Automating Regression Verification

Dennis Felsing

Abstract. Regression verification is an approach to prevent regressions in software development using formal verification. The goal is to prove that two versions of a program behave equally or differ in a specified way. We have extended Strichman and Godlin's approach [4,5] for regression verification by relational equivalence and two ways of using counterexamples to refine verification conditions. Furthermore, we have developed a new approach for program equivalence that reduces the equivalence of two programs to Horn constraints over uninterpreted predicates and uses state-of-the-art SMT solvers like Z3 and Eldarica to find these predicates. We have implemented and evaluated our approach and found promising results on a set of non-trivial integer programs that can be proved equivalent automatically.

1 Introduction

Preventing unwanted behaviour, commonly known as *regressions*, is a major concern during software development. Currently the main quality assurance measure during development is *regression testing*. Regression testing uses a manually crafted test suite to check the behaviour of new versions of a program.

For example, consider the following two functions in ANSI C in Figure 1, which both calculate the greatest common divisor of two positive numbers:

```
int gcd1(int a, int b) {        int gcd2(int x, int y) {
  if (b == 0) {                   int z = x;
    return a;                     if (y > 0) {
  } else {                          z = gcd2(y, z % y);
    a = a % b;                    }
    return gcd1(b,a);             return z;
  }                             }
}
```

Fig. 1: Example functions calculating the GCD

To test such a function, multiple test cases would have to be written to cover the entire function behaviour. Designing such regression tests requires such an amount of manual work that typically more than 50%

of the development time is spent on designing test cases [7]. Still, there is no guarantee of finding all introduced bugs.
Another approach to this problem is *formal verification*: The functions *gcd1* and *gcd2* can individually be proved correct with respect to a formal specification of the greatest common divisor, which would imply their equivalence. This requires the software engineer to provide said formal specification. Additionally, it is often necessary to manually guide the proof.
Regression verification offers the best of both worlds: As in formal verification, full coverage is achieved and no test cases are required. No formal specification of function behaviour is required, just as in regression testing. Instead of comparing the two programs to a common formal specification, regression verification compares them to each other. The old program version serves as specification of the correct behaviour of the new one. Note that the "correctness" that regression verification proves is different from that shown using formal verification: In formal verification there may be a degree of freedom for the program behaviour, that allows bugs to be introduced if the specifaction does not fully capture the behaviour of a function.
In regression verification the use of an old program version as specification assures that the full behaviour is preserved, so no new bugs can be introduced at all.
So far regression verification is limited to proving functional relations, such as equivalence, between program versions. Regression testing on the other hand can also be employed to test for nonfunctional requirements, such as performance.
Our contribution is the implementation and extension of the automatic regression verification approach by Strichman and Godlin, as well as the implementation and evaluation of a new approach for regression verfication.
The rest of this paper is structured as follows: In Section 2 Strichman and Godlin's approach, which uses uninterpreted functions for regression verification, is presented and extended. In Section 3 we detail a new approach we developed which performs program equivalence proofs by constraining uninterpreted predicates. Finally in Section 4 the implementation and evaluation of this approach are presented.

2 Overapproximation using uninterpreted functions

An initial approach for regression verification has been developed by Strichman and Godlin [4] and is illustrated in Figure 2.
Proving equivalence of *gcd1* and *gcd2* is difficult since the programs call themselves recursively, potentially an unbounded number of times. Strichman and Godlin propose to replace recursive calls by the same placeholder in both programs, a so called *uninterpreted function U*. They compare the main rule describing their approach to Hoare's rule for recursive invocation:

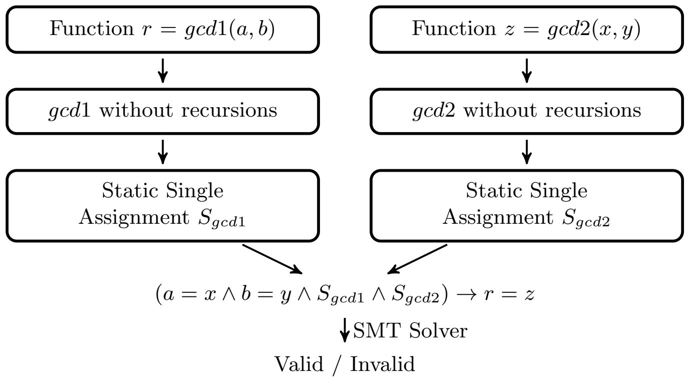
Function $r = gcd1(a, b)$
Function $z = gcd2(x, y)$
$gcd1$ without recursions
$gcd2$ without recursions
Static Single Assignment S_{gcd1}
Static Single Assignment S_{gcd2}
$(a = x \wedge b = y \wedge S_{gcd1} \wedge S_{gcd2}) \to r = z$
SMT Solver
Valid / Invalid

Fig. 2: Regression verification approach by Strichman and Godlin

> [Hoare's rule for recursive invocation] was described by Hoare as follows: *The solution... is simple and dramatic: to permit the use of the desired conclusion as a hypothesis in the proof of the body itself.* The correctness of [the rule] is proved by induction, where the base case corresponds to the base(s) of the recursion, namely the nonrecursive run(s) through the procedure. ([5])

Consider the two functions in Figure 1. Initially we transform them by simply replacing the recursive function calls by an uninterpreted function U, as can be seen in Figure 3.

```
int gcd1(int a, int b) {
  if (b == 0) {
    return a;
  } else {
    a = a % b;
    return U(b,a);
  }
}
```

```
int gcd2(int x, int y) {
  int z = x;
  if (y > 0) {
    z = U(y, z % y);
  }
  return z;
}
```

Fig. 3: Replaced function calls with calls to the uninterpreted function U

The next step is to transform the bodies of these functions to static single assignment (SSA) form. Subsequently SSA formulas are constructed as a conjunction of assignments expressed as equalities. These formulas model the behaviour of *gcd1* and *gcd2* respectively, relating function outputs to inputs. This means that in all assignments of the form *x := exp;* we replace x with a new unused variable x_i. Every conjunct of the SSA formula represents a state of the corresponding procedure.
Hence, S_{gcd_1} and S_{gcd_2} imply the equality of respective output values ($result_{gcd_1} = result_{gcd_2}$), assuming equality of inputs ($a = x \wedge b =$

$$S_{gcd_1} = \begin{pmatrix} a_0 = a & \wedge \\ b_0 = b & \wedge \\ b_0 = 0 \rightarrow result_0 = a_0 & \wedge \\ b_0 \neq 0 \rightarrow a_1 = a_0 \% b & \wedge \\ b_0 \neq 0 \rightarrow result_1 = U(b_0, a_1) & \wedge \\ result_{gcd_1} = result_1 & \end{pmatrix}$$

(a) SSA formula of *gcd1*

$$S_{gcd_2} = \begin{pmatrix} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ y_0 > 0 \rightarrow z_1 = U(y_0, (z_0 \ \% \ y_0)) & \wedge \\ y_0 \leq 0 \rightarrow z_1 = z_0 & \wedge \\ result_{gcd_2} = z_1 & \end{pmatrix}$$

(b) SSA formula of *gcd2*

Fig. 4: Example of transforming functions into static single assignment formulas

y). The following formula can now be constructed to express functional equivalence:

$$(\underbrace{a = x \wedge b = y}_{\text{Equal inputs}} \wedge S_{gcd_1} \wedge S_{gcd_2}) \rightarrow \underbrace{r = z}_{\text{Equal outputs}} \tag{1}$$

This formula is passed to an SMT solver, like Z3 or Eldarica, which can have three results:

1. The SMT solver proves the formula (1) valid:. This implies the two functions are actually equal.
2. The SMT solver finds a counterexample dissatistfying (1). As we overapproximate the recursive function calls, this does not necessarily imply actual inequality.
3. The SMT solver times out.

Strichman and Godlin use the bounded model checker CBMC to prove this formula for C programs on bitvectors.
We implemented this approach in the tool *simplRV* (**Sim**ple **P**rogramming **L**anguage **R**egression **V**erification), which is capable of performing regression verification on unbounded integer and array functions in a simple imperative programming language featuring recursions as well as loops, but no global variables. For this programming language we use an interpreter developed in a student project in the tool *Kammerjäger*.
simplRV produces an SMT formula which is passed to state-of-the-art SMT solvers like Z3 and Eldarica. Our implementation differs from that provided by Strichman and Godlin in their *RVT* tool. Instead of building a logical formula, *RVT* creates a C function that calls the functions to be compared with the same, but uninterpreted, inputs, and compares their results. Instead of an SMT solver, the bounded model checker *CBMC* is then used to verify this new function.

We developed and implemented the following extensions to the existing approach within *simplRV*:

Conditional Equivalence Total equivalence between the functions to be compared is not always desired. Consider our example in Figure 1: The proof of equality fails for negative numbers, but one can imagine that these functions are only called with positive numbers. In this case we require *conditional equivalence* for nonnegative inputs by assuming an additional precondition $a \geq 0$:

$$(\mathbf{a} \geq \mathbf{0} \wedge a = x \wedge b = y \wedge S_{gcd1} \wedge S_{gcd2}) \rightarrow r = z \tag{2}$$

Another common example for conditional equivalence are bug fixes in the program. Once a bug has been fixed, an equivalence proof is still desirable to prevent the introduction of new bugs. But simple equivalence of all outputs for all inputs would not be correct in this case. Instead the case of the bug fix have to be excluded using a precondition.

Relational Equivalence We implemented *relational equivalance* (denoted as "$\simeq$"), which is a superset of conditional equivalence, so that the user can specify relations between the inputs and outputs of functions. An example for this are functions which are off by one in a new version, but otherwise should still behave equally. By default equality is used as the relation.

Counterexamples Our tool makes use of *counterexamples* returned by the SMT solver on a failed proof. The functions are automatically run on the input values of the counterexamples. If their outputs differ, the programs are not equivalent and the user is informed about this with the counterexample.

Additional Information Spurious counterexamples can be returned by the SMT solver because we overapproximate the functions using an uninterpreted function. We use the information won from spurious counterexamples as additional constraints on the uninterpreted function U and rerun the proof. This is a decision procedure for the cases where all function arguments are inductive in their recursive calls.

A summary of our extensions to the initial approach is given in Figure 5.

Using a collection of examples from various sources including compiler optimizations, refactorings and other publications we evaluated our approach and found it to work well for a wide range of examples.

Utilizing the information of spurious counterexamples can lead to an endless loop of new spurious counterexamples. These so called *edge cases* occur when at least one of multiple parameters is not inductive. Proving equivalence of functions of this kind is a limitation of the approach just described. A more intricate view on the problem can help, which we will discuss in the next section.

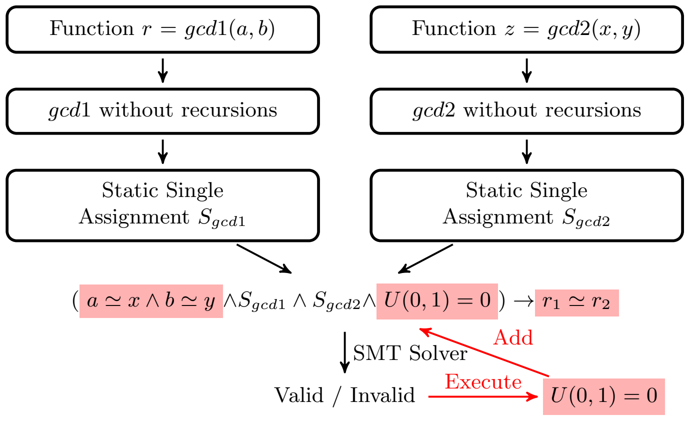
Function $r = gcd1(a,b)$
Function $z = gcd2(x,y)$
$gcd1$ without recursions
$gcd2$ without recursions
Static Single Assignment S_{gcd1}
Static Single Assignment S_{gcd2}
$(a \simeq x \wedge b \simeq y \wedge S_{gcd1} \wedge S_{gcd2} \wedge U(0,1) = 0) \rightarrow r_1 \simeq r_2$
SMT Solver
Add
Valid / Invalid
Execute
$U(0,1) = 0$

Fig. 5: Extended regression verification approach

3 Approximation using uninterpreted predicates

So far we overapproximated recursive function calls using an uninterpreted function, so that we transform $r = gcd_1(a,b)$ and $z = gcd_2(x,y)$ into a formula of the pattern

$$\forall U.(\bigwedge_i \varphi \rightarrow \forall U.\ \ldots \wedge S_{gcd_1} \wedge S_{gcd_2} \rightarrow r = z). \tag{3}$$

We can modify this by abstracting both recursive calls using a single predicate R, a so called *mutual function summary*, that describes the relation between the arguments and result values of both recursive calls to one another:

$$\exists R.(\ldots \wedge R(a,b,r,x,y,z)) \rightarrow r = z. \tag{4}$$

We use the same symbolic execution of both functions as before. The only difference is in the interesting part, the handling of recursive calls. Universal quantification over all uninterpreted functions U was implicitly assumed in the first approach. Now instead of showing that the formula is true for all predicates U, we have to find a single common predicate R that satisfies the formula.
The mutual function summary replaces and approximates the recursive function calls. It is required that the mutual function summary R is a faithful abstraction, which is achieved by making it model both functions' behaviour.
In the cases where only one function recurses, an uninterpreted function summary R_1 for the first function, respectively R_2 for the second function, is inserted instead. These predicates only relate one function's inputs to its outputs, just like the uninterpreted functions in the previous approach.

State-of-the-art SMT solvers like Eldarica and Z3 can be used to automatically infer a predicate R, which satisfies our requirements, namely is a faithful abstraction of the behaviour of both functions and implies the equivalence of outputs.
We developed a similar approach for functions containing while loops instead of recursive function calls. An uninterpreted *mutual loop invariant* C is used as a predicate relating the variables in a loop after each loop iteration. The invariant C is true at the start of the loop. Assuming C to hold, after the single execution of both loop bodies, the invariant C holds again, relating the values in both loops to each other. Once neither loop guard holds, the mutual loop invariant C is used to imply the equality of the rest of the functions. In this approach an invariant C can be inferred automatically as well.
In order to derive invariants and mutual loop invariants verification conditions are represented in form of *Horn clauses* over uninterpreted relation symbols. A big advantage of our approach is that the invariants and mutual loop invariants for similar programs are linear, which makes them easy to automatically infer.

4 Implementation and Experiments

Implementation We have implemented our second approach for a subset of ANSI C in a tool named RÊVE[1]. Program data is limited to local variables and function parameters of type `int`, which is interpreted as unbounded (mathematical) integers. Bounded integers can be simulated by instrumenting programs with modulo operations, at the cost of increased reasoning complexity. Supported control structures are if-then-else and while statements, function calls and returns. The return statement must always be the last statement of a function and must return a local variable. Recursive function calls may not occur within the conditions of if or while statements. Using our tool, pairs of recursive functions as well as functions containing while loops can be proven equivalent, but not functions containing both recursive calls as well as while loops.
The tool (i.e., the *wlp* calculus) is implemented in Standard ML. As Horn constraint solvers we used Z3 (unstable branch, 2013-11-27) and Eldarica (version from 2014-04-16).

Experiments We have evaluated the effectiveness and performance of our tool on a collection of benchmarks. The benchmarks vary in size from 10–50 lines of code (for both programs together). Benchmark results are summarized in Table 1. We also give results from the only automatic tool that is directly comparable to ours, the Regression Verification Tool (RVT) by Strichman and Godlin [4].
The programs in the first group in Table 1 are recursive, while the ones in the second group contain loops. Benchmarks where the two programs were not equivalent are in the third group, and their names end with a bang (`!`). All other benchmarks contain equivalent programs; the ✗ outcome is in this case a false negative.

[1] RÊVE is available at `http://formal.iti.kit.edu/improve/ase2014/`

Benchmarks `limit1` to `limit3` were given by Strichman and Godlin as beyond the limits of their approach to regression verification. Benchmarks `barthe2-big` and `barthe2-big2` embed the benchmark `barthe2` into a larger program that is syntactically identical in both versions. We could not prove equivalent the `ackermann` benchmark, as the result of a recursive function call is used as the argument to another recursive function call. Furthermore, we originally could not prove the `limit1` benchmark, as two steps of the first loop are equivalent to one step of the second loop, an issue that we solve in the next section and illustrate with the larger `digits10` benchmark.

The `triangular-mod` benchmark corresponds to the illustrating example instrumented with modulo operations to simulate integer overflow.

As far as we are aware, RVT does not supply additional information to assist the user in case of a failed proof attempt. While, in theory, the model checker underlying RVT produces a counterexample, such a counterexample can be spurious due to the fixed abstraction employed. The Eldarica solver that we use, in contrast, returns a genuine counterexample for many failed proofs. We found these counterexamples useful in diagnosing problems with the programs, even though we currently do not translate these counterexamples into source code terms.

4.1 An Example for Loop Equivalence

We consider a real-world example from [1]. The program P_1 in Figure 6(a) computes the number of digits in the decimal expansion of `n` through a series of integer divisions by 10. The program P_2 in Figure 6(c) computes the same result but (asymptotically) about seven times faster. This speedup is accomplished by reducing the strength of operations. The loop has been unrolled four times[2] *and* the majority of divisions have been replaced by pure comparisons.

Unsurprisingly, P_1 and P_2 cannot be proved equivalent automatically. To do so, the tool would in the least need to figure out the (very complex) relation between one iteration of the loop in P_1 and four iterations of the loop in P_2. To overcome this barrier, the software engineer needs to supply to the tool the knowledge that an unrolling transformation took place. At the moment, we achieve this transfer by manually carrying out the unrolling on P_1 and producing the intermediate program P_1' shown in Figure 6(b). We then prove automatically that P_1' and P_2 are equivalent. Note that P_1' is still significantly different from P_2, as unrolling is not the only optimization that has been performed originally. The program P_1' still performs four times as many divisions as P_2. The if-conditions directly follow the divisions and depend on them, which slows the program down, while the four if-conditions in P_2 are all dependent on the same division result.

[2] Loop unrolling is a simple transformation, in which the loop body is replicated within the loop and guarded by the loop guard. This transformation preserves the semantics of the program.

```
int f(int n) {
  int r = 1;
  n = n/10;

  while (n > 0) {
    r++;
    n = n / 10;
  }
  return r;
}
```

(a) basic version P_1

```
int f(int n) {
  int r = 1;
  n = n/10;

  while (n > 0) {
    r++;
    n = n / 10;
    if (n > 0) {
      r++;
      n = n / 10;
      if (n > 0) {
        r++;
        n = n / 10;
        if (n > 0) {
          r++;
          n = n / 10;
        }
      }
    }
  }
  return r;
}
```

(b) intermediate version P_1'

```
int f(int n) {
  int r = 1;
  int b = 1;
  int v = -1;

  while (b != 0) {
    if      (n<    10) { v = r;   b = 0; }
    else if (n<   100) { v = r+1; b = 0; }
    else if (n<  1000) { v = r+2; b = 0; }
    else if (n<10000) { v = r+3; b = 0; }
    else {
      n = n / 10000;
      r = result + 4;
    }
  }
  return v;
}
```

(c) optimized version P_2

Fig. 6: Computing the number of digits (`digits10`) from [1]

Table 1: Benchmark results

	Run time (seconds)			
Benchmark	RVT	Rêve+ Z3	Rêve+ Eldarica	Source
ackermann	0.8	–	–	[4]
mccarthy91	1.1	1.8	1.7	[4]
limit1	✗	–	–	[4]
limit2	✗	–	5.2	[4]
limit3	✗	–	4.5	[4]
add-horn	✗	–	4.3	
triangular	✗	–	3.3	
triangular-mod	✗	–	–	
inlining	✗	–	5.7	
simple-loop	0.8	0.1	5.3	
loop	✗	–	2.8	
loop2	✗	–	3.2	
loop3	✗	–	5.4	
loop4	✗	–	27.4	
loop5	✗	–	26.4	
while-if	✗	–	3.8	
digits10	✗	–	11.3	[1]
barthe	✗	–	4.8	[2]
barthe2	0.5	10.2	3.9	[2]
barthe2-big	1.6	–	5.7	
barthe2-big2	1.7	–	8.0	
bug15	1.0	0.1	1.8	[4]
nested-while	1.5	–	5.2	[4]
ackermann!	✗	0.1	4.9	
limit1!	✗	0.0	1.4	
limit2!	✗	0.7	10.9	
add-horn!	✗	0.1	1.9	
triangular-mod!	✗	0.5	28.1	
inlining!	✗	0.1	2.7	
loop5!	✗	0.0	2.2	
barthe!	✗	1.8	21.9	
nested-while!	✗	0.1	5.2	

Dash (–) denotes timeout at 600 seconds, cross (✗) denotes that the tool terminates but cannot prove equivalence. All times have been measured on a 2.5 GHz Intel Core2 Quad machine.

After 11.3 seconds, RÊVE with Eldarica succeeds in proving equivalence with the following automatically inferred coupling predicate:

$$\begin{aligned} &(b_2 = 1 \land r_1 = r_2 \land 10n_1 \leq n_2 \land n_2 \leq 10n_1 + 9) \\ \lor\ &(b_2 = 0 \land r_1 = v_2 \land n_2 \geq 10n_1 \land n_1 \leq 0) \end{aligned}$$

Here, n_1 and r_1 denote the variables of P_1', and n_2, b_2, r_2 the variables of P_2. The variable b_2 indicates whether the loop will ($b_2 = 1$) or will not ($b_2 = 0$) be executed once more. The coupling predicate is hence a disjunction over these two cases: While the loop is iterated, r_1 and r_2 hold the same value and n_1 is one division by 10 ahead of n_2, i.e., $n_1 = n_2 \text{ DIV } 10$. Exactly this fact is expressed by the linear constraint $10n_1 \leq n_2 \land n_2 \leq 10n_1 + 9$. When the loop of P_2 has finished, its negated loop guard $n_1 \leq 0$ holds and the final results are stored in r_1 and v_2.

4.2 Discussion

In addition to the promising experimental results, we argue below that our procedure is applicable for a wide range of practical regression verification problems. On the other hand, since our method exploits structural similarities between the compared programs, it cannot be expected to perform well when exchanging complete algorithms, or when changing the design of a system in a fundamental way. For example, replacing a bubble sort procedure with a quicksort within a bigger program may preserve overall behavior, but automatically proving this is currently not feasible.

Our method works well whenever sufficiently "simple" coupling predicates exist that prove program equivalence. This applies to a number of important cases:

– as a baseline, our procedure will always be able to prove that a program is equivalent to itself, by applying a greedy reduction, and choosing the coupling predicates $\bar{x}_1 = \bar{x}_2$.
– the procedure is also complete when applied to two programs with the same control structure, and locally equivalent (but not necessarily identical) loop and function bodies. In this case, the same coupling predicates $\bar{x}_1 = \bar{x}_2$ can be chosen for the entry points of the bodies.
– the procedure is complete for program transformations that correspond to affine mappings of program states; this includes renaming or exchanging variables, shifting the value of a variable by a constant offset, or changing the sign of some variable.

We currently do not consider equivalence of programs that use arrays or heap data structures, but we intend to work on lifting these limitations in the future. It is, for instance, known that assertions about arrays can be encoded in Horn clauses [3]. An approach to regression verification of programs with tree-shaped heap structures can be presumably adapted from [4].

5 Related work

Regression Verification and variations of it have been subject to many studies:
Godlin and Strichman introduced the term Regression Verification to describe the problem of proving the equivalence of two closely related, successive versions of a program. [4, 5] Their approach is simple, using only a single inference rule to prove the equivalence of two related functions. The equivalence of whole programs is then shown by applying this inference rule bottom-up to the call graphs of both programs. A Regression Verification Tool (RVT) for the C language is provided. All loops are transformed to recursions before the proof starts. The use of a bounded model checker as the base of RVT leads to some limitations compared to an SMT solver, for example no further information about recursive functions can be encoded.
Hawblitzel et al. [6] provide a framework for the user to encode coupling predicates for a simple While language in Boogie, an intermediate verification language developed at Microsoft Research.
Another approach is taken by Barthe et al. [2] Instead of regressions their focus are optimisations. They generalise the self composition commonly used in non-interference checking to two programs and merge them into a single *product program* in a user-specified way.
Verdoolaege et al. [8] have developed an automatic approach for equivalence proofs over static affine programs by matching dependence graphs of two programs. This enables successfull proofs of complex transformations on array programs. The approach is implemented for a subset of ANSI C in the isa tool. The abstraction of arithmetical operations prevents proving many of our examples.

6 Conclusion and future work

We have extended the reach of regression verification by building on the foundation of the approach by Godlin and Strichman, making it usable for a wider range of programs and use cases such as bugfixes. Furthermore, we have developed a new approach that uses invariant inference techniques to conduct regression proofs fully automatically, while still allowing for the user to specify the condition of equality if it is asked for. Our evaluations have shown that a wide range of examples work well with the new approach presented in this paper. The provided implementation is made available publicly.
So far in the new approach only integer programs have been considered. Extending our approach to other constructs like heaps and arrays will make the tool more powerful and enable more use cases.

References

1. Andrei Alexandrescu. Three optimization tips for C++, 2012. `https://www.facebook.com/notes/facebook-engineering/three-optimization-tips-for-c/10151361643253920`.

2. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.
3. Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 246–266, Berlin, Heidelberg, 2010. Springer-Verlag.
4. Benny Godlin and Ofer Strichman. Regression verification. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 466–471. IEEE, 2009.
5. Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.
6. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Mutual summaries: Unifying program comparison techniques. In *Proceedings, First International Workshop on Intermediate Verification Languages (BOOGIE)*, 2011.
7. Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.
8. Sven Verdoolaege, Martin Palkovic, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Experience with widening based equivalence checking in realistic multimedia systems. *J. Electronic Testing*, 26(2):279–292, 2010.