38, 36,39]. 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 [9, 17], arithmetic [18], or sensor readings [5,40]; high-level mechanisms include approximating loop computations [42], generating multiple approximate candidate implementations [1,20,46,30], or leveraging neural networks [12].

S. He, Z. Rakamarić
University of Utah, Salt Lake City, UT, USA
E-mail: {shaobo,zvonimir}@cs.utah.edu

S. Lahiri
Microsoft Research, Redmond, WA, USA
E-mail: shuvendu@microsoft.com

```
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.

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 [39], to static analyses [8], and interactive theorem provers [7] 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 [3]) automated *differential program verifiers* [4,19] for specifying and verifying properties of approximations. Such verifiers (e.g., SymDiff [22,23]) leverage SMT 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* [7,23] 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 [39]. Relative accuracy criteria specify the acceptable difference between precise and approximate outputs for specific approximations [7]. In addition to these established criteria, we propose the concept of relative termination [19,11] 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

1.1.1 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 [26]) 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 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,[1] without access to additional runtime states or preconditions (Sec. 5.2). 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., [26,39]) to ensure the safety of the approximation.

1.1.2 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 not 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 $\underline{\text{i := i + 1}}$ is (mistakenly) replaced with $\underline{\text{i := i - 1}}$, 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 [19], instead of establishing each program terminates in isolation.

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

```
var str:[int]int;
const 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.

1.1.3 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 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 [28]. Approximating statements that impact control flow often leads to serious problems such as 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 [39]. 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. Hence, our approach that leverages differential verification to check for control flow safety, which allows for precise analysis (Sec. 3.2), can verify this approximation. 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 (Sec. 3.2), and (ii) a relative termination property that the sets of terminating executions are identical in the two programs.

1.2 Our Approach and Contributions

In this paper, we perform a feasibility study of using a differential verifier (Sec. 2) for expressing and verifying various relative specifications related to approximations (Sec. 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 [22,23,19] to express and (mostly) automatically verify these specifications. We describe some of the extensions needed to improve the automation for the benchmarks we considered (Sec. 4), including automatic inference of *mutual summaries.* Overall, our verifier requires less than one manually supplied predicate on average to verify the safety of the approximations (Sec. 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 [27] for approximate compilers such as ACCEPT [39]).

2 Background

2.1 Programs

A program P consists of a set of procedures and a set of global variables and constants. Each procedure p 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 (we require for loops to be encoded using tail-recursive procedures) consisting of a set of nodes $Nodes_p$ and edges $Edges_p \subseteq Nodes_p \times Nodes_p$, with an entry node $n_p^e \in Nodes_p$ and an exit node $n_p^x \in Nodes_p$. Each node $n \in Nodes_p$ in the control flow graph contains one of the following statements in *Stmts*:

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

where x, x_i represent program variables and $e, e_i \in Exprs$ 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 [3].

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 [2]. 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 [2].

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 $\mathcal{T}_p$ 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 ease of presentation, in the rest of the paper we assume that we are given two versions P_1, P_2 of a program with disjoint sets of procedures and globals (note that the disjointness is easy to achieve with simple renaming). We distinguish components of the two versions using subscripts 1 and 2 respectively. We use X to denote the set of input parameters and globals of procedure p — each variable $x \in X$ is named x_1 (resp. x_2) in program P_1 (resp. P_2). As a notational convenience, we define the predicate $\mathit{EqInitState}(p_1, p_2) = \mathbf{old}(\bigwedge_{x \in X} x_1 = x_2)$ to indicate that the two versions $p_1 \in P_1$ and $p_2 \in P_2$ of a procedure p start out in the same initial state.

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 $\mathbf{old}(e)$ 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 $\neg\,(\mathbf{old}(g_1 \leq g_2) \wedge x_1 \leq x_2 \wedge g_1 + y_1 > g_2 + y_2)$ 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 $(\sigma_1, \sigma_1') \in \mathcal{T}_{p_1}$ and $(\sigma_2, \sigma_2') \in \mathcal{T}_{p_2}$, the value of e is obtained by evaluating the inputs (resp. outputs) of p_i under σ_i (resp. σ_i').

Definition 1 *(Mutual Summary [19]).* 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 $\mathcal{T}_{p_1} \times \mathcal{T}_{p_2}$.

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 [19]).* 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 in Sec. 3.1 and control flow in Sec. 3.2) and relative accuracy (Sec. 3.3) for approximations. Finally, we describe the use of relative termination specifications for describing approximations (Sec. 3.4).

3.1 Preserving Assertion Safety

Recall from Sec. 1.1.1 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:

> Relative assertion safety of p_2 with respect to p_1 holds if for all common input states σ such that $(\sigma, \sigma_1) \in \mathcal{T}_{p_1}$ and σ_1 is not an error state, all $(\sigma, \sigma_2) \in \mathcal{T}_{p_2}$ are such that σ_2 is also not an error state.

Recall that assertions are desugared using a ghost variable *OK* (Sec. 2.1). Relative assertion safety for p_1 and p_2 is then encoded as the mutual summary specification $\mathit{EqInitState}(p_1, p_2) \Rightarrow (\mathit{OK}_1 \Rightarrow \mathit{OK}_2)$, which is checked in the end of the generated product program as a postcondition of its top-level procedure (see Sec. 4.1).

3.2 Preserving Control Flow Safety

Preserving control flow safety has been identified as an important robustness criterion for approximations (Sec. 1.1.3). 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 $\mathit{cflow} := \mathit{trackCF}(\mathit{cflow}, \mathit{blockID})$, where *trackCF* is an uninterpreted function defined as $\mathit{trackCF}(\mathit{int}, \mathit{int})$ *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 approximate program. Then the mutual summary $\mathit{EqInitState}(p_1, p_2) \Rightarrow \mathit{cflow}_1 = \mathit{cflow}_2$ 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 $(\sigma, \sigma_1) \in \mathcal{T}_{p_1}$ and $(\sigma, \sigma_2) \in \mathcal{T}_{p_2}$ 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 [28]. 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

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

procedure Swish(maxR:int, n:int) returns (numR:int) {
  var oldMaxR:int;
  oldMaxR := maxR; havoc maxR; assume RelaxedEq(oldMaxR, maxR);
  numR := 0;
  while (numR < maxR && numR < n) numR := numR + 1;
  return;
}
```

Fig. 3 Swish++ open-source search engine example.

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. [7]. The example is a simple model that abstracts many implementation details. It takes as input a threshold for the maximum number of results to display `maxR` and the total number of search results `n`, and returns the actual number of results to display `numR` bounded by `maxR` 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* [7,32]) as the mutual summary $\mathbf{old}(maxR_1 = maxR_2 \wedge n_1 = n_2) \Rightarrow RelaxedEq(numR_1, numR_2)$.

3.4 Preserving Termination

We use relative termination conditions (Sec. 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. Note that the presence of a **`havoc`** statement introduces nondeterminism in p_2 (Fig. 2), in which case the specification only guarantees that p_2 has at least one terminating execution on a common input to p_1 (i.e., it does not guarantee that some executions of p_2 do not terminate). To address this, we transform such *internal* nondeterminism into an input nondeterminism using a standard trick of modeling a **`havoc`** statement as a read from a global stream of unconstrained values [22]. This can be done by introducing an unconstrained global array `a` and a counter `c` into the array, and replacing **`havoc`** `x` with `x := a[c++]`. The array becomes a part of the input, and hence the internal nondeterminism is converted into an input nondeterminism. 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 [22,23,19], 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 manually annotate all intermediate mutual summaries and relative specification conditions for every pair of procedures [19]; 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 [23]. 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 [23] and checking relative termination conditions [19] 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 [23], which advanced state-of-the-art in several ways. First, it can handle procedures (including recursion) in P_1 and P_2 unlike most other product constructions that are intraprocedural [4]. 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 $P_{1\times 2}$ consists of procedures in P_1, P_2 and a set of product procedures described below. The set of globals of $P_{1\times 2}$ 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 $p_{1\times 2}$ whose input (resp. output) parameters are the disjoint union of input (resp. output) parameters of p_1 and p_2. The body of $p_{1\times 2}$ 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 $p_{1\times 2}$ where q_1 and q_2 are executed with inputs i_1, i_2 resp. and produce outputs o_1, o_2 resp. (where both inputs and outputs include global mutable state), we constrain (o_1, o_2) to be the output of executing $q_{1\times 2}$ 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. Fig. 4 gives an example of a product program.

```
var str1:[int]int, str2:[int]int;
const x1:int, x2:int, y1:int, y2:int;

procedure Helper_HelperApprox(i1:int, i2:int)
ensures MS_Helper_HelperApprox(i1,i2,...,str1,str2); // mutual summary
{
  var tmp1:int, tmp2:int, in_i1:int, in_i2:int;
  // auxiliary variables
  var in_str1:int[int], out_str1:int[int], st_str1:int[int],
      in_str2:int[int], out_str2:int[int], st_str2:int[int];
  var b1:bool, b2:bool;                     // call witness vars

  b1, b2 := false, false;                   // initialize call vars

  // inline procedure Helper
  if (str1[i1] != 0) {
    tmp1 := str1[i1];
    str1[i1] := tmp1==x1 ? y1 : tmp1;
    in_i1, in_str1 := i1+1, str1;           // store inputs
    call Helper(i1+1);
    b1 := true;                             // record call
    out_str1 := str1;                       // store outputs
  }

  // inline procedure HelperApprox
  if (str2[i2] != 0) {
    tmp2 := str2[i2];
    havoc tmp2;
    str2[i2] := tmp2==x2 ? y2 : tmp2;
    in_i2, in_str2 := i2+1, str2;           // store inputs
    call HelperApprox(i2+1);
    b2 := true;                             // record call
    out_str2 := str2;                       // store outputs
  }

  // constrain calls
  if (b1 && b2) {                           // for pair of calls
    st_str1, st_str2 := str1, str2;         // store globals
    str1, str2 := in_str1, in_str2;
    call Helper_HelperApprox(in_i1, in_i2);
    assume(str1 == out_str1);               // constrain outputs
    assume(str2 == out_str2);               // constrain outputs
    str1, str2 := st_str1, st_str2;         // restore globals
  }
}

procedure ReplaceChar_ReplaceCharApprox()
ensures MS_ReplaceChar_ReplaceCharApprox(...); // mutual summary
{...}
```

Fig. 4 Product program constructed from the example in Fig. 2 and its approximation.

In this paper, we realize that the resultant product program has the following property that was not observed before:

> For any product procedure $p_{1\times 2} \in P_{1\times 2}$, if a 2-program 2-state expression e is satisfied by every $(\sigma_{1\times 2}, \sigma'_{1\times 2}) \in \mathcal{T}_{p_{1\times 2}}$, 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 $p_{1\times 2}$, it is a valid mutual summary for the pair of procedures p_1 and p_2. Informally, this follows from the fact that the bodies of procedures p_1 and p_2 are simply inlined into $p_{1\times 2}$, the added statements for the recording of global state do not alter their executions (but only influence the auxiliary ghost state), and the assumed mutual summaries get proven in the called procedures. 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 Sec. 3) as a specification over the product procedure $p_{1\times 2}$, and verify $P_{1\times 2}$ using any off-the-shelf program verifier. The observation also allows us to automatically infer mutual summaries, as described next, which has not been done before.

4.2 Invariant Inference

To verify a mutual summary, we annotate the resultant product program $P_{1\times 2}$ with a summary of the top-level procedures, and let a program verifier infer intermediate specifications (preconditions and postconditions of intermediate $q_{1\times 2}$ 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 [24]) results in generation of single-program invariants and fails to infer relational 2-program specifications. Therefore, SymDiff exploits the mapping between parameters and globals to automatically 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 [16] or Houdini [14]. SymDiff leverages Houdini (that only infers subsets of these predicates) since it is typically fast and predictable, and has been shown to scale to very large programs [45]. 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 $P_{1\times 2}$ 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 [19]. 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 each 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 (Sec. 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., R_p overapproximates $\mathcal{T}_p$). 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 $(\sigma_2, \sigma_2') \in R_{p_2}$:

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

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** $\exists \sigma_2' \mathbin{.} R_{q_2}(\sigma_2, \sigma_2')$, 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. Since R_{q_2} is uninterpreted, the only way to prove this assertion is to use an axiom like above (just instantiated for procedure q), which requires R_{q_1} and $RT_{q_{1\otimes 2}}$ to hold. Intuitively, such an assertion before every call (which is the only way to cause non-termination in the absence of loops) when combined with the introduced axioms ensures that any 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. A failing assertion indicates there exists a path where a call to q_2 is not preceded by a call to q_1, and hence we failed to prove that q_2 is called less often than q_1, in which case termination cannot be ensured.

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 Sec. 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.

In this work, 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.

5.1 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. [7]: *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*.
- Loop approximation examples: *Cube Root*, *Gradient Descent*, *Loop Perforation*, and *Pointer Perforation*.

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

– Image processing programs taken from the ACCEPT benchmark suite [37]: *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. To make our experiments easily reproducible, we created a custom Apt platform[2] profile containing SymDiff and all benchmarks available at `https://www.aptlab.net/p/fmr/approx-nfm2016`.

5.2 Results

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*) such that simple relational predicates can be inferred to facilitate the proofs. For example, unrolling once the

[2] Apt is an open platform for sharing research developed at the University of Utah.

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

procedure SelectionSort() {
  var c:int, position:int, tmp:int;
  c := 0; position := 0; tmp := 0;
  while (c < (n - 1)) {
    call position := Find(c);
    if (position != c) {
      tmp := array[position];
      array[position] := array[c];
      havoc tmp;
      array[c] := tmp;
    }
    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. 5 Selection sort.

loop in procedure `Strcpy` (see Fig. 1) helps to establish a simple but important invariant $i_1 = i_2$. (This can be automated by trying in parallel all combinations of unrollings: unroll first procedure, unroll second procedure, unroll both.) 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 $j_2 \le i_2$ (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 Houdini-based inference techniques successfully generated most of the required specifications automatically, indicating that relative specifications do not heavily depend on complex program-specific invariants.

5.3 Experience

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

5.3.1 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 defin-

```
function Relaxed(x:int, y:int, e:int) returns (bool) {
  x <= y + e && y <= x + e
}
function RelaxedAll(x:[int]int, y:[int]int, e:int) returns(bool) {
  (forall j:int :: (j>=0 && j<=63) ==> Relaxed(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) ==> Relaxed(x[j], y[j], e))
}
function ShortInt(x:int) returns (bool) {-32768<=x && x<=32767}

const quantTable:[int]int;
var tmp:[int]int;

procedure Quantization(data:[int]int)
modifies tmp;
requires (forall j:int :: ShortInt(data[j]));
{
  var i:int, value:int, dataOld:[int]int;
  dataOld := data; havoc data; assume RelaxedAll(data, dataOld, 16);
  i := 63;
  while(i >= 0) {
    value := data[i] * quantTable[i];
    value := sdiv(value + 16384, 32768);
    tmp[i] := value;
    i := i - 1;
  }
}
```

Fig. 6 JPEG quantization.

ing a quantified predicate template $ArrayEqAfter(str_1, str_2, i_1) \doteq \forall j \,.\, j \geq i_1 \Rightarrow str_1[j] = str_2[j]$. The proof of control flow safety for the selection sort example shown in Fig. 5 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. Our experience shows that *ArrayEqAfter* and *ArrayEqBefore* are needed for most examples with arrays, and hence we automatically instantiate them using our inference engine.

5.3.2 JPEG Quantization

Fig. 6 shows the source code of a JPEG encoder quantization stage taken from the ACCEPT benchmark suite [37]. Each element in `data` gets its quantized value stored in `tmp` 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

```
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. 7 Simple cube root calculation.

an approximation that allocates `data` in approximate memory since the error `e` introduced to the stored value (denoted by the predicate *Relaxed*) 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:

$$\textbf{old}(data_1 = data_2) \Rightarrow (\forall i\ .\ (i \geq 0 \wedge i \leq 63) \Rightarrow Relaxed(tmp_1[i], tmp_2[i], 2))$$

The most involved manually provided predicate $RelaxedAfter(tmp_1, tmp_2, i)$ is similar to *ArrayEqAfter*. It is based on the observation that after each iteration of the loop, all corresponding elements of the arrays tmp_1 and tmp_2 after index i should satisfy *Relaxed* with the error bound of 2. The main difference between predicates *RelaxedAfter* and *ArrayEqAfter* is the posteriori of the universally quantified expression, which also shows up in the accuracy specification. Therefore, it is possible to completely automate the proof of this example, and other similar examples in the ACCEPT benchmark suite, by extracting the quantity predicate (*Relaxed*) and plugging it into the template derived from predicates *RelaxedAfter* and *ArrayEqAfter*.

5.3.3 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.

5.3.4 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` (see Fig. 7). 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 $r_1 \leq r_2$. Moreover, the proof is completely automated with relative termination condition inference described in Sec. 4.3. Boolean expression $r_1 \leq r_2$ is an inferred precondition of the product program to verify mutual summaries of the two programs in Fig. 7. Although it is obtained with the assumption that both programs terminate for the verification of mutual summaries, this expression serves as a valid relative termination condition under which the approximate program terminates at least as often as its original version. The heuristic of using inferred preconditions of the product program for checking mutual summaries as relative termination condition is proven effective by this benchmark as well as other benchmarks in our expriment.

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 approaches, and (iii) dynamic approaches. Language based approaches propose language constructs and annotations to make approximations explicit in a program. EnerJ [39] introduces approximate types and ensures that such values do not impact precise computations, including conditional statements. ACCEPT [37] automatically searches for code regions that can be approximated based on type annotation and static compiler analysis pass. FlexJava [29] allows users to annotate scoped variables (e.g., return values), and then it automatically infers safe-to-approximate variables and operations using a simple taint analysis. Our work can be used to improve the soundness and precision of these analyses. For example, ACCEPT aims to prevent unacceptable approximations of a program by executing it on a sample of its inputs, while our approach can rigorously prove relative accuracy specifications over all the input values. On the other hand, FlexJava relies on an imprecise taint analysis to discover parts of a program that can be approximated, while our SMT-based approach is path-sensitive and hence more precise.

Carbin et al. [7] develop a special-purpose language and constructs for introducing approximations and relaxed specifications (based on *relational Hoare logic* [4]), and prove correctness of transformations using the general purpose Coq theorem prover [10]. 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 Sec. 5), thereby showing that mutual summaries and SMT-based verification can significantly improve the automation for most transformations covered by this approach.

Rely [8] is a programming language that allows users to verify probabilistic quantitative reliability guarantees of programs running on unreliable hardware using an associated static analysis. Chisel [25] 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 and it ensures control flow equivalence using a simple dependence analysis. On the other hand, Chisel can reason about probabilities, which our approach currently does not support. ExPAX [30] generates a set of safe-to-approximate operations based on a dataflow taint analysis, and then computes allowed approximation for each operation to minimize energy consumption while satisfying reliability constraints.

DECAF [6] combines static type inference, dynamic tracking, and runtime check to give probabilistic guarantee on the quality of approximate programs.

Learning based approaches have become popular to control the quality of program approximations. Green [1] builds a quality of service (QoS) model by sampling a training set that describes QoS loss with respect to approximations. Capri [43] uses machine learning to construct an error model that quantifies the probability that an approximation configuration generates acceptable results. Both Green and Capri leverage learned models to synthesize approximate applications that satisfies accuracy specifications. However, these two approaches provide only statistical or even little guarantee on the accuracy of approximate output. Our approach, on the other hand, can be easily extended to synthesize approximate programs with formal accuracy guarantees.

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 [41] and LLFI [44] 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., [33, 34]). 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 [35], the reasoning is only for a subset of all the possible executions of the program.

Finally, our work is related to previous approaches to translation validation [27, 31] and regression verification [13, 15], which leverage SMT solvers to discharge equivalence properties. In contrast, our mutual summaries and product construction allow for richer relaxed specifications other than equivalence, interprocedural reasoning [19, 23], 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 extending this work with support for reasoning about floating-point programs, which often contain ample opportunities for introducing approximations.

Acknowledgements We thank Adrian Sampson for his feedback and for helping out with benchmark selection, and Akash Lal for assisting with Houdini. This work was supported in part by NSF awards 1255776 and 1552975, and SRC contract 2013-TJ-2426.

References

1. W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *ACM SIGPLAN Conf. on Programming*

Language Design and Implementation (PLDI), pages 198–209, 2010.

2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2006.
3. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press, 2009.
4. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 14–25, 2004.
5. J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain$< T >$: A first-order type for uncertain data. In *ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 51–66, 2014.
6. B. Boston, A. Sampson, D. Grossman, and L. Ceze. Probability type inference for flexible approximate programming. In *ACM SIGPLAN Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 470–487, 2015.
7. M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 169–180, 2012.
8. M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM SIGPLAN Intl. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 33–52, 2013.
9. L. N. Chakrapani, J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. Probabilistic design: A survey of probabilistic CMOS technology and future directions for terascale IC design. In *Intl. Conf. on Very Large Scale Integration of System on Chip (VLSI-SoC)*, pages 101–118, 2006.
10. The Coq proof assistant. `http://coq.inria.fr`.
11. D. Elenbogen, S. Katz, and O. Strichman. Proving mutual termination. *Formal Methods in System Design*, 47(2):204–229, 2015.
12. H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. *Commun. ACM*, 58(1):105–115, 2014.
13. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *ACM/IEEE Intl. Conf. on Automated Software Engineering (ASE)*, pages 349–360, 2014.
14. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Intl. Symp. of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*, pages 500–517, 2001.
15. B. Godlin and O. Strichman. Regression verification. In *Design Automation Conf. (DAC)*, pages 466–471, 2009.
16. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 72–83, 1997.
17. P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, and D. Sylvester. Underdesigned and opportunistic computing in presence of hardware variability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(1):8–23, 2013.
18. J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *IEEE European Test Symp. (ETS)*, pages 1–6, 2013.
19. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Towards modularly comparing programs using automated theorem provers. In *Intl. Conf. on Automated Deduction (CADE)*, pages 282–299, 2013.
20. H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–212, 2011.
21. L. Kugler. Is "good enough" computing good enough? *Commun. ACM*, 58(5):12–14, 2015.
22. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 712–717, 2012.
23. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE)*, pages 345–355, 2013.
24. K. L. McMillan. Lazy annotation revisited. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 243–259, 2014.

25. S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. *SIGPLAN Not.*, 49(10):309–328, 2014.
26. S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *ACM/IEEE Intl. Conf. on Software Engineering (ICSE)*, pages 25–34, 2010.
27. G. C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2000.
28. J. Nelson, A. Sampson, and L. Ceze. Dense approximate storage in phase-change memory. In *Ideas and Perspectives session at ASPLOS*, 2001.
29. J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. FlexJava: Language support for safe and modular approximate programming. In *ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE)*, pages 745–757, 2015.
30. J. Park, K. Ni, X. Zhang, H. Esmaeilzadeh, and M. Naik. Expectation-oriented framework for automating approximate programming. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.
31. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 151–166, 1998.
32. M. Rinard. Acceptability-oriented computing. In *ACM SIGPLAN Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 221–239, 2003.
33. M. F. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Dynamic analysis of approximate program quality. Technical Report UW-CSE-14-03-01, University of Washington, 2014.
34. M. F. Ringenburg, A. Sampson, L. Ceze, and D. Grossman. Profiling and autotuning for energy-aware approximate programming. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.
35. P. Roy, R. Ray, C. Wang, and W.-F. Wong. ASAC: Automatic sensitivity analysis for approximate computing. In *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 95–104, 2014.
36. A. Sampson. *Hardware and Software for Approximate Computing.* PhD thesis, University of Washington, 2015.
37. A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. ACCEPT: A programmer-guided compiler framework for practical approximate computing. Technical Report UW-CSE-15-01-01, University of Washington, 2015.
38. A. Sampson, J. Bornholt, and L. Ceze. Hardware-software co-design: Not just a cliché. In *Summit on Advances in Programming Languages (SNAPL)*, pages 262–273, 2015.
39. A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 164–174, 2011.
40. A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 112–122, 2014.
41. V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan. Towards formal approaches to system resilience. In *IEEE Pacific Rim Intl. Symp. on Dependable Computing (PRDC)*, pages 41–50, 2013.
42. S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Joint Meeting of the European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE)*, pages 124–134, 2011.
43. X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. Proactive control of approximate programs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 607–621, 2016.
44. A. Thomas and K. Pattabiraman. LLFI: An intermediate code level fault injector for soft computing applications. In *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.
45. J. Vanegue and S. K. Lahiri. Towards practical reactive security audit using extended static checkers. In *IEEE Symp. on Security and Privacy*, pages 33–47, 2013.
46. Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 441–454, 2012.