Hypotheses Generation and Minimization in VCs

Heather K. Harton

Abstract—Automated Verification Condition (VC) Generation has been shown to be practical with the implementation of the Clemson University RESOLVE VC Generator. The current VC generator has been used to create VCs for numerous enhancement level realizations. As more complex software is processed by the VC generator, the simplification of the clauses becomes more important. The number of hypotheses that the prover will need to consider should be minimized in order to keep the proof process efficient. The VC generation process should be able simplify VCs by using the knowledge based on the information inherent to the goal-oriented generation process.

1 Introduction

THIS talk will discuss the current implementation of the RESOLVE VC Generator as it applies to the simplification process for verification conditions. This will include a brief overview of the implementation details of the generation process and the simplification steps. In order for the simplification process to be useful, it needs to make use of the information available by the VC Generation process that would not be readily available to a VC prover.

A goal-oriented VC generation process has been implemented [1]. This process modifies the goal using proof rules for each line of code until only a mathematical verification condition remains. The resulting VC is then processed by a prover which attempts to show the VC is true.

2 Hypotheses Generation and Simplification

At present, the VC Generator provides limited simplification of the generated assertions. In an effort to ensure that all information that may be needed to show the goal is true is provided in a VC, any hypotheses that could ever be useful have been included. For example, the constraints on conceptual variables, types, or parameters to a module are included as assumptions in any resulting VCs. Often this information may not be used in proving a VC and could even cause the prover to perform many unnecessary steps.

The current simplification process used by the VC generator is quite simple. There are some simplifications made by the proof rules themselves. For example, when a variable declaration rule is applied, if the variable is not used in the VC, then the variable declaration rule does not modify the VC. Simplification steps similar to this one that are built into proof rules seem ideal.

Currently, some simplification rules have been implemented that are not based on a proof rule but are instead attempting to do other types of basic simplifications. For example, if an assumption is identical to the goal, then assertion can be simplified to true. Additionally if the goal is obviously true (X=X), then the assertion may be simplified to true. However, these types of simplification might be more efficiently handled by a prover instead of the VC generator. In fact, it may be that the RESOLVE prover can act as a simplifier. The impact of calling the prover after each proof rule application to perform simplification steps needs to be studied.

If we consider the example in Figure 1: VC from Stack Flip ExampleFigure 1, there are several different causes for the various hypotheses. min_int <= 0 < max_int is a requires clause for the Integer Template. min_int <= Max_Depth < max_int is a constraint that must hold true for all integers. Max_Depth > 0 is a requires clause for the Stack Template and |S| <= Max_Depth is a constraint that must hold true for all Stacks. However, in this particular example none of these are needed to prove the goal true. Can any of these hypotheses be simplified out?

```
Goal:
S = (Rev(empty_string) o S)

Given:
(((min_int <= 0)and
(0 < max_int) and
((min_int <= Max_Depth)and
(Max_Depth < max_int)
(Max_Depth > 0)) and
(|S| <= Max_Depth))
```

Figure 1: VC from Stack Flip Example

Some simplification steps that are currently being considered arrive with the addition of a new type of Verification statement, *Presume*. Currently, when generating VCs, we use an *Assume* statement. There are a few times when the assume rule is simplified in the current VC Generator. For example, if we can assume X = Y, we will replace all instances of X in our VC with Y. However, with the implementation of *Presume* this intelligence can increase. If a *Presume* statement exists, this will be considered an intermediate predicate and must be included as a hypothesis. However, anywhere that an *Assume* is used, an addi-

• *Heather K. Harton is a student in the Department of Computing, Clemson University, Clemson, SC. E-mail: hkeown@g.clemson.edu.*

tional hypothesis may not have to be added when processing the *Assume* statement. So, for each conjunct in the assumption, if the goal contains a variable in that portion of the assume statement, then the portion needs to be included as a hypothesis. If the goal does not contain any of the variables in a conjunct in an assume statement, that conjunct can be discarded.

Upon inspection, one can see that the use of *Presume* in addition to *Assume* should allow further simplification of VCs. It is unsure if there are other modifications to the VC Generation process that can decrease the number of hypotheses in the VCs.

This talk will also include some discussion on the internal manipulation of the VCs during the generation process.

3 Example

As an example, a Stack Flip enhancement realization can be considered. This example generates 6 VCs. Although these VCs can be proved automatically currently, how many of the hypotheses are unnecessary? As seen in Figure 2, there are many unused hypotheses. Unfortunately, it does not seem that in this example any of the hypotheses are eliminated with the use of *Assume* and *Presume*.

VC	*Number of Hypotheses*	*Number used in Proof*	*Using Presume and Assume*
1	4	0	4
2	7	1	7
3	7	2	7
4	7	2	7
5	7	1	7
6	6	1	6

Figure 2: Use of Hypotheses in a VC

4 Conclusion

It is important that verification conditions are as simple as possible when provided to the prover. Any simplifications that can more easily be made by the VC generator instead of a prover should be handled during the generation process. In addition, to provide an efficient verification process any steps that the prover has been optimized to perform should not be attempted by the VC generator.

5 Acknowledgements

This research is funded in part by NSF grants CCF0811748 and DMS0701187.

References

[1] H. Harton, J. Krone, and M. Sitaraman, "Formal Program Verification," *Wiley Encyclopedia of Electrical and Electronics Engineering, Software Engineering Volume*, 2007.