Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping?

Jason Kirschenbaum[1], Bruce Adcock[1], Derek Bronish[1], Hampton Smith[2], Heather Harton[2], Murali Sitaraman[2], and Bruce W. Weide[1]

[1] The Ohio State University, Columbus OH 43210, USA
{kirschen,adcockb,bronish,weide}@cse.ohio-state.edu
http://www.cse.ohio-state.edu/rsrg
[2] Clemson University, Clemson SC , USA
{hamptos,hkeown,msitara}@clemson.edu
http://www.cs.clemson.edu/~resolve/

Abstract. Anecdotal experience constructing proofs of correctness of code built from reusable software components reveals that they tend to be relatively trivial bookkeeping exercises: they rarely require a substantive mathematical deduction. A careful empirical analysis of hundreds of verification conditions (VCs) for a library of component-client code shows the level of sophistication each proof requires, and suggests how to use the results to characterize a notion of mathematical "obviousness."

1 Introduction

Perhaps the most powerful tool of modern programming languages—and a feature that makes new languages enticing to prospective developers—is a rich library of well-documented software components. Not only is the code in these catalogs (or "APIs" or "libraries") reusable, but code that makes use of them tends to be reusable as well, because it builds on a cleanly-modularized foundation: programmers use components to build more specialized and sophisticated components. Thus, when considering the task of verifying the correctness of software, one logical place to begin is with verification of code that is written from the perspective of clients using these "catalog components."

Sitaraman *et al.* [1] have explained how to reason about and formally verify the execution-time behavior of reusable software components. The use of programming-by-contract with abstract mathematical models, along with corresponding sound proof systems, allows software developers to reason about the correctness of newly-created code that uses components. One limitation of this process, however, is its reliance on people to check the verification conditions (VCs) that result from the syntax-driven proof system. Given a human being's propensity for error, one natural solution is to automate the reasoning process; VCs can be discharged automatically in a "push-button" manner.

The verifying compiler grand challenge [2,3] is to build a compiler that both generates executable code and performs the reasoning process described above.

S.H. Edwards and G. Kulczycki (Eds.): ICSR 2009, LNCS 5791, pp. 31–40, 2009.

This tool would give developers more confidence in the correctness of implemented software components, providing the ability to reuse software using only the contracts (or "interfaces") of the components.

One stumbling block in the path of meeting this challenge, of course, is the difficulty of fully automating mathematical theorem proving. In the general case, the problem of automatically proving or disproving a VC is undecidable [4]. We proceed on the notion that, because software engineers write code that is motivated by practical mathematical insight, real software will not normally result in VCs that are inherently undecidable. Indeed, our thesis is that most VCs generated from most programs are "obvious." A precise definition of **obvious** is a long term research question that is important for software verification; we would like to be able to characterize what makes a program automatically provable. Automated provers will always require sufficient justification, as part of the program itself, to make correctness manifest. In this paper we examine VCs generated from code that builds on reusable software components, and classify them based on the reasoning methods needed for their proofs—a first step towards an eventual rigorous characterization of obviousness. This work is, to our knowledge, the first that classifies VCs in accordance with these objectives and makes a case for the simplicity—as opposed to the oft-assumed difficulty—of automatically verifying "everyday" programs.

Based on our earlier work [5], we generate VCs from a battery of programs that make use of a basic component library. Of course, the results of this analysis are arguably dependent on the particular proof system and programming language used. To a lesser extent, the analysis may also be dependent on the particular components used and implementations verified, but we argue that the conclusions generalize along these dimensions—a testable thesis offered as a challenge to the community.

Section 2 describes RESOLVE, the specification and programming language used in this work. In Section 3 we elaborate our taxonomy of VC classification. Section 4 presents our analysis of one example for illustration, and more general results gathered across a large component catalog. Section 4 discusses related work and Section 6 contains concluding remarks and directions for future work.

2 RESOLVE: VC Generation and an Example

The language used in this work is RESOLVE [6], an imperative language designed with fostering reusable, component-based software as the foremost objective. RESOLVE provides a catalog of abstract data type interfaces (e.g., Queue, Set, Map) on which client code can rely. The client code is written in a programming-by-contract discipline with model-based mathematical specifications, and thus adheres to a strict encapsulation boundary between abstract interface contracts and concrete implementation details: only the contracts, which formally define a component's behavior, are necessary to write and reason about client code.

More specifically, RESOLVE enforces so-called "two-level thinking": reasoning about the correctness of a component implementation requires *only* the specification of that component along with the specifications of any components used

in the implementation. This clear delineation of roles and emphasis on mathematical formalism facilitates *modular* verification of RESOLVE programs via a two-stage process of (1) VC generation and (2) VC proof. The purpose of the proof system is to transform, syntactically, a RESOLVE program into a set of logical formulas whose validity is equivalent to the correctness of the program.

One other major difference between RESOLVE and other languages is the use of **swap**, denoted by `:=:`, as the primary data movement operator. Swapping serves a role similar to assignment in other languages, and has been argued to be a better choice for data movement, particularly for the purposes of reasoning about code [7]. The statement `x :=: y` exchanges the values of `x` and `y`.

A final key attribute of RESOLVE is that the language includes syntactic slots for mathematical code annotations, which allow for a sound and relatively complete proof system [8,9]. These annotations include loop invariants, termination metrics for loops and recursive operations, and indeed arbitrary mathematical assertions, which must themselves be proved, that may "assist" the automated reasoning process in discovering mathematical insights underlying the code. Our hope is to minimize the number of extra programmer-supplied assertions necessary to verify code, and in fact none of the code verified in the work for this paper required any extra assertions beyond standard loop invariants and progress metrics (which are always required).

We generate VCs automatically in accordance with the RESOLVE program proof system defined formally in [9] and informally in [1]. As outlined in [1], this process can best be characterized as filling out a symbolic tracing table. We generate one VC for each line in the program where the next statement requires a pre-condition to be satisfied or a loop invariant/progress metric property to be upheld. We also generate a VC which states that, at end of the code being verified, the implementation has met its obligation as stated in its contract. Note the modularity of this approach: requirements on legally calling operations are proven to be met on the client's side, whereas the correct behavior of program units assuming satisfied pre-conditions is verified once-and-for-all in isolation.

The VC generator works in two phases. The by-rote VC generation of phase one introduces a large number of mathematical variables: one for each program variable in each program state (roughly speaking, we consider the execution of each line of code in the program to define a new "program state" or "program point"). For instance, x_i stands for the value of program variable `x` in state i. In a program with n variables and m statements, the VCs contain a total of nm distinct mathematical variables at the end of phase one, and of course each variable may actually occur multiple times.

The simplification phase of the VC generator then applies a few theory-independent logical restructuring rules to each VC. The most obvious and useful simplification is basic substitution. A majority of the nm-many variables are removed by this step. For instance, the hypotheses in the original VCs often contain clauses which indicate that the value of some variable was preserved from one program state to the next, *e.g.,* $x_{10} = x_9$. Upon encountering this situation we replace x_{10} by x_9 throughout the VC and remove this clause altogether.

Any effective back-end automated prover would certainly be able to apply such substitutions to achieve the same effect, but the VC generator does this before handing the VCs to an automated prover so the human-readable output of the VC generator is more concise.

The second phase also makes various structural changes to VCs in an effort to render them more tractable for processing by an automated theorem prover. An example is the explicit introduction of case analysis. In our experience, useful case analysis can rarely be performed spontaneously by a back-end automated prover. Requiring human advice about appropriate cases seems to be the norm, and certainly is necessary in general. However, the kinds of case splits required for discharging VCs, rather than arbitrary mathematical sentences, are relatively simple and can usually be deduced from the structure of the code. The VC generator therefore makes case splits explicit. The simplification phase divides each VC into appropriate cases based on the control flow of the code, thus obviating the need for any special insight or assistance on the prover's behalf.

The net result of the two-phase VC generation process is that, while there are normally many VCs even for a relatively short piece of code, based on anecdotal experience, each VC tends to be relatively small and "prover-friendly."

To aid our explanation of the VC classification in Section 3, we present an example software component contract along with a component extension, and code that purports to implement that extension. The software component is a stack and the extension defines an additional operation that reverses a stack. The contract is given in Fig. 1(b), and describes the behavior of the stack in terms of a mathematical model, namely a mathematical string. Each operation contract, for example `Push`, describes the effects of the operation on the model via `requires` and `ensures` clauses (the pre- and post-conditions of the operation, respectively). The symbol `#` in the `ensures` clause denotes the incoming value of a parameter.

The contract for the `Reverse` operation is depicted in Fig. 1(c). This contract uses a *mathematical* function `reverse` in the `ensures` clause; its meaning should be unsurprising.

Finally, realization code that is purported to implement the `Reverse` operation is given in Fig. 1(a). Note the use of the swap operator `:=:` in this code. Also, this code illustrates the requisite code annotation in the form of a loop invariant (the `maintains` clause), and a loop progress metric (the `decreases` clause).

3 Classification of VC Proof Techniques

We use the implementation of a stack reverse extension shown in Fig. 1(a) to illustrate the VC classification process. Figures 3 through 6 show example VCs generated by the method discussed in Section 2. Note that a VC always takes the form of a logical implication whose antecedent is a conjunction of zero or more clauses. We refer to clauses of the antecedent as "hypotheses" and the consequent as the "goal". We represent our VCs by showing all of the antecedent's clauses conjoined above the solid line, with the consequent written below the line.

```
realization Iterative
  implements Reverse
    for StackTemplate

    procedure Reverse
         (updates s: Stack)
        variable tmp: Stack
        loop
            maintains
             reverse (s) * tmp
              = reverse (#s) * #tmp
            decreases |s|
        while not IsEmpty (s) do
            variable x: Item
            Pop (s, x)
            Push (tmp, x)
        end loop
        s :=: tmp
    end Reverse

end Iterative
```

(a) Stack Reverse Implementation

```
contract
  StackTemplate (type Item)

   math subtype
      STACK_MODEL
        is string of Item

   type Stack
        is modeled
         by STACK_MODEL
       exemplar s
       initialization
         ensures
          s = empty_string

   procedure Push
      (updates s: Stack,
       clears x: Item)
        ensures
         s = <#x> * #s

   procedure Pop
      (updates s: Stack,
       replaces x: Item)
        requires
         s /= empty_string
        ensures
         #s = <x> * s

   function IsEmpty
      (restores s: Stack)
              : control
        ensures
           IsEmpty =
       (s = empty_string)
end StackTemplate
```

(b) Stack Contract

```
contract Reverse
  enhances StackTemplate

   procedure
     Reverse
      (updates s: Stack)
        ensures
         s = reverse (#s)
end Reverse
```

(c) Stack Reverse Contract

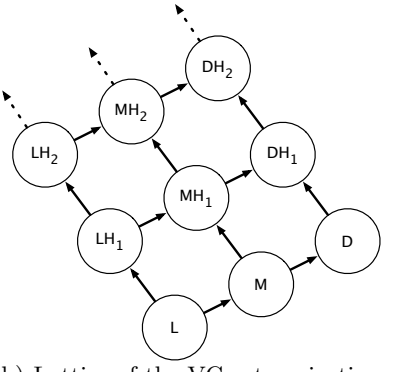
Fig. 1. Stack Example

Our classification of the difficulty of the proof requirements of VCs is presented in Fig. 2(a). Note that the categories (i.e., sets of VCs) in Fig. 2(a) have a natural structure. The L category is a subset of every other category. Each LH_i is a subset of LH_{i+1}, each MH_i is also a subset of MH_{i+1}, and each DH_i is a subset of each DH_{i+1}. Finally, each LH_i is a subset of MH_i; also, each MH_i is a subset of DH_i. This structure forms a lattice shown in Fig. 2(b).

Our methodology for analyzing the RESOLVE component catalog is to instrument an in-house VC prover, SplitDecision, to analyze each proven VC to

Label	What is needed in the proof
L	Rules of mathematical logic
H_n	At most n hypotheses from the VC needed ($n > 0$)
M	Knowledge of mathematical theories used in the specifications
D	Knowledge of programmer-supplied definitions based on mathematical theories above

(a) VC Classification



(b) Lattice of the VC categorization

Fig. 2. Table of VC Categorization and Diagram of Category Relationships

determine the "lowest" category the VC is in. We do this by checking if the goal is a tautology, checking to see how many hypotheses are needed in the proof, etc. The automatic nature of this can only guarantee we find an upper bound for any VC. For example, a particular VC categorized as MH_1 might in fact be M or LH_1, but certainly not MH_2 or DH_1. For VCs that are true but are not proven automatically (12.5% of VCs in the study), we manually examine those VCs to determine the category.

Figure 3 shows a VC that falls into the L category. Its proof relies only on an axiom of first order logic with equality: $\forall x.x = x$. We note that VCs of type L are the "most obvious" kind: no knowledge of the underlying mathematical theory nor even any of the hypotheses of the VC are needed to establish the goal. Any rigorous, technical definition of "obvious" should include all VCs of this type.

$$\Rightarrow \text{reverse}(s_0) * \Lambda = \text{reverse}(s_0) * \Lambda$$

Fig. 3. Example from category L

A VC that falls into an LH_n category is given in Fig. 4. More specifically, we label this VC as LH_1 because it is provable by noting simply that the goal is one of the hypotheses. LH_1 VCs clearly should be considered "obvious," whereas other LH_n VCs may or may not be obvious, depending on the amount of logical deduction necessary to actually establish the implication.

$$\begin{array}{rr} & s_2 \neq \Lambda \\ \wedge & \text{reverse}(s_2) * tmp_2 = \text{reverse}(s_0) * \Lambda \\ \wedge & |s_2| \geq 0 \\ \wedge & \text{is\_initial}(x_3) \\ \hline \Rightarrow & s_2 \neq \Lambda \end{array}$$

Fig. 4. Example from category LH_1

A VC in category M is given in Fig. 5. This VC depends only on the validity of the goal in the underlying mathematical theory. In this case, the goal is a direct consequence of

$$\frac{|s_2| \geq 0}{\Rightarrow s_0 \neq \Lambda \rightarrow |s_0| > 0}$$

Fig. 5. Example from category M

a useful theorem in the mathematical theory of strings; it can be derived using the proof rules of first order logic, plus the theory of integers to account for 0 and >. If the theorem is known and available, then this VC is surely obvious. If not, it might seem obvious to humans on semantic grounds, but a symbolic proof is several steps long.

Finally, in Fig. 6 we include a VC from an implementation of the well-known Egyptian multiplication algorithm to illustrate the D category. Its proof requires three hypotheses. Without knowledge of the programmer-supplied definition of IS_ODD—either an expansion of the definition or the algebraic properties thereof—the VC cannot be proven.

$$\begin{array}{rr} & \text{...[9 irrelevant hypotheses]} \\ \wedge & n_0 * m_0 = n_3 * m_3 + p_3 \\ \wedge\ m_3 = m_8 + m_8 \vee m_3 = m_8 + m_8 + 1 & \\ \wedge & \text{IS\_ODD}(m_3) \\ \hline \Rightarrow & n_0 * m_0 = (n_3 + n_3) * m_8 + p_3 + n_3 \end{array}$$

Fig. 6. Example from category DH_3

The distinction between the D and M categories is based on the current state of the components. As more theories are developed it is possible that programmer-supplied definitions are moved into the mathematical theories because of their general utility. However, it is also clear that there will always be examples of mathematical definitions that are programmer-supplied, for example the format of a particular type of input file.

4 Analysis of Verification Conditions

The VCs analyzed for the paper are generated from roughly fifty components that comprise a total of roughly two thousand lines of source code. The code ranges from simple implementations of arithmetic using unbounded integers, to sorting arbitrary items with arbitrary orderings, and includes classical code such as binary search using bounded integers.

Figure 7 shows classification data for all VCs generated from a sample catalog of RESOLVE component client code that relies on existing, formally-specified components to implement extensions,

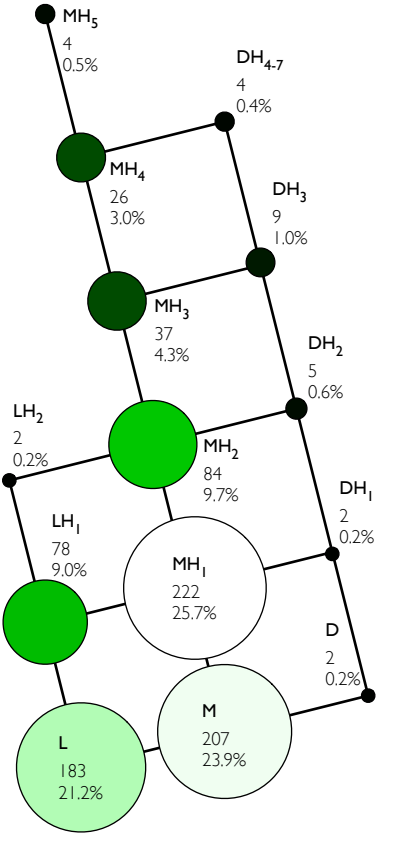


Fig. 7. VCs as categorized

which add additional functionality (*e.g.,* the aforementioned `Stack Reverse`). Here, the area of each bubble in the lattice is proportional to the fraction of the 865 VCs that fall into that category. The number and percentage of all VCs falling into each category is shown in or near its bubble.

Several interesting features are exhibited in Fig. 7. First, over 30% of the VCs can be proved *without* the use of any mathematics or more than one hypothesis. Moreover, over 75% of the VCs can be proved using no more than general mathematical knowledge along with at most one hypothesis.

However, programmer-supplied definitions do tend to result in more sophisticated proofs; more hypotheses tend to be needed than in the general case. It is rare for a VC to require only a programmer-supplied definition. The programmer-supplied definitions tend to encapsulate complexity; this complexity is apparent from the larger number of hypotheses needed to prove such VCs.

While the number of hypotheses needed to prove VCs is interesting, the metric may mask some other properties of the VCs. For example, a VC whose proof requires a larger percentage of the hypotheses might be considered "less obvious" than a VC whose proof requires a small fraction of the hypotheses. Based on the results, this is fairly rare; the majority of VCs (over 90%) can be proved using only 30% of the hypotheses.

5 Related Work

We are unaware of any prior work that empirically examines the structure or proof difficulty of VCs. Work in this area has focused instead on the initial subproblems of generating VCs and creating tools to prove them.

For example, the problem of generating VCs has been tackled using the Why methodology [10], which involves a simplified programming language, annotated with logical definitions, axioms, pre-conditions, post-conditions and loop invariants, from which VCs can be generated. A subset of both C (with annotations) and Java (with JML specifications) can be translated into the simplified programming language, such that the VCs generated are claimed to represent the correctness of the original C or Java code. The translation process from C or Java must explicitly capture the memory model of the original source language (C or Java). As a result of using RESOLVE, we do not need an explicit memory model, dramatically simplifying the generated VCs.

Also addressing the problem of generating VCs, the tool Boogie [11] takes as input annotated compiled Spec# [12] code and generates VCs in the BoogiePL language. This language has support for mathematical assumptions, assertions, axioms and function definitions along with a restricted set of programming language constructs. The BoogiePL representation is used by Boogie to generate first order mathematical assertions. The method [13] used for generating VCs is similar to the method presented in Section 2.

Addressing both the problem of generating VCs from program source and of creating tools that can prove the generated VCs automatically, Zee *et al.* [14] have used a hybrid approach of applying both specialized decision procedures

and a general proof assistant to prove that code purporting to implement certain data structure specifications is correct. However, the use of Java as a starting language requires that the specifications use *reference* equality for comparison. Our approach proves properties that depend on the *values* of the objects instead.

To repeat, none of the above papers nor any others we know about present empirical data about the sources of proof difficulty across even a small selection of VCs, let alone hundreds as reported here.

6 Conclusion and Future Work

This paper has presented the first, to our knowledge, examination of the structure of VCs generated for client code built on reusable software components. The statistics support our hypothesis that VC proofs are mostly simple bookkeeping. Only a few VCs are structurally complicated. Moreover, the vast majority of VCs can be proved using at most three hypotheses.

We have done some work using formal proof rules underlying a goal-directed approach, discussed in [8]. Preliminary results suggest that the goal-directed proof system generates fewer VCs that are comparably simple, according to taxonomy and the metrics, as the tabular method of VC generation. Further investigation of this phenomenon is in order.

Our goal is to extend this work toward a more rigorous definition of **obvious** with a finer grained evaluation of the current VCs that is more directly connected to a particular type of prover. Performing a more rigorous comparison of different VC generation strategies along with a comparison of VCs generated using different programming languages should provide valuable additional information about the sources of difficulty of VC proofs.

Acknowledgments

The authors are grateful for the constructive feedback from Paolo Bucci, Harvey M. Friedman, Wayne Heym, Bill Ogden, Sean Wedig, Tim Sprague and Aditi Tagore. This work was supported in part by the National Science Foundation under grants DMS-0701187, DMS-0701260, CCF-0811737, and CCF-0811748.

References

1. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S.M., Hollingsworth, J.E.: Reasoning about software-component behavior. In: Frakes, W.B. (ed.) ICSR 2000. LNCS, vol. 1844, pp. 266–283. Springer, Heidelberg (2000)
2. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. J. ACM 50(1), 63–69 (2003)
3. Woodcock, J., Banach, R.: The verification grand challenge. Journal of Universal Computer Science 13(5), 661–668 (2007), `http://www.jucs.org/jucs_13_5/the_verification_grand_challenge`

4. Enderton, H.: A Mathematical Introduction to Logic. Harcourt/Academic Press (2001)
5. Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental benchmarks for software verification tools and techniques. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 84–98. Springer, Heidelberg (2008)
6. Ogden, W.F., Sitaraman, M., Weide, B.W., Zweben, S.H.: Part I: the RESOLVE framework and discipline: a research synopsis. SIGSOFT Softw. Eng. Notes 19(4), 23–28 (1994)
7. Harms, D., Weide, B.: Copying and swapping: Influences on the design of reusable software components. IEEE Transactions on Software Engineering 17(5), 424–435 (1991)
8. Krone, J.: The Role of Verification in Software Reusability. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH (December 1988)
9. Heym, W.D.: Computer Program Verification: Improvements for Human Reasoning. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH (December 1995)
10. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
11. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
12. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview, `http://citeseer.ist.psu.edu/649115.html`
13. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE 2005: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 82–87. ACM, New York (2005)
14. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. SIGPLAN Not. 43(6), 349–361 (2008)