Automating Regression Verification

Dennis Felsing Sarah Grebing Vladimir Klebanov Mattias Ulbrich

Karlsruhe Institute of Technology, Germany

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 worked on an approach for regression verification, extending Strichman and Godlin's work by relational equivalence and two ways of using counterexamples.

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;
  }                             }
}
```

Figure 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. Writing these regression tests requires such an amount of manual work that typically more than 50% of the development time is spent on designing test cases [5]. 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. As in regression testing no formal specification of function behaviour is required. 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 is a degree of freedom for the program behaviour, which allows to introduce certain bugs even when a bug is present.

In regression verification the use of an old program version as specification assures that the exact behaviour is preserved, so no new bugs can be introduced at all.

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.

A number of approaches for regression verification have been developed. [1, 2, 4, 6]

2 Overapproximation using uninterpreted functions

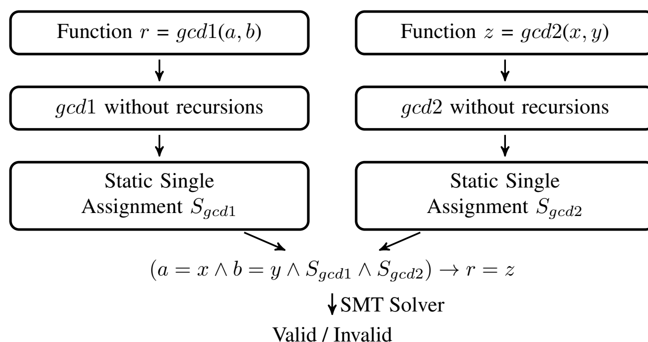

Figure 2: Regression verification approach by Strichman and Godlin

An initial approach for regression verification has been developed by Strichman and Godlin and is illustrated in Figure 2 [3]:

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.

Afterwards the programs can be converted into logical formulae S_{gcd_1}, S_{gcd_2}, which incorporate U. These formulae model the behaviour of *gcd1* and *gcd2* respectively, relating function outputs to inputs. Hence, S_{gcd_1} and S_{gcd_2} imply the equality of respective output values, assuming equality of inputs, in the following way:

$$(a = x \wedge b = y \wedge S_{gcd1} \wedge S_{gcd2}) \rightarrow r = z \qquad (1)$$

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. simplRV outputs an SMT formula, which is passed to state-of-the-art SMT solvers like Z3 and Eldarica. We developed and implemented the following extensions to the existing approach within *simplRV*:

Total equivalence between the functions to be compared is not always desired. Consider our example in Figure 1: 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:

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

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 has to be excluded using a precondition.

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. By default equality is used as the relation.

Our tool makes use of *counterexamples*, which are returned by the SMT solver on a failed proof. The functions are automatically tested using these counterexamples, and if their outputs differ, the programs are not equivalent and the user is informed about this with an actual counterexample.

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 successfully handles the cases where a finite number of function values serve as the non-recursive base case of the function.

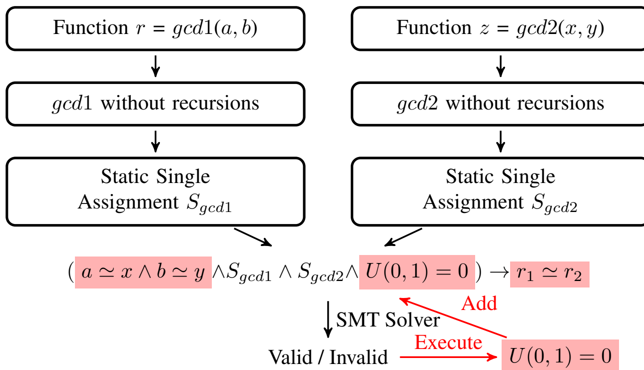

Figure 3: Extended regression verification approach

A summary of our extensions to the initial approach is given in Figure 3.

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 only one of multiple parameters has a base case. 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 is ongoing research.

3 Conclusion and future work

We have extended the reach of regression verification. This enables us to prove a greater class of functions and to still prove equivalence for the relevant cases when a bug has been fixed in the program.

So far only integer programs have been considered. Extending our approaches to other constructs like heaps and objects will improve comparability to other regression verification approaches and enable more realistic use cases.

References

[1] José Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Verifying cryptographic software correctness with respect to reference implementations. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, volume 5825 of *LNCS*, pages 37–52. Springer Berlin / Heidelberg, 2009.

[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] Benny Godlin and Ofer Strichman. Regression verification. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 466–471. IEEE, 2009.

[4] 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.

[5] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[6] 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.