

# A Demonstration of the RESOLVE Verifying Compiler

Hampton Smith<sup>1</sup>, Chuck Cook<sup>1</sup>, Heather Harton<sup>1</sup>, Murali Sitaraman<sup>1</sup>

<sup>1</sup> School of Computing, Clemson University, Clemson, SC 29634, USA  
{hamptos | ccook | hkeown | msitara}@clemson.edu

**Abstract.** RESOLVE includes integrated programming, specification, and proof languages necessary to form the basis of a full end-to-end tool chain for creating verified software in a modular, component-based fashion. The RESOLVE Verifying Compiler that we are developing at Clemson is sufficiently mature to demonstrate the advantages of this system. We propose to demonstrate the development and verification of reusable software components in the RESOLVE programming language, from their specification through to their use in executable programs.

## 1 Introduction

The problem of mechanized program verification is perhaps best summed up in Tony Hoare's vision of a verifying compiler [1]. By verifying software correct, we guarantee—statically, at run-time—that it will work for any input. The RESOLVE language [2] is an imperative, object oriented language designed to be amenable to verification. For the tool demo, we would like to showcase our work on the RESOLVE Verifying Compiler. The technical underpinnings of this work are the topic of [3].

In the RESOLVE system, a verified component begins with a software developer specifying the component in the RESOLVE specification language, then realizing it in the RESOLVE programming language. This source code, along with its specification, is passed to the verification condition (VC) generator. VCs are mathematical assertions necessary and sufficient to prove the correctness of the code. The VCs correspond to showing that the specification of an operation holds at the end of the code for that operation, requirements of external operations called in the code are satisfied, and internal annotations such loop invariants and loop termination progress metrics are legitimate. They are generated by the mechanical application of *proof rules* [4] that transform an implementation intended to realize a specification into assertions. For code correctness, all generated VCs must be proved.

VCs can be generated in multiple formats: in the input format of the Isabelle proof assistant, in a human-readable style for manual inspection, or in a symbolic format which can be passed directly to RESOLVE's prover. Once the component is verified, the compiler generates equivalent Java code that can be executed or incorporated into an existing program.

Since the VCs are presented in a mathematical format, proofs for them are supported by RESOLVE's mathematical sub-system. This subsystem consists of a variety of mathematical theorems organized into theories. The intent is that these theories are written ahead-of-time by trained mathematicians. Since an incorrect theorem could introduce unsoundness into the system, each theorem must be

backed by a corresponding proof in RESOLVE's formal proof language. Our proof checker can thus mechanically check the proof to make sure that the theorem is correct.

## 2 Demo

The components we would like to demonstrate are as follows:

**Verified Code Generation.** Since it is the focus of our submitted paper [5], it would be appropriate to demonstrate generating correct-by-construction Java code from verified RESOLVE source code.

**Verification Condition Generation.** We have developed a web interface that enables the user to walk through the steps of selecting a component interface and a realization thereof, then generate VCs live. The nature of verification conditions is the topic of [6].

**Proving with Isabelle.** Isabelle is a proof assistant that can double as a simple automated prover. Our web demo allows the user to pass the VCs generated in the VC-generation step to Isabelle and see the results of the proof attempt

**Proving with the RESOLVE prover.** Our web demo also allows the user to pass generated VCs to RESOLVE's prover and see the results, so that they can be compared with those achieved using Isabelle. A screenshot of our web demo showing the results from the RESOLVE prover is shown in Figure 1.

The screenshot shows a web interface titled "Verification Demo" with a "Math Units" link. It features a table with four columns: "Concept", "Enhancement", "Concept Realization", and "Enhancement Realization". The "Concept" column lists files like "Unbounded\_Stack\_Template.co". The "Enhancement" column lists "UST\_Transfer\_Capability.en" and "UST\_Flip\_Capability.en". The "Concept Realization" column is empty. The "Enhancement Realization" column lists "Manual\_Transfer\_Realization.rb". Below the table are "View" buttons for each column and a "Compile" button. Below the table is a section titled "Results of RESOLVE verification:" containing a text area with the following output:

```

for isabelle);
-typecheck to check mathematical typing; -proofcheck to check mathematical
proofs;
-sanitycheck to perform various common sense checks; -prove to attempt to
prove VCs.
0 1 Proved in 70 milliseconds. Overall, 421 proofs were directly considered
and 155 useful backtracks were performed.
0 2 Proved in 2430 milliseconds. Overall, 34977 proofs were directly
considered and 12616 useful backtracks were performed.
1 1 Proved in 1 milliseconds. Overall, 1 proofs were directly considered and 0
useful backtracks were performed.
1 2 Proved in 712 milliseconds. Overall, 39333 proofs were directly considered
and 14143 useful backtracks were performed.
2 1 Proved in 9 milliseconds. Overall, 421 proofs were directly considered and
155 useful backtracks were performed.
2 2 Proved in 512 milliseconds. Overall, 34977 proofs were directly considered
and 12616 useful backtracks were performed.

```

On the right side of the results section, there are buttons for "Generate Code", "Generate VCs", "Verify with Isabelle", and "Verify with RESOLVE". Below these buttons is a checkbox labeled "Display Proofs".

Figure 1: Results of the RESOLVE prover

**Proof Checker.** The RESOLVE proof checker provides instant feed-back on theorem proofs given in our formal proof language. By changing the proofs and checking them in real time, errors can be discovered and corrected.

### 3 Relation to Other Work

In this section we will briefly present three other popular systems intended for integrated verification and compare and contrast the RESOLVE system with them.

#### 3.1 Verification of Java Components

Other examples of static verification tools for Java are ESC/Java [7] and Jahob [8]. However, unlike RESOLVE, each attempts to identify certain kinds of runtime errors rather than guarantee full functional correctness. ESC/Java's specification language is similar JML [9], while Jahob uses Isabelle. ESC/Java uses Simplify [10] as its back-end prover. The next generation of ESC/Java, ESC/Java2, is used to perform static checks for JML [11]. An interesting feature of Jahob is that, where RESOLVE requires programmers to simply provide invariants, Jahob can use shape analysis to assist in automatically determining them [12, 13].

#### 3.2 Verification of C# Components

Spec# is an extension of the C# programming language to allow formal specification and static verification. Like RESOLVE, Spec# can be used to provide full functional specifications, which can be passed to an automated prover. Unlike RESOLVE, Spec# does not require such specifications and fits the resulting components with dynamic checks of method preconditions [14].

### 4 Summary

In summary, we will demonstrate Java code generation capabilities by considering sample reusable components in RESOLVE and producing executable Java code for them. We will show generation of VCs that correspond to various correctness conditions for an example implementation. In the process, we will note the modularity of the VC generation process. We will also show how the VCs are proved automatically. Finally, we will demonstrate the proof checker.

## 5 Acknowledgements

This research is funded in part by grants from the National Science Foundation, CCF-0811748 and DMS-0701187.

## 6 References

1. Hoare C.A.R.: The Verifying Compiler: A Grand Challenge. In: Goos, G., Hartmanis, J., van Leeuwen, J. (eds.) Proc. CC 12, LNCS, vol. 2622, pp. 262-272. Springer, Berlin (2003)
2. Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W.: Part II: specifying components in RESOLVE. SIGSOFT Softw. Eng. Notes 19(4) pp 29-39 (1994)
3. Sitaraman M., Adcock B., Avigad J., Bronish D., Bucci P., Frazier D., Friedman H., Harton H., Heym W., Kirschenbaum J., Krone J., Smith H., Weide B.W.: Building a Push-button RESOLVE Verifier: Progress and Challenges. Technical Report RSRG-09-01. <http://www.cs.clemson.edu/~resolve/reports.html> (2009)
4. Harton H.K., Sitaraman M., Krone J.: Formal Program Verification. In: Wah B.W. (ed.) Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons (2008)
5. Smith H., Harton H., Frazier D., Raghuvver M., Sitaraman M.: Generating Verified Java Components through RESOLVE. ICSR 11, to appear in proceedings (2009)
6. Kirschenbaum J., Adcock B., Bronish D., Smith H., Harton H., Sitaraman M., Weide B.W.: Verifying Component-Based Software: Deep Mathematics or Simple Book-keeping. ICSR 11, to appear in proceedings (2009)
7. Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R.: Extended Static Checking for Java. In: Proc. ACM SIGPLAN 2002 Conference on Programming language Design and Implementation. Berlin, pp. 234-245 (2002)
8. Kuncak K., Rinard M.: An Overview of the Jahob Analysis System: Project Goals and Current Status, In: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, pp. 323 (2006)
9. Leavens G.T., Baker A.L., Ruby C.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM Software Engineering Notes 31:1-38 (2006)
10. Detlefs D., Nelson G., Saxe J. B.: Simplify: A Theorem Prover for Program Checking. J. ACM 52, 3, pp 365-473 (2005)
11. Poll E., Kiriya J., Cok D.: Introduction to JML. <http://secure.ucd.ie/products/open-source/ESCJava2/ESCTools/papers/CASSIS2004.pdf>
12. Wies T.: Symbolic Shape Analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, September (2004)
13. Wies T., Kuncak V., Lam P., Podelski A., Rinard M.: Field Constraint Analysis. In Proc. Int. Conf. Verification, Model Checking, and Abstract Interpretation (2006)
14. Bennett M., Leino K.R.M., Schulte W.: The Spec# programming system: An overview. In: Barthe G., Burdy L., Huisman M., Lanet J., Muntean T. (eds.) Proc. CASSIS 2004, LNCS, vol. 3362, pp. 49-69. Springer, Berlin (2004)