Specification and Reasoning in SE Projects using a Web IDE

Charles T. Cook
Svetlana V. Drachova-Strang
Yu-Shan Sun
Murali Sitaraman
Clemson University
School of Computing
Clemson, SC 29634, USA
{ctcook, sdracho, yushans, msitara}@clemson.edu

Jeffrey C. Carver
University of Alabama
Computer Science
Tuscaloosa, AL 35487, USA
carver@cs.ua.edu

Joseph Hollingsworth
Indiana University Southeast
Computer Science
New Albany, IN 47150, USA
jholly@ius.edu

Abstract

A key goal of our research is to introduce an approach that involves at the outset using analytical reasoning as a method for developing high quality software. This paper summarizes our experiences in introducing mathematical reasoning and formal specification-based development using a web-integrated environment in an undergraduate software engineering course at two institutions at different levels, with the goal that they will serve as models for other educators. At Alabama, the reasoning topics are introduced over a two-week period and are followed by a project. At Clemson, the topics are covered in more depth over a five-week period and are followed by specification-based software development and reasoning assignments. The courses and project assignments have been offered for multiple semesters. Evaluation of student performance indicates that the learning goals were met.

1. Introduction

There is continued interest in teaching students the connections between mathematics and computing, and helping them understand the benefits of mathematical reasoning for producing high quality software, as evidenced through SIGCSE panel discussions over the last decade from [1] to [2]. Indeed, while several educators, including us, have attempted to inject mathematical reasoning methods in software engineering, a majority of such efforts have been done through "pencil and paper" methods. Significant code-based software engineering projects that rely on using formal contracts for objects are difficult to design, develop, and assign. So even at institutions where mathematical reasoning methods are taught in software engineering courses, students leave without any "hands on" experience in applying the principles they learn directly to executable software.

This paper summarizes our experience in developing and assigning reasoning projects at two institutions at two different levels of depth using the RESOLVE Web IDE (henceforth, referred to as just the IDE in this paper) [3][4][5]. The IDE supports contract-based software development and is especially tailored for teaching reasoning. At Alabama, the students were exposed to the principles over a two-week period and assigned a set of reasoning exercises using the environment. At Clemson, over a five-week period, students reused existing components as well as developed new ones, according to formal contracts and reasoned about the components they built. Other institutions have attempted similar projects as well, though this paper concentrates only on two institutions. The projects expose a range of possibilities for

978-1-4673-5140-9/13 © 2013 IEEE CSEE&T 2013, San Francisco, CA, USA

educators (not just in software engineering, but in other courses) to introduce "hands on" experimentation with analytical reasoning in their classrooms.

We have repeated these experiments multiple times at Alabama and at Clemson, and have found that at both institutions, students had little difficulty in completing the assignments. Some elements of the Reasoning Concept Inventory (RCI) [6], a set of principles we have identified as necessary for students to reason about software formally, are reinforced through these projects, so we have assessed student exam performance on those topics to study the impact of the project and found the benefits to be positive in terms of student learning.

The rest of this paper is organized as follows. Section 2 discusses the Alabama experience and reasoning assignments with pre-developed components. Section 3 explains the Clemson experience, including new component development and reasoning. Along the way, sections 2 and 3 explain the workings of the integrated web environment. Section 4 presents results from our experimentation and evaluation. Section 5 contains related work and our conclusions.

2. Reasoning project summary

2.1 Summary of reasoning principles

This section summarizes the reasoning principles included in a software engineering course at Alabama. The enrollees in the course are primarily second semester sophomores and first semester juniors. This course is organized as a more theoretical course without a full semester-long team project and generally follows the standard software engineering textbook by Pressman [7] covering topics including: software processes, requirements analysis, modeling, design, formal methods/reasoning, quality assurance, and maintenance. We have used the web environment to support the curriculum module on formal methods/reasoning.

Two course meetings are devoted to theoretical concepts behind formal reasoning (formal methods and formal reasoning) and the IDE has been used to aid in explaining the materials. The students also learn how to interact with the IDE during this demonstration. Following the lectures, a reasoning project is assigned as explained in the next section. It is typically a 1-week or 2-week project.

2.2 Assignment details

This is an individual assignment made up of six tasks designed to exercise the student's reasoning skills and make use of the IDE's verification condition (VC) generation and proving capabilities. The first two tasks focus on Integers, but the later tasks involve objects and their interface specifications.

i. Generate VCs (Verification Conditions of correctness) and attempt to prove a defective Max function in Int_Max_Example_Facility. Explain how to fix the defect.
ii. Generate VCs for the Integer_Template Add_to operation with specification in Adding_Capability and implementation in Recursive_Add_to_Realiz. Prove all VCs.
iii. Develop a reasoning table for a Clear_2 operation for the Stack_Template. Provide assume and confirm assertions for each state.
iv. Generate VCs for task iii and compare the generated VCs with the assertions in your table. Write a proof for each generated VC using the givens.
v. Generate VCs for Queue_Template's Remove_Last operation. Write a proof for each generated VC using the givens.

vi. Consider the previous task. Make each of the following changes to the code, regenerate VCs, and state which VC(s) have become unprovable.
   a. Comment out the first Dequeue operation.
   b. Change the maintaining clause (loop invariant) to #Q = <E> o Q.
   c. Change the decreasing clause (termination progress metric) to |T|.

2.3 Solution approach using the RESOLVE Web IDE

The tasks listed above lead students to reason about the code using analytical methods and the assignment provided in the section 3 (below) will do the same and additionally integrate testing into the process. The students are given all necessary specifications and implementations for these tasks. The central point is for students to learn to reason about programs analytically and understand the connection between proofs and software correctness. For this assignment, students use the IDE in support of the reasoning and proving questions. The IDE can be accessed at http://resolve.cs.clemson.edu. To begin task 1 for example, she uses the Facilities ("main" modules) tab and selects "Int_Max_Example_Facility" shown in Figure 1.

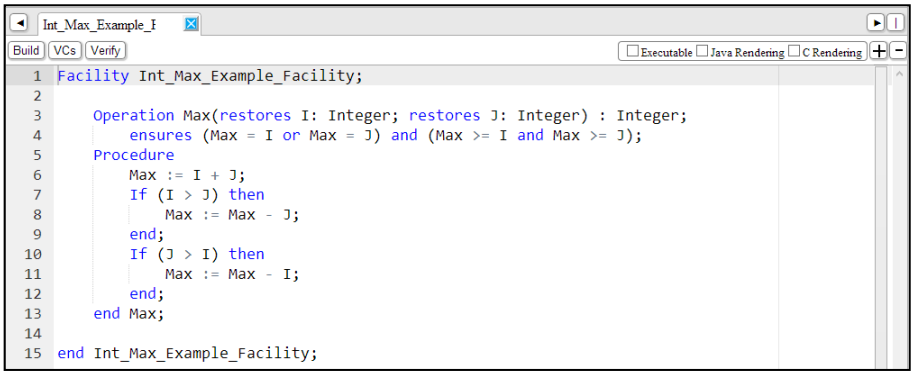

Figure 1 Max function specification and code

The IDE is based on RESOLVE, an integrated specification and programming language combination, and a mathematically sound verification system [5]. The specification notation is quite standard. The programming code is similar to popular languages, except for minor syntactic differences (e.g., the use of := for assignment and = for expressing mathematical equality in formal specifications). So sophomore/junior-level students need to learn basic specification notations but can understand the executable code with relatively little effort.

To see the necessary and sufficient conditions for correctness, a student selects the "VCs" button. The student then attempts to prove the goal for each VC using only the associated givens only to discover that VC 3_2, the VC corresponding to the "If" statements on line 7 and 10 both evaluating to true, cannot be proven (Figure 2). Observing that this equates to the state where I = J, she realizes that not taking this state into account is a flaw in the implementation logic and that adding an "If" statement for this case that sets Max := I is the solution.

Here, the function Max is specified to return the maximum of the two Integers I and J. The given code is erroneous by design. It is faulty for two reasons, one of which is more apparent than the other. The more obvious problem is that it functions incorrectly when I and J are equal. The less obvious problem is that there could be an Integer overflow in the first statement.

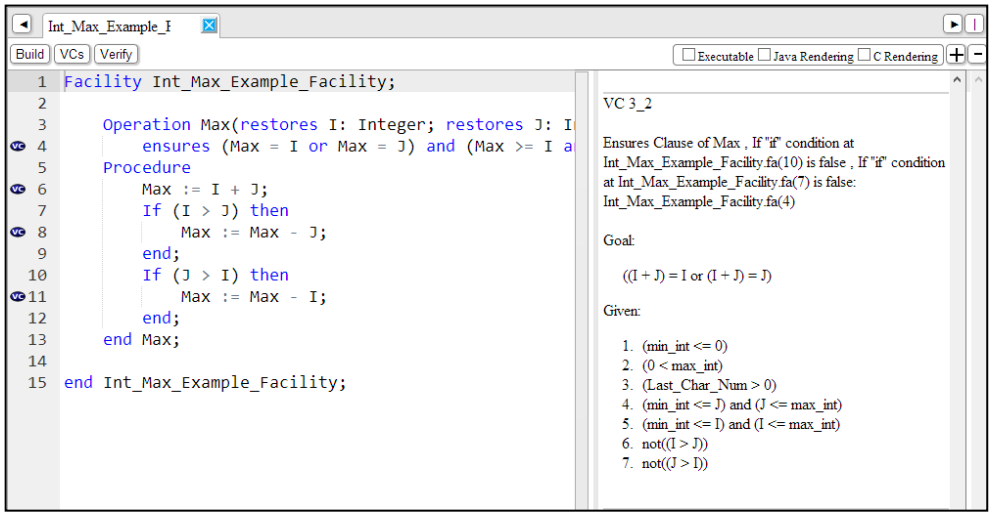

Figure 2 Unprovable VC generated for function Max

Task #6 is more advanced and it is based on a Queue data abstraction (Figure 3).

```
1  Concept Queue_Template(type Entry; evaluates Max_Length: Integer);
2      uses Modified_String_Theory;
3      requires Max_Length > 0;
4
5      Type Family Queue is modeled by Str(Entry);
6      exemplar Q;
7      constraint |Q| <= Max_Length;
8      initialization ensures Q = empty_string;
9
10     Operation Enqueue(alters E: Entry; updates Q: Queue);
11         requires |Q| < Max_Length;
12         ensures Q = #Q o <#E>;
13
14     Operation Dequeue(replaces R: Entry; updates Q: Queue);
15         requires |Q| /= 0;
16         ensures #Q = <R> o Q;
```

Figure 3 Excerpt of Queue_Template specification

Details of the specification language may be found at [5][8][9][10]. A queue is modeled mathematically as a string of entries, and queue operations are specified using string operators, such as concatenation (denoted by "o") and length (denoted by |*|). In the ensures clauses, # notation is used to denote incoming values. A discussion of this Queue abstraction and other abstractions may be found in [8].

The task demands students to make changes to the loop invariant and progress metric for code that implements an extension (or enhancement) operation Remove_Last (the specification appears in Figure 4). The ensures clause of the specification says that the outgoing Queue concatenated with the outgoing entry E is the same as the incoming Queue (#Q). The implementation calls primary queue operations from Queue_Template.

```
1  Enhancement Remove_Last_Capability for Queue_Template;
2
3      Operation Remove_Last(updates Q: Queue; replaces E: Entry);
4          requires |Q| /= 0;
5          ensures #Q = Q o <E>;
6
7  end Remove_Last_Capability;
```

Figure 4 Specification of Remove_Last operation

The change in part (b) leads to an assertion that is not a valid loop invariant and hence, unprovable code. After analyzing the VCs generated by the IDE, a student can see that a VC that was initially provable is now unprovable, as shown in Figure 5.

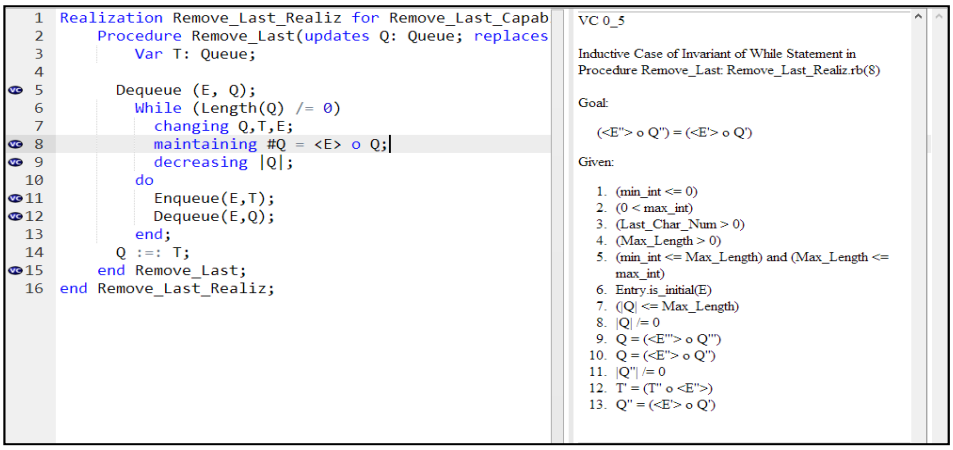

Figure 5 An unprovable VC from modified invariant

After interacting with the IDE to complete the assignment the students at Alabama generated a report to submit to the course instructor. The instructor graded these reports based upon the correctness of the VCs and the proofs.

3. Development and reasoning project summary

3.1 Summary of principles

At Clemson, students learn to use the IDE to develop components according to their formal contracts and reason about their correctness. They are assigned to a 3-person team project where each student is required to develop different pieces that are subsequently integrated. If all components satisfy their contracts, then the integrated code should require little or no testing. During a 5-week period in this course students learn to understand logical specifications, the ideas of external contract (interfaces), internal contracts (object invariants within an implementation), and to reason about software correctness and proofs; details may be found in [8][9][10].

3.2 Team software development overview

This 4-week assignment introduces software engineering students to component design with contracts. While students in a team are certainly encouraged to consult and collaborate, each assignment part consists of tasks to be completed individually, so that team members can work independently to develop components relying solely on provided formal contracts, yet face little trouble at integration time. When complete, the components are combined into a working system with minimal integration costs.

3.2.1. Part 1: The first part involves three tasks to be completed individually:
The specification of a Sequence_Template is given.

i. Implement Realization Seq_Realiz_1 for Sequence_Template.
ii. Implement Realization Seq_Realiz_2 for Sequence_Template.
iii. Implement a test module Seq_Test_Facility to test sequence operations as well as an operation to transfer contents of a queue to a sequence.

Tasks (i) and (ii) require developing realizations for a given concept. The students are given different internal contracts (Figure 6) for the two realizations, forcing each to be implemented differently. The concrete representation for a concept like Sequence is a

record. In Seq_Realiz_1 the record comprises two private fields Contents and Len. An internal contract for an implementation specifies representation invariants (conventions) and an abstraction function or a relation (correspondence) that says how the representation is mapped to the abstract specification. For example, for the two different contracts shown in Figure 6, because of the different conventions the initialization code (i.e., the constructor code) will need to be different; specifically, the Next field cannot be initialized to 0 in the second case.

Task (iii) is to develop test facilities that would fully exercise the implementations from (i) and (ii) by declaring and testing instances of Queue and Sequence templates.

The internal contract for Seq_Realiz_1 is given below.

```
Type Sequence = Record
    Contents: Array 1..Max_Length of Entry;
    Len: Integer;
end;
convention
    0 <= S.Len <= Max_Length;
correspondence
    Conc.S = (Concatenation i: Integer
    where 1 <= i <= S.Len, <S.Contents(i)>);
```

The internal contract for Seq_Realiz_2 is given below.

```
Type Sequence = Record
    Contents: Array 1..Max_Length of Entry;
    Next: Integer;
end;
convention
    1 <= S.Next <= Max_Length + 1;
correspondence
    Conc.S = (Concatenation i: Integer
    where 1 <= i <= S.Next -1, <S.Contents(i)>);
```

Figure 6 Internal contracts for realizations

3.2.2. Part 2: Part 2 of the assignment consists of four tasks, the first three to be done individually and the final one as a group; all these tasks are concerned with a key idea: implementing one reusable component by using another.

i. Implement Sequence_Template with Queue_Template enhanced with a capability for Insert_after and Remove_after
ii. Implement Preemptable_Queue_Template with Sequence_Template
iii. Implement Search_Capability for Preemptable_Queue_Template
iv. Develop a driver for Preemptable_Queue_Template that tests the features above.

3.2.3. Part 3: Next the students are given a reasoning assignment (similar to the Alabama assignment) that builds upon the work done in the team-based software development project. The first task involves a test plan and makes no direct use of the IDE.

i. Generate a test plan for Sequence_Template, giving suitable test points (input/output pairs) based solely on the specification of each operation.
ii. Consider the implementation of Preemptable_Queue_Template in Part 2. Use the IDE to generate and prove the VCs that are provable. When VCs are unprovable, check your code and fix any defects.
iii. Generate VCs for the Transfer operation in Seq_Test_Facility. Doing this requires you to supply loop annotations, an invariant and a progress metric.

3.3 Component diagrams

A UML diagram illustrating a complete solution for Part 1 is shown in Figure 7. In the figure, both interfaces (formal contract specifications) and their implementations are

shown. Developing this diagram helps students understand the dependencies and note that well-designed software has no direct implementation-to-implementation coupling.

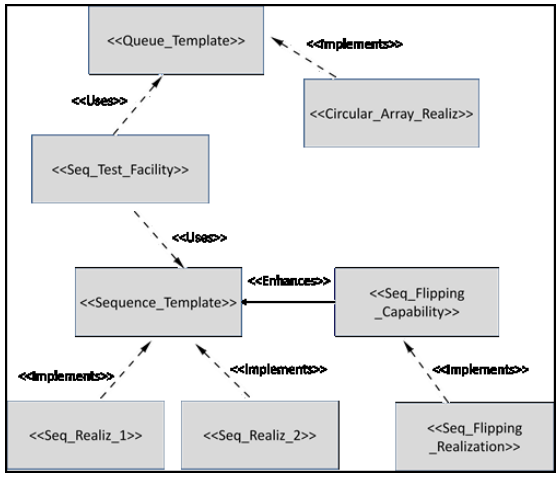

Figure 7 Student solution UML diagram for Part 1

3.4 Team development with RESOLVE Web IDE

The IDE provides open access to a default library of components, such as stacks, queues, sequences, lists, maps, etc. It also includes support for team development by permitting the creation of multiple, unrelated component projects. Also, within a project, instructors can control student access to the components by creating a custom project that allows student access to some components but not to others. This project is only available to registered users of the IDE; thus students are required to register and login to successfully complete the assignment. Figure 8 (below) is a screen shot demonstrating the syntax highlighting and real-time syntax checking.

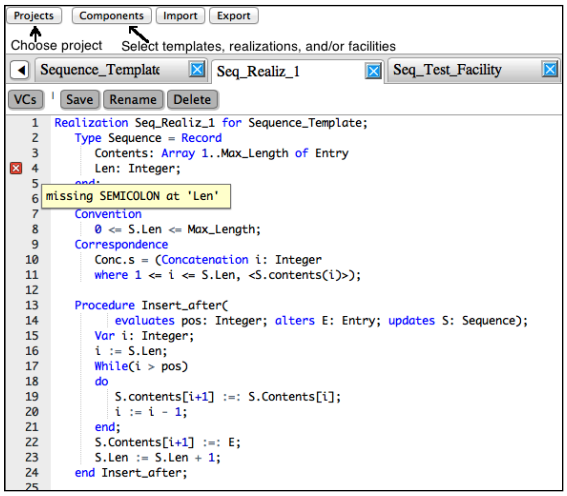

Figure 8 Real-time syntax error highlighting

Team members combine their individual modules into a single submission using the import option which adds each team member's snapshot to the team leader's workspace in the IDE. To run the software, the team selects a driver Facility (i.e., main "module") and clicks the "Build" button to translate the RESOLVE code into an executable JAVA version (JAR) that can be downloaded and executed.

When the team is satisfied with their work, a final snapshot of their combined work is exported and submitted to the instructor electronically. The instructor then imports and evaluates the completed assignment. Through the process of using the IDE for team development and checking independent modules against their contracts, students understand how formal contracts minimize integration costs.

4. Experimentation and evaluation

4.1 Reasoning assignment results

At Alabama, the reasoning materials were taught and the reasoning assignment was given in Spring 2011, Fall 2011, and Spring 2012 semesters. All students who submitted solutions were able to generate VCs and prove the assertions. The table below summarizes the data. While a third of the students did not complete the assignment (the first time the ideas were attempted), in subsequent semesters, that has been less of an issue. Students did as well on the reasoning projects as on others.

	Spring 11	Fall 11	Spring 12
Submission Avg.	83%	87%	80%
# of submission	17	14	35
# of omissions	9	1	8

Figure 9 Reasoning project performance

The reasoning assignment has also been given in software engineering courses at Western Carolina, DePauw and Cleveland State, but no hard data is available.

4.2 Contract-based development and reasoning assignment results

At Clemson, the relevant materials were taught, and contract-based software development and reasoning assignments materials were given in Fall 2010, Spring 2011, Fall 2011, and Spring 2012 semesters. Between 20 and 32 students were registered for these classes. At Clemson, almost no one failed to complete the project. Student project averages were much higher than their course averages. Since the submissions were tested using components from different groups, there is reason to believe that most projects adhered to contracts and would have needed little integration testing.

	Fall 10	Spring 11	Fall 11	Spring 12
Parts 1 and 2	93%	86%	95%	96%
Part 3	89%	92%	98%	87%

Figure 10 Average development and reasoning project scores

4.3 Impact of Project as seen through Reasoning Concept Inventory correlation results

To determine the effectiveness of the IDE as a learning aid, we studied exam performance on RCI items directly affected by the topics covered by the project: precise specification of Pre-Post Conditions (RCI 3.4.3), the notion of Design by Contract (RCI 4.2) and the idea of Internal Contracts and Assertions (RCI 4.3).

	Semesters	Students	Average	% scoring 70% or higher
RCI 3.4.3	3	82	87%	85%
RCI 4.2	2	56	76%	59%
RCI 4.3.1	1	24	77%	67%

Figure 11: Assessment of exam performance on RCI Items influenced by the project

Since the project was the main instrument to reinforce the RCI principles above and the results summarized in the table above are at or higher level than student performance on other topics, an inference can be made that the project had a positive impact on learning of key software engineering principles.

4.4 Results from web usability surveys

To aid in refining the IDE's user interface to increase its effectiveness as a teaching tool, usability surveys have been given over the past several semesters to sixty undergraduate students at Clemson University and University of Alabama, and five graduate students at Cleveland University. Beyond background questions, the survey was made up of five-level Likert items to assess the perceived benefits of the IDE to students. The statement with the highest agreement level was "The web interface has helped me understand the relationship between mathematical reasoning and software development," receiving a score of 3.92 out of 5. The statements "… helped me understand how to build component-based software" and "The presentation of components within web interface has helped me understand the relationships between specifications and implementations" received ratings of 3.86 and 3.80, respectively.

5. Related work and summary

The IDE presented here differs from popular IDE's such as Eclipse or Netbeans in that it supports formal specification and reasoning. It also differs from efforts in integrating formal reasoning in software engineering courses with tools. The tool developed at the Universite Catholique de Louvain help write code given specifications and vice-versa [11], but is not a full-fledged, web-based reasoning environment.

JAIDE (Java Integrated Development Environment) incorporates formal notation, UML diagrams, and Java code and is designed to type check, prove theorems, and analyze source code [12]. It has been used for a limited multi-semester field-test in courses, but it is not a comprehensive reasoning tool and it is not openly available.

The focus of Alloy is on specifications and it has been used in undergraduate SE classes [13]. ProverEditor is a plugin used to integrate the theorem proving capabilities of Coq with the file management and editing capabilities of Eclipse [14]. ProofWeb is a web-based proof assistant that uses a version of Coq modified to produce output encoded as XML. It has been used in logic courses at several European Institutions [15].

Given the importance of reasoning for developing high quality software, there are several ongoing industrial and research verification efforts. One environment that uses a theorem prover as a backend for a reasoning environment is VeriFast. It works with Java and C source code [16]. Jahob is a specification and verification system built for a subset of Java. Its specification language is JML-like and generates verification conditions in a variety of formats, allowing it to be used in conjunction with many third party automated provers [17]. Microsoft has developed a suite of tools for verification and analysis, but their focus is on industrial usage [18]. The Ohio State University has also produced a system that uses RESOLVE for component verification [5]. These efforts have not been tried or evaluated in an undergraduate educational setting.

The work presented in this paper complements the work in test-driven learning in [19], because task 1 of part 3 of the Clemson assignment is intended for similar learning.

In this paper, we have summarized our experiences in teaching mathematical reasoning to undergraduate software engineering students using an integrated web environment, especially designed for education. The level of exposure can be varied, as illustrated by

the approaches taken at two institutions. Based on consistently high student performance over several offerings of the projects at the institutions, we conclude that reasoning can be taught effectively with the environment. Results from usability surveys are also positive. As other instructors and institutions attempt these projects, we are actively engaged in minimizing the effort in developing, teaching, and assigning reasoning projects. To make the environment useable in introductory classes, one current direction is to render code in multiple programming languages.

Acknowledgments

This research has been funded in part by the NSF grants CCF-0811748, CCF-1161916, DUE-1022191, and DUE-1022941.

References

[1] Gries, D., Marion, B., Henderson, P., Schwartz, D., "Panel: How Mathematical Thinking Enhances Computer Science Problem Solving," *Procs. 32nd ACM SIGCSE Conference*, February 2001.

[2] Krone, J., Baldwin, D., Carver, J. C., Hollingsworth, J. E., Kumar, A., and Sitaraman, M., "Panel: Teaching Mathematical Reasoning across the Curriculum," *Procs. 43rd ACM SIGCSE Conference*, March 2012.

[3] Cook, C. T., *A Web-Integrated Environment for Component-Based Reasoning*, M. S. Thesis, Clemson, September 2011.

[4] Cook, C. T., Harton, H. K., Smith, H., and Sitaraman, M., "Specification engineering and modular verification using a web-integrated verifying compiler," *ICSE*, Zurich, 2012, 1379-1382.

[5] Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, B., Frazier, D., Friedman, H.M., Harton, Heym, W., Kirschenbaum, J., Krone, J., Smith, H., and Weide, B.W., "Building a Push-Button RESOLVE Verifier: Progress and Challenges," *Formal Aspects of Computing*, Springer, 2011.

[6] Drachova, Svetlana. "Teaching and Assessment of Mathematical Principles for Software Correctness Using a Reasoning Concept Inventory," Ph. D. Dissertation, Clemson University, 2013.

[7] Pressman, Roger. A Practitioner's Approach 7e. New York, McGraw-Hill, 2010. Print.

[8] Cook, C.T., Drachova, S., Hallstrom, J.O., Hollingsworth, J., Jacobs, J.P., Krone, J., and Sitaraman, M., "A Systematic Approach to Teaching Abstraction and Mathematical Modeling." *ACM SIGCSE ITiCSE*, 2012.

[9] Leonard, D.P., Hallstrom, J.O., and Sitaraman, M., "Injecting Rapid Feedback and Collaborative Reasoning in Teaching Specifications," *ACM SIGCSE*, 2009.

[10] Sitaraman, M., Hallstrom, J.O., White, J., Drachova-Strang, S., Harton, H., Leonard, D., Krone, J., and Pak, R. Engaging students in specification and reasoning: "hands-on" experimentation and evaluation. *ACM SIGCSE ITiCSE*, 2009, 50-54

[11] Dony I. and Le Charlier B. "A Tool For Helping Teach A Programming Method". In *Proc. 11th Annual SIGCSE ITiCSE*, 2006.

[12] S. Skevoulis, M.Falidas. "Integrating Formal Methods Tools Into Undergraduate Computer Science Curriculum." *ACM SIGCSE ITiCSE*, 2002.

[13] Alloy: A language and tool for relational models, accessed at: http://alloy.mit.edu/alloy/

[14] Charles J. and Kiniry J. R., "A Lightweight Theorem Prover Interface for Eclipse". INRIA Sophia Antipolis. 2008.

[15] Hendricks M., et al. "Teaching Logic Using a State-of-the-Art Proof Assistant". *Acta Didacta Napocensia*. 3(2). 2010. 35-48.

[16] Jacobs B., et al. "A Quick Tour of the Verifast Program Verifier". In *Proc. APLAS 2010, tool paper track*. LNCS 6461. 2010. 304-311.

[17] Kunac V. and Rinard M. "An overview of the Jahob Analysis System – Project Goals and Current Status". In *NSF Next Generation Software Workshop*. 2006.

[18] Schulte, W., "Ten Years of Automated Code Analysis at Microsoft," *ICSE*, Zurich, 2012.

[19] Dvornik, T., Janzen, D.S., Clements, J., Dekhtyar, O., "Supporting Introductory Test-Driven Labs with WebIDE". IEEE *CSEET*, 2011.