Engaging Students in Specification and Reasoning: "Hands-On" Experimentation and Evaluation

[†]Murali Sitaraman, [†]Jason O. Hallstrom, [†]Jarred White, [†]Svetlana Drachova-Strang, [†]Heather Harton, [†]Dana Leonard, [‡]Joan Krone, [*]Rich Pak

[†]Clemson University
School of Computing
Clemson, SC 29634-0974
1.864.656.3444

[‡]Denison University
Math. and Computer Science
Granville, OH 43023-0810
1.740.587.6484

[*]Clemson University
Psychology
Clemson, SC 29634-0974
1.864.656.1584

ABSTRACT

We introduce a "hands-on" experimentation approach for teaching mathematical specification and reasoning principles in a software engineering course. The approach is made possible by computer-aided analysis and reasoning tools that help achieve three central software engineering learning outcomes: (*i*) Learning to read specifications by creating test points using only specifications; (*ii*) Learning to use formal specifications in team software development while developing participating components independently; and (*iii*) Learning the connections between software and mathematical analysis by proving verification conditions that establish correctness for software components. Experimentation and evaluation results from two institutions show that our approach has had a positive impact.

Categories and Subject Descriptors

K.3.2. [**Computers and Education**] Computer and Information Science Education; D.2.2. [**Software Engineering**] Design Tools and Techniques; D.2.4. [**Software Engineering**] Software/Program Verification; F.3.1. [**Logics and Meanings of Programs**] Specifying and Verifying and Reasoning about Programs.

General Terms

Design, Documentation, Human Factors, Languages, Verification.

Keywords

assertions, components, mathematical thinking, tools.

1. INTRODUCTION

This paper introduces an approach to teaching mathematical specification and reasoning principles in a software engineering course. The important objective of teaching mathematical principles in undergraduate computing has had several pioneers [1][2][3][4][5][6]. The approach presented in this paper builds on this prior work, yet is distinguished by several factors. Specifically, we have built and employed a collection of relatively independent reasoning tools and have designed "hands-on" experiments using those tools to help students learn, practice, and appreciate the principles introduced. We use topics typically taught in software engineering courses to motivate a rigorous approach to specification and analysis.

While we have spent six weeks of a semester to present and teach the principles listed in this paper at Clemson, the methods and tools are set up in such a way that selected constituents of the approach can be easily adapted elsewhere to fit a two or three-week schedule at other institutions. For example, only the principles of proving verification conditions for code correctness and the corresponding tool were introduced at Denison in their software engineering course. Moreover, while this paper centers on software engineering, our methods and tools are designed to be useful *across* the curriculum in multiple courses at varying levels of rigor. For example, we are using only a subset of the reasoning methods and tools in a sophomore-level software development course where students learn principles of object-oriented computing using Java. In our classrooms, we use an active learning approach to engage students and encourage learning from peers. Our collaborative methods and tools seem effective based on our experimentation and evaluation.

Curriculum materials, software tools, and detailed evaluation results are available at `www.cs.clemson.edu/~resolve`.

Paper Organization: Section 2 gives an overview of the topics covered, salient details of how they are motivated, and summary descriptions of our methods and tools. Section 3 discusses experimentation activities and evaluation results. The last section surveys related work and presents our conclusions.

2. ENGAGING STUDENTS

The syllabus for the software engineering course discussed in this paper is quite typical. We teach requirements analysis, design and specification, component-based implementation, and quality assurance techniques. We also introduce our students to process models, metrics, and project management issues. Students engage in team projects as well.

Once we have covered large-scale software engineering issues in the first half of the semester, we focus on component-based software construction in the second half, and this is where we

ITiCSE'09, July 3–8, 2009, Paris, France.

introduce principles of specification and reasoning. Noting that expensive integration costs can be avoided if the contracts used by team members are unambiguous, we introduce formal specifications of components. The teaching principles discussed in this paper are independent of programming or specification languages and techniques, and have general applicability. But some of the tools we have developed are language specific.

For formal specification, we use RESOLVE [7], a language in which all objects are conceptualized abstractly using mathematical models. Educators may find RESOLVE overviews elsewhere [8][9]. For an example, consider the specification of a `Queue` **concept** in which a queue ADT is modeled mathematically as a string of entries and primary operations to enqueue, dequeue, and find length are included. Once students are exposed to the basics of specifications, we introduce them to more complex examples. Given below, for instance, is the specification of a secondary `Queue` operation (i.e., an operation implementable using only primary operations) that we call an **enhancement**:

```
Rotate (updates Q: Queue, restores n: Integer);
requires |Q| <= n ;
ensures there exist S, T: Str(Entry) such that
        |S| = n and #Q = S o T and Q = T o S ;
```

The **requires** clause states the pre-condition: the length of `Q` must be less than `n`. The **ensures** clause states the post-condition; `#Q` denotes the pre-conditional value of `Q`, and `o` denotes concatenation. In the assertions, all variables denote their mathematical values.

The literature contains several good ways to teach students to understand mathematical specifications such as the ones above. In a contract-oriented software development setting, the motivation is straightforward because only by understanding specifications, can students reuse the components built by others and develop their own components. Before students can use formal specifications, they need to be able to read and understand them. This is the topic of the first sub-module.

2.1 Reading Specifications

One way to check that students understand logical assertions, such as the ones above, is to ask them to develop a variety of test points based only on their understanding of the relevant specification, perhaps even before an implementation has been developed. We teach them to select inputs that satisfy the pre-condition, and to plug those values into the post-condition to determine the corresponding output values. Consider a test point for `Rotate()`, assuming `Max_Length` is at least 5 and type `Entry` is `Integer`:

Valid inputs:	**Q =** <17, 1, 6, 9, 20>	**n =** 3
Expected outputs:	**Q =** <9, 20, 17, 1, 6>	**n =** 3

To check that students are reading the assertions (and not guessing), we often use unhelpful names in our introductory exercises. We have also built and applied a special reasoning tool [10] for experimenting with specification understanding.

2.2 Team Software Development Using Formal Contracts

Teaching modular design is a central goal of the software engineering course. A key enabler of modular design is the use of formal component specifications that allow developers to work in parallel, relying only on the specifications. If the implementation of each component satisfies its specification and does not violate any requirements of the components it reuses, then, in principle, there should be no integration difficulties. This is the focus of the second part of our approach.

In the team project, some concepts are implemented using built-in container structures, such as arrays, while others rely on different components. Students also develop multiple implementations of the same concept. After the project is complete, we have them replace implementations to illustrate that when using a specification-based development approach, component implementations can be switched without affecting the calling code. Multiple implementations also help them to appreciate the performance trade-offs among components.

It is useful to consider the team project in more detail: It is a component-based software development assignment involving several concepts, implementations, and enhancements. All implementation elements are fully-specified. Each team has 3 members and is responsible for developing 3 to 4 (disconnected) component implementations used in a larger system. The goal is for the students to work independently, relying only on contract specifications, and to then integrate their implementations prior to submission. When there are integration errors, it becomes easier to pinpoint the source. To emphasize the critical role of formal specifications, we use a uniform framework to present both specifications *and* implementations.

For "hands-on" experimentation, we use the *RESOLVE Contract Analyzer* (Figure 1) along with a translator to *Java*. The Analyzer applies a series of checks to enforce contract programming principles. For example, in developing the implementation of an enhancement, such as `Rotate()`, if a student uses implementation details specific to a particular `Queue` implementation, a violation will be detected by the tool.

2.3 Formal Reasoning and Proofs

Beyond enabling modular development, formal specifications enable modular analysis of programs – in particular, modular *verification*. In the software engineering courses at Clemson and Denison, we use a verification condition (VC) generator based on RESOLVE. In Figure 2, the VC generator is shown integrated as an Eclipse plug-in. The VC generator is part of a larger verification effort. Formal proof rules underlying the generator may be found in [8] and a collection of benchmarks are discussed in [11].

One of the verification conditions generated by the tool corresponds to the post-condition of the operation being verified. Equally important, the tool generates conditions to ensure that the pre-conditions of reused operations are respected. Given loop invariants and progress metrics for termination, the VC generator also generates assertions to check the correctness of those invariants. So it can be used to teach students which invariants are appropriate and which are not.

A given implementation is correct with respect to its specification only if all the corresponding verification conditions can be proved. One of the collaborative activities in the class is for a group of students to work together to prove the verification conditions for an example component. When one of the

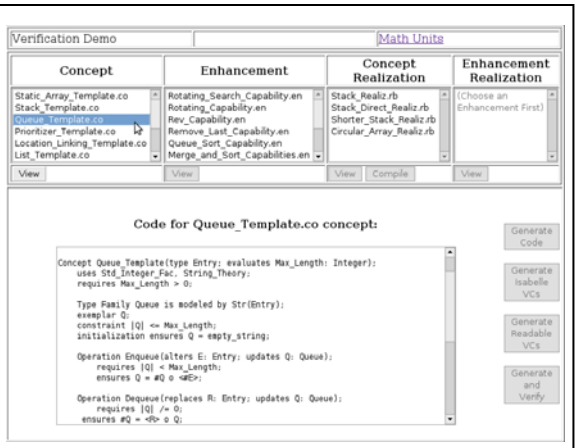

Figure 1. Team Development with Formal Contracts

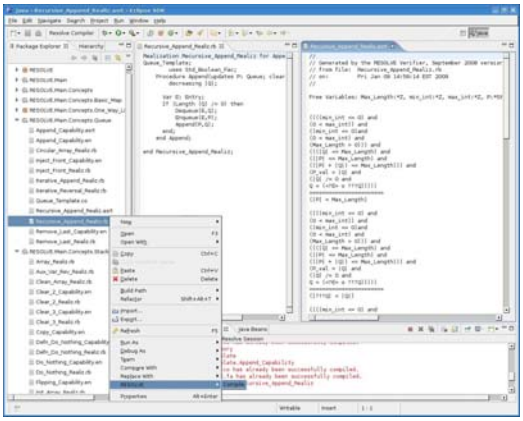

Figure 2. Verification Condition Generation

conditions cannot be proved, an interesting exercise is to trace the problem to the point of error in the code. (The generator is currently being augmented to perform this task automatically, but the manual exercise is useful in illustrating the connections between code and generated assertions.) In this process, students not only apply proof techniques learned in their discrete mathematics class, but also see the important connections between mathematical thinking and high quality software development.

Given below is one of the three conditions generated for a code fragment that clears a 2-element stack using two calls to `Pop()`. (The other conditions arise to be sure that the calls are not placed on an empty stack.) In this assertion, `S`, `?S`, and `??S` are strings of entries corresponding to various values of a stack `S` in different states. The `Next_Entry` variables are defined analogously.

```
((|S| <= Max_Depth) and (|S| = 2) and
(S = (<??Next_Entry> o ??S)) and
(??S = (<?Next_Entry> o ?S)))
➔
     (?S = empty_string)
```

Several experiments can be conducted using the VC generator, including the one conducted at Clemson and discussed in the evaluation section. The tool was also used at Denison. Though formal results from Denison are unavailable, students found the connections between logic and software engineering interesting, and at least two decided to pursue follow-up independent studies on the topic.

3. EVALUATION

Results from our experimentation have been positive to the point that the faculty of the *School of Computing* at Clemson voted unanimously to integrate reasoning principles in two of the required undergraduate courses beginning in the fall of 2009. Denison plans integration within their courses as well. Some key evaluation results from Clemson are summarized in this section.

3.1 Evaluation of Reasoning Tools

To evaluate the usefulness of the specification understanding tool (Section 2.1), after introducing formal specifications, we gave a short quiz consisting of 7 formally-specified operations, all with unhelpful names. Students were required to provide test cases for each. The class average was 75%, showing that most students can understand formal specifications readily. The performance improved further with the tool. After its introduction, a similar quiz was given. The class was partitioned, with only one-half allowed to use the tool during the quiz. The class average was 84%. Students using the tool had an average of 94%, compared to a 78% average for students who did not use the tool. Two class periods later, students who did not use the tool were allowed to retake the quiz using the tool. The average across these students increased to 94.3%. Details of the tool for teaching formal specifications and its use in encouraging "collaborative reasoning" are reported in [10].

One of the most exciting results of applying the team development tools (Section 2.2) is that students get to observe the benefits of their collaborative, yet independent efforts first-hand. When errors arise during component integration, they refer to the contracts to see if the violator was a user or an implementer. On the day students turned in their team projects, the instructor integrated different components from different teams – the software worked perfectly with no integration difficulties! In fact, for some components, the instructor used code from different *methods* based on matching *internal* component contracts (e.g., representation invariants). Whereas such mixing would have worked only 25% of the time based on a pure probabilistic analysis if students hadn't used or properly understood the role of internal contracts, in our experiment, it would have worked 80% of the time. The students, including two co-authors of this paper, were amazed, and hopefully learned the powerful role of formal contracts in software engineering.

To evaluate the impact of formal assertions and the VC generator tool (Section 2.3), we conducted a comparative experiment to analyze students' abilities to discover errors in code. In the first case, students were given only the verification conditions and asked, for each example, if one or more VCs could not be proved. (This would indicate an error in the corresponding code fragment with respect to its specification.) In the second case, students were given several code examples and corresponding specifications, and asked to detect the errors by reasoning about the code directly. The examples across the two trials were similar, but not identical.

Statement	Initial Hypothesis	Pre-Survey Average	Post-Survey Average
1	+	3.44	3.87
2	**+**	**3.05**	**3.76**
3	+	3.63	3.84
4	**?**	**3.29**	**3.97**
5	+	4.02	4.34
6	**-**	**2.05**	**1.39**
7	**+**	**3.02**	**3.86**
8	**+**	**3.02**	**4.29**
9	**+**	**3.31**	**3.76**
10	**-**	**2.51**	**1.74**
11	**-**	**4.41**	**3.05**
12	?	2.41	2.55
13	+	2.66	2.84
14	?	3.41	3.34
15	-	2.29	2.13
16	+	3.22	3.58
17	**?**	**2.90**	**2.37**
18	**-**	**2.80**	**2.08**
19	?	2.66	2.66
20	**-**	**3.56**	**2.63**
21	**?**	**4.24**	**3.74**
22	+	3.44	3.55
23	+	3.61	3.84
24	+	3.93	3.68
25	+	3.88	3.89

Table 1. Attitudinal Survey Results

The average score for the assertion-based assignment was 73.1%, as compared to 66.9% for the code-based assignment. This suggests that the students were at least as able to understand and analyze verification conditions generated by our tool as they were able to analyze code. For some questions, however, students clearly performed better using the code directly. In the assertion-based assignment, these questions involved an inference that many students overlooked. For other questions, the assertions were much easier to analyze than the code. We mention just two examples: Out of 17 students asked to determine if a given code fragment was correct, 15 answered incorrectly because they did not realize that an object was being pushed on a full stack. In contrast, only 8 failed to notice the boundary condition problem when looking at the generated assertions. The remainder of the students answered incorrectly because they missed an inference regarding the final goal of the code. The most significant improvement came from a question involving complex control and data flow. Only four students missed this when looking at the assertions, compared to 14 when studying code.

3.2 Attitudinal Evaluation

To assess the impact of the approach on student attitudes toward software development, we administered (identical) pre- and post-semester surveys in our software engineering course. The surveys maintained student anonymity, though we used "secret codes" known only to the students to correlate pre- and post-semester results. The surveys included both attitudinal questions and essay-type questions. Here we focus only on the attitudinal results.

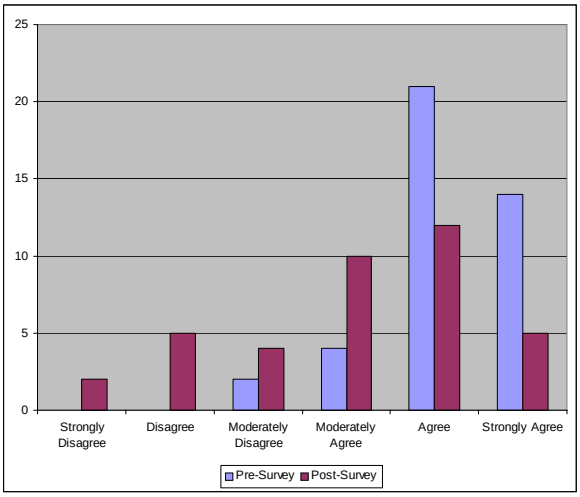

Figure 3. Survey Results for Statement 11

The survey included 25 statements; students were required to rank their level of agreement with each from *strongly disagree* (0) to *strongly agree* (5). Table 1 summarizes data collected from 25 students. In the table, "+" indicates that we expected more agreement at the end, and "-" indicates that we expected less. A "?" indicates that the anticipated outcome was uncertain.

Due to space limitations, we are unable to discuss all of the statements, but it is easy to see that student attitudes moved in the expected direction. (The only exception was statement 24, "Developing quality software is difficult".) Indeed, statistical analysis reveals that for about half the statements, shown in bold, the movement is highly significant in the desired direction. To give an example of a statement where the expected direction was not obvious, consider statement 12: "It is easy to combine components from different team members and produce working software." Given our emphasis on formal contracts, it seems reasonable to expect that more students would agree at the end. However, if students realized that developing these contracts is not easy, their reaction would be the opposite. Graphs for two representative statements appear in figures 3 and 4:

- (**S11**) Testing software thoroughly is the most important way to ensure correctness.
- (**S18**) When working in teams, natural language (e.g., English) descriptions of the different components are sufficient for communication among team members.

Other statements range from more general to reasoning-specific:

- (**S3**) The difficulty in understanding and modifying a 10,000 line software system has more to do with the style in which the software is written, and less to do with how smart I am.
- (**S20**) Reasoning about programs involving components requires a thorough understanding of pointers and/or references.

Our comparative evaluation with a traditional software engineering course showed that students in the experimental class were more attuned to formal specification and reasoning.

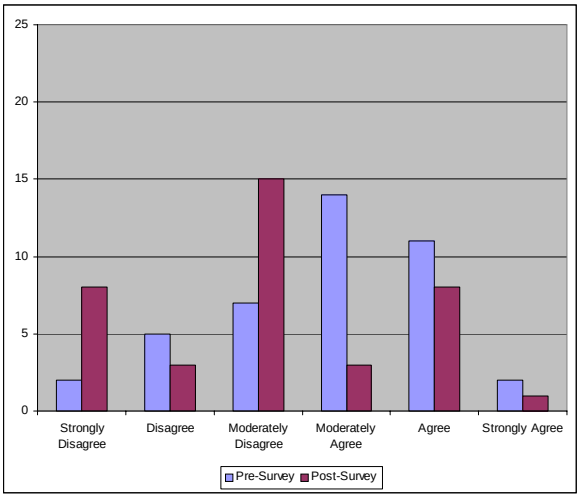

Figure 4. Survey Results for Statement 18

4. RELATED WORK AND CONCLUSIONS

The approach to teaching reasoning principles presented in this paper differs from our own previous work [12], as well as from other related efforts in that we focus on engaging students with "hands-on" experimentation using computer-aided reasoning tools in a software engineering context. Some of the prior efforts have focused on the introductory course sequence. Whereas components are the focus of the educational efforts in [3], introducing students to and exciting them about mathematical thinking is the theme of [4]. McLoughlin and Hely introduce formal reasoning early in the computing curriculum by using templates as one of a collection of techniques for students to learn and apply [13]. Dony and LeCharlier developed a tool for checking the correctness of simple programs, which can provide feedback to students by detecting programming and/or reasoning errors and providing typical counter-examples [14]. Beyond the well-known work of Henderson in software engineering [2], Hilburn has introduced the use of collaboration among students to learn reasoning through a peer review process and incorporated it within an introductory course in formal methods [15].

While it is nice to teach the benefits of formal methods, "in principle," there is no substitute for employing them in classrooms. Tools allow for "hands-on" experimentation, help engage students, and make it possible to introduce the principles in small increments at interested institutions. Specifically, they aid in teaching specifications, development of software relying on contract specifications, and analytical reasoning. Evaluation results and feedback from students in a software engineering course have been positive at two institutions.

5. ACKNOWLEDGMENTS

This work is funded in part by U. S. NSF grants DUE-0633506, DMS-0701187, CCF-0811748, and CNS-0745846.

6. REFERENCES

[1] Gries, D., "A Principled Approach to Teaching OO First," *The 39th SIGCSE Technical Symposium on Computer Science Education*, ACM, 2008, 31-35.

[2] Henderson, P. B., "Mathematical Reasoning in Software Engineering Education," *Communications of the ACM*, **46**(9), ACM, 2003, 45-50.

[3] Bucci, P., Long, T., and Weide, B., "Do We Really Teach Abstraction?" *The 32nd SIGCSE technical symposium on Computer Science Education*, ACM, 2001, 26-30.

[4] Tomer, T.S., Baldwin, D., and Fox, C. J., "Integration of Mathematical Topics in CS1 and CS2," *The 31st SIGCSE Technical Symposium on Computer Science Education*, ACM, 1998, 364-365.

[5] Almstrum, V., Dean, C., Goelman, D., Hilburn, T., Smith, J., "Support for Teaching Formal Methods," *ACM SIGCSE Bulletin (ITiCSE 2000 Working Group Reports)*, **33**(2), ACM, 2001, 71-88.

[6] Henderson, P.B., Baldwin, D., Dasigi, V., Dupras, M., Fritz, J., Ginat, D., Goelman, D., Hamer, J., Hitchner, L., Lloyd, W., Marion, B., Riedesel, C., Walker, H., "Striving for Mathematical Thinking," *ACM SIGCSE Bulletin (ITiCSE 2001 Working Group Reports)*, 33(4), ACM, 2001, 114-124.

[7] Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W., "Specifying Components in RESOLVE," *Software Engineering Notes*, **19**(4), ACM, 1994, 29-39.

[8] Harton, H., Krone, J., and Sitaraman, M., "Formal Program Verification," *Encyclopedia of Computer Science and Engineering*, Ed. B. Wah, John Wiley & Sons, 2008.

[9] Kulczycki, G., Sitaraman, M., Yasmin, N., and Roche, K., "Formal Specification," *Encyclopedia of Computer Science and Engineering*, Ed. B. Wah, John Wiley & Sons, 2008.

[10] Leonard, D.P., Hallstrom, J.O., and Sitaraman, M., "Injecting Rapid Feedback and Collaborative Reasoning in Teaching Specifications," *The 40th SIGCSE Technical Symposium on Computer Science Education*, ACM, 2009, (*to appear*).

[11] Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., and Frazier, D., "Incremental Benchmarks for Software Verification Tools and Techniques," *Verified Software: Theories, Tools, and Experiments*, Springer-Verlag, 2008, 84-98.

[12] Sitaraman, M., Long, T.J., Weide, B.W., Harner, E.J., and Wang, L., "A Formal Approach to Component-Based Software Engineering and Its Evaluation," *The 23rd International Conference on Software Engineering*, IEEE, 2001, 601-609.

[13] McLoughlin, H. and Hely, K., "Teaching Formal Programming to First Year Computer Science Students," *The 29th SIGCSE Technical Symposium on Computer Science Education*, ACM, 1996, 155-159.

[14] Dony, I. and Le Charlier B., "A Tool for Helping Teach a Programming Method," *The 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ACM, 2006, 212-216.

[15] Hilburn, T.B., "Inspections of Formal Specifications," *The 29th SIGCSE Technical Symposium on Computer Science Education*, ACM, 1996, 150-15.