A Web-Integrated Verifying Compiler for RESOLVE: A Research Perspective

Daniel Welch, Charles Cook, Yu-Shan Sun, and Murali Sitaraman
School of Computing, Clemson University
Clemson, South Carolina, 29634
{dtwelch, ctcook, yushans, murali}@clemson.edu

ABSTRACT
RESOLVE is an integrated language that combines imperative programming and mathematical specifications for full functional verification of component-based programs. From a researcher’s perspective, this paper summarizes the elements of an integrated RESOLVE web IDE that includes a verifying compiler. The paper introduces elements of the language and features of the IDE with a variety of illustrative examples, including the following: Extensible mathematical units that contain definitions and results, specifications of generic components that use those mathematical units, alternative implementations of specifications, and automated generation of verification conditions and proofs for implementation correctness. While verification and research are the focus of this paper, the compiler also translates RESOLVE code to Java (or C) for execution and has been used in undergraduate computer science classes at multiple institutions for over five years.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Verification—VCs, automated proving, modular software

General Terms
Reliability, Verification, Languages

Keywords
automation, components, formal methods, specification, verifying compiler

1. INTRODUCTION
The task of creating a compiler capable of verifying the correctness of software is a long standing challenge for the computing community [1]. This challenge demands a compiler that can scale to verification of component-based systems. Any language and supporting system that aims to meet this challenge must be designed in such a way that it encourages – and indeed, requires – software and supporting mathematics to be reusable, extensible, and capable of verifying individual components in isolation. The challenge is also one of automation. Though the verification literature is replete with notable achievements in verification (e.g., [2, 3]), they require significant human intervention. So a key part of the challenge is automation [4].

A variety of papers that span a spectrum of topics in specification and verification using RESOLVE can be found in the literature (e.g., [5, 6]). Some recent papers have also discussed elements of the web IDE (e.g., its design [7, 8] and use in software engineering projects [9]). In this paper, using illustrative examples, we provide a comprehensive overview of RESOLVE’s approach to component based program verification, while demonstrating how the web IDE can become an invaluable tool for researchers to more-easily author and verify component based systems.

2. CHARACTERISTICS OF A LANGUAGE FOR VERIFIED SOFTWARE
Before delving into a more complete discussion of the RESOLVE web IDE and its role as a research tool, in this section we first explain why we place such emphasis on having an easy-to-use IDE front-end, backed by a specialized language for building verified software components. The reason ultimately stems from the sheer difficulty and breadth of the verification challenge. Elsewhere we have outlined the RESOLVE vision for verified software [10] and have noted that at the language level, in order to be amenable towards automated verification of practical programs, languages must possess in some form or another the following characteristics.

- **Integrated Specificational Capabilities** First and foremost, a language designed to facilitate verification must include a specificational counterpart to an implementation language. Further, if these two pieces are to be fully integrated, then they must share consistent semantics that will allow a verifier to ensure that an implementation provides the specified behavior. Thus, the language must make it possible to write separable interface specifications for components.

Furthermore, in order to show that a piece of code is correct, a verifying compiler must be given sufficient “hints” along the way to make certain justifications obvious. This capability typically comes through code sufficiently annotated with interface-level specifications describing not only operational pre and post
conditions, but also loop invariants, progress metrics for establishing termination, representation invariants, and abstraction functions/relations within ADT implementations. It is therefore imperative that the language includes suitable syntactic slots for supplying these assertions.

- **Clean Semantics** Any language that keeps the effect of code local to some restricted subset of a program’s overall state space is said to be clean in the sense that only those objects directly targeted by a particular programmatic construct are permitted to change. The push for clean semantics is seriously hindered in languages that allow nonlocal reasoning to occur through uncontrolled referencing, aliasing, and mutation [11]; while there are various ways to mitigate the impact of aliasing, RESOLVE includes a swap operator to move objects [12].

- **Reusable Mathematical Theories** The ability to model a software component through specifications that make use of abstract mathematical models is yet another important capability for a language supportive of verification. It is unlikely however that only a handful of models will suffice when one considers the full spectrum of applications – most of which undoubtably lie outside the boundaries of “typical,” well worked mathematics. Thus, the ability to create new, extensible theories personalized for a given application is a necessity.

- **Reusable Components** Finally, languages must have some way to amortize the cost of specifying and verifying components over a collection of applications. This amortization is only possible if the language enforces a strict separation between the implementation of a component and its corresponding specification (or, contract). By enforcing this strict separation of concerns, it becomes possible to cleanly verify that a realization is correct with respect to it’s mathematical, abstract specification without binding the specification to one particular implementation.

A language with these characteristics, backed by an easy to use web IDE will allow us to “sidestep” the many difficulties in trying to retrofit verification friendly facilities into existing languages, and instead more easily explore one component of our central thesis: That given suitable annotations, reasoning about the correctness of programs is as straightforward for automated provers as it is for competent human programmers who hold an intuitive knowledge that their programs work as intended [13].

While the focus of this paper is on verification using RESOLVE and its web IDE, we note that these same ideas have been moved across the board to other languages, such as C++ and Java [14, 15].

3. A COMPREHENSIVE EXAMPLE

In this section, we use mathematical and programmatic elements of a Queue component family to give a concrete illustration of the necessary language features described in Section 2, and familiarize readers with the layout and available functionality of the web IDE\(^1\). Upon first visiting the web interface – shown in Figure 1, users are presented with the option of loading one or more different components ranging from specifications (Concepts) to mathematical theory constituents (Theories).

![Figure 1: Users may draw from a variety of reusable, verified components or create their own, new theories and components.](image)

3.1 Concept Specification

Every RESOLVE component has a formal interface specification. Even typically built-in objects such as arrays, integers, and pointers have formal descriptions of their behavior through interfaces.

An example Concept (or interface specification) is the Queue_Template abstraction presented in Figure 2. This specification is parameterized by a generic type Entry and an integer Max_Length indicating the maximum number of Entries a Queue can hold. The evaluates parameter mode preceding Max_Length indicates that the actual argument passed may be an arbitrary Integer expression and that it is evaluated and used as a parameter. In order to instantiate and use this generic concept, the client is required to supply suitable arguments for Entry and Max_Length (additionally, users are also required to select a realization as we will demonstrate later).

The uses line gives the specification access to a mathematical theory of strings called String Theory, which contains a number of useful definitions intended to help specifiers author succinct, correct specifications for a given ADT. (The theories also contain proven theorems involving these definitions for use in verification, a point explained in a future section.)

The Type Family clause introduces a collection (i.e. family) of abstract types called Queues that are modeled by strings of Entries. The exemplar Q clause that immediately follows can be thought of as an example queue that acts as a representative to our ADT family. Use of the exemplar Q can be seen in the constraint clause, where we use it to assert that not all strings of Entries can be models of valid queues, but rather, only those of length Max_Length or less.

The Initialization clause notes that all queues are guaranteed empty when initialized, which is to say – drawing
Figure 2: The Queue_Template specification as it appears in RESOLVE’s web IDE. Here we also showcase some built-in educational features, such as explanations of keywords on demand.

Comparisons to programming – it specifies a constructor.

Now that the framework for our abstract model is specified, we may now define operations on Queue such as Enqueue, Dequeue, and Length. Each of these operations carries with it associated pre and post-conditions, signified by requires and ensures clauses, respectively. As with the parameters to the Concept module, formal parameters to operations are similarly preceded by a specification parameter mode which makes explicit what effect the operation will have on the parameter. In particular, alters means that only the input value of that parameter is relevant and that no guarantees are made concerning its value after the operation; use of this mode allows implementations to avoid a need to copy either references (which causes unnecessary aliasing) or copy data representations (which can be expensive) [12]; updates means a parameter passed with a meaningful value will be changed to a new meaningful value specified in the ensures clause by the end of the call. For a replaces mode parameter the passed input value is ignored, because it is being replaced with the ensures clause's meaningful value.

The requires and ensures clauses are strictly mathematical assertions, and as such, their variables refer to the mathematical values of the parameters. Thus, the \( Q \) parameter of operation Enqueue refers to the mathematical value of a Queue – as defined within the Concept, as opposed to the programmatic value of a Queue as defined in a realization of the concept.

Since mathematical variables have only one unchanging value, a pound sign \( # \) is used in specifications when we wish to explicitly refer to the value of a parameter at the start of a call, as opposed to its value at the end. By way of example, the ensures clause of the Enqueue operation in Figure 2 uses \#Q to refer to the incoming value of the Queue as opposed to simply \( Q \) which implicitly denotes its outgoing value at the conclusion of the call.

3.1.1 Mathematical Support

Queue_Template specification employs a mathematical theory of strings called String_Theory. The precis of this theory contains the definitions of the terms used, such as Str used in modeling and string notations. The angled braces \(< . . . >\) and \( \circ \) operator found in the ensures clause of Enqueue are notation defined in String_Theory used to represent a singleton string and string concatenation, respectively. Given this, the specification of the ensures clause of Enqueue may be read in English as: “The outgoing queue is equal to the incoming queue concatenated with the incoming singleton string.”

Responsible however for more than simply defining operators and notational conveniences, any number of definitions and useful results (theorems) are also listed within a theory’s Precis. Figure 3 shows a portion of String_Theory’s Precis to provide examples of theorems used by the verifier involving various string notations.

Figure 3: A snippet of RESOLVE’s mathematical theory of strings open in the web IDE.

Taking a page from the world of object oriented programming’s header-implementation relationship, readers will note that no proofs of the theorems appear at the Precis level, but rather, are proven offline and relegated to a separate proof-unit, since they provide fine grained information unnecessary to most general users of the theory.

RESOLVE’s current library of theories is by no means complete, or for that matter, exclusive to strings. Rather, any number of typical theories such as numbers to more
sophisticated ones such as trees and spirals have been developed and ultimately will be used in specification of programming concepts similar to that described above in our queue example.

3.2 Realization

Once a Concept has been specified, an implementation has to be provided in a Realization for it to be used. Shown in Figure 4 is a Circular Array Realization of Queue_Template. It is worth noting that there could be any number of implementations of the same concept. In this paper however, we focus exclusively on a particular “circular” array-based implementation.

![Figure 4: A Circular array based realization of the Queue_Template specification. The blue VC buttons on the left are verification conditions corresponding to particular portions of code. Ultimately, each of these VCs must be proven to establish correctness.](image)

The first point to notice in the realization is that we chose to represent our queue programmatically as a Record (analogous to a C struct) containing an array, Contents, and integers, Length and Front to denote the size of, and front of our queue, respectively. One thing to note about the Contents array is that it is not built-in, but rather, is specified the same as any other component. While some “syntactic sugar” for arrays is provided, a pre-processing step translates all array manipulation into interactions with a normal component, allowing reasoning to proceed via the normal concept machinery described above.

We impose restrictions on this programmatic representa-
The organizing principle governing enhancements and their associated realizations is the same between concepts and concept realizations: The enhancement specification is purely conceptual, and hence implementation neutral. Thus, there can be any number of specific realizations for any one particular specification. For example, in our `Append_Capability` specification shown in Figure 5, we conceptually dictate what exactly it means to stick (or, append) two queues, \( P \) and \( Q \) together while two separate methods might implement the operation using different techniques (i.e., one recursively and the other iteratively).

Figure 6: The specification of `Append_Capability`.

Although realizations of an enhancement are concrete in the sense that they spell out specific ways of implementing a specification, they do not have access to the internal implementation details used to realize the base concept. Thus, since `Iterative_Append_Capability` makes no mention of the internal `Contents` array or any other implementation-specific construct used to represent our programmatic circular-array based queue, enhancements are permitted to exist independent of any one particular realization.

Consider the realization of `Iterative_Append_Capability` shown in Figure 7. This particular `Append` procedure works by iteratively emptying queue \( Q \) - preemptively suggested by its `clears` parameter mode – and repeatedly enqueueing the most recently dequeued element from it, into the designed “updates” queue, \( P \).

While there are some efforts to infer invariants, most automated systems [4], require programmers to supply loop invariants that will be checked and used by the verifier. As Figure 7 shows, the invariant is captured through a `maintaining` clause; and a `decreasing` clause is used to specify a progress metric that will be used to show termination of the loop (or recursion).

### 3.4 Automated Verification

Using RESOLVE’s integrated term-rewrite prover, we are able to fully and mechanically verify the `Append` procedure and others, including ones for generic sorting [17]. Figure 9 shows some of the VCs generated by `Iterative_Append_Realiz`, while Figure 10 displays the same method post verification, with VCs that were successfully dispatched displayed in green, and any others unable to be proven marked with a red “X”.

In proving this particular VC, a helpful result to have in our arsenal of theorems would be one that asserts associativity of string concatenation. Specifically, with givens 9 and 11 (shown in Figure 9), and the following associativity theorem in `String_Theory` reproduced from Figure 3 in Figure 8, the goal is straightforward enough for the prover to establish automatically. In general however, a variety of theorems from different mathematical precis may be necessary to prove some verification conditions. The actual steps involved in transforming givens, and applying theorems of the form shown above is beyond the scope of this paper. Those readers interested in a more in depth discussion of this and other proof-related processes might refer to [17].

### 3.5 Putting It Together

Now that we have specified a fully fledged component and an extension on it, we now give a brief demonstration with the goal of illustrating how a typical client might use this generic component with extensions (enhancements) to reverse a queue. In doing so, we also hope to detail how users of the web IDE can author their own verified RESOLVE concepts, realizations, extensions, and client facilities.
Among the well-known efforts for specification and verification of Java programs is Java Modeling Language (JML) [21], a language that serves as the basis and supports a broad range of applications including runtime assertion checking, as well as design-by-contract based components. While there are similarities in the goals of RESOLVE and JML in the shared emphasis on design by contract, RESOLVE specifications are closer in spirit to assertions in mathematics, whereas JML specifications are in Java-style.

Our sister group at Ohio State also has developed a verification system for RESOLVE. The verification conditions generated can use either SplitDecision [22] (an in house decision procedure prover), Z3, or Isabelle [23]. The Ohio State system, though not full-fledged like the IDE described here, provides a web-based user interface where a user can access an existing library of components and generate verification conditions.

4. RELATED WORK

Currently, there are a number of ongoing projects that seek to design an automated-system capable of producing verified software. While we present a brief overview of several here, readers are encouraged to refer to [4] for a more comprehensive discussion of the systems mentioned.

VeriFast [18] is a system that allows annotations to be inserted as special comments into (possibly) multi-threaded Java or C code. With these annotations in place, VeriFast achieves verification using Microsoft’s Z3 [19] SMT solver as a verification checking backend.

Microsoft Research has designed Dafny from the ground up to be a programming language capable of verifying linked structures and other demanding verification related benchmarks. It uses Boogie [20] as an intermediary language to generate verification conditions, which are then fed directly into Z3. Both Dafny and VeriFast also feature an interactive online tool in which users are able to author their own programs and follow embedded tutorials directly in the website.

5. SUMMARY

This paper provides a comprehensive overview of an integrated language for verification that brings together mathematical specifications and software components. It describes in detail a realization of these principles through a web IDE that supports the RESOLVE language and its verifying compiler. With the web IDE and a language supportive of the characteristics summarized in Section 2, we have made the development process of verified components and component-based systems as simple as possible. The language and the compiler have been used in research and computer science education at multiple institutions for several years, and continues to undergo improvements on all fronts.

6. ACKNOWLEDGMENTS

We thank past and present members of our research groups at Clemson and Ohio State for their contributions to the ideas contained in this paper. This research has been funded in part by the NSF grants CCF-0811748, CCF-1161916, and DUE-1022941.

7. REFERENCES

Figure 9: Hovering the cursor over the blue VC buttons reveals which VC corresponds to which line of code.

Figure 10: The Iterative_Append_Realiz, fully verified.


[13] Jason Kirschenbaum, Bruce Adcock, Derek Bronish, Hampton Smith, Heather Harton, Murali Sitaraman, and Bruce W. Weide. Verifying component-based software: Deep mathematics or simple bookkeeping?


