Regression Verification - a practical way to verify programs

Ofer Strichman Benny Godlin

Technion, Haifa, Israel.
Email: ofers@ie.technion.ac.il bgodlin@cs.technion.ac.il

1 Introduction

When considering the program verification challenge [8] one should not forget a lesson learned in the testing community: when it comes to industrial size programs, it is not realistic to expect programmers to formally specify their program beyond simple assertions. It is well known that large parts of real code cannot be described naturally with high level invariants or temporal properties, and further that it is often the case that the process of describing what a code segment should do is as difficult and at least as complicated as the coding itself. Indeed, high-level temporal property-based testing, although by now supported by commercial tools such as Temporal-Rover[4], is in very limited use. The industry typically attempts to circumvent this problem with *Regression Testing*, which is probably the most popular testing method for general computer programs. It is based on the idea of reasoning by induction: check an initial version of the software when it is still very simple, and then check that a newer version of the software produces the same output as the earlier one, given the same inputs. If this process results with a counterexample, the user is asked to check whether it is a bug or a legitimate change. In the latter case the testing database is updated with the new 'correct' output value. Regression Testing does not require a formal specification of the investigated system nor a deep understanding of the code, which makes it highly suitable for accompanying the development process, especially if it involves more than one programmer. We propose to learn from this experience and develop techniques for *Regression Verification*. The underlying proof engine is still a certifying compiler as envisioned by the grand challenge, so this proposal should be thought of as another application of this technology that makes the verification picture more complete.

While formally proving equivalence between two programs is generally undecidable, if one is willing to sacrifice completeness this becomes not only a decidable problem, but also one that can be built on top of existing tools. Without completeness, the equivalence problem can be reduced to one of proving an assertion on a merged program (see next section), for which functional verification techniques can be used. Thus, Regression verification should be thought of as an additional layer, or dimension, to be dealt with as part of the verification challenge.

When can Regression-verification be useful? A natural question to ask is whether proving equivalence is relevant in the context of a real software development process, as in such a process the program is expected to produce a different output after every revision. While this is in general true, consider the following scenarios, all of which are targeted by our approach:

- Checking side-effects of new code. Suppose, for example, that from version 1.0 to version 1.1 a new flag was added, that changes the result of the computation. It is desirable to prove that as long as this flag is turned off, the previous functionality is maintained. The RV tool we propose will allow the user to express a condition (the activation of the flag in this case) under which the two programs are expected to produce equal outputs.
- Checking performance optimizations. After adding an optimization of the code for performance purposes, it is desirable to verify that the two versions of the code still produce the same output.
- Manual *Re-factoring* (a popular set of techniques for rewriting existing code for various purposes). To quote Martin Fowler [6, 5], the founder of this field, '*Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.'* Equivalence proof, before and after refactoring, seems valuable in this case.

This list demonstrates, but does not exhaust, the scenarios in which such a tool can be used.

□

At the Technion we are currently developing a method for formally verifying the equivalence of two closely related C programs, under certain restrictions inherited from the functional verification tool we use. One of the main challenges is to find ways to benefit from the similarity of the two programs, in order to scale-up the verification system beyond what is currently possible for verifying a single program. In particular, we are trying to make the complexity depend only on the *differences* between the two programs (including the propagation of these changes to other parts of the program), rather than on their original sizes. We dedicate the next section for giving a brief description of this project, after which we will summarize what we believe are the main challenges in making verification by regression widely used.

2 Proving equivalence of two C programs

In the rest of this note we concentrate on C programs. Automated (incomplete) verification tools that work directly on widely used programming languages such

as C were first made available only a few years ago. We are aware of three categories of such tools: 1) Predicate-abstraction-based tools such as BLAST [7], SLAM [2] and MAGIC [3], 2) Symbolic search of a bounded model of the program, as in SATURN [14] and CBMC [10], and 3) Explicit state tools such as SPIN (in combination with FEAVER [9]) and CMC [11]. Each of the tools in these three categories is a candidate for serving as an infrastructure for Regression Verification, although adapting each one of them for this task presents a research challenge by its own right.

Our prototype implementation uses CBMC as the underlying decision procedure, since it is one of the two tools (together with CMC) that supports full C and C++. In the rest of this note we exclusively focus on this category.

2.1 The CBMC tool

CBMC [10], developed by D. Kroening, is a Bounded Model Checking tool for full ANSI-C and C++ programs. For each loop i in the given program, the user is required to specify a bound k_i on the number of iterations. This enables CBMC to symbolically characterize the full set of possible executions restricted by these bounds, with a propositional formula f. The existence of a solution to $f \wedge \neg a$, where a is a user defined assertion, implies that there is a path in the program that violates a. Otherwise, it is still possible that the given bounds are not sufficiently high. CBMC allows the user to check whether the bounds are high 'enough' by generating special *unwinding assertions* for each loop. An unwinding assertion for a loop i, given k_i, is satisfied iff the condition of this loop cannot be true after iterating k_i times. Thus, CBMC can be thought of as a complete tool as long as all the loops are terminating.

2.2 A Regression Verification tool for C programs

We started building a Regression Verification tool that generates a combined C program that is unrolled and checked by CBMC. Our program performs several simplifications and abstractions with Uninterpreted Functions as we explain in Section 2.3. The user is involved in two phases:

- The user needs to supply a list of pairs

$$(\langle label\#1, expression\#1\rangle, \langle label\#2, expression\#2\rangle)\ldots$$

  representing his/her specification that $expression\#1$ in the location specified by $label\#1$ in the first program, should always be equal to $expression\#2$ in location $label\#2$ in the second program. In other words, the sequence of values of these two expressions in these locations should be equal regardless of the inputs. A special case of this option is specifying that the outputs of the program are equal.
- When a counterexample is found, the user needs to confirm whether it represents a bug or a legitimate change resulting from further development of

the investigated program. The problem is that there can be an exponential number of examples due to a single change, so approving them one by one is not a desirable option. We plan to give the user the option of describing symbolically the allowed changes between the two programs, and also of determining dynamically the subset of outputs to concentrate on.

2.3 Optimizations with Uninterpreted Functions

Verification can be made simpler by using automated abstraction and decomposition. In the context of proving equivalence, Uninterpreted Functions has proven to be a highly effective tool (see, for example, the case of Translation Validation [13, 12]).

In the following description we use unprimed and primed variables to distinguish between variables that belong to the old (unprimed) and new (primed) versions of the code. Consider, for example, a function $\texttt{int}f(x_1 \dots x_n)$ that is syntactically equivalent to its counterpart $\texttt{int}f'(x'_1 \dots x'_n)$ in the new code, and assume that they do not call other functions. We would like in this case to hide the content of f and f' while assuring that if the input to both functions is the same, then so is their output and side effects. Assume that f is reading the values of the global variables in a set G_r and writes to a set of global variables G_w (G_r and G_w are not necessarily disjoint)[1]. We observe that if f and f' are called with the same arguments, and all variables in G_r have the same values as their counterparts in G'_r when f and f' are called, then the return value of f and f' is the same, as well as the values in G_w and G'_w. Based on this observation, we represent the functions f and f' with two new variables of type `int`, say f_v and f'_v, and add the following constraint:

$$(\bigwedge_{i=1}^{n} x_i = x'_i \wedge \forall g_r \in G_r . g_r = g'_r) \rightarrow (f_v = f'_v \wedge \forall g_w \in G_w . g_w = g'_w) \qquad (1)$$

This type of (conservative) abstraction can be seen as a simple extension of a method suggested by Ackermann [1], who considered the case in which there are no side effects. There are various complications when using this kind of simplification, some of which are:

- Recursive and mutually recursive functions require proofs by induction.
- A change in a function renders all its ancestors in the call-graph unsuitable for replacement with Uninterpreted Functions. We attempt to minimize this effect in two ways. First, we attempt to prove that although the two functions are syntactically different, they are still semantically the same. Second, we consider *Uninterpreted Scopes*, which are more fine-grained than Uninterpreted Functions, and hence less sensitive to the propagation of changes.

[1] We consider here local variables declared as `static` as a special case of global variables.

– Function arguments and global variables can be pointers, which makes Equation (1) unusable (obviously the two pointers represent different memory addresses). We are currently investigating various syntactic analysis methods in order to be able to check, at least in some cases, whether two pointers point to two isomorphic objects.

2.4 Summary

Comparing to existing C verification tools, all of which are property based, Regression Verification has two things to offer. First, code that cannot be easily checked against a formal specification can still be checked throughout the development process by examining its evolving effect on the output (or, in fact, on internal variables as well, which helps pin-pointing the cause for the difference). Second, comparing two similar systems is in most cases computationally easier than property-based verification[2]. The reason for this is that there are various optimizations and decomposition opportunities that are only relevant when comparing two closely related systems. We described one such optimization based on Uninterpreted Functions in section 2.3.

3 The road ahead

Verification by regression poses numerous technical challenges, some of which are:

– Adapting other existing techniques for formally verifying a single program to proving equivalence, most notably predicate abstraction. Our current choice of the C language and the tool CBMC as a starting point is largely due to the maturity of the tool rather than some deep theoretical reason.
– Achieving scalability and automation beyond what is possible in functional verification. Abstracting portions of the program with uninterpreted functions, as explained above, is one possible technique that is challenging by itself in the presence of dynamic data structures, aliasing, arrays and so forth. Various Static Analysis techniques (like Pointer Analysis and Shape Analysis) seem relevant to this question.
– Identifying (either manually or automatically) what variables (outputs or others) and in which locations should be equal in order to increase the confidence in the correctness of the changes.
– Identifying the connection between functional properties and equivalence properties: assuming that a certain early version of the program satisfies some invariants or other high level property, which variables should be followed through the evolvement of the program to guarantee that this property still holds ? under which conditions this is decidable?

[2] The same observation is well known in the hardware domain, where equivalence checking of circuits is considered computationally easier in practice than model-checking.

– Finding the ideal gap between two versions of the same program, for making Regression Verification most effective. There is an apparent tradeoff between larger gaps, which reduce the overhead of proving equivalence, and the effectiveness of comparing the two versions of the code.

References

1. W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001. SLAM.
3. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE), To appear*, 2003. magic.
4. D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of SPIN2000*. Springer-Verlag, 2000.
5. M. Fowler. http://www.refactoring.com.
6. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002. BLAST.
8. T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, January 2003.
9. G. Holzmann and M. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, 28(4):364–377, April 2002.
10. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
11. M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI 2002*, 2002.
12. A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and computation*, 178(1):279–293, Oct. 2002.
13. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *4th Intl. Conf. TACAS'98*, volume 1384 of *Lect. Notes in Comp. Sci.*, pages 151–166. Springer-Verlag, 1998.
14. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages.* ACM Press, 2005.