Refuting Untrue Verification Conditions

Diego S. Zaccai
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
Email: zaccai@cse.ohio-state.edu

***Abstract* — The RESOLVE tools should try to refute false verification conditions (VCs), perhaps using the "small scope hypothesis" for guidance.**

I. Introduction

All VCs fall into either of two disjoint sets; true VCs and false VCs. Much of the focus of RESOLVE tool development has been on expanding the provable subset of true VCs. However, little work has been done on the opposite side. As a result of this, our tools do not refute incorrect VCs, but time-out instead. In this paper, I propose to explore this issue and explain the general guidelines of a tool I will be implementing to expand the refutable subset of false VCs with the hope of providing opportunities for richer interaction between the tool and its users.

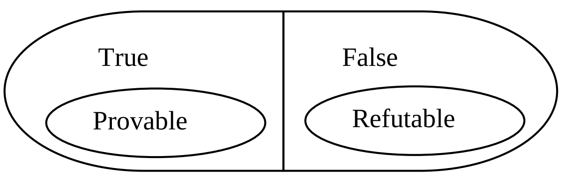

Figure 1. The division of VCs into its possible values

II. The Structure of a VC

The OSU RESOLVE tools now generate VCs with a very specific form. Each is a logical implication whose premise is a conjunction of literals or the negation of literals and whose conclusion is a literal or the negation of one. These literals have variables in them that are implicitly universally quantified. This universal quantification is part of the reason why most of the provers we use do not refute incorrect VCs unless they can be reduced to true → false. This is overreaching, since in order to refute a VC we are not required to prove that it reduces to false, but instead we only need to find a counterexample.

The problem, of course, is that in general finding this counterexample can be an extremely complicated task. However, we will leverage the small scope hypothesis [1] that argues that a high proportion of the bugs in code can be found by testing the program for all "small" inputs, i.e., in a "small" scope. In this case however, we are not interested in testing the code itself, but rather the VCs that are required to be satisfied in order to prove the code correct. We will not be able to disprove all of the false VCs for the same reasons we cannot prove all the correct ones. However, we can hope that because of the small scope hypothesis and previous work by this group indicating that most VCs fall into the "obvious" realm [3] we will be able to disprove a large fraction of false VCs with a relatively simple tool.

A sample incorrect VC from erroneous code

Given:

$$0 \le n_0$$
$$0 \le m_0$$
$$0 \le 0$$
$$0 \le 0 + 1$$
$$0 \le m_0 - 1$$
$$0 \le n_0 \times (m_0 - 1)$$
$$0 \le n_0 \times (m_0 - 1) + n_0$$
$$0 + 1 < m_0$$
$$m_0 \ne 0$$

Prove:

$$m_0 - 1 = m_0$$

Figure 2. Sample VC from an incorrect implementation of UnboundedNaturalMultiply. The implementation is available online as: RecursiveIncBug implementation of Multiply enhancement for UnboundedNaturalFacility in the RESOLVE Catalog [2].

III. Proposed Methodology

All variables in a VC have a mathematical model type that describes which values those variables can take. For example, in the sample VC above, both variables are mathematical integers. Thus the possible values on which this VC could be tested is a two dimensional subspace. As one applies the small scope hypothesis, choosing all possible combinations on a small predefined range of, say, 0 to 2 should be enough to provide a counterexample. In other words, there are 9 potential counterexamples in this small scope. Indeed, the combination $n_0=1$ and $m_0=2$ does provide a counterexample to the given VC.

This sample VC is especially simple in the sense that its variables stand for numbers, where the notion of "small" is easily interpreted. The RESOLVE language does provide opportunities for much more complex data types with more complex mathematical models. However, considering only the built-in mathematical types in the OSU version of RESOLVE, each such model is a combination of simpler mathematical entities: integers, strings, finite sets, and tuples. We can view each one of these as a new subspace that we need to sample. For example, a Text variable would provide a two dimensional candidate subspace where one of the dimensions would be the length of the string that models a Text object, and the other would be provided by the value of the elements in each position. As a result, the candidate counterexamples could be chosen from a range of lengths and a subset of the possible characters in each location. Similar methods to construct candidate counterexamples can be used with other mathematical types.

A third data abstraction feature has to be considered when generating counterexamples. Generics are a vital part of RESOLVE, but their state spaces are not completely determined at component design time (hence, at verification time). For example, the component QueueTemplate provides queue functionality regardless of what type of item is stored in the queue. However, such generics should not create problems much harder than the complex data types described before, as the properties of the item stored in them should not affect the validity of VCs. This is actually an advantage: if a counterexample is to be found, it should suffice to consider candidates for which the entries in generic collections are, say, integers.

Once a set of possible variable assignments has been created, the process of refuting a VC becomes a simple succession of assignments and evaluations. The tool would simply assign all possible combinations of the values of each variable, and report the VC as false if one such combination evaluates to false.

IV. Improving the Search of States

Many VCs have very specific boundaries to which the given variables can be constrained. For example, the sample VC provides the information that m_0 has to be greater than 1 (eighth given), and a heuristic could be used to leverage these cases that would provide more candidate counterexamples with little additional expense. However, the seventh given provides a clear example as to why the tool should not rely on this type of heuristic as its only method of producing counterexamples. The seventh given is a relatively complex relation between two variables. Such relationships might be extremely hard to exploit in the programmatic generation of candidate counterexamples. As a result, the heuristic generation of candidate counterexamples would be a tool that complements the proposed one but would not replace it.

V. Improving Efficiency

The third and fourth givens in the sample VC could be evaluated immediately to see if they are true. If so, they can be removed from the VC entirely, since a true literal in a conjunction will not affect the evaluation of the conjunction. Similarly, if either evaluates to false, the VC becomes trivially true and there is no point in continuing the search for counterexamples.

Conclusion

There is a need to provide more conclusive answers to users when a program is not correct, and I believe that the proposed tool will be a first step in that direction. Of course, when using this methodology, failure to find a counterexample does not necessarily imply that the VC is true. This in turn could leave the user of the tool in the same position as before, where the tools time-out without a definite answer.

Acknowledgements

The author is grateful for input from Bruce Weide, Bill Ogden, Paolo Bucci, Bruce Adcock, Jason Kirschenbaum and Derek Bronish. This material is based upon work supported by the National Science Foundation under Grants No. DMS-0701260 and CCF-0811737. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. Jackson and C. A. Damon. Elements of style: analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7), July 1996.

[2] The Resolve VC Generator [Online]. Available at: http://resolve.cse.ohio-state.edu:8080/ResolveVCWeb/

[3] Kirschenbaum, J., Adcock, B., Bronish, D., Smith, H., Harton, H., Sitaraman, M., Weide, B.W., Verifying component-based software: deep mathematics or simple bookkeeping? *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, Springer-Verlag, LNCS 5791, September 2009, pp. 31-40.