Modular Verification of Generic Components Using a Web-Integrated Environment

Charles T. Cook, Heather Harton, Hampton Smith, and Murali Sitaraman

Technical Report RSRG-11-03
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

September 2011

Modular Verification of Generic Components Using a Web-Integrated Environment

Charles T. Cook, Heather Harton, Hampton Smith, and Murali Sitaraman

School of Computing
Clemson University
100 McAdams Hall
Clemson, SC 29634
{ctcook, hkeown, hamptos, murali}@clemson.edu

Abstract. This paper presents a web-integrated environment that has been especially built to capture component relationships and allow construction and composition of verified generic components. The environment facilitates team-based software development and has been used in undergraduate CS education at multiple institutions. The environment makes it easy to simulate "what if" scenarios, and has spawned much research and experimentation. The paper describes a generic sorting component enhancement that is layered over a base component. Sorting is parameterized both by the mathematical ordering used in the specification and the actual operation used to realize the mathematics. The paper also describes a facility component that instantiates sorting for a particular case. The facility component, sorting components, and base components are verified totally independently. In the process, the paper illustrates the issues in generic software verification and the role of higher-order assertions.

Keywords: Generics, education, specification, system description, and verification.

1 Introduction

For software verification to be viable, it must be scalable. Software implementations must be component-based and the components must be designed to allow for verification of one component at a time. The verification process should avoid re-verification of components, even when they are generic (i.e., parameterized). Given that a component-based system often consists of several components, it is important for the verification to take place within an environment that respects component relationships. To ease construction and composition of components for students and researchers, this environment is web integrated. It has been designed to run fully within the user's web browser, removing the need to install and configure any additional software. It makes use of the latest browser technologies to provide an intuitive, dynamic, and feature-rich environment for cross-platform component development and reasoning. The environment is available freely on the web.

The web-integrated environment presented in this paper facilitates component-based development and modular verification. The environment emphasizes the relationships among the components, instead of taking the traditional file system browser approach used by most integrated development environments (IDEs). It also provides tools for reasoning about the correctness of component-based software. Verification conditions (VCs) can be generated and their correctness can be proved via an automated proving system.

The specifications, implementations, and verifications are presented in this paper in RESOLVE, an integrated specification-programming language [1], designed to build verifiable software from components. Theorems used by the automated prover are independently established through a proof assistant, such as Isabelle [2], though other systems can be used as well. Verified RESOLVE software is automatically translated to Java code [3] for execution.

To illustrate both the novelty of the research and system contributions in this paper, we use verification of a generic queue sorting component whose specification is parameterized by the mathematical order to be used in sorting (that is constrained to be a total pre-ordering) and implementation is parameterized by an operation that computes that relation. Such a component is among the verification benchmarks proposed in [4]. The annotations in the specification of the sort operation and the internal assertions in the implementation, such as loop invariants, are higher-order assertions. The assertions and the supporting mathematical library have been designed to avoid the need for quantifiers. While literature contains automated verification of instances of the sorting problem, verification of a component with higher order assertions remains difficult [5]. A related aspect is the generation of suitable verification conditions that must be satisfied at the time the generic component is used. Specifically, VCs need to guarantee that the user-supplied actual mathematical ordering and operation at the time of instantiation satisfy the formal requirements.

The paper is organized as follows. Using the various specification and implementation pieces necessary to define and use a generic queue sorting component as examples, the paper presents elements of the web-integrated environment that highlight components and their relationships. Then will be a discussion of the RESOLVE verification condition generator and minimalistic prover as applied to the example components, followed by a section on verification of client code. A later section summarizes the use of the environment in undergraduate software engineering courses at multiple institutions. The paper ends with a description of related work and a presentation of our conclusions.

2 Relationship-Centered Component Management

While traditional, stand-alone IDEs are often used to build and manage component-based systems, none have been specifically designed to take advantage of and exploit the relationships between the components. The package managers that most employ are simply a visual representation of the file system hierarchy of the project storage space. The web-integrated environment takes a different approach in present-

ing RESOLVE components to users. The environment's user interface is divided into tabs representing each of the RESOLVE component types. **Fig. 1** shows the upper level of the user interface that highlights concepts (data abstraction specifications), realizations (implementations), enhancements (specifications that inherit and extend concept specifications), enhancement realizations, and facilities (modules that compose and use some combination of the above). Typical implementations of components are built reusing other components, and keeping their verification modular is a key goal of the research. Mathematical theories used in the specifications are not explicitly shown, but are available under the browser tab.

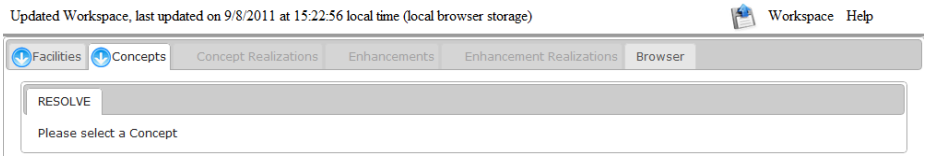

Fig. 1. Web-integrated environment user interface

To illustrate the issues, a layered sorting implementation is used as a central example in this paper (see **Fig. 2**). The selection sort code, instead of being written using an array, is based on queues, which themselves are built layered on arrays using the interface specification Static_Array_Template. The figure shows one possible implementation of each involved from among many possible alternatives.

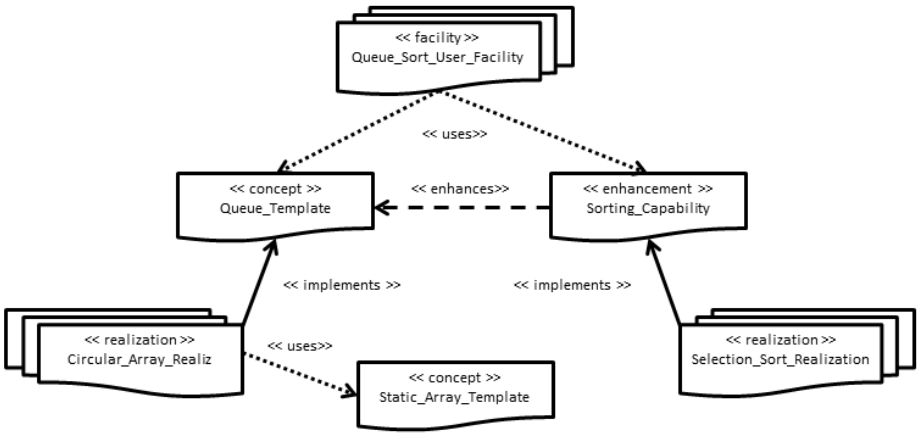

Fig. 2. Relationships among example specifications and implementations

Upon entering the web-integrated environment's URL, users can choose to browse the RESOLVE workspace component library or create a new concept or facility. Lists of available concepts and facilities are provided through dropdown menus opened by clicking the blue down arrows on the respective tabs.

A sampling of concepts available online can be seen by activating the Concept dropdown menu. New concepts can be created or loaded from file using options

available in that menu. Queue_Template concept is selected and shown in **Fig. 3**. The specifications in the editor are syntax highlighted and can be freely modified by the user. While open, the contents of the editor are syntax checked in real time. Lines with syntax errors are indicated by a red icon to the left of the line number; the exact error is retrieved by hovering over the icon.

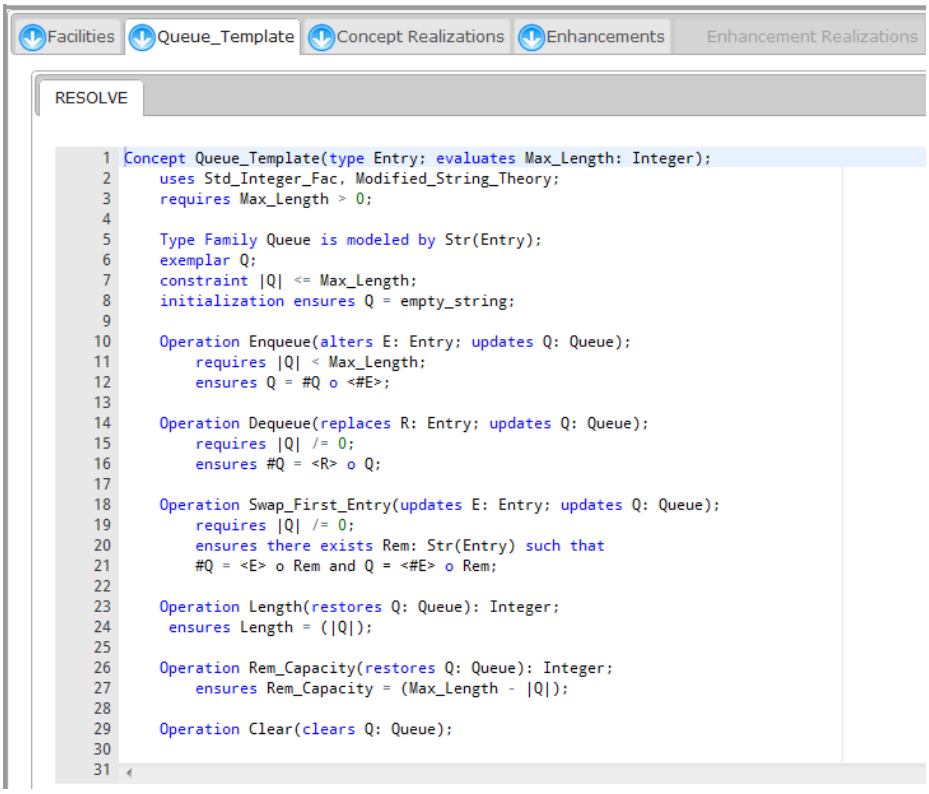

Fig. 3. Specification of Queue_Template in the editor

The Queue_Template concept (**Fig. 3**), like all RESOLVE concepts [6,7], begins with a mathematical model of the provided abstract data type Queue. The template is parameterized by the type entries in the Queue and a maximum length. In this specification, the concept uses mathematical String_Theory and models a Queue (of Entry) as a string (of Entry). The constraints clause restricts lengths of queues from this concept to be less than or equal to Max_Length. Whereas implementations of Queue_Template must guarantee through the abstraction mapping (or relation) of their internal data structure that the mappings are always within the constraints, users of queues may always assume such is the case; so the verifier obligates implementations that the abstraction mappings are legitimate and assumes the constraints in proving client code. Queue_Template also contains specifications for operations it provides. An ensures clause, (i.e., post-condition), is a guarantee of the functionality of

the operation, but only if the caller has satisfied the requires clause (i.e., pre-condition) of the operation. In RESOLVE notation, ensures clauses describe the incoming value of a variable by prepending a '#' to the variable name. For example, the ensures clause for the Enqueue operation (line 12) could be read as "the outgoing queue Q equals the incoming Q concatenated with the string containing the incoming value of E." RESOLVE notation also supports parameter passing modes, keywords describing the expected behavior of the variable. Additional details may be found in [6,7].

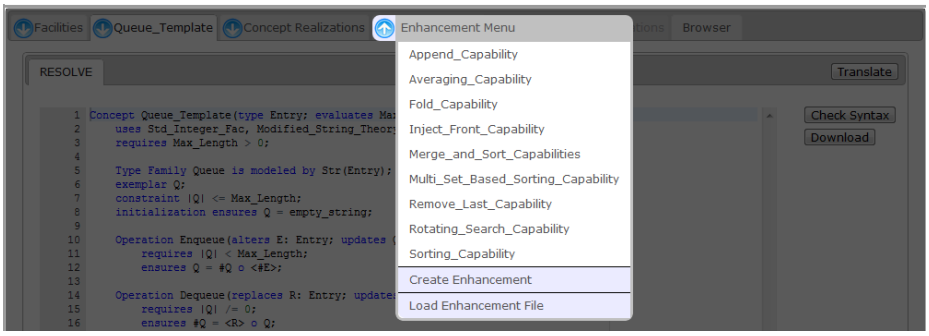

Fig. 4. Available example enhancements for Queue_Template

In addition to displaying the concept, the environment also activates the dropdown icons for realizations and enhancements. A realization component contains RESOLVE code for implementing either a concept or an enhancement and will be related to a single concept or enhancement; however a concept or enhancement can have many different realizations. An enhancement of a concept provides additional functionality and is implemented *independent* of the realizations of the concept.

Like the concept dropdown menu, the enhancement menu also has options to create, load, or open one (**Fig. 4**). In contrast to typical IDE package managers, the only specifications visible here are those that inherit from and enhance Queue_Template.

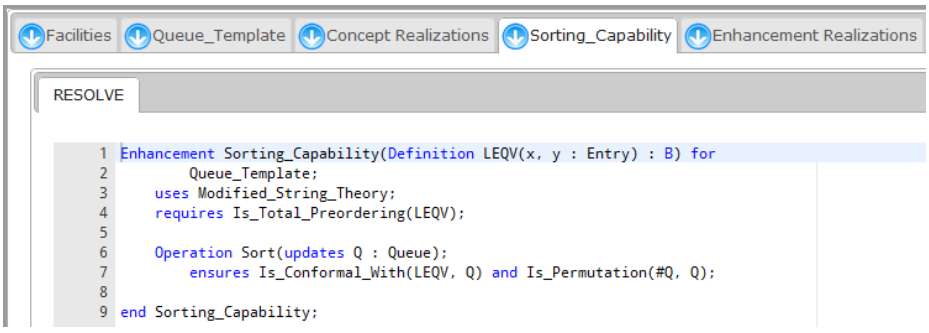

Fig. 5. Specification of Queue Sorting_Capability in the editor

The specification of Sorting_Capability (**Fig. 5**) includes a mathematical definition LEQV as a parameter which is required to be a total preordering (i.e. total and transitive). The ensures clause of the Sort operation states that the updated Q is a permutation of the input Q (#Q) and that it is conformal with the given predicate LEQV. The definition of Is_Permutation, the higher-order predicate Is_Conformal_With (that states that every two entries in the given string conform to the given ordering), and the theorems available for use by the prover are found in String_Theory. Whereas quantifier avoidance makes the specifications more readable from a software engineer's perspective, quantifier avoidance and the corresponding mathematical development of the string theory ease the task of automated proving even when dealing with higher-order assertions.

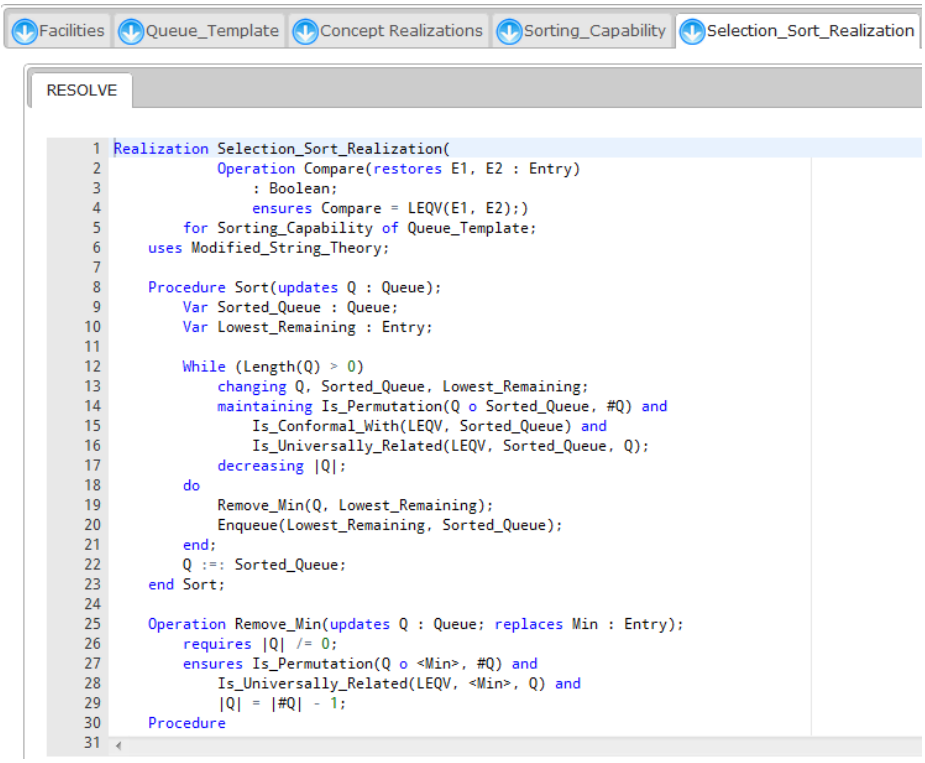

Fig. 6. Selection_Sort_Realization code in the editor

Given the enhancement specification, the Enhancement Realization dropdown icon can be used to browse implementations specific to the selected enhancement. Selecting a realization, such as Selection_Sort_Realization, opens it in the editor (**Fig. 6**). Selection_Sort_Realization is a generic sorting realization that implements the Sort operation specification in the Sorting_Capability enhancement. It is important to note that the realization parameter is a fully specified formal operation called Compare whose ensures clause is based on the total preording LEQV discussed earlier. The

user must provide an actual operation at the time of instantiation, depending on the Entry type. The actual operation must satisfy the specifications given here.

RESOLVE requires minimal code annotations [8]. The loop in the Sort procedure is annotated with a maintaining clause (i.e. an invariant), a decreasing clause (i.e. a progress metric for termination), and a changing clause (i.e. a list of modified variables) that helps simplify a class of invariants. The predicate Is_Universally_Related holds if all the entries in the first string precede all the entries in the second string, according to the given ordering. The figure also shows the specification of a local operation Remove_Min that is used in the sorting code. Related information on assertions for sorting components may be found in [5,9].

3 Modular Verification of a Generic Sorting Implementation

To verify the correctness of the implementation in **Fig. 6**, the verifier needs no knowledge of how queues are represented or implemented. The verifier does not need any information on how the component might be used either; in other words, the verification is purely modular and requires no code beyond that of the module being verified.

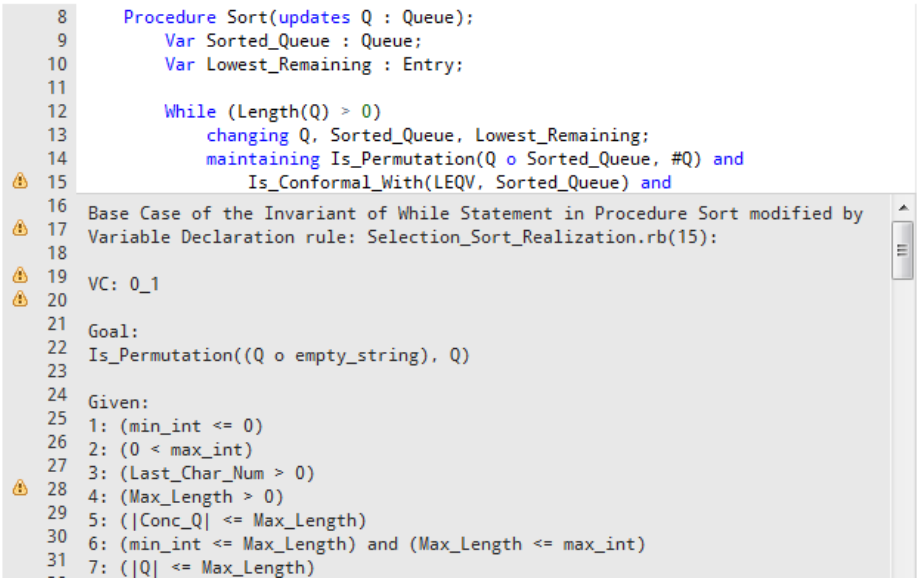

Fig. 7. Sort operation with VC overlay

For each realization, the system's verification condition (VC) generator will generate VCs that if proven will show that the component realization satisfies the specification of the component. Mechanical proof rules used to create the VCs are summarized in [10]. VCs come from various sources in the code. There will be a VC to show that the post-condition (ensures clause) of each operation holds and a VC to show that the

pre-condition (requires clause) of any operation called is satisfied. VCs will also need to establish invariants and termination. The system uses a goal-oriented approach to creating these VCs [10]. This normally begins with the post-condition of each operation as the goal. The goal is then modified via the proof rules for each programming language element until there are no more proof rules to apply.

Fig. 7 shows the effect of clicking the VC button to generate verification conditions of correctness for this generic realization. The VCs are shown at the approximate place in the code so that the user can connect the VCs with the code; for example, if a called operation has a precondition, the VC corresponding to proving the requirement is shown just before the call. The figure shows a part of a VC necessary to establish that the given invariant is an invariant for the base case.

4 Automated Proving

Many verification systems [11,12,14] rely primarily on an SMT-style prover for the brunt of the verification. SMT provers trade generality for raw speed by limiting the domains of VCs over which the prover can operate (to, for example, integers) and then using heuristics to very efficiently bit-blast possible valuations until a solution is found. The RESOLVE minimalist prover [1] within web-integrated environment explores the opposite approach: maintaining generality even at the cost of some speed and relying on a flexible verification-system and a well-engineered spec to result in easy-to-prove VCs (a topic explored more in [16]). Results from attempting to verify the selection sort realization are shown in **Fig. 8**.

Whereas one VC is shown not provable, because of missing theorem(s) on total preordering, the rest are established by the prover. The minimalist prover is, at its core, a simple re-write prover, using a body of available theorems to replace like with like in the VC until it becomes obvious. This can result in large proof spaces (to explore 6-step proofs potentially involving any of 10,000 theorems requires no less than 10,000^6 proofs be considered and potentially many more), so the prover attempts to make intelligent choices about likely proof paths (for example, theorems that introduce new types or relations are de-prioritized) even while reasonably assuming that VCs from routine software represent straightforward logic [16]. Additionally, the mathematical system is designed to exploit modularity techniques from the realm of software engineering [17] to limit available theorems.

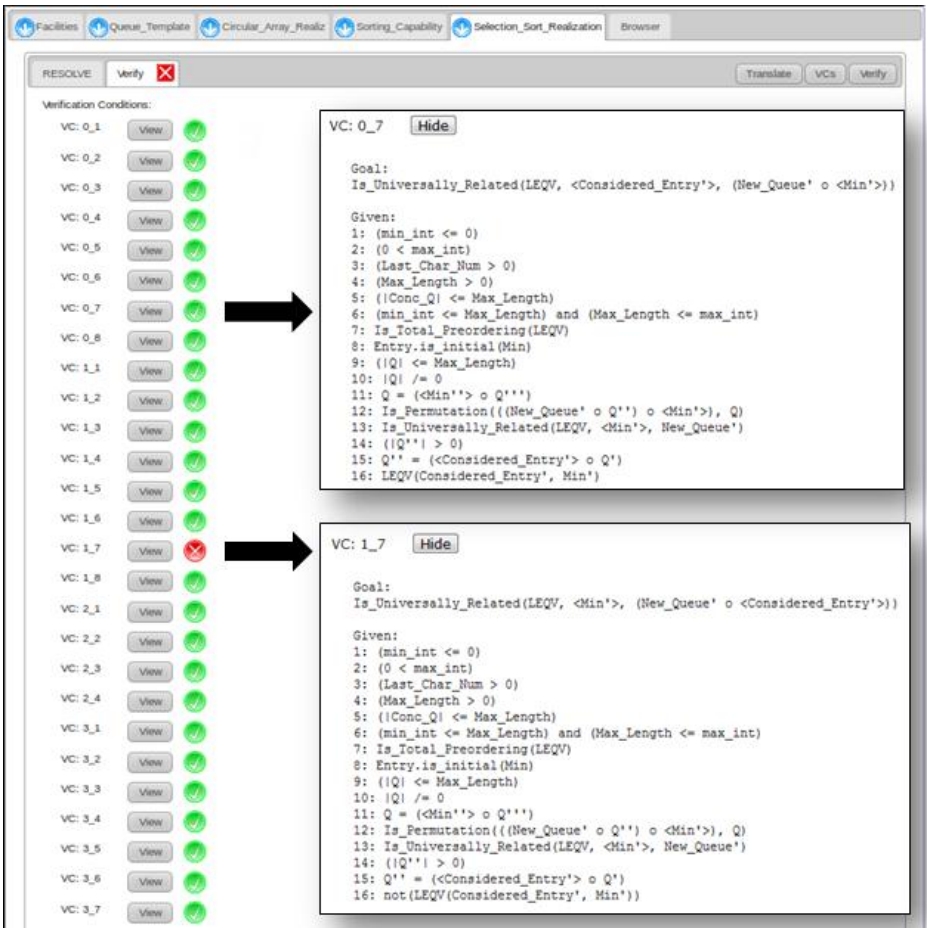

Fig. 8. Automated verification results for Selection_Sort_Realization

5 Modular Verification of Client Code

RESOLVE facilities make it possible to instantiate and use concepts, enhancements, and realizations. They can themselves be used within realizations when a new component is built using an existing one. Like the other types of components available through the environment, facilities have a dedicated tab and dropdown icon. **Fig. 9** shows a facility, Queue_Sort_User_Facility, open in the editor.

This facility uses the previously mentioned components to build and enhance a queue. Here, Queue_Template is instantiated with Student information on lines 18-21. An array-based realization (not shown) has been chosen for implementation. The Queue facility is enhanced with Sorting_Capability (and a realization is chosen) to give access to the Sort operation. Of particular interest are the definition Rank_LEQV and the operation Rank_Order. These are the mathematical ordering and comparison

operation mentioned earlier that are passed as arguments to Sorting_Capability and Selection_Sort_Realization, respectively. (The figure also shows a second definition Age_GEQV to illustrate that the same user facility can instantiate and use sorting in multiple ways.)

Queue_Sort_User_Facility | Queue_Template | Concept Realizations | Sorting_Capability

RESOLVE

```
1  Facility Queue_Sort_User_Facility;
2        uses Std_Boolean_Fac, Std_Integer_Fac, Std_Char_Str_Fac, Queue_Template;
3
4     Type Student_Info = Record
5         Name: Char_Str;
6         Rank: Integer;
7         Age: Integer;
8     end;
9     Definition Rank_LEQV(S1,S2: Student_Info): Boolean = (S1.Rank <= S2.Rank);
10    Definition Age_GEQV(S1, S2: Student_Info): Boolean = (S1.Age >= S2.Age);
11
12    Operation Rank_Order(restores S1, S2: Student_Info): Boolean;
13       ensures Rank_Order = (Rank_LEQV(S1, S2));
14      Procedure
15        Rank_Order := (S1.Rank <= S2.Rank);
16    end Rank_Order;
17
18    Facility QF is Queue_Template(Student_Info, 10)
19       realized by Circular_Array_Realiz
20       enhanced by Sorting_Capability(Rank_LEQV)
21         realized by Selection_Sort_Realization(Rank_Order);
22
23    Operation Main();
```

Fig. 9. An example user program for queue sorting

There are a few interesting aspects in generating VCs for the facility. When the Queue facility is created, VCs must be created to show that the parameters provided to the facility meet the specifications. For the present example, Max_Length must be greater than 0 (because of the requires clause in the concept Queue_Template), so a VC is generated to show that 10 > 0. A VC must also be generated to show that Rank_LEQV is a total pre-ordering (because of the requires clause in the Sorting_Capability enhancement). **Fig. 10** shows these two VCs.

In the facility declaration, the Sorting_Capability enhancement is realized by Selection_Sort_Realization. For this realization, Rank_Order is passed as an argument. Thus VCs must be generated to show that the Rank_Order operation is strong enough to be used as the Compare operator (given in the realization of Selection_Sort_Realization). One VC must show that the requires clause of Compare is strong enough to show that the requires clause of Order is true. This is easy because both operations have no requires clause. The resulting VC will simply have a goal that we need to prove 'true' with an assumption 'true'. There will be another group of VCs that must show that the ensures clause of Rank_Order implies the ensures clause of Compare. These are more intricate to generate but simple to prove and require

replacing the parameters with the actual values so that it can be proven. This must also take into account the parameter modes. Both Rank_Order and Compare restore their parameters and a VC must show that to be the case in Rank_Order since Compare restores the parameters. These VCs are shown in **Fig. 11**. The VCs generated are simple and all of them can be automatically proven by a minimalist prover.

```
1  Facility Queue_Sort_User_Facility;
2        uses Std_Boolean_Fac, Std_Integer_Fac, Std_Char_Str_Fac, Queue_Template;
3
4     Type Student_Info = Record
5         Name: Char_Str;
6         Rank: Integer;
7         Age: Integer;
8  Requirement for Facility Declaration Rule for QF: Sorting_Capability.en(5):
9
10 VC: 3_1
11
12 Goal:
13 Is_Total_Preordering(Rank_LEQV)
14
15 Given:
16
17
18 Facility Declaration Rule: Queue_Template.co(42):
19
20 VC: 3_2
21
22 Goal:
23 (10 > 0)
24
25 Given:
```

Fig. 10. Some VCs pertaining to a Queue with sorting facility declaration

```
12    Operation Rank_Order(restores S1, S2: Student_Info): Boolean;
13 Ensures from QF: Queue_Sort_User_Facility.fa(12):
14
15 VC: 2_1
16
17 Goal:
18 Rank_LEQV(S1, S2) = Rank_LEQV(S1, S2)
19
20 Given:
21 1: Rank_Order = Rank_LEQV(S1, S2)
22 2: #S1 = S1
23 3: #S2 = S2
24
25 Ensures from QF: Queue_Sort_User_Facility.fa(12):
26
27 VC: 2_2
28
29 Goal:
30 S1 = S1
31
```

Fig. 11. Verification that the actual Rank_Order operation satisfies the formal requirements

In addition to VC generation, the environment includes the ability to translate RESOLVE components to Java source code [3] and build executable Java programs for facilities. Environment-generated Java programs are packaged as an executable jar archive that can be downloaded and saved to a user's local file system. The jar file can be double-clicked to execute the program. **Fig. 12** shows part of the code and the results of running Queue_Sort_User_Facility

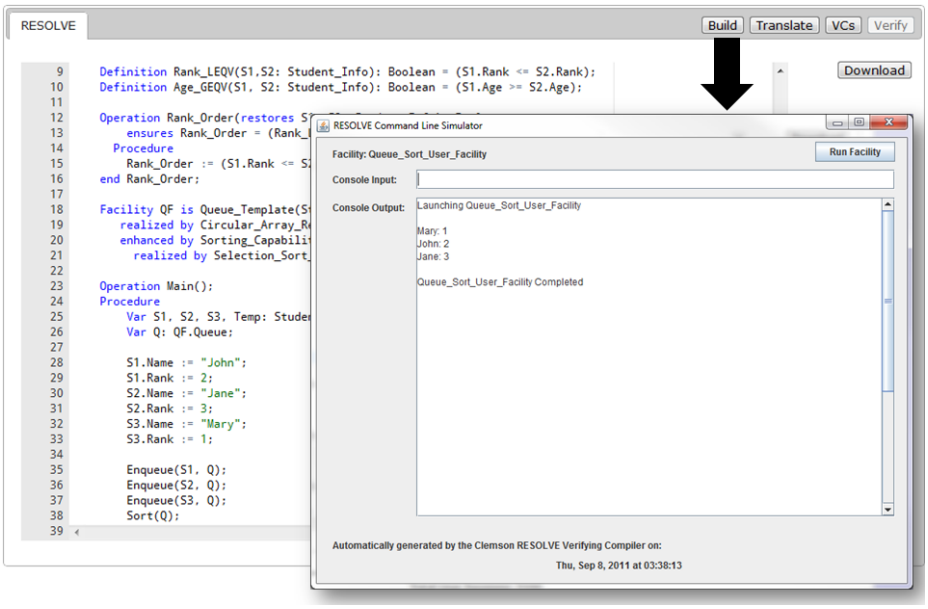

Fig. 12. Running Queue_Sort_User_Facility

6 Use in Software Engineering Education

The web-integrated environment is meant to be used as a collaborative software development platform [13]. It has been used in research and has also been put to the test in CPSC 372, a junior level software engineering course at Clemson University for the past several semesters for a project made up of a series of component-based software engineering assignments. The assignments were cumulative in nature; each built upon the results of the previous one. A typical assignment involved the development of at least three components, and the overall project involved over a dozen components. Teams of three students were instructed to individually use the system to develop, verify, and save their respective work. When complete they were to combine their individually-developed modules into a single workspace, save the workspace to a local file, and submit the file to the instructor for evaluation.

The overall goal for these projects is to teach the students the importance and usefulness of specifications and verification in contract-based software development. For each assignment, the students were given one or more component specifications and

were often required to create multiple implementations of a given specification, satisfying different design constraints. In some assignments, students were given the internal contract for a component implementation, such as representation invariants and abstract functions, which must be satisfied. In every assignment, a third team member was required to create a facility component that exercised each of the implementations. If all of the implementations were correct with respect to their specifications, the particular implementation choices in the facility would not matter; the program results would be identical. Students would switch among implementations to understand the ideas of modular verification.

The results from the experiments were positive. Most of the students were able to successfully complete and submit the project on time. Students in the fall 2010 course averaged 92.7% and those in spring 2011 averaged 85.6%. Both averages are significantly higher than their course averages.

The system's verification condition (VC) generation capabilities were also tested in in software engineering courses at Clemson, Alabama, Western Carolina, DePauw, and Cleveland State. Students in these courses were given an individual assignment to reinforce reasoning techniques covered in class. Students generated VCs and proofs for examples, and also explored "what-if" scenarios by making changes to specifications and code. These class averages were 89% and 92% for the fall 2010 and spring 2011 assignments at Clemson.

7 Related Works

An excellent summary of ongoing verification efforts is given in [8]. This section will briefly touch on some of these and a few others

While technically a research language for experimenting with the verification stack, Dafny [14] has gained some legitimacy as a full-strength programming languages thanks to its successes in verifying linked components and addressing challenging problems [15]. It uses boogie [18], a shared intermediate verification language that handles VCs generation in a language-independent way before passing them off to the Z3, an SMT style automated prover.

VeriFast [19] is verification system for Java and C programs. It provides a user interface allowing users to add contractual annotations as specially marked comments to source C and Java source code. VeriFast also features an interactive verifier that uses Z3 and a debugger to diagnose verification errors.

JMLEclipse [20] is a tool supporting the Java Modeling Language (JML). It provides an environment based on Eclipse that provides Runtime Assertion Checking, Estended Static Checking, and Full Static Program Verification for Java. Using JML to specify and verify Java components allows the environment to be used for contract-based component development.

Jahob [21] is a specification and verification system built for a subset of Java. Its specification language is JML-like, and VCs are compiled down to formats acceptable to a diverse set of automated provers. Jahob has a focus on integrating its proof

checker and allowing the programmer to add in-line hints to the various prover backends.

KeY [22], is an environment geared toward producing high-quality object-oriented programs. It uses a subset of UML notation to provide formal specifications and generates proof obligations in JavaDL, a subset of Java used for embedded devices. These obligations can be verified automatically through a built-in theorem prover, or exported to an external prover, such as Z3.

The Ohio State University verification system [23] also uses RESOLVE for component verification. The OSU system also provides a web-based user interface with access to a library of components; verification conditions can be generated, but only for the implementations found in the library. This system also supports automated proving of VCs using Isabelle [2] or SplitDecision, an OSU developed proving tool that uses simplification techniques to prove or disprove VCs.

The system proposed here is unique in its web integration, emphasis on component relationships, modular reasoning, and its features for studying interactions among specifications, implementations, and verification.

8 Conclusions

The web-integrated environment is the result of efforts to provide a user-friendly, installation-free environment focused on component-based software development and modular verification. It provides a start to finish solution for constructing and composing verified component-based systems. It allows users to reuse or create new specifications and implementations for use in larger systems. It makes it possible to reason about the correctness of an implementation independently. The ability to generate Java executable programs from RESOLVE facilities and components provides an outlet for demonstrating the functionality of the environment as a whole. The environment has been used by students in software engineering courses spanning several different institutions for both team-based software development and modular reasoning exercises as a tool reinforcing principles introduced in course lectures. Positive feedback from users has shown that it is a valuable addition to the arsenal of teaching and research tools available for component-based software verification.

9 Acknowledgements

Funding for this research was provided in part through NSF grants CCF-0811748, DUE-1022941 and DUE-0633506.

10 References

1. Sitaraman M., et al. “Building a Pushbutton RESOLVE Verifier: Progress and Challenges”. In *Formal Aspects of Computing*, Springer. 2011. P. 607-626.

2. Nipkow T., et al. “Isabelle/HOL—A Proof Assistant for Higher-Order Logic”. LNCS 2283. Springer. 2002.
3. Smith H., et al. “Generating Verified Java Components through RESOLVE”. In: *Proc. ICSR*, LNCS 5791. Springer. 2009. p 11-20.
4. Weide B W., et al. “Incremental Benchmarks for Software Verification Tools and Techniques”. In *Proc. of the 2nd international Conference on Verified Software: Theories, Tools, Experiments*. LNCS 5295. Springer. 2008.
5. Tagore A. and Weide B. W. “To Expand or Not to Expand: Automatically Verifying Software Specified With Complex Mathematical Definitions”. The Ohio State University. 2011.
6. Sitaraman M. and Weide B W. “Component-Based Software Using RESOLVE”. In *Software Engineering Notes*. 19(4). 1994. P 21-22.
7. Kulczycki G., et al. “Formal Specification”. In *Wiley Encyclopedia of Computer Science and Engineering*. Wiley. 2009.
8. Muller P. et al. “The 1st Verified Software Competition: Experience Report”. In *Proc. 17th International Symposium on Formal Methods*. LNCS 6664. Springer. 2011.
9. Kirschenbaum J., et al. “A Case Study in Automated Verification” .In *Proc. AFM’08: Third Workshop on Automated Formal Methods*. ACM. 2008. P. 53-58.
10. Harton H K., et al. “Formal Program Verification”. In *Wiley Encyclopedia of Computer Science and Engineering*. Wiley. 2009.
11. Bertot. Y. and Casteran P. “Interactive Theorem Proving and Program Development”. Springer. 2004.
12. Barnet M., et al. “The Spec# Programming System: An Overview”. In *Proc. CASSIS 2004*. LNCS 3362. Springer. 2004. P. 49-69.
13. Cook C. “A Web-Integrated Environment for Component-Based Software Reasoning”. MS Thesis. Clemson University. 2011.
14. Leino K R M. “Dafny: an automatic program verifier for functional correctness”. In *Proc. Of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Springer-Verlag. 2010. p. 348-370.
15. Leino K R M. and Monahan R. “Dafny meets the Verification Benchmarks Challenge”. In *Proc. of the Third international conference on Verified software: theories, tools, experiments*. Springer-Verlag. 2010. P. 112-126.
16. Kirschenbaum J., et al. “Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping?”. In *Proc. ICSR*. LNCS 5791. Springer. 2009. p 31-40.
17. Smith H., et al. “Integrating Math Units and Proof Checking for Specification and Verification”. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*.
18. Barnet M., e. al. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In *Formal Methods for Components and Objects*. LNCS 4709. 2006. P. 364-387.
19. Jacobs B., et al. “A Quick Tour of the VeriFast Program Verifier”. In *Proc. APLAS 2010, too paper track*. LNCS 6461. 2010. P. 304-311.
20. Chalin P., et al. “Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML”. In *Formal Aspects of Computing*, Springer. 2011.
21. Kuncak V. and Rinard M. “An Overview of the Jahob Analysis System – Project Goals and Current Status”. In *NSF Next Generation Software Workshop*. 2006.
22. Ahrendt W., et al. “The KeY tool: Integrating object oriented design and formal verification”. LNCS 4334 Springer-Verlag. 2007.
23. Kirschenbaum J., et al. “Automatic Full Functional Verification of Clients of User-Defined Abstract Data Types”. The Ohio State University. 2010.