Formal Reasoning Using an Iterative Approach with an Integrated Web IDE

Nabil M. Kabbani, Daniel Welch, Caleb Priester, Stephen Schaub,
Blair Durkee, Yu-Shan Sun, and Murali Sitaraman

Clemson University,
Clemson SC 29631, USA

{nkabban,dtwelch,cpriest,sschaub,bdurkee,yushans,msitara}@clemson.edu

This paper summarizes our experience in communicating the elements of reasoning about correctness, and the central role of formal specifications in reasoning about modular, component-based software using a language and an integrated Web IDE designed for the purpose. Our experience in using such an IDE, supported by a 'push-button' verifying compiler in a classroom setting, reveals the highly iterative process learners use to arrive at suitably specified, automatically provable code. We explain how the IDE facilitates reasoning at each step of this process by providing human readable verification conditions (VCs) and feedback from an integrated prover that clearly indicates unprovable VCs to help identify obstacles to completing proofs. The paper discusses the IDE's usage in verified software development using several examples drawn from actual classroom lectures and student assignments to illustrate principles of design-by-contract and the iterative process of creating and subsequently refining assertions, such as loop invariants in object-based code.

1 Introduction

An IDE equipped with a formal verification system at its back end can facilitate an alternative style of developing software that involves using feedback from the verifier to locate and correct errors statically, instead of a more classical testing and debugging approach. This paper illustrates how such an approach can work in practice based on our experience in employing it to teach a software engineering course, where students are asked to develop software components that are provably correct with respect to a set of given specifications.

The discussion in this paper is in the context of teaching analytical reasoning to undergraduate CS students. The overall details of our educational approach for teaching mathematical reasoning, including an evaluation of student learning over multiple years in two required courses at Clemson, may be found in [11, 12]. Details of the types of software component development and reasoning assignments given to students are the topic of [7]. The purpose of the present paper is to explain the iterative approach that students and software engineers, in general, can employ for developing verifiably correct software using the feedback from the Web IDE and its integrated prover.

The IDE discussed in this paper is web-integrated, easy to use, and freely available online. It has been used at multiple institutions over the span of several years for teaching [9] and research [36] purposes, and is designed for RESOLVE, an integrated specification and programming language supported by a push-button verifying compiler [31]. The characteristics that distinguish the RESOLVE language and approach from most others include the following [21]:

- Mathematical theories used in specifying programming concepts are extensible and they are described in reusable mathematical units; The theories are purely mathematical and do not involve any computational considerations. They are carefully engineered to ease automated proving. Number theory and a theory of strings over arbitrary types (used in this paper) are some examples.

- Specifications of programming concepts that encapsulate abstract data types are kept strictly separate from implementations to facilitate design-by-contract [26] and to allow for multiple ways of realizing the same concept and permit efficiency trade-offs. Examples of such concepts include programming integers with computational bounds, arrays, stacks, queues, and lists.
- The notion of clean semantics [21] makes it inessential to introduce and reason about object references explicitly in typical user code.

While the literature includes several integrated development environments based on formal techniques related to ours (see section 6 for a complete description), the one closest in spirit to the IDE discussed in this paper is the online Dafny IDE [24]. For most of the exercises discussed in this paper, Dafny could be used as well. However, the key difference that manifests itself the most for the purposes of this paper is our system's usage of a VC generation system [15] that underlies the integrated Web IDE. Using the VCs and a supporting prover capable of revealing which VCs fail to prove, it is possible to determine why a proof was unsuccessful from givens in the context. However, unlike the Dafny approach, which is backed by Z3 [23], the IDE presented here cannot be used to provide counter examples when verification fails. The integrated prover does not use the proof-by-refutation technique, thus requiring a different sort of debugging to take its place. For example, a user contrasts what goal needs to be proved from the givens, tries to understand which givens would be more useful in attempting to prove the goal, and then adjusts the code or assertions as needed.

The reasoning process using the RESOLVE Web IDE is quite similar to what might be employed by one using a Coq-style proof assistant, except that the proofs to be done are mostly 'obvious' due to the simple nature of VCs arising from well-designed software [19]. This characteristic has allowed us thus far to forego the need for manual proof assistance for VCs.[1]

To illustrate how the IDE helps produce correct code based on realtime feedback, we begin with a simple example that involves only the use of the `Integer` datatype. This is followed by two object-based erroneous code examples: one that is recursive, and another that is iterative. These are examples presented to students as part of a software engineering course at Clemson. In all cases, we follow an iterative approach that eventually leads to the correct code or adequate annotations. The discussion concludes with a non-trivial queue copy example code with invariants that students were asked to develop for an assignment using the iterative approach. We note that the examples discussed in this paper are meant to give an idea of the iterative process. Several more complex components are available at the Web IDE; even more can be created by logging in to the site. We conclude the paper with a discussion of the most related work and a summary.

2 Understanding Design by Contract Using the IDE

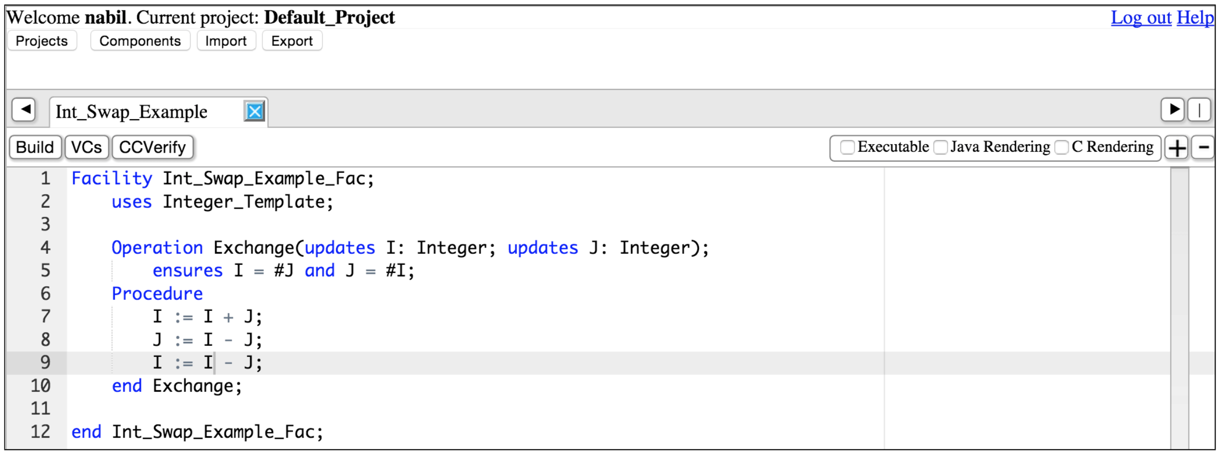
In this and following sections, we provide several illustrative examples, each building in complexity, that demonstrate the iterative process we envision when using the Web IDE to develop provably correct code. All examples discussed have a shared emphasis on the debugging aspect: that is, each requires

[1]A proof assistant such as Coq or Isabelle [29] is indeed necessary for proving the results in reusable mathematical units employed by the automated prover, but the focus of this paper is on code correctness and VCs, assuming that the necessary theorems have been established previously [20].

sufficient knowledge of design by contract to correctly identify and amend a variety of errors in code or annotations based on interactive feedback from the prover in the form of VCs.

The first, relatively simple example presents an operation that arithmetically swaps the values of two `Integer` objects. Taking advantage of the conceptual simplicity of the code comprising this initial example, we also use this as an opportunity to familiarize readers with RESOLVE style specifications, syntax, and layout of the Web IDE. More details on the design of the RESOLVE language and its IDE may be found elsewhere [9, 31, 32].

Upon opening our first example, `Int_Swap_Example_Fac` (Fig. 1), students are presented with code for a single operation, `Exchange`, that takes two `Integer` objects, denoted `I` and `J`. The essence of the specifications that formally communicate what exactly `Exchange` does can be found in the `ensures` clause (the postcondition), where we formally assert that `I = #J` $\wedge$ `J = #I`. This assertion, when stated informally, can simply be read as "the outgoing value of `I` is equal to the incoming value of `J` and the outgoing value of `J` is equal to the incoming value of `I`."[2] Notice also that there is no return value for the `Exchange` operation. Instead, we prefix each parameter with mode `updates` to indicate to clients that each of the `Integer` values passed will contain a purposeful value as specified by the conclusion of the operation.



```
Facility Int_Swap_Example_Fac;
    uses Integer_Template;

    Operation Exchange(updates I: Integer; updates J: Integer);
        ensures I = #J and J = #I;
    Procedure
        I := I + J;
        J := I - J;
        I := I - J;
    end Exchange;

end Int_Swap_Example_Fac;
```

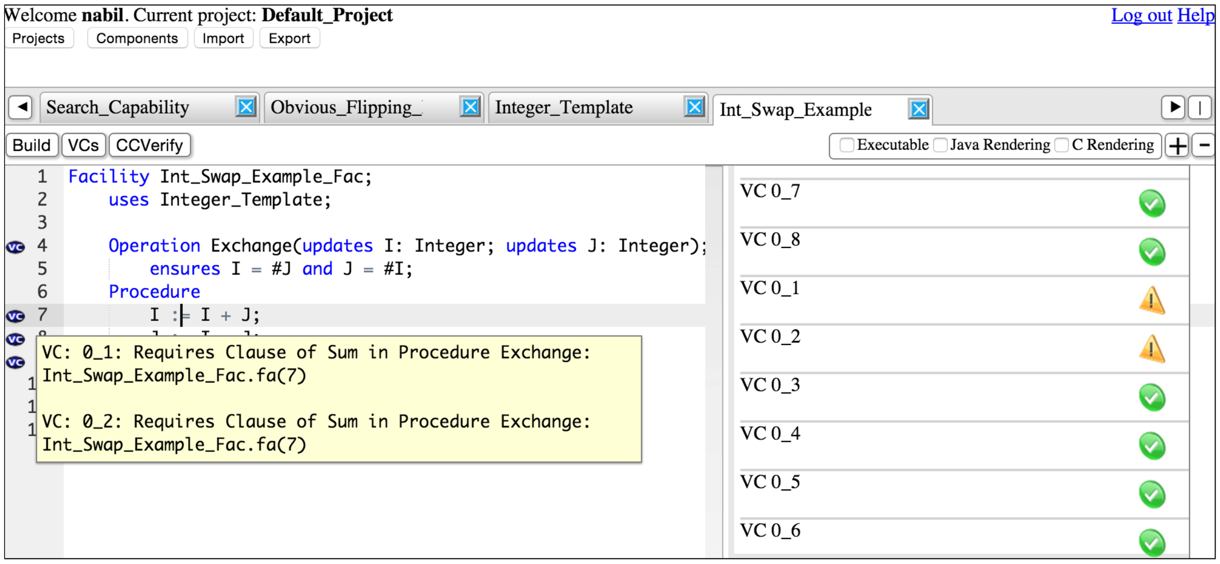
Figure 1: `Exchange` operation with missing requires clause.

Software developers are free to edit both the specifications (formal contracts) of `Exchange`, as well as its executable body (sandwiched between the `Procedure` and `end` keywords). When ready to verify the operation, students may invoke an integrated prover. The exact prover used is of less importance to the discussion in this paper. It's worth noting here, however, that our system is supportive of three approaches: one based on term-rewriting (accessible via the *RWVerify* button) [34], another that is currently under development and uses a congruence closure algorithm in conjunction with a matcher for quantified expressions (accessible via *CCVerify* button), and (optionally) an external SMT solver.[3] The second one that is designed to be just sufficient to prove VCs arising from program correctness (as opposed to arbitrary mathematical assertions) is summarized in section 5, and that is the one used for the examples in this paper.

Upon attempting to verify the `Exchange` operation, students are presented with a screen summarizing proof results, as shown in Fig. 2. The system generates eight distinct VCs [15]. VCs are mathematical

[2]In RESOLVE ensures clauses, # denotes the *incoming* value of a formal parameter.
[3]Z3 [27] is currently being incorporated as a proving option.

assertions that are both necessary and sufficient for the code to be proven correct. To understand why there are eight VCs, we briefly describe the design-by-contract idea in this setting. Two VCs arise from the two conjuncts of the `ensures` clause of the `Exchange` operation, guarantees to be provided by the implementer of the code. Six VCs, two each for the `requires` clauses (preconditions) of each of the three statements in the code, namely that the sum or differences do not go outside computational integer bounds (i.e., `min_int` and `max_int`), for a total of eight VCs. This is because preconditions of called operations are the responsibility of the calling code in design-by-contract. Placing the cursor near the line number of a statement causes a box to appear referring to the names of one or more VCs generated if the statement produces VCs.



```
1  Facility Int_Swap_Example_Fac;
2      uses Integer_Template;
3
4      Operation Exchange(updates I: Integer; updates J: Integer);
5          ensures I = #J and J = #I;
6      Procedure
7          I := I + J;
```

```
VC: 0_1: Requires Clause of Sum in Procedure Exchange:
Int_Swap_Example_Fac.fa(7)

VC: 0_2: Requires Clause of Sum in Procedure Exchange:
Int_Swap_Example_Fac.fa(7)
```

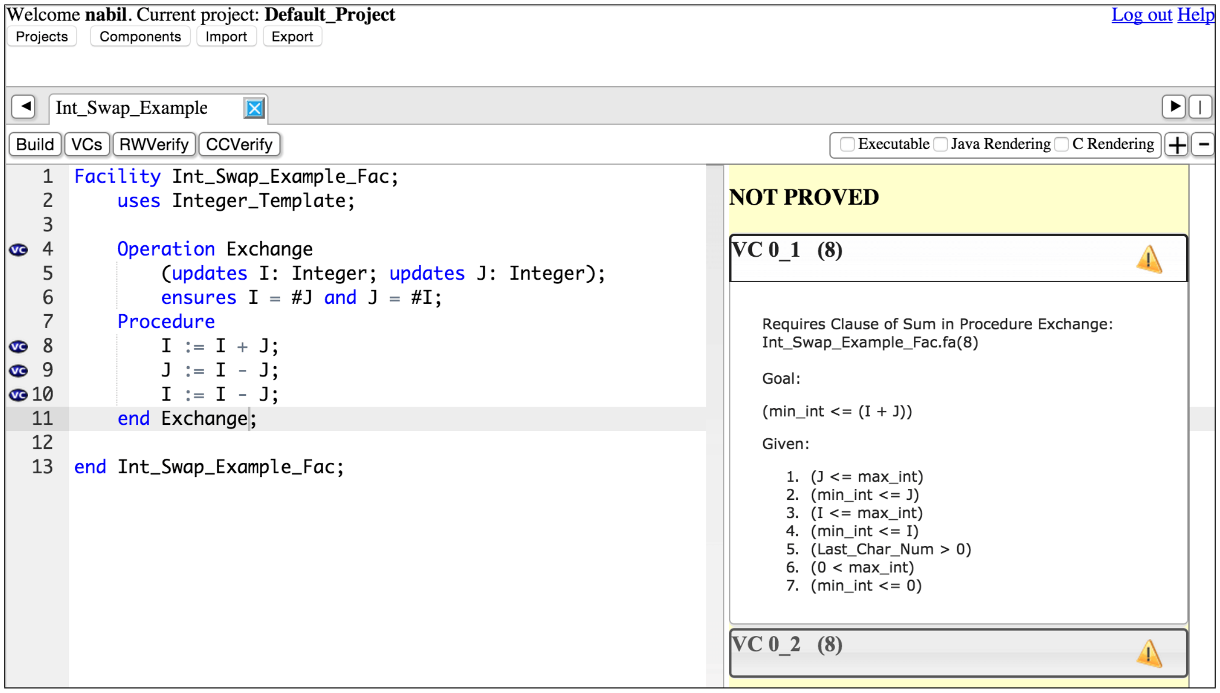
Figure 2: Proof attempt of `Exchange` operation with missing requires clause.

Of the eight VCs, two are unable to be proven, as indicated by the yellow exclamation marks beside VC_01 and VC_02 (Fig. 3). The line numbers in code corresponding to the VC are given in parentheses. While VCs in general might arise from any number of sources within a block of executable code, those unable to be established here arise from the requires clause of the sum operation that is implicitly invoked when `I` and `J` are added via the + operator. We leave it to a reader to convince themselves that an overflow or an underflow can occur in this code only for the first statement.

To aid students in arriving at the particular insight necessary to debug this code, we encourage them to interactively explore the unprovable VCs by mousing over context sensitive VC buttons appearing next to lines of code that generated VCs. Upon clicking any of these buttons, the panel on the right hand side of the Web IDE updates with relevant, finer grained information about the particular VC queried, including a succinct description of what must be established (the goal) and the various facts (givens) the system currently knows.[4]

In terms of the `Exchange` example, it is easy to observe that the system is unable to infer from the givens that `min_int <= (I + J)` (VC 0_1) and `(I + J) <= max_int` (VC 0_2). It then becomes possible to infer that the system currently lacks knowledge suitable to realize that `Integer` overflow (or underflow) will not occur when the + operation is carried out. To remedy this, and 'provide' the

[4]A parsimonious approach to the generation of givens is under research and several of the unrelated givens are expected to disappear in the next version of the IDE.



```
Facility Int_Swap_Example_Fac;
    uses Integer_Template;

    Operation Exchange
        (updates I: Integer; updates J: Integer);
        ensures I = #J and J = #I;
    Procedure
        I := I + J;
        J := I - J;
        I := I - J;
    end Exchange;

end Int_Swap_Example_Fac;
```

```
Requires Clause of Sum in Procedure Exchange:
Int_Swap_Example_Fac.fa(8)

Goal:

(min_int <= (I + J))

Given:

  1. (J <= max_int)
  2. (min_int <= J)
  3. (I <= max_int)
  4. (min_int <= I)
  5. (Last_Char_Num > 0)
  6. (0 < max_int)
  7. (min_int <= 0)
```

Figure 3: Full display of a VC in the IDE.

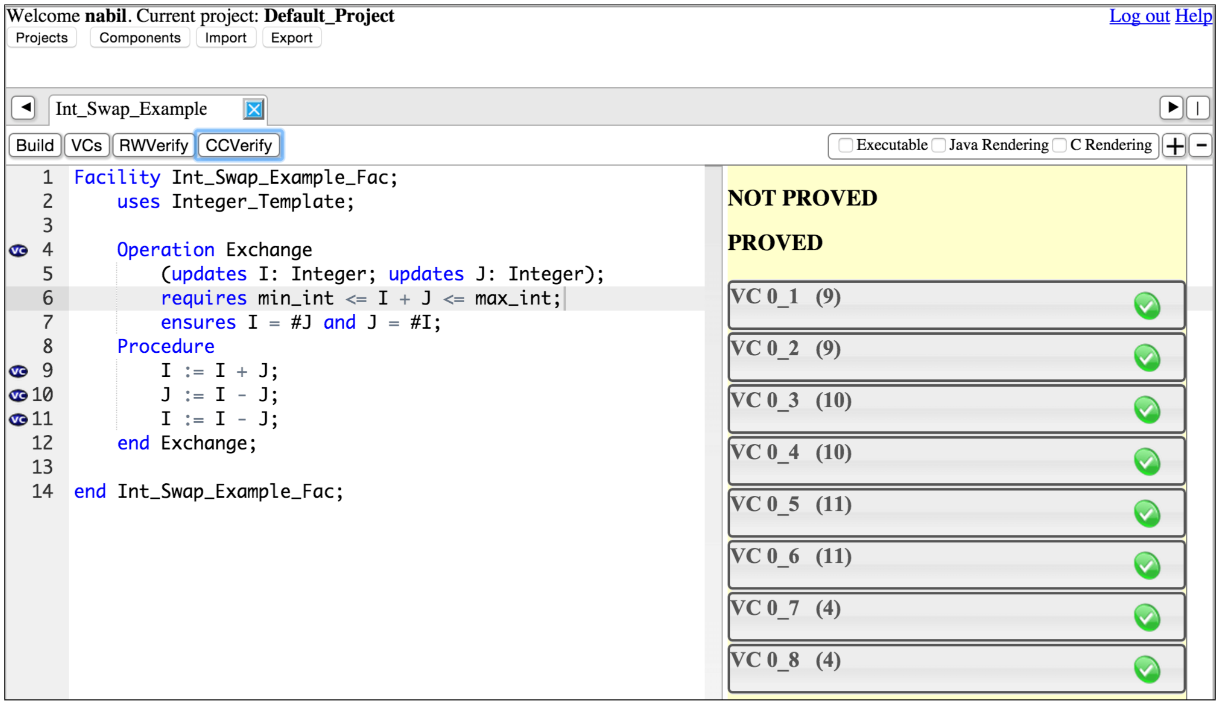
system with the assurance that this will not happen, students must defer to their knowledge of design-by-contract, amending the specification of `Exchange` with a suitable requires clauses as shown in Fig. 4. Again, under design-by-contract, the requirements become givens to be used in proofs. The figure shows that the Web IDE successfully verifies the code using the improved operation specification.

3 Debugging Recursive Code

For the second example, we consider an example operation which inverts the order of items in a queue (see Fig. 5). The `Invert` operation is specified in an enhancement (an extension using specification inheritance) to the `Preemptable_Queue` concept and implemented in an enhancement realization, using only operations provided in the `Preemptable_Queue` concept. This separation of concerns makes it possible to verify the enhancement realization in a modular fashion without referring to or refining to any one implementation of `Preemptable_Queue` concept. A preemptable queue differs from a regular queue in that it has operations to "inject" new items at the front of the queue (i.e., preempt the regular queue order), in addition to regular queue operations, such as `Enqueue` and `Dequeue`.

A complete specification of the `Preemptable_Queue` concept is shown in Appendix A. In the `Preemptable_Queue` concept, the contents of the queue are conceptualized as a mathematical string (a structure similar to but simpler than a sequence in Z, because no positions are involved). So for this operation, the ensures clause (or post-condition) states that the outgoing value of the parameter `Q` should be the mathematical reverse of the input parameter (denoted by `#Q`). Suppose that this operation is implemented using faulty code such as is shown in Fig. 6. Three of the VCs are verified, but VC 0_3 is not. So as we did before, we encourage students to take a close look at that particular unprovable VC.

In the goal, `E'` is the dequeued entry, `Q'` is the conceptual string that stands for the value of the queue



```
Facility Int_Swap_Example_Fac;
    uses Integer_Template;

    Operation Exchange
        (updates I: Integer; updates J: Integer);
        requires min_int <= I + J <= max_int;
        ensures I = #J and J = #I;
    Procedure
        I := I + J;
        J := I - J;
        I := I - J;
    end Exchange;

end Int_Swap_Example_Fac;
```

```
NOT PROVED

PROVED
VC 0_1  (9)   ✓
VC 0_2  (9)   ✓
VC 0_3  (10)  ✓
VC 0_4  (10)  ✓
VC 0_5  (11)  ✓
VC 0_6  (11)  ✓
VC 0_7  (4)   ✓
VC 0_8  (4)   ✓
```

Figure 4: `Int_Swap_Example_Fac` verified.

passed into the recursive call of `Invert`, and `Q` stands for the value of the queue at the beginning of the procedure. The goal is that `E’` concatenated with the reverse of `Q’` is equal to the reverse of `Q`. In order to debug this VC a user may first write down the goal and then apply transformations until we can clearly observe why the goal is unprovable. The purpose here is to show a general process when the problem with the unprovable VC is less obvious.[5]

```
Goal: Q’ = Reverse(Q)
```

Our first transformation will be to use given #1 and apply it to the left-hand side of the goal. We will label this transformation Step 1.

```
Step 1: <E’> o Q’’ = Reverse(Q)
```

Next, we will apply given #2 to transform the left-hand side once again:

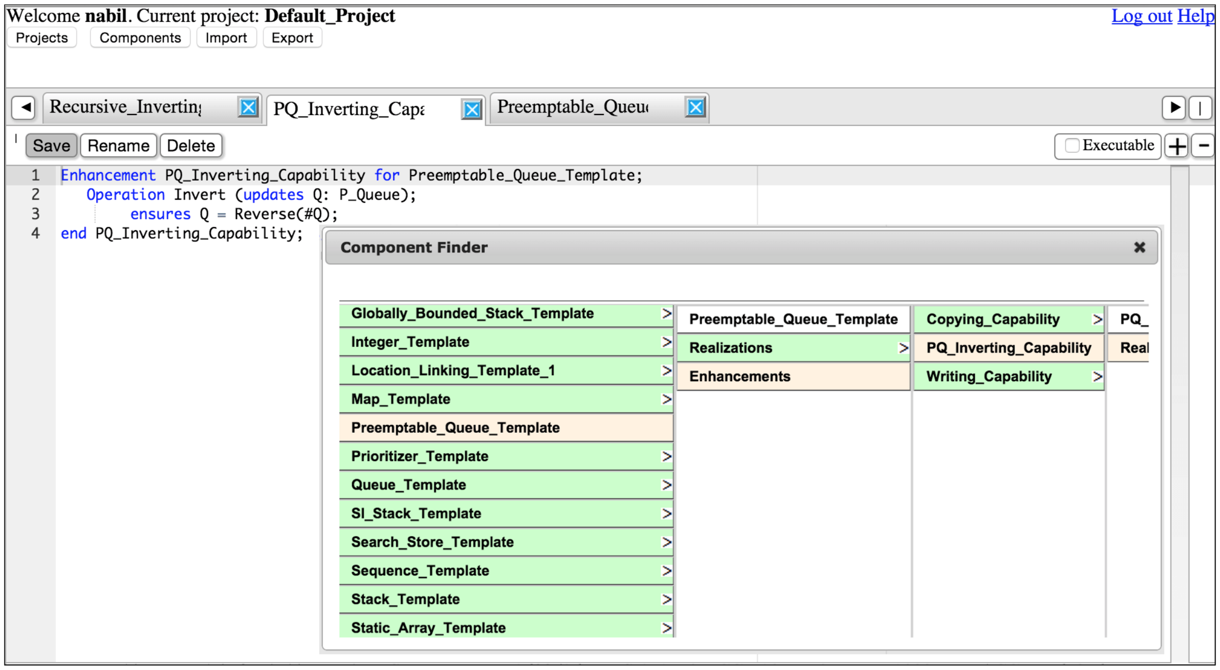
```
Step 2: <E’> o Reverse(Q’’’) = Reverse(Q)
```

And then we apply given #3 to the right-hand side:

```
Step 3: <E’> o Reverse(Q’’’) = Reverse(<E’> o Q’’’)
```

Next, one can attempt to use a theorem from `String_Theory`, which defines string notations and results involving those notations for mathematical strings. The theorem we need here states the following:

[5]While we show the details of these steps here, in actual debugging, such a detailed analysis may not be necessary; understanding of such principles as string concatenation is not commutative is straightforward and the problem may be inferred more readily.



Welcome **nabil**. Current project: **Default_Project**

```
Enhancement PQ_Inverting_Capability for Preemptable_Queue_Template;
    Operation Invert (updates Q: P_Queue);
        ensures Q = Reverse(#Q);
end PQ_Inverting_Capability;
```

Figure 5: Selection of an enhancement in the Web IDE.

```
For all u, v : String, Reverse (u o v) = Reverse(v) o Reverse(u)
```

This transformation will produce the following result:

```
Step 4: <E’> o Reverse(Q’’’) = Reverse(Q’’’) o Reverse(<E’>)
```

Finally, we apply a theorem that states that the reverse of a single-length string is itself, which gives us step 5:

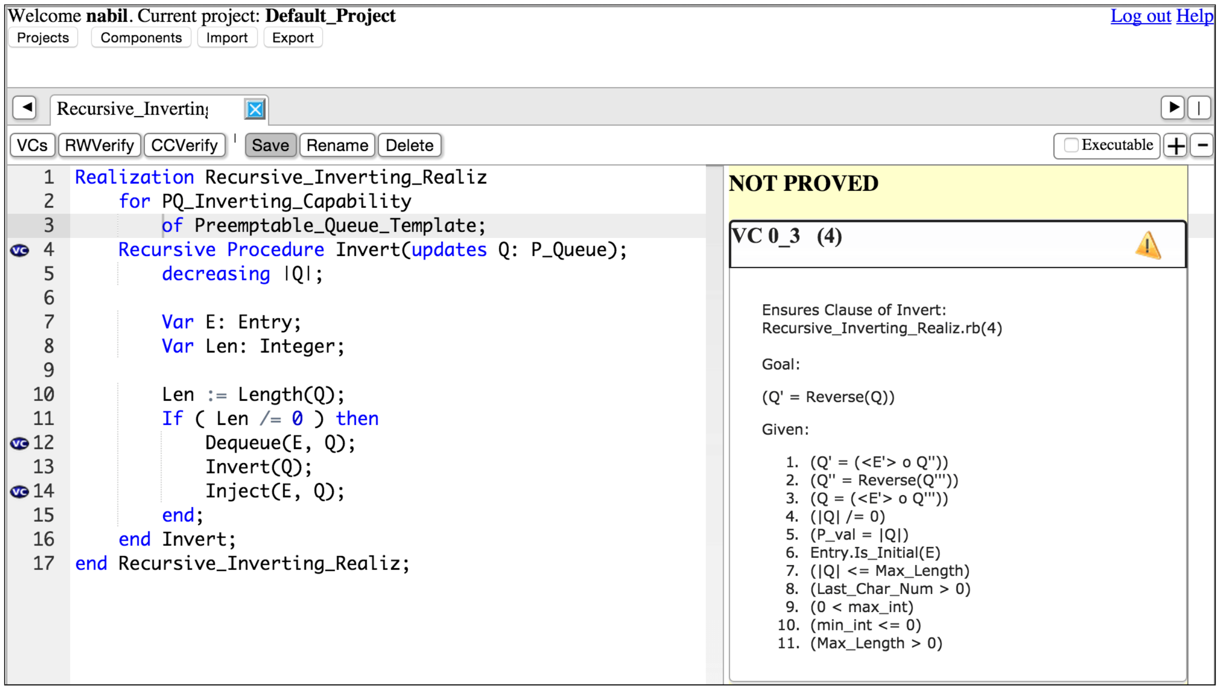
```
Step 4: <E’> o Reverse(Q’’’) = Reverse(Q’’’) o <E’>
```

At this point, it is obvious to see that the goal is categorically false, as the concatenation operator is not commutative. Thus, the problem with the code is that the call to Inject is placing the dequeued entry on the wrong side of the recursively inverted queue. This example illustrates how the VC can serve as a guide to pinpoint the source of the error in formal reasoning.

In the case, the correction is to fix the code: Specifically, the call to `Inject` needs to be replaced with a call to `Enqueue`.

4 Loop Invariants

This section outlines creation and iterative development of loop invariants for code using object-based examples. Stacks and queues are abstract data types represented as objects in RESOLVE. Their behavior is specified in a *Concept*, which is an abstract description of the methods all implementations must contain. It concludes with a discussion of an assignment given to students in a junior level software engineering course.



```
Realization Recursive_Inverting_Realiz
    for PQ_Inverting_Capability
        of Preemptable_Queue_Template;
    Recursive Procedure Invert(updates Q: P_Queue);
        decreasing |Q|;

        Var E: Entry;
        Var Len: Integer;

        Len := Length(Q);
        If ( Len /= 0 ) then
            Dequeue(E, Q);
            Invert(Q);
            Inject(E, Q);
        end;
    end Invert;
end Recursive_Inverting_Realiz;
```

```
NOT PROVED
VC 0_3   (4)

Ensures Clause of Invert:
Recursive_Inverting_Realiz.rb(4)

Goal:

(Q' = Reverse(Q))

Given:

 1. (Q' = (<E'> o Q''))
 2. (Q'' = Reverse(Q'''))
 3. (Q = (<E'> o Q'''))
 4. (|Q| /= 0)
 5. (P_val = |Q|)
 6. Entry.Is_Initial(E)
 7. (|Q| <= Max_Length)
 8. (Last_Char_Num > 0)
 9. (0 < max_int)
10. (min_int <= 0)
11. (Max_Length > 0)
```

Figure 6: Inverting Code with Error.

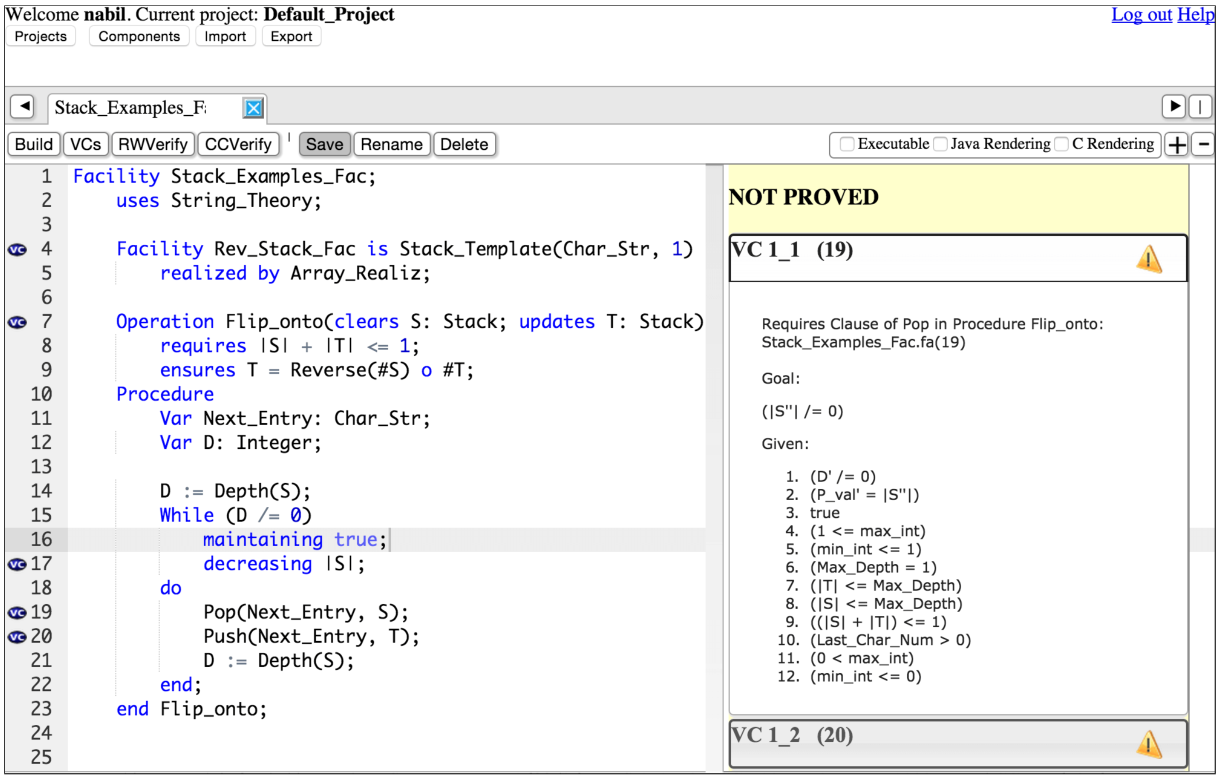
4.1 Learning Iterative Invariant Development

We begin with a simple example involving stacks to highlight the iterative steps we commonly see students working through with our Web IDE in reasoning about, and ultimately arriving at, appropriate assertions for loop invariants. Stacks, like queues, are modeled mathematically using strings; operations such as `Push` and `Pop` are specified using string notations.

Fig. 7 shows an example operation presented in a classroom to teach the idea of invariants. `Flip_onto` iteratively transfers entries from a source stack, `S`, to a destination stack, `T`, resulting in a version of `T` that is prefixed by a 'flipped' version of `S`. As expected, the intuition describing this outcome is formalized in the operation's `ensures` clause by the following succinct assertion: `T = Reverse(#S) o #T`. With the operation's input/output behavior formally expressed, students must turn to the task of deriving a suitable invariant for the while loop, expressed in RESOLVE using the `maintaining` clause. (The `decreasing` clause is used to document the progress metric necessary to prove termination.)

Starting with a `maintaining` clause that simply reads "`true`"—which is an appropriate starting point for beginners to understand the process of developing adequate invariants—the system (unsurprisingly) fails to establish correctness. Aside from the obvious inability to prove VCs corresponding to the operation's overall `ensures` clause, students using the Web IDE are able to see—with the help of the interactive VC buttons next to the line numbers—that VC 1_1 and 1_2 arising from calls to `Pop` and `Push` within the body of the loop are currently unprovable, as shown in Fig. 7.

Examining VC 1_1, students are immediately informed that the `requires` clause of `Pop` (`|S''| /= 0`) cannot be established. Referring to the list of available givens, students can see that while the system is aware that `D' /= 0`, it still lacks any knowledge relating the length of `S` to the current depth, `D`. To address this roadblock and provide the prover with the information it needs to meet the precondition criteria of `Pop`, students might start by amending the maintaining clause with the assertion that `|S| = D`.



```
Facility Stack_Examples_Fac;
    uses String_Theory;

    Facility Rev_Stack_Fac is Stack_Template(Char_Str, 1)
        realized by Array_Realiz;

    Operation Flip_onto(clears S: Stack; updates T: Stack)
        requires |S| + |T| <= 1;
        ensures T = Reverse(#S) o #T;
    Procedure
        Var Next_Entry: Char_Str;
        Var D: Integer;

        D := Depth(S);
        While (D /= 0)
            maintaining true;
            decreasing |S|;
        do
            Pop(Next_Entry, S);
            Push(Next_Entry, T);
            D := Depth(S);
        end;
    end Flip_onto;
```

NOT PROVED

VC 1_1 (19)

Requires Clause of Pop in Procedure Flip_onto:
Stack_Examples_Fac.fa(19)

Goal:

(|S''| /= 0)

Given:

1. (D' /= 0)
2. (P_val' = |S''|)
3. true
4. (1 <= max_int)
5. (min_int <= 1)
6. (Max_Depth = 1)
7. (|T| <= Max_Depth)
8. (|S| <= Max_Depth)
9. ((|S| + |T|) <= 1)
10. (Last_Char_Num > 0)
11. (0 < max_int)
12. (min_int <= 0)

VC 1_2 (20)

Figure 7: Loop with insufficient invariant.

Sure enough, upon re-running the prover, students are given validation in the form of a green checkmark, indicating that one roadblock to verification of the current operation has been successfully dealt with.

In motivating further construction of the `maintaining` clause, students once again look to unproven VCs as a guide to development. In this case, looking specifically to VC 2_2, students can see that the `ensures` clause to the overall operation is still unable to be established. Using this insight, combined with the goal this VC is attempting to establish—specifically, that `T' = (Reverse(S) o T)`—one way for students to proceed is to simply append this assertion to the evolving `maintaining` clause, yielding `|S| = D and T = (Reverse(#S) o #T)`. Upon doing so, students can indeed see that the prover is now able to establish the ensures clause of the operation (indicated by VC 2_3), but is unable to establish the VC corresponding to the invariant of the while statement—suggesting that something is still lacking from the assertion. However, in examining the (now provable) goal of the overall `ensures` clause addressed in the previous step, students can see that it reads as follows: `(Reverse(S) o T) = (Reverse(S) o T)`. Thus, mirroring the same technique and intuition employed to make the ensures clause provable earlier—that is, adding `T = (Reverse(#S) o #T)`—students can now make the necessary cognitive leap to realize the clause must be changed to read: `Reverse(S) o T = Reverse(#S) o #T`, resulting in a final, provable assertion that reads: `D = |S| and Reverse(S) o T = Reverse(#S) o #T`.

4.2 Applying Iterative Invariant Development

Following an introduction to the iterative development of loop invariants and discussion, students used the Web IDE to complete reasoning assignments. The assignments required students to write verified code for pre-specified concepts and enhancement operations. The specification of one such operation to copy a generic `Preemptable_Queue` is given below.

```
Operation Copy_Queue
   (restores Q: P_Queue; replaces Q_Copy:  P_Queue);
   ensures Q_Copy = Q;
```

Table 1 is a summary of student performance for each of the invariant writing assignments. In addition to copying a queue, students wrote code for outputting a queue, reversing a sequence, and an end user application assignment that involved use of custom-made mathematical definitions and operations involving non-trivial types. The definitions in the end user assignment were not complemented by necessary results and hence, the prover was not of use in establishing the invariants. The complexity of the assignment, the mathematics involved, and the absence of prover support are among possible reasons for the low success rate of students in developing appropriate invariants for those operations.

	Writing (Queue)	**Copying (Queue)**	**Reversal (Sequence)**	**Facility Operations**
Correct	70%	90%	60%	30%
Insufficient Invariant	20%	10%	30%	
Other	10%		10%	70%

Table 1: Evaluation of Invariant Assignments

Fig. 8 is an example of code developed by a student for the `Copy_Queue` assignment.[6] Neither the code nor the invariant is necessarily optimal. Proofs of all 18 VCs generated for this copy operation are completed in an average time of 3 seconds (total) on the server that hosts the Web IDE. As noted earlier in the introduction, other provers, such as Z3, could be used to discharge the VCs.

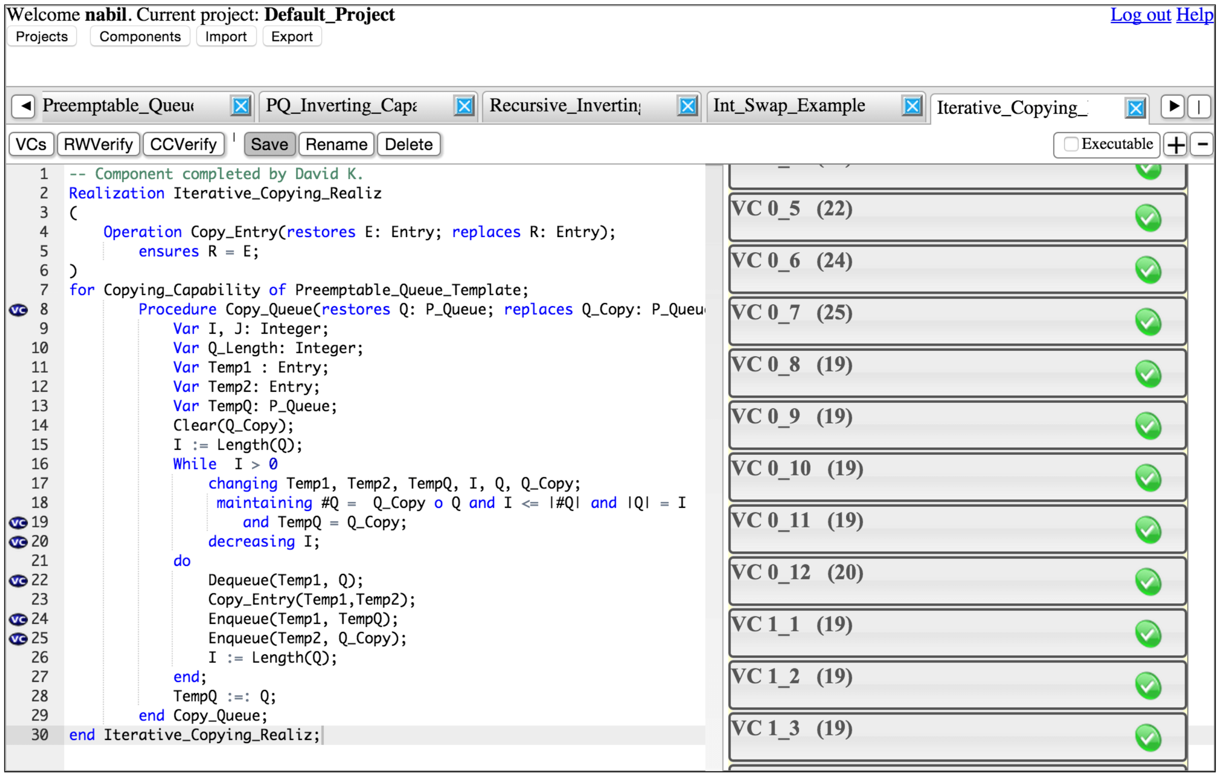
5 Summary of the Prover Underlying the Web IDE

The verifying compiler that serves as the back end of the web interface contains a modular VC generation subsystem [8, 15] which provides input to an automated prover. The automated prover relies on previously proven results in a library of mathematical theories that are reused in the specification of programming concepts [34].

At the core of the *CCVerify* automated VC prover is a congruence closure algorithm that incorporates the Theory of Equality, similar in spirit to that described in [28]. An outer layer that incorporates pattern matching techniques for expressions containing universally quantified variables is engaged, similar to the matcher described in [10]. In this way, a single component can handle problems from multiple domains. *CCVerify*, which contains fewer than 2000 lines of Java code, is designed to be fast, simple, and effective. As it is fully integrated into the compiler, there are no issues with portability, licensing, or translation of the assumptions to other formats as there might be if an external tool were used.

There is an implied division of labor in the production of proofs. Specifications must be written so that the consequent of the VC eventually produced is a predicate with constants as arguments. It

[6] In the figure, `changing` clause is optional and it lists variables potentially affected by the loop; variables not mentioned are assumed to be unchanging. In the absence of this clause, all variables are assumed to be affected. This clause is useful to simplify some routine invariants [15].



```
-- Component completed by David K.
Realization Iterative_Copying_Realiz
(
    Operation Copy_Entry(restores E: Entry; replaces R: Entry);
        ensures R = E;
)
for Copying_Capability of Preemptable_Queue_Template;
        Procedure Copy_Queue(restores Q: P_Queue; replaces Q_Copy: P_Queu
            Var I, J: Integer;
            Var Q_Length: Integer;
            Var Temp1 : Entry;
            Var Temp2: Entry;
            Var TempQ: P_Queue;
            Clear(Q_Copy);
            I := Length(Q);
            While  I > 0
                changing Temp1, Temp2, TempQ, I, Q, Q_Copy;
                 maintaining #Q =  Q_Copy o Q and I <= |#Q| and |Q| = I
                    and TempQ = Q_Copy;
                decreasing I;
            do
                Dequeue(Temp1, Q);
                Copy_Entry(Temp1,Temp2);
                Enqueue(Temp1, TempQ);
                Enqueue(Temp2, Q_Copy);
                I := Length(Q);
            end;
            TempQ :=: Q;
        end Copy_Queue;
end Iterative_Copying_Realiz;
```

```
VC 0_5  (22)  ✓
VC 0_6  (24)  ✓
VC 0_7  (25)  ✓
VC 0_8  (19)  ✓
VC 0_9  (19)  ✓
VC 0_10  (19)  ✓
VC 0_11  (19)  ✓
VC 0_12  (20)  ✓
VC 1_1  (19)  ✓
VC 1_2  (19)  ✓
VC 1_3  (19)  ✓
```

Figure 8: Student code for the Queue copy assignment.

turns out that typically it is sufficient to use only instances of previously proven universally quantified statements (these are members of a reusable theorem library) to construct a proof, assuming that the specifications used to generate the VC make such a proof relatively obvious. Sophisticated techniques used in automated theorem provers may not be required to discharge the VCs. We are currently testing this hypothesis using a limited version of Z3 as well.

Our mathematical specification system is feature–rich. It allows for polymorphic types and first class functions. These features make a *direct* translation of *some* of our mathematical theories to the standard many-sorted first-order logic language [2] used in SMT proving impossible, though it is possible to support these features relatively simple (in a sound, but not complete way) within the matching component of the integrated prover.

6 Related Work

A summary of related specification/verification languages may be found in [16], and tools or IDEs to facilitate their usage are discussed in the first proceedings of this workshop [13]. We discuss only the most related work in this section.

Like RESOLVE that underlies the Web IDE discussed in this paper, Dafny is a programming and specification language intended for verification of functional correctness [24]. The Eiffel integrated language effort [25], though initially focused on runtime assertion checking, is now supported by Eve [14], the Eiffel Verification Environment, which includes the AutoProof [35] tool for static verification. Both Dafny and AutoProof translate code and specifications into Boogie [1], an intermediate verification language. The Boogie tool can create VCs suitable as input to an SMT solver. An important distinction is that the RESOLVE compiler handles VC generation internally, and displays them in a format that makes it easy for users to connect problematic VCs with the code and specifications that produced them.

Java Modeling Language (JML) is a specification/verification language for Java programs, and tools for the language include the ability to do runtime assertion checking [22]. JML does not have an IDE, but there are efforts to integrate JML as plugin to Eclipse [4, 5]. Tools are available for ProB, an animation and a model checker for the B-Method, which is a formal method based on abstract machine notation [3, 37]. The VeriFast effort is aimed at verifying single/multi-threaded C and Java programs [17, 33]. VeriFast also includes a GUI that is packaged into their code distribution.

We are not alone in employing a formal methods IDE in education. Whereas our educational focus is mostly on software engineering aspects (though we have used the Web IDE to a limited extent in a discrete mathematics course), teaching discrete mathematics and specifications using an IDE is the focus of FoCaLiZe—an IDE that takes source code, specification properties, and machine-checkable proofs to produce executable OCaml code and checkable Coq input values [18, 30]. Though our Web IDE does not support inferring loop invariants, invariant inference is a useful feature; an Eclipse plugin with a goal to infer object and loop invariants for C programs is discussed in [6].

7 Conclusions

This paper has detailed an iterative approach for creating, debugging, and developing components that are correct with respect to their specifications, using an IDE equipped with a verification system. Using several illustrative examples drawn from lectures and student assigments, we have explained how students and software engineers, in general, can develop provably correct software iteratively based on the VCs and feedback received from the RESOLVE Web IDE. Extensive experience with the IDE in the classroom indicates that students are indeed capable of producing correct software using the the IDE as discussed in this paper. While the present paper has focused only on functional correctness of code, the IDE includes features to create and view mathematical units and data abstraction realizations with representation invariants and abstraction relations, as well as for generating executable Java code from RESOLVE code [36].

A variety of improvements to the IDE are in progress, ranging from minor visual improvements, such as highlighting VC buttons that correspond to unprovable VCs, to more significant ones, such as the creation and development of performance specifications and related correctness checks.

8 Acknowledgements

The RESOLVE verifying compiler is a multi-decade project involving researchers at several universities, including, but not limited to, Clemson University, Ohio State University, and Denison University. We acknowledge the ideas and support of members of the group in this endeavor. This research has been funded in part by the US NSF grants CCF-0811748, CCF-1161916, and DUE-1022941.

References

[1] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K. Rustan M. Leino (2005): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem P. de Roever, editors: *Formal Methods for Components and Objects, 4th International*

Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures, *Lecture Notes in Computer Science* 4111, Springer, pp. 364–387, doi:10.1007/11804192_17.

[2] Clark Barrett, Aaron Stump & Cesare Tinelli (2010): *The SMT-LIB Standard Version 2.0*. *SMT-LIB.org*. Available at `http://SMT-LIB.org`.

[3] Jens Bendisposto, Sebastian Krings & Michael Leuschel (2014): *Who watches the watchers: Validating the ProB Validation Tool*. In Dubois et al. [13], pp. 16–29, doi:10.4204/EPTCS.149.3.

[4] Patrice Chalin, Perry R. James & George Karabotsos (2008): *JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML*. In Natarajan Shankar & Jim Woodcock, editors: *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, *Lecture Notes in Computer Science* 5295, Springer, pp. 70–83, doi:10.1007/978-3-540-87873-5_9.

[5] David R. Cok (2014): *OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse*. In Dubois et al. [13], pp. 79–92, doi:10.4204/EPTCS.149.8.

[6] David R. Cok & Scott C. Johnson (2014): *SPEEDY: An Eclipse-based IDE for invariant inference*. In Dubois et al. [13], pp. 44–57, doi:10.4204/EPTCS.149.5.

[7] Charles T. Cook, Svetlana Drachova-Strang, Yu-Shan Sun, Murali Sitaraman, Jeffrey C. Carver & Joseph E. Hollingsworth (2013): *Specification and reasoning in SE projects using a Web IDE*. In Tony Cowling, Shawn Bohner & Mark A. Ardis, editors: *26th International Conference on Software Engineering Education and Training, CSEE&T 2013, San Francisco, CA, USA, May 19-21, 2013*, IEEE, pp. 229–238, doi:10.1109/CSEET.2013.6595254.

[8] Charles T. Cook, Heather K. Harton, Hampton Smith & Murali Sitaraman (2012): *Specification engineering and modular verification using a web-integrated verifying compiler*. In Martin Glinz, Gail C. Murphy & Mauro Pezzè, editors: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, IEEE, pp. 1379–1382, doi:10.1109/ICSE.2012.6227243.

[9] Charles T. Cook, Yu-Shan Sun & Murali Sitaraman (2015): *Experience report: evolution of a web-integrated software development and verification environment*. *Softw., Pract. Exper.* 45(6), pp. 857–872, doi:10.1002/spe.2259.

[10] David Detlefs, Greg Nelson & James B. Saxe (2005): *Simplify: a theorem prover for program checking*. *J. ACM* 52(3), pp. 365–473, doi:10.1145/1066100.1066102.

[11] Svetlana Drachova-Strang (2013): *Teaching and Assessment of Mathematical Principles for Software Correctness Using a Reasoning Concept Inventory*. Ph.D. thesis, Clemson University. Available at `http://www.cs.clemson.edu/resolve/research/docs/Dissertation-Drachova-Strang.pdf`.

[12] Svetlana Drachova-Strang, Jason Hallstrom, Murali Sitaraman, Joe Hollingsworth, Joan Krone et al. (2015): *Teaching Mathematical Reasoning Principles for Software Correctness and its Assessment*. *ACM Transactions on Computing Education*, pp. accepted, to appear.

[13] Catherine Dubois, Dimitra Giannakopoulou & Dominique Méry, editors (2014): *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014*. *EPTCS* 149, doi:10.4204/EPTCS.149.

[14] Carlo A. Furia, Julian Tschannen & Bertrand Meyer (2014): *The Gotthard Approach: Designing an Integrated Verification Environment for Eiffel*. In Dubois et al. [13], pp. 1–2, doi:10.4204/EPTCS.149. Abstract of invited talk.

[15] Heather Harton (2011): *Mechanical and Modular Verification Condition Generation For Object-Based Software*. Ph.D. thesis, Clemson University. Available at `http://www.cs.clemson.edu/resolve/research/docs/Dissertation-Harton_2_1.pdf`.

[16] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller & Matthew J. Parkinson (2012): *Behavioral interface specification languages*. *ACM Comput. Surv.* 44(3), p. 16, doi:10.1145/2187671.2187678.

[17] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx & Frank Piessens (2011): *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*. In Mihaela Gheorghiu Bobaru,

Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, *Lecture Notes in Computer Science* 6617, Springer, pp. 41–55, doi:10.1007/978-3-642-20398-5_4.

[18] Mathieu Jaume & Théo Laurent (2014): *Teaching Formal Methods and Discrete Mathematics*. In Dubois et al. [13], pp. 30–43, doi:10.4204/EPTCS.149.4.

[19] Jason Kirschenbaum, Bruce M. Adcock et al. (2009): *Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping?* In Stephen H. Edwards & Gregory Kulczycki, editors: *Formal Foundations of Reuse and Domain Engineering, 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009. Proceedings*, *Lecture Notes in Computer Science* 5791, Springer, pp. 31–40, doi:10.1007/978-3-642-04211-9_4.

[20] Gregory Kulczycki, Murali Sitaraman et al. (2013): *A Language for Building Verified Software Components*. In John M. Favaro & Maurizio Morisio, editors: *Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*, *Lecture Notes in Computer Science* 7925, Springer, pp. 308–314, doi:10.1007/978-3-642-38977-1_23.

[21] Gregory W. Kulczycki (2004): *Direct Reasoning*. Ph.D. thesis, Clemson University. Available at `http://www.nvc.vt.edu/gregwk/assets/research-papers/kulczycki04direct.pdf`.

[22] Gary T. Leavens, Albert L. Baker & Clyde Ruby (2006): *Preliminary design of JML: a behavioral interface specification language for java*. *ACM SIGSOFT Software Engineering Notes* 31(3), pp. 1–38, doi:10.1145/1127878.1127884.

[23] K. Rustan M. Leino (2010): *Dafny: An Automatic Program Verifier for Functional Correctness*. In Edmund M. Clarke & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, *Lecture Notes in Computer Science* 6355, Springer, pp. 348–370, doi:10.1007/978-3-642-17511-4_20.

[24] K. Rustan M. Leino & Valentin Wüstholz (2014): *The Dafny Integrated Development Environment*. In Dubois et al. [13], pp. 3–15, doi:10.4204/EPTCS.149.2.

[25] Bertrand Meyer (1988): *Eiffel: A language and environment for software engineering*. *Journal of Systems and Software* 8(3), pp. 199–246, doi:10.1016/0164-1212(88)90022-2.

[26] Bertrand Meyer (1997): *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.

[27] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[28] Greg Nelson & Derek C. Oppen (1980): *Fast Decision Procedures Based on Congruence Closure*. *J. ACM* 27(2), pp. 356–364, doi:10.1145/322186.322198.

[29] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.

[30] François Pessaux (2014): *FoCaLiZe: Inside an F-IDE*. In Dubois et al. [13], pp. 64–78, doi:10.4204/EPTCS.149.7.

[31] Murali Sitaraman, Bruce M. Adcock et al. (2011): *Building a push-button RESOLVE verifier: Progress and challenges*. *Formal Asp. Comput.* 23(5), pp. 607–626, doi:10.1007/s00165-010-0154-3.

[32] Murali Sitaraman & Bruce Weide (1994): *Component-based Software Using RESOLVE*. *SIGSOFT Software Engineering Notes* 19(4), pp. 21–22, doi:10.1145/190679.199221.

[33] Jan Smans, Bart Jacobs & Frank Piessens (2013): *VeriFast for Java: A Tutorial*. In Dave Clarke, James Noble & Tobias Wrigstad, editors: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, *Lecture Notes in Computer Science* 7850, Springer, pp. 407–442, doi:10.1007/978-3-642-36946-9_14.

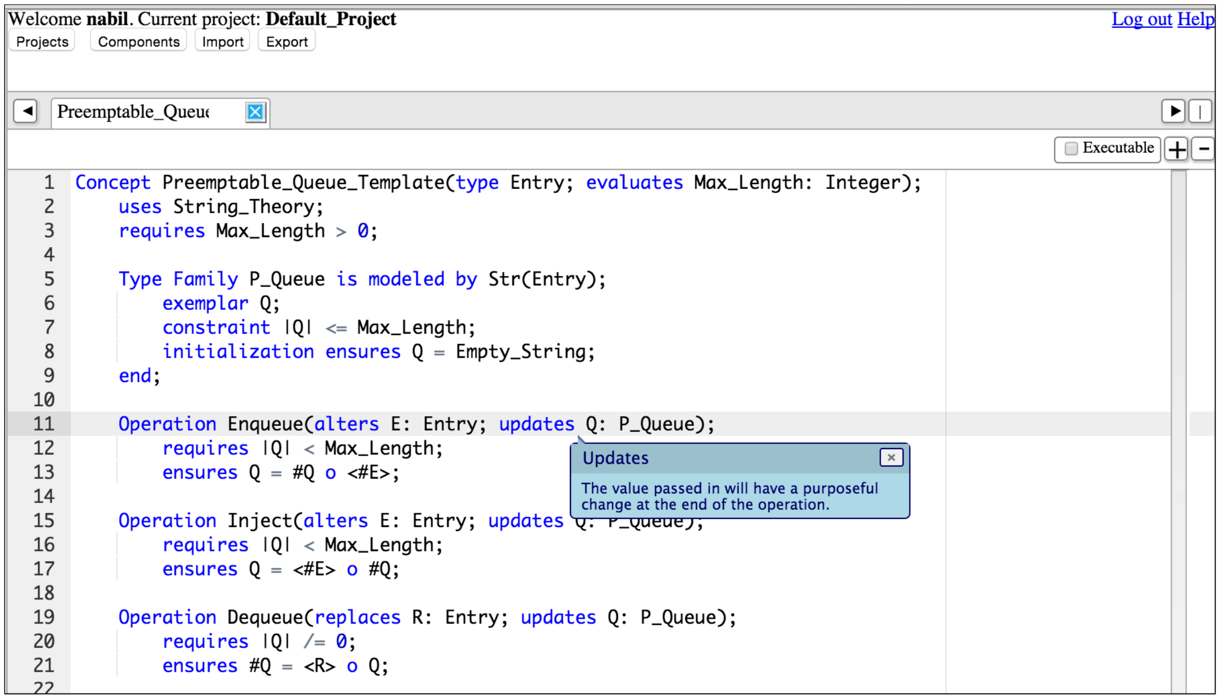
[34] Hampton Smith (2013): *Engineering Specifications and Mathematics for Verified Software*. Ph.D. thesis, Clemson University. Available at `http://www.cs.clemson.edu/resolve/research/docs/Dissertation-Smith.pdf`.

[35] Julian Tschannen, Carlo A. Furia, Martin Nordio & Bertrand Meyer (2011): *Verifying Eiffel Programs with Boogie*. *CoRR* abs/1106.4700. Available at `http://arxiv.org/abs/1106.4700`.

[36] Daniel Welch, Charles T. Cook, Yu-Shan Sun & Murali Sitaraman (2014): *A web-integrated verifying compiler for RESOLVE: a research perspective*. In Dharanipragada Janakiram, Koushik Sen & Vinay Kulkarni, editors: *7th India Software Engineering Conference, Chennai, ISEC '14, Chennai, India - February 19 - 21, 2014*, ACM, pp. 12:1–12:6, doi:10.1145/2590748.2590760.

[37] John Witulski & Michael Leuschel (2014): *Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB*. In Dubois et al. [13], pp. 93–105, doi:10.4204/EPTCS.149.9.

A Preemptable_Queue Specification



```
Concept Preemptable_Queue_Template(type Entry; evaluates Max_Length: Integer);
    uses String_Theory;
    requires Max_Length > 0;

    Type Family P_Queue is modeled by Str(Entry);
        exemplar Q;
        constraint |Q| <= Max_Length;
        initialization ensures Q = Empty_String;
    end;

    Operation Enqueue(alters E: Entry; updates Q: P_Queue);
        requires |Q| < Max_Length;
        ensures Q = #Q o <#E>;

    Operation Inject(alters E: Entry; updates Q: P_Queue);
        requires |Q| < Max_Length;
        ensures Q = <#E> o #Q;

    Operation Dequeue(replaces R: Entry; updates Q: P_Queue);
        requires |Q| /= 0;
        ensures #Q = <R> o Q;
```

Updates

The value passed in will have a purposeful change at the end of the operation.

Figure 9: Partial `Preemptable_Queue` specification.