Towards Automated Differential Program Verification for Approximate Computing

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

Zvonimir Rakamarić
School of Computing, University of Utah, UT, USA
zvonimir@cs.utah.edu

Abstract

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

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

1. Introduction

Continuous improvements in per-transistor speed and energy efficiency are fading, while we face increasingly important concerns of power and energy consumption, along with ambitious performance goals. The emerging area of *approximate computing* aims at lowering the computational effort (e.g., energy) of an application through controlled (small) deviations from the intended results. There is a growing need to develop *formal* and *automated* techniques that allow approximate computing trade-offs to be explored by developers.

Prior research has ranged from the use of types [18], static reliability analysis [8] or interactive theorem provers [7] to study the effects of approximations while also providing formal guarantees. While these techniques have significantly increased the potential to employ approximate computing in practice, a drawback is that they either lack the required level of precision or degree of automation. More importantly, these works do not harness the continuous advances in Satisfiability Modulo Theories (SMT) [5] based automatic software verification [3, 15]. SMT-based approaches have the potential of providing a good balance of precision and scalability, without sacrificing automation, at least for a large class of programs written in imperative languages such as C/C++, Java, or C#.

In this work, we apply *automated differential program verification* [10, 13] (implemented in SymDiff [12]) towards the problem of *logical*[1] reasoning about program approximations. Previous work has shown that structural similarity of closely-related programs can be exploited to perform automated verification of relative safety for assertions [13]. Although formalisms based on *Relational Hoare Logic* [6] have been around for reasoning about relational properties, such verifiers are mostly based on interactive theorem provers (e.g., Coq [1]). This precludes leveraging automatic verification condition generation [4], SMT-based checking, and invariant inference. In this work, we unify two ideas in SymDiff to harness the power of SMT solvers towards differential verification. First, we use the concept of *mutual summaries* for specifying relational (two-program) properties related to approximation [10]. Second, we use a novel product program construction for *differential assertion checking* that permits procedural programs, and allows leveraging off-the-shelf program verifiers and invariant inference engines [13]. We describe how the construction can be used to check mutual summary specifications as well. The framework enables inference of simple relational invariants using existing program verifiers. In particular, we can use a form of predicate abstraction in a scalable manner to infer many relational properties.

We have applied SymDiff towards an approximate computing case study to illustrate the modeling, specification, and proof of several acceptability conditions for approximate transformations studied by Carbin et al. [7]. Carbin et al. developed a domain-specific language for specifying approximations and acceptability conditions, and performed the verification of several examples using interactive theorem prover Coq. These examples cover approximations due to truncating loops, unreliable memory, and relative memory safety [13]. Overall, their proofs for three examples required around 955 lines of Coq proof script — this makes it difficult to scale the effort to larger programs or hundreds of such programs. In contrast, our verification in SymDiff requires less than 10 lines of specifications. [2]

2. Differential Program Verification

We briefly cover our recent works on differential program verification [10, 13] to verify (relational) properties over two programs, as implemented in the SymDiff tool [12]. SymDiff uses *mutual summaries* [10] as a specification mechanism for relational properties. Given two versions of a procedure, a mutual summary is an expression over their input and output variables that relates the pre- and post-states of the two versions. The following is a simple mutual summary example:

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

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

[2] More details can be found in an accompanying technical report [14].

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

Figure 1. Swish++ Example

We check such mutual summary specifications using a method based on a novel product program transformation [13]. Although the transformation was proposed for *differential assertion checking*, we show that the construction can be used to check more general mutual summary specifications. These mechanisms are well-suited for reasoning about programs with multiple (recursive) procedures. More importantly, the technique allows for leveraging any off-the-shelf invariant inference engine to infer intermediate specifications required to prove the desired specification. In this work, we leverage the implementation of the Houdini [9] (monomial predicate abstraction) inference technique available in Boogie [11]. This allows SymDiff to communicate domain-specific (mutual specifications) to the invariant inference engine. Using this inference technique, we are able to infer many intermediate invariants for many realistic examples, as we discuss next. [3]

3. Acceptability of Approximate Programs

We illustrate the application of differential verification towards two examples from a recent work by Carbin et al. [7]. The authors developed a special-purpose language to specify the transformations and reason about the relaxed specifications. They used the general purpose Coq theorem prover [1] to discharge proof obligations; each proof required roughly 300 lines of proof scripts according to the authors. By leveraging existing program verifiers and SMT solvers, we obtain the proofs almost completely automatically. In both cases, the final verification takes less than a second.

3.1 Dynamic Knobs

Fig. 1 gives the example from an open-source search engine *Swish++*. The approximation (referred to as Dynamic Knobs) allows the search engine to trade-off the number of search results to display to the user under heavy server load. The approximation is justified as users are typically interested in the top few results, and care more about the performance of displaying the search results. The program `swish` takes as input (a) a threshold for the maximum number of results to display `max_r`, and (b) the total number of search results `N`. It returns the number `num_r` denoting the actual number of results to display, which has to be bounded by `max_r` and `N`.

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

[3] Details of other examples and benchmarks can be found at `http://1drv.ms/1A5wuuj`

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

Figure 2. LU Decomposition Example

Relaxed Specification The relaxed specification (akin to *acceptability property* [7, 17]) can be expressed as a mutual summary over the original and approximate version (prefixed with `v1.` and `v2.` respectively) of `swish` as follows:

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

The user only has to specify the postcondition

$$\mathbf{ensures}\ \mathtt{v1.RelaxedEq}(\mathtt{v1.num\_r}, \mathtt{v2.num\_r})$$

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

Verification We required few additional intermediate specifications (beyond the relaxed specification) for the proof. Recall that loops are automatically extracted as tail-recursive procedures in SymDiff. The additional intermediate specification is the relational expression $\mathtt{v1.RelaxedEq}(\mathtt{v1.num\_r}, \mathtt{v2.num\_r})$ as both **requires** and **ensures** for the product of the two loop-extracted procedures. All the remaining invariants are automatically inferred. In comparison, the Coq proof comprised of 330 lines of proof script.

3.2 Approximate Memory and Data Type

Fig. 2 gives a portion of the *LU Decomposition* algorithm implemented in SciMark2 benchmark suite [2]. The algorithm computes the index of the pivot row `p` for a column `j`, where the pivot row contains the maximum value among all rows in the column. It returns the index `p` of the pivot in addition to the value of the maximum element in column `j`.

Approximation The underlined statements model the introduction of an error value `e` if the matrix is stored in approximate memory [16]. As before, the predicate `RelaxedEq` denotes the relationship between the original and approximate value read from the memory, which are bounded by a non-negative constant `e`.

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

$$\mathbf{ensures}\ \mathtt{RelaxedEq}(\mathtt{v1.max}, \mathtt{v2.max}).$$

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

References

[1] The Coq proof assistant. `http://coq.inria.fr`.

[2] SciMark 2.0. `http://math.nist.gov/scimark2`.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[4] 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 *International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2006.

[5] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 14–25, 2004.

[7] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM SIGPLAN Conference 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 International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 33–52, 2013.

[9] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe (FME)*, pages 500–517, 2001.

[10] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Towards modularly comparing programs using automated theorem provers. In *International Conference on Automated Deduction (CADE)*, pages 282–299. Springer, 2013.

[11] S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. Intra-module inference. In *International Conference on Computer Aided Verification (CAV)*, pages 493–508, 2009.

[12] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification (CAV)*, pages 712–717, 2012.

[13] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 345–355, 2013.

[14] S. K. Lahiri, A. Haran, S. He, and Z. Rakamarić. Automated differential program verification for approximate computing. Technical report, Microsoft Research, 2015. `http://research.microsoft.com/apps/pubs/default.aspx?id=246381`.

[15] K. L. McMillan. An interpolating theorem prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 16–30, 2004.

[16] J. Nelson, A. Sampson, and L. Ceze. Dense approximate storage in phase-change memory. In *Ideas and Perspectives session at ASPLOS*, 2001.

[17] M. Rinard. Acceptability-oriented computing. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 221–239, 2003.

[18] 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 Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174, 2011.