

The beginning of the end of debugging as we know it

Bruce M. Adcock

Department of Computer Science & Engineering

The Ohio State University

Columbus OH 43210

Email: adcockb@cse.ohio-state.edu

Abstract—When speaking of the verifying compiler effort, we often talk of the “humans in the loop”. At least one of them is a programmer, who almost invariably writes code with defects in it. After attempting to compile the program and failing, the programmer is then supposed to attempt to correct the defect, and repeat until the program finally compiles. At some point, we must examine how such problems can be presented to the user to aid them in their task. A separate, but equally important issue, is to explore ways to optimize the verified code, so that a choice does not need to be made between both verifiability and performance. We will discuss why it is important to start working on these problems now.

I. INTRODUCTION

In the current state of programming, code is handed to a compiler and it compiles or it does not, depending on how well the code meets the syntactic structure required by the language. We are working to change this, keeping the syntactic checking, but then having the compiler declare the code correct (and then actually compile it), or incorrect (and not compile). Compare this with our overview of how a verifying compiler’s system might work in Figure 1. It is clearly a change from how things are done presently with standard practices. New to the programmer in this system, there are now formal specifications detailing preconditions and postconditions of operations, progress metrics and loop invariants within the operations, and mathematical definitions and results that might be of use in the specifications (and potentially in the programmer’s understanding of how the code should work). And yet, ideally, the lower two-thirds of Figure 1 would remain opaque to the programmer even when something goes wrong. Correct code would bring about successful compilation, while incorrect code must provide feedback that does not suggest the particular inner workings we are developing.

To verify a particular piece of code, the formal specification properly detailing its preconditions and post-conditions *must* be given. Using these, proposed conjectures called verification conditions (VCs) are generated, and the proof of the validity of these VCs demonstrates that a code realization meets its specification. The VCs have many components to them. The first two are the most relevant: an obligation required by the particular specification, and the collection of facts known up to that point from which to establish that the obligation is met. A VC can also pass relevant mathematical definitions and results to assist in the proof. For code to be valid, the facts, aided by the mathematical definitions and results, must imply

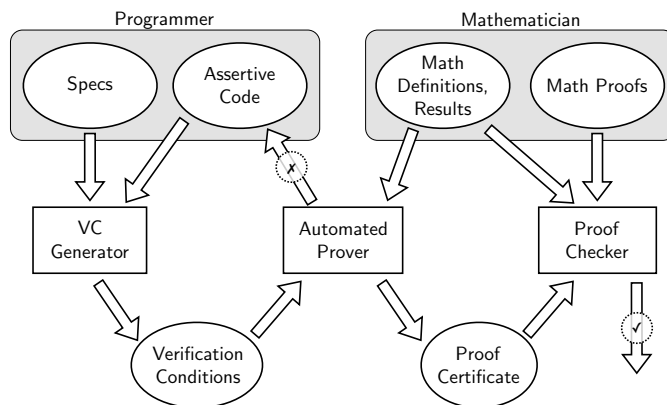


Fig. 1. System Overview

the obligation. Notably, the goal in any verifying compiler is to have the actual proof be fully automated.

Generating and proving VCs is different from taking code and compiling it. A compiler attempts to generate object code to do what the high-level language code actually says, within the constraints of any static type information and the semantics of the programming language in general. A VC generator, and subsequently any VC, has greater information as to the intent of the programmer. Ergo, there is potential to use this extra information both to aid the programmer in finding defects, and to (for example) improve the performance of the resulting object code.

It is hopefully clear from this description that having a verifying compiler changes the development of software. The automatic processing of a VC has three possible outcomes: it is fully verified, it is found to be defective, or more mathematical theory is needed to decide. A programmer has little need for standard debugging techniques, as there is no need to debug correct code, and no ability to compile incorrect code. However, a programmer still requires help in *understanding* the results of verification attempts. As it stands, we have taken the sisyphian task of debugging problematic code in to a sisyphian task of debugging VCs given to automated provers.¹

Besides having programmers change code that contains bugs, we are also potentially asking them to change code that is correct, but cannot be automatically proven. Which is easier,

¹In reality, we must not present this to a programmer in the form of a VC, but in some more suitable context, which is part of the challenge.

adding new theories, or simplifying the code? This means we are likely asking the programmer to remove optimizations from the code, merely so that it will compile.

We have therefore taken the task of debugging, and made it more cryptic to programmers. We have also taken their proudly developed optimizations and stripped them out. In short, our verification has added a barrier to anyone who would consider writing a verified program. It therefore *cannot* be reasonably argued that these are problems we can forever avoid.

II. EXAMPLES OF FAILED VCS

To see the difficulty presented to the programmer, simply look at the VCs shown in Tables I and II. One is valid, while the other is not (To avoid spoiling the solution, do not look at the following footnote until trying to determine which is which.)² They are shown out of context to demonstrate the difficulty a programmer would have in understanding the problem with debugging a VC outright. In reality, we have much more information which can be used to present the programmer with, such as state the VC was generated for, reason the VC was generated, and, of course, the code itself.

TABLE I
A VC, UNPROVEN

$$\begin{array}{l}
m_0 = m_2 + k_2 \\
\wedge n_2 + m_2 = n_0 + m_0 \\
\wedge n_2 \leq \max \\
\wedge n_0 \leq \max \\
\wedge m_2 \leq \max \\
\wedge m_0 \leq \max \\
\wedge k_2 \leq \max \\
\wedge n_2 + 1 \leq \max \\
\wedge n_0 + m_0 \leq \max \\
\wedge k_2 + 1 \leq \max \\
\wedge 0 \leq n_2 \\
\wedge 0 \leq n_0 \\
\wedge 0 \leq \max \\
\wedge 0 \leq m_2 \\
\wedge 0 \leq m_0 \\
\wedge 0 \leq k_2 \\
\wedge 0 \leq n_2 + 1 \\
\wedge 0 \leq k_2 + 1 \\
\wedge m_2 \neq 0 \\
\hline
\Rightarrow n_0 + m_0 = n_2 + m_2 + 1
\end{array}$$

III. THE RESOLVE ADVANTAGE

Before explaining why this is a problem that needs to be worked on *now*, it is worth noting the advantage we have compared to verification with other languages. Most other VC generators take formally annotated code from mainstream languages, such as C or Java (using Caduceus [1], Krakatoa [2], ESC/Java [3], etc.), and produce VCs from them. As such, although the process of generation remains in many ways similar, the VCs themselves for C or Java are potentially much more difficult due to the need to take into account side effects

²Table I comes from `NaturalFacility_Add_IterativeDecBug` (VC 7), which does not have a necessary decrement statement (and the VC is therefore incorrect). Table II comes from `SetTemplate_Intersect_Iterative` (VC 12), and originally could not be solved due to a lack of theory at the time. It is, however, correct.

TABLE II
ANOTHER VC, UNPROVEN

$$\begin{array}{l}
tmp_2 = tmp_2 \cap t_0 \\
\wedge tmp_2 \cap s_2 = \emptyset \\
\wedge t_0 \cap s_0 = t_0 \cap (tmp_2 \cup s_2) \\
\wedge is_initial(x_6) \\
\wedge is_initial(x_3) \\
\wedge s_2 \neq \emptyset \\
\wedge x_4 \in t_0 \\
\wedge x_4 \in s_2 \\
\hline
\Rightarrow t_0 \cap (tmp_2 \cup \{x_4\}) = tmp_2 \cup \{x_4\}
\end{array}$$

the languages themselves allow. We therefore are “gifted” with much cleaner debugging of VCs. Our work will, hopefully, translate to be usable to these other attempts at verification, but it will indeed be a larger effort for them.

The advantage can also be disadvantageous, too. For our purposes, we write simpler code to aid the verification and subsequent debugging. Other groups are working to verify (or at least prove properties of) preexisting code, so that the original optimizations remain. But is this a true disadvantage to us? Programmers have handed over many tasks to compilers that were once their own domain, beginning with giving up control of writing assembly. There will always be specialized domains with unique optimizations, but many optimizations are quite common. It is therefore in our interests to demonstrate the usefulness of all the additional information we generate for the needs of verification, by providing optimizations to programmers who write code that is easier to understand.

IV. WHY NOW?

The question then arises: why work on this problem now? There clearly are many other pressing pieces to the puzzle in Figure 1. The first point is somewhat obvious: this is an essential part of the puzzle. For anyone to actually use the system we have presented for a verifying compiler, we have a potential high cost in being able to handle the mathematical theories already. Issues from greater difficulty in finding why code will not compile, and being forced to remove optimizations, only increase that cost.³ Due to its importance, it is also therefore necessary to make sure we have answers to these issues when the other pieces are in place.

The other significant part is of component design (and also the mathematical models we choose for our ADTs). Consider, for example, the use of pointers. There has been some work towards this on the RESOLVE side [4], [5], which may very well be the way we should use when dealing with pointers. We must note, however, that while the VCs themselves can be automatically generated, as of yet the VCs have not all been automatically *proven*. A programmer who uses pointers must therefore handle the fallout of the unproven VCs.

We can now compare this with the verification done elsewhere, such as with ownership types and/or separation logic. Using Verifast [6] to verify a standard C implementation of a

³Clearly, there is a large upside as well from having fully verified code, but a high entry cost will not help adoption of any verifying compiler.

stack, 15 mathematical definitions and lemmas are required, along with 54 manual assertions in the code to be proved (this includes two small test operations that subsequently use the stack type). Does this mean their method is worse? The tool happens to have much less rich built-in mathematical theory than ours (somewhat explaining the necessary lemmas and definitions), and it is able to verify the code. If we remove some assertions or lemmas, would a programmer just as easily be able to debug why a VC cannot be proven? Or we make a bug while coding? We do not know the answers to these questions, meaning we also do not know which is better from the programmer's viewpoint.

It is therefore integral that we choose a proper modeling of pointers that takes into account how a programmer can view problems. Pointers are not unique to this problem, but the uncharted-ness of our handling them demonstrates the questions we need to face early on. Any hope of mainstream use of it demands these issues to be solved.

REFERENCES

- [1] J.-C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus platform for deductive program verification," in *19th International Conference on Computer Aided Verification*, ser. LNCS, vol. 4590/2007. Berlin, Germany: Springer-Verlag, July 2007, pp. 173–177. [Online]. Available: <http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf>
- [2] C. Marché, C. Paulin-Mohring, and X. Urbain, "The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML," *Journal of Logic and Algebraic Programming*, vol. 58, no. 1–2, pp. 89–106, 2004. [Online]. Available: citeseer.ist.psu.edu/marche03krakatoa.html
- [3] K. R. M. Leino, G. Nelson, and J. B. Saxe, "ESC/Java user's manual," Compaq Systems Research Center, Technical Note, Oct. 2000.
- [4] G. Kulczycki, M. Sitaraman, B. W. Weide, and A. Rountev, "A specification-based approach to reasoning about pointers," in *SAVCBS '05: Proceedings of the 2005 conference on Specification and verification of component-based systems*. New York, NY, USA: ACM, 2005, p. 7.
- [5] G. Kulczycki, H. Keown, M. Sitaraman, and B. W. Weide, "Abstracting pointers for a verifying compiler," in *SEW '07: Proceedings of the 31st IEEE Software Engineering Workshop*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 204–213.
- [6] B. Jacobs and F. Piessens, "The verifast program verifier," Department of Computer Science, Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW-520, August 2008.