

Tool-Aided Learning of Code Reasoning with Abstraction in the CS Curriculum

Megan FOWLER^{1,*}, Jason HALLSTROM²,
Joseph HOLLINGSWORTH⁴, Eileen KRAEMER¹,
Murali SITARAMAN¹, Yu-Shan SUN¹, Jiadi WANG⁴,
Gloria WASHINGTON³

¹*Clemson University, USA*

²*Florida Atlantic University, USA*

³*Howard University, USA*

⁴*Rose-Hulman Institute of Technology, USA*

*e-mail: mefowle@g.clemson.edu, jhallstrom@fau.edu, hollings@rose-hulman.edu,
etkraem@clemson.edu, msitara@clemson.edu, yushans@g.clemson.edu,
wangj19@rose-hulman.edu, gWASHINGTON@scs.howard.edu*

Received: February 2021

Abstract. Computer science students often evaluate the behavior of the code they write by running it on specific inputs and studying the outputs, and then apply their comprehension to a more general understanding of the code. While this is a good starting point in the student’s career, successful graduates must be able to reason analytically about the code they create or encounter. They must be able to reason about the behavior of the code on arbitrary inputs, without running the code. Abstraction is central for such reasoning.

In our quest to help students learn to reason abstractly and develop logically correct code, we have developed tools that rely on a verification engine. Code involves assignment, conditional, and loop statements, along with objects and operations. Reasoning activities involve symbolic reasoning with simple assertions and design-by-contract assertions such as pre-and post-conditions as well as loop invariants with data abstractions. Students progress from tracing and reading code to the design and implementation of code, all relying on abstraction for verification. This paper reports some key results and findings from associated studies spanning several years.

Keywords: abstraction, design by contract, online tool, software engineering, symbolic reasoning.

*Corresponding author.

1. Introduction

1.1. Foundations of Computer Science

Computer Science is a rapidly evolving field. It requires students to remain up to date on the newest techniques, languages, and practices. Yet one aspect that doesn't change is the theoretical foundations of computing, such as abstract reasoning about code correctness. In order for students to enter the work force as successful developers, they need to have a strong foundation in creating stable, well-designed code (Hinchey *et al.*, 2008). It has been established that students who are able to effectively trace code are better at writing code (Lister *et al.*, 2004, 2009). Current standards for tracing code involve running code on multiple test case input values. While this is a valuable process, it can lead to limited student understanding of code. It is just not possible to run every possible test scenario. For example, in a junior level software engineering course, 50% of students found the code segment in Listing 1 was a correct implementation for finding the maximum value between two integers (Cook *et al.*, 2018).

Without students providing detailed work such as seen in Table 1, we can only speculate as to the cause for their oversight of the case in which i and j are equal. Their reasoning, for example, may have been ad hoc.

What appears to be a simple oversight in a code sample is an argument for students to learn to generalize their reasoning through the use of abstraction. Symbolic reasoning is

```

int Max (int i, int j){
    int max = i + j;
    if (i > j){ max = max - j; }
    if (j > i){ max = max - i; }
    return max;
}

```

Listing 1. Code Exercis.

Table 1
Example Trace for Max with Concrete Values

Code	max(1, 2)	max(2, 1)	max(2, 2)
<code>int max = i + j;</code>	max = 1 + 2; max = 3;	max = 2 + 1; max = 3;	max = 2 + 2; max = 4;
<code>if (i > j) { max = max - j; }</code>	false	true max = 3 - 1; max = 2;	false
<code>if (j > i) { max = max - i; }</code>	true max = 3 - 1; max = 2;	false	false
<code>return max;</code>	max = 2	max = 2	max = 4;

Table 2
Example Trace using Symbolic Reasoning

Code	max(i, j)
<code>int max = i + j ;</code>	max = i + j;
<code>if (i > j) { max = max - j; }</code>	max = i + j - j max = i
<code>if (j > i) { max = max - i; }</code>	max = i + j - i max = j
<code>return max;</code>	max = i or max = j or max = i + j

reasoning about code on arbitrary symbolic input values, as opposed to specific concrete inputs. In this example, if students traced the code with abstract symbolic values – making no assumptions about their concrete values – they will have to consider every path of the code. Such symbolic tracing, done manually or facilitated by an automated tool, can help them understand where the code fails to compute the maximum. A simplified version of such symbolic reasoning is shown in Table 2. In Listing 1, notice that the values of `i` and `j` never change. Using `i` and `j` as initial symbolic place holders, we see that the `max` value returned would be either `i`, `j`, or `i+j`.

1.2. Abstraction and Symbolic Reasoning

In general, our goal is for students to learn to reason abstractly. For example, to reason that a given example function computes the positive square root of its input value on **all** allowed inputs and not only that it computes 3.0 on input 9.0 or 5.0 on input 25.0.

A symbolic approach to reasoning aims for students to learn to generalize and understand the overall purpose of code. A low level of understanding can be demonstrated by explaining what is happening in the code line by line. For example, suppose that Listing 1 in Section 1.1 were named `Mystery` instead of `Max`, and that the code were corrected by changing the first `if` statement to greater than or equal to. An example of a line-by-line explanation would be as follows:

- `mystery` is assigned the value of `i` plus `j`
- if `i` is greater than or equal to `j`, `j` is subtracted from `mystery`
- if `j` is greater than `i`, `i` is subtracted from `mystery`
- `mystery` is returned

Essentially, the above explanation is a translation of the code into common vernacular. Abstraction requires the student to view the logic of the operation in a broader scope and therefore produce a more holistic understanding of the functionality. Even for an operation with the name `Mystery`, a student who develops such an understanding would conclude that it returns the maximum of input values.

We hope to help students achieve this higher level of understanding through symbolic reasoning. Students will still utilize tracing through code, but instead of only reasoning with concrete values, they will also use symbolic input values. For the example

Table 3
Example Trace for Swap

Code	i	j
$i = i + j;$	$\#i + \#j$	$\#j$
$j = i - j;$	$\#i + \#j$	$\#i + \#j - \#j$ $\#i + \#j - \#j$ $\#i$
$i = i - j;$	$\#i + \#j - \#i$ $\#i + \#j - \#i$ $\#j$	$\#i$
Final Result	$i = \#j$	$j = \#i$

above, the input values of i and j will be *remembered* as $\#i$ and $\#j$. So if the student were to write $i = i + j;$ the value of i would be equal to $\#i + \#j$ (the input value of i plus the input value of j). An illustration of the idea is in Table 3, which shows that the code effects a swap operation.

In the end we see that i now equals the input value of j and j equals the input value of i . We were able to determine the purpose of this code without running multiple test cases using concrete values and having to identify the pattern.

1.2.1. Data Abstraction

Whereas for primitive objects such as Integers and Reals, the corresponding mathematical abstractions are implicit and obvious, that is not the case for data structuring objects, such as stacks, queues, and lists.

Abstraction is key when tracing method calls so students are not overwhelmed by internal data structures (Bucci *et al.*, 2001). The emphasis should not be placed on how the data structure is represented within the operation, but rather on the abstract behavior of the operations. For example, when using a stack object, it is not important to know if a stack is represented by an array or linked structure, but what stack's operations such as `push()` and `pop()` do based on an understanding of a formal stack data abstraction. While it is possible to observe and learn this behavior through the use of concrete examples, symbolic reasoning can help provide a stronger foundation and allow the transfer of knowledge across data structures.

1.3. Research Objectives

The over arching goal of this research is to help students learn abstract reasoning at various levels with the aid of tools that help students practice and help instructors identify and understand their difficulties. Beginning with symbolic reasoning with simple assertions as a foundation, students proceed to understand data abstractions. Studies were conducted between 2016 and 2021 across 8 semesters which we label as F1, S1, F2, S2, F3, S3, F4, and S4.

A key aim of using automated tools is to understand the specific fine-grain learning difficulties students face when reasoning about code so that appropriate interventions may be developed (Cook *et al.*, 2018; Fowler *et al.*, 2019; Priester *et al.*, 2016). By “fine-grain,” we mean understanding student difficulties at a resolution that exceeds identifying high-level constructs that might present challenges (e.g., functions, loops, parameter passing) to reveal the underlying cause(s) of a learning roadblock (e.g., a missing algebraic foundation or a flaw in the student’s mental model of variable storage). When these difficulties are not readily apparent to instructors, it is hard to devise suitable interventions, especially for students with the most need. Achieving this level of resolution is prohibitively time-consuming in the absence of automation.

Our Educational Research Questions (ERQs), summarized below, will be discussed in context in later sections. Together they focus on learning difficulties along four broad themes of abstraction.

1. Learning symbolic reasoning basics with assignment statements; in Section 4. *This study utilized semesters F1, S1, and F2.*

ERQ 1.1: With or without intermediate steps, can a majority of students learn the basics of tracing code using symbolic input values instead of specific input values (1) strictly with the help of an online reasoning tool and (2) with instruction in addition to the tool?

2. Learning symbolic reasoning with conditional statements; in Section 5. *This study utilized semester S4.*

ERQ 2.1: What impact does the online tool have on student performance regarding the tracing of conditional statements using arbitrary symbolic values?

ERQ 2.2: What impact does the online tool have on student self-efficacy regarding the tracing of conditional statements using arbitrary symbolic values?

3. Learning to use design-by-contract assertions in reasoning with data abstractions; in Section 6. *This study utilized semester S2.*

ERQ 3.1: What common learning difficulties in reading and writing formal Design-by-Contract (DbC) assertions can be pinpointed with an automated tool and collected data?

ERQ 3.2: Which difficulties persist on a final exam, when students do not have access to the tool?

4. Learning to develop loop invariants for code involving data abstractions; in Section 7. *This study utilized semesters F3, S3, and F4.*

ERQ 4.1: What common difficulties do students face, specifically as it concerns developing loop invariants?

ERQ 4.2: With respect to developing loop invariants, a) what do student responses reveal about their level of understanding of the concepts and b) how suitable are their responses for identifying actionable items for intervention?

The research questions related to the first two themes additionally consider the benefits of an automated tool itself, whereas the questions on the last two themes use

the tool only as a means to collect data for analysis. Results presented from all themes, except for ERQ 2.1 and 2.2, are synthesized from prior publications. Additional details may be found in the upcoming 2021 PhD dissertation of the primary author (Fowler, 2021).

2. Related Work

2.1. *Abstraction and Reasoning*

A debate exists in learning theory regarding the depth of understanding achieved by students when introduced to a new topic through use of concrete examples, versus through abstraction (Carbonneau *et al.*, 2013; De Bock *et al.*, 2011; Kaminski *et al.*, 2008; McCallum, 2008), though the domain there is not computing. The findings in (Kaminski *et al.*, 2008) suggest that “...giving college students multiple concrete examples may not be the most efficient means of promoting transfer of knowledge” and that “because the difficulty of transferring knowledge acquired from concrete instantiations may stem from extraneous information diverting attention from the relevant mathematical structure, concrete instantiations are also likely to hinder transfer for young learners who are less able than adults to control their attentional focus.” Her study showed that through the use of an abstract generalized structure, students were able to transfer that knowledge to novel situations whereas when they used concrete instantiations there was little or no transfer. McCallum rebutted this work claiming that the two treatment groups were not working with the same mathematical structure which led to a bias in the transfer task (McCallum, 2008).

A study focusing on the teaching of an electronic circuit-wiring task found that when experts taught novices, they used more abstract statements compared to beginner instructors who used more concrete examples to teach novices. They found that the beginner-instructed novices performed better than the expert-instructed novices when completing the target task. However, the reverse happened when trying to apply this knowledge to a different task within the same domain (Hinds *et al.*, 2001). Here, the novices instructed by experts performed better than their counterparts. This reinforces the idea that concrete examples may be useful for learning repeated tasks, but may not necessarily help with the transfer of knowledge.

Within computing, the standard for all students is to learn how to code through the use of concrete values. The automation of grading has led to the current laboratory practice in which students are writing code to pass given test cases. The emphasis is less on understanding overall code behavior. This can lead to bloated, clunky, or inefficient code. Worse, it can be incredibly frustrating to a student to write code that appears to meet example test cases, yet be told that it fails the overall purpose. It is also possible for automated testing to approve a flawed solution (Forišek, 2006). The students who are able to reason well in laboratory exercises and competitions focus of the logic of the problem using induction (Ginat, 2014), which is key to abstraction.

2.2. Data Abstraction and Loop Invariants

Following an introduction to data abstraction with formal design-by-contract assertions (Fowler *et al.*, 2020), students are primed to develop and reason with loop invariants for code that involves objects. Despite the importance of loop invariants for understanding and debugging of algorithms, few computer science or software engineering graduates are able to use them effectively (Henderson, 2003). When students learn to write loop invariants for iterative code, they can achieve a level of understanding not possible otherwise (Gries, 1981).

2.3. Reasoning Tool Design

Online tutoring of programming concepts has received much attention in the CS literature (e.g., (Alevan *et al.*, 2009; Bhattacharya *et al.*, 2011; Guo; Kumar *et al.*, 2007; Li *et al.*, 2011; O'Brien *et al.*, 2014; Price *et al.*, 2017; Wiggins *et al.*, 2015)). There are several IDEs that provide compile-time error feedback and numerous useful capabilities (e.g., finding the declaration or all uses of a method). The novel online reasoning tool that we have developed, unlike other tutors and IDEs, is backed by a software verification engine (Cook *et al.*, 2012a; Sitaraman *et al.*, 2011). This allows it to facilitate reasoning over abstract input values. It has a strong theoretical basis and has been used for nearly a decade at multiple institutions. Thousands of undergraduate students have employed symbolic reasoning approaches using this reasoning tool in CS courses (Cook *et al.*, 2012b; Drachova *et al.*, 2015; Hallstrom *et al.*, 2014; Heym *et al.*, 2017) and in software engineering projects (Cook *et al.*, 2013; Priester *et al.*, 2016). It suffices to say that the engine is far more powerful than demanded by the reasoning activities discussed here. The engine can enhance learning through a variety of logical “what if” questions. Since answers are verified automatically, answer keys are not stored or used.

3. Tool Design and Research Framework

3.1. Tool Design and Reasoning Activities

The reasoning tool utilizes a verification engine (Sitaraman *et al.*, 2011) so that a proof can be automatically generated and students do not have to complete the proof to check correctness. This allows instructors to appropriately challenge students without overwhelming them. The use of an online tool also means that students can make multiple attempts by having the tool check the correctness of their assertions.

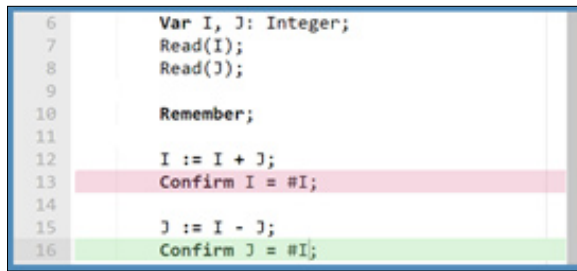
The tool has been designed with scaffolding to reduce cognitive load, defined as the ease with which information can be processed in working memory (Paas *et al.*, 2010; Sweller *et al.*, 2011), as seen from the screen shot in Fig. 1. The activity description is

Lesson 0 - Constants and assignments		
<< Prev	Next >>	
Activity: Please complete the Confirm assertion(s) and check correctness.		<pre> 1 Facility BeginToReason; 2 uses Integer_Ext_Theory; 3 4 Operation Main(); 5 Procedure 6 Var I, J, K: Integer; 7 8 I := 1; 9 J := 2; 10 K := 3; 11 12 J := I + J; 13 I := J - K; 14 15 Confirm I = /*expression*/; 16 end Main; 17 end BeginToReason;</pre>
Reference Material: := is the assignment operator		
<< Prev	Next >>	Click here to check correctness
Reload lesson	End survey	

Fig. 1. Online Reasoning Tool Interface.

at the top left with reference material below it showing only what is relevant to reason about the code on the right. The first few lessons ask students to trace code on specific input values, such as in Fig. 1. The use of the tool is the first introduction to all the constructs. In the Pascal-like code shown (with which most of our students learning Java apparently have no trouble and need no introduction), we use `:=` to denote assignment to distinguish it from the mathematical equality used in assertions. The student's task is to trace the given code and replace `/* expression */` after the **Confirm** with a logical assertion (utilizing “=” means equals, as in logic). The student can make changes only in the assertion. Unlike an `assert` statement, nothing is executed; however, the assertion is given to the verification engine to verify that the assertion holds.

An example of symbolic reasoning is shown in Fig. 2. The activity asks students to state the values of `I` and `J` in terms of their input values, remembered to be `#I` and `#J`, respectively, at the line marked Remember. The tool has been designed to provide visual feedback that is immediate (Azevedo and Bernard, 1995) to reduce cognitive load (Chen *et al.*, 2011; Moreno, 2004). An example of this feedback can be seen in Fig. 2. Since `I` is changed on line 12, it is not true that `J = #I - #J`, and hence, line 16 is wrong (and has been highlighted with a red background). A number of correct, logically equivalent answers exist. While all answers are verified, the system expects reasonable answers, rejecting trivial answers such as **Confirm** `I = I`. When a student's answer is wrong (i.e., does not verify, or is trivial), the lesson may be repeated, or a follow-on lesson is given. Though not a focus of the current paper, a key benefit of this tool is that it can pinpoint obstacles specific to subsets of learners (Cook *et al.*, 2018).



```

6      Var I, J: Integer;
7      Read(I);
8      Read(J);
9
10     Remember;
11
12     I := I + J;
13     Confirm I = #I;
14
15     J := I - J;
16     Confirm J = #I;

```

Fig. 2. Symbolic Reasoning with Visual Correct and Incorrect Feedback.

3.2. Research Framework

In our attempt to help students to understand and use abstraction in reasoning about code, we have integrated symbolic reasoning concepts into a sophomore level software development course and a junior level software engineering course, both required courses for CS majors at our institution. Introducing students to using logic to evaluate code correctness early in their education helps build a foundation for abstract reasoning. The subsequent course then allows students to write and develop their own code to meet specifications provided by the instructor. By dividing this process across two courses, students are less likely to be overwhelmed. This approach also promotes reinforcement of these concepts.

3.2.1. Symbolic Reasoning with Simple Assertions

The software development course description includes specification and reasoning among its topics and is required for all CS majors at our institution. Details of this course are discussed in (Hallstrom *et al.*, 2014). The course has a unit dedicated to symbolic reasoning. Students receive a lecture on the topic and are then introduced to the online tool *BeginToReason*. The tool aids in pinpointing student difficulties when learning symbolic reasoning. Users are presented with a code sample and are asked to complete assertions regarding the logic of the code.

3.3. Reasoning with Data Abstraction Assertions

The subsequent software engineering course description includes program specification and reasoning as core topics and is required of all CS majors. The course integrates the use of formal contracts. Pre-and post-condition assertions are now included in the tool. Additionally, students learn to how to develop loop invariants through using a web IDE that allows for the design and implementation of code. Though the web IDE's interface is different from the tool used in the software development course, this tool functions similarly in that it relies on the same verification engine to generate and check proofs for correctness.

4. Learning Symbolic Reasoning Basics

4.1. Introduction

The first learning objective is to understand whether we can effectively teach the general population of computer science students the basics of reasoning about code on all input values using a symbolic approach with the (non-exclusive) aid of a reasoning tool. Symbolic reasoning basics concerns the ability to reason about a sequence of assignment statements, and that is the focus of this section. Additional details of symbolic reasoning studies in pinpointing specific difficulties for individual students and subgroups may be found in (Cook *et al.*, 2018). A related study in (Fowler *et al.*, 2019) explores the role of motivation in learning symbolic reasoning.

4.1.1. Educational Research Question

Given the growing importance of online education in reaching a diverse audience, and the role of online tools in ensuring that at least a portion of CS education can happen outside a classroom, this research involves understanding the role of a reasoning tool we have built. In helping students learn the basics of reasoning, we consider the role of a step-by-step approach (i.e., intermediate steps) in reasoning correctly about code composition and whether students will follow such an approach if learned.

ERQ 1.1-Reasoning Basics: With or without intermediate steps, can a majority of students learn the basics of tracing code using symbolic input values instead of specific input values (1) strictly with the help of an online reasoning tool and (2) with instruction in addition to the tool?

The hypothesis here is that a significant majority of students will indeed be able to learn to reason with symbolic values; after all, students learn algebra in high schools. But such learning will require classroom instruction in conjunction with a tool. We also consider if students perform as well on symbolic reasoning questions as on other topics in a CS course.

4.2. Experimental Methods

The reasoning system and the activities were administered in lab sessions. Data for the experiments discussed in this section were obtained over the course of three semesters; F1, S1, and F2. When multiple lecture sections were involved, the students were intermingled in the lab sections.

We have varied when intermediate steps are provided in the tool activities. Intermediate steps are one way to ask a learner to show their understanding after each program statement in a code segment (through **Confirm** assertions), instead of providing a single summary **Confirm** assertion at the end of the code. This compels the student to explicitly think about the values of variables after each statement. We included a sym-

bolic reasoning question on the exam to determine student learning. We also varied exam questions: asking for steps, not asking for steps, and provided scaffolding.

4.3. Results

4.3.1. ERQ 1-Reasoning Basics: Ability to Learn Symbolic Reasoning Basics

For the F1 experiment, 114 students participated in the study across five sections of the lab. Students did not receive any formal instruction regarding symbolic reasoning before tool use. This tool served as their first introduction to symbolic reasoning about code. Though TA help was available, few students asked for help. Students were not time constrained and took between 20 minutes to an hour to complete all the activities.

About six weeks after working with the tool in the lab, students were asked to complete a symbolic reasoning question on a regular final exam. The question was worth a total of five points, which was 20% of the exam. There were four versions of the reasoning question and the versions were randomly assigned at testing time. Two versions of the reasoning question involved reasoning about a piece of swapping code (similar, but not identical to the one given in the introduction). The other two versions had to do with the sum of two variables. These two question versions were then further separated by one version of the question asking students to show their work, while the other did not.

Across all versions of the questions in F1, students received an average of at least 4 points, which translates into completing approximately 80% of the question correctly. It would appear the online tool can successfully introduce all students to the basics of symbolic reasoning. At the same time, it is equally possible that the question was too easy. This is a useful, but not definitive result. The performance of students on such questions have been sufficiently impressive that instructors in later semesters have resorted to giving more demanding questions (in terms of both the number of variables and the number of statements).

Learning Reasoning w/wo Classroom Instruction. The S1 and F2 experiments had 91 and 92 students respectively across five lab sections. We used a Mann-Whitney U-test to assess if the student populations are comparable because it is more suitable to compare performance of different groups of students across semesters and because it does not require that the two independent samples are normally distributed.

In F2, tool-based instruction was supplemented with a one-hour classroom lecture before the exam. The difference in scores for the exam reasoning question between the semesters was found to be statistically significant with a p value less than 0.05. In the spring, only 59.1% of students completed at least 80% of the question correctly, while in the fall, 74.2% did. A similar pattern is observed between the overall exam scores, where fall students performed significantly better, with a p value of 0.0005. While this would point directly to the benefit of classroom instruction, student motivation may be a confounding factor and is discussed in (Fowler *et al.*, 2019). Regardless, the important point is that symbolic reasoning can be learned by a majority of students. The next section explains how students can go from these basics to generalize the purpose of code

containing a conditional statement. Later sections illustrate how students are able to connect the formally stated purpose of code (contracts) given in symbolic form with code that realizes that purpose.

5. Learning to Reason about Conditional Statements

5.1. Introduction

Whereas the initial findings have shown the usefulness of an online tool to help students, they focused on students reasoning about code involving only assignment statements. Reasoning about conditional statements symbolically is naturally more challenging. An exploration of that topic is the focus of the research discussed in this section.

5.1.1. Educational Research Questions

ERQ 2.1-Performance on Conditionals: What impact does the online tool have on student performance regarding the tracing of conditional statements using arbitrary symbolic values?

ERQ 2.2-Self-Efficacy on Conditionals: What impact does the online tool have on student self-efficacy regarding the tracing of conditional statements using arbitrary symbolic values?

This research has focused on two questions, both of which consider the benefits of the reasoning tool. Since the classroom instruction was on symbolic reasoning basics with assignments, any learning here can be attributed more directly to the practice and learning resulting from tool usage.

5.2. Experimental Methods

The reasoning system and the activities were administered in closed lab sessions. Data for the experiments discussed in this section were obtained in the S4 study at three universities.

A total of 106 students completed a pre-quiz, followed by a lab activity, and then a post-quiz. This chapter focuses on the results obtained from the pre-and post-quiz with regard to student performance and reported self-efficacy. While there were multiple lecture sections, the students were intermingled in the lab sections (but not across universities).

The assessment quiz provided to students organized the questions in order of increasing difficulty. We presented three types of multiple choice questions to students that were essentially the same, except that the answers were in different formats, Table 4. The first type of answer choices were about a high-level, holistic understanding of the purpose of the code. The second type of answer choices made use of simple math functions to summarize code behavior. The third type of answer choices were logical, using only common relational operators.

Table 4
Conditional Questions on Assessments

Holistic	Functional	Relational
<pre>//Remember the value of I at this point as # I, etc. If (J <= K) { K = J; }</pre>	<pre>//Remember the value of I at this point as #I, etc. If (J < I) { I = J; }</pre>	<pre>//Remember the value of I at this point as #I, etc. If (J > I) { I = J; }</pre>
<p>a. J is unchanged b. K is unchanged c. J is the minimum of #J and #K d. J is the maximum of #J and #K e. K is the minimum of #J and #K f. K is the maximum of #J and #K</p>	<p>a. I = #I b. J = #J c. I = Min(#I, #J) d. I = Max(#I, #J) e. J = Min(#I, #J) f. J = Max(#I, #J)</p>	<p>a. I >= #I b. I <= #I c. J >= #J d. J <= #J e. I >= #J f. J <= #I</p>

5.3. Results

5.3.1. ERQ I-Performance on Conditionals

We found that overall, there was a statistically significant improvement in student performance using symbolic reasoning about conditional statements. Scores from all three conditional questions on the assessments were averaged together for the overall analysis.

Fig. 3 plots each students' results so the average between the pre-and post-quiz is on the x-axis and the difference between the pre-and post-quiz on the y-axis. The re-

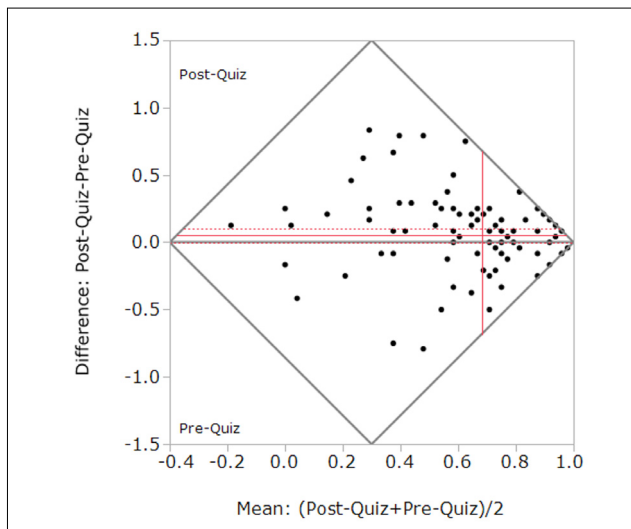


Fig. 3. Matched Pairs Analysis.

Table 5
Performance on Conditional Statements

Question	Pre-Quiz Score	Post-Quiz Score	Difference	p-value
Holistic	0.668	0.764	0.096	0.0124
Total	0.658	0.707	0.049	0.0339

sulting t-test indicates that the shift in student scores was statistically significant with a p-value of 0.0339, as seen in Table 5. The average difference was an improvement of 0.049.

When accounting for the various response methods for each question, we found that there was statically significant improvement for students when answering the conditional question that required a holistic understanding of the purpose behind the code segment, Table 5. The other two question formats did not show any statistical significance, so they were omitted from the table. This leads us to believe that using symbolic reasoning while working with the online tool helped improve student ability to reason abstractly about the purpose behind the functionality of the code.

5.4. ERQ 2-Self-Efficacy on Conditionals

The last two questions presented to students in the pre-and post-quiz were about student ability to reason about assignments and conditionals. They were designed to be a self evaluation in order for us to evaluate student self-efficacy. Students received a 7 point Likert scale as seen in Table 6 which was transformed into numerical representation. To evaluate if a student's perceived understanding of reasoning with conditional statements changed, we subtracted the pre-score from the post score.

Table 6
Likert Scale transformation

Strongly Dislike	Dislike	Slightly Dislike	Neutral	Slightly Agree	Agree	Strongly Agree
-3	-2	-1	0	1	2	3

Table 7
Student Difference in Self-Efficacy

Statement	Pre-Quiz	Post-Quiz	Diff	p-value
I can trace the execution of code involving conditional statements, using symbolic input values, such as #I and #J	1.02	1.80	0.78	<0.0001

We found that student perception of their own ability to reason about conditional statements shifted from an average of approximately 1 (Slightly Agree) closer to 2 (Agree). This shift is statistically significant. This indicates that the tool and its lessons may have helped students to feel more confident in their ability to reason about conditional statements.

6. Learning Design-by-Contract (DbC) Assertions for Data Abstractions

6.1. Introduction

This section focuses on results in a software engineering course in which students learn to read and write formal assertions using symbolic reasoning, in the context of data abstractions. In software engineering using a design-by-contract approach, software components encapsulate objects and they have interface contracts. The contracts include a mathematical abstraction for the encapsulated objects and contracts for operations to manipulate the objects, in the form of pre-and post-conditions. Students complete exercises whereby they write pre-and post-conditions using the same automated reasoning tool described in Section 4. Our findings are based on a quantitative analysis of data collected by the tool, as well as qualitative data from task-based interviews. To assess the persistence of student learning difficulties, we also study student performance on relevant exam questions.

6.1.1. Educational Research Questions

ERQ 3.1-DbC Basics: What common learning difficulties in reading and writing formal Design-by-Contract (DbC) assertions can be pinpointed with an automated tool and collected data?

ERQ 3.2-DbC Persistence: Which difficulties persist on a final exam when students do not have access to the tool?

The results help to inform computer science education efforts, not only in software development and software engineering courses, but also in discrete mathematics and formal languages courses where precise notations are important (Herman *et al.*, 2008; Zaccai *et al.*, 2014).

While this research is based on a specific formalism dictated by an underlying tool, we note that the results are generally more applicable because the core ideas of mathematical modeling and logic are shared by many formal approaches. We note that syntactic difficulties with formal expressions arise for beginning formal methods learners no matter what the language. At the same time, semantic difficulties, such as the one discussed in this section concerning the distinction between the input and output values of a parameter in an operation's post-condition, also arise across formal specification approaches.

6.2. Lessons and the Reasoning Tool

For the work reported here, the tool utilizes the same interface as the studies discussed in Section 4 and 5, but now is instrumented with six symbolic reasoning activities involving DbC assertions on data abstractions. The activities are presented with scaffolding that includes instructions and reference materials, helping to reduce extraneous cognitive load (Moreno, 2004; Sweller et al., 2011; Wouters et al., 2008).

6.2.1. DbC Assertion

Basics Given the focus on assertions, the code in each lesson is relatively simple. After a short introduction, students have little to no difficulty understanding the code, though the syntax is slightly different than what they are used to (e.g., the distinguished argument is passed as an explicit parameter – `Push(K, S)`, instead of `s.push(k)`) (Cook et al., 2018).

Formal specifications rely on using mathematical models, such as numbers, tuples, sets, or strings to explain the behavior of programming objects. For example, mathematical sets are useful when modeling a container where the order of the items in the container does not matter. Mathematical strings, on the other hand, are useful for modeling the behavior of programming objects, such as a stack, a queue, or a list, where order of the items in the container is important. Often, the same mathematical concept is used to explain a variety of programming concepts. This distinction between mathematical models (e.g., sets and strings) and programming concepts (e.g., lists and hash maps) become clear to students with a few examples. The students learn and appreciate, for example, that numbers in mathematics have no bounds whereas programming integers are necessarily bounded.

The mathematical string notation used to describe the behavior (but not a representation or specific implementation) of a data abstraction include `Empty_String`, `o` for string concatenation, `|S|` for string length, and `<E>` to denote a string containing a single entry. After a few in-class activities, students become comfortable with basic string notations.

Several of the lessons involve `Stack` objects and operations. The contract for object construction ensures that `stack S = Empty_String`, whereas the contract for `Pop` **requires** (a pre-condition) that the stack not be empty, i.e., `|S| > 0`. In the **ensures** clause (post-condition) of an operation's contract, it is often necessary to distinguish between input and output values. For example, the `Push` specification ensures the value of stack `S` after invoking `Push` as `S = <#E> o #S`; i.e., the concatenation of the input entry (`#E`) and the input stack (`#S`). These specifications are entirely abstract and are independent of how the objects might be represented and how the operations might be implemented.

6.2.2. Lessons

The first two lessons ask students to consider formal contracts for operations such as `Push` and `Pop`, and then to reason about code involving those operations. To facilitate


```

Confirm S = Empty_String;
Read (K);
Remember;
Push (K, S);
Confirm S = /* expression */;

```

Listing 2. Lesson 1.

symbolic reasoning, as opposed to the use of concrete values, a **Remember** construct is used.

The code segment for Lesson 1 is shown in Listing 2, with the type declarations omitted. The lesson begins with a newly constructed stack S . The **Confirm** line $S = \text{Empty_String}$ serves as an explicit reminder for the students of the stack's initial value. Subsequently, an integer K is read (from standard input) and pushed onto the stack. The **Activity** section on the tool's screen reminds students to replace all `/* expression */` blocks with a mathematical assertion that expresses the behavior of the provided code.

To answer the question correctly, students must be able to read and understand the contract of `Push`. A correct answer is $S = \langle \#K \rangle$. Trivial answers, such as $S = S$, are not accepted. Other incorrect answers include the following.

- $S = \langle \rangle$ – Nothing has been pushed on the stack.
- $S = K$ – Type mismatch; S is a string of entries; K is an integer.
- $S = \langle K \rangle$ – The `Push` contract is (purposely) written so that K may be changed during the call. (This answer would be correct if `Push` were specified not to change K .)

Activities 3 and 4 ask students to fill in a suitable pre-condition (or **requires** clause) for an operation so that when the clause is assumed, the operation's code is correct. Activities 5 and 6 focus on post-conditions (or **ensures** clauses), where students must specify an operation's behavior based on its code. Taken together, these activities cover the essence of operation calls in DbC.

6.3. Experimental Methods

6.3.1. Online Tool

In the S2 study, seventy-one students interacted with the tool across two sections with different instructors. The tool was used outside of the classroom with no restrictions on the amount of time available. However, completion of the activities was required before a specified due date. It is important to note that students' answers did not affect their grades, allowing them to interact with the tool without fear of penalty. Students were also told that a (paper and pencil) final exam question would be given, similar to the activities encountered when using the tool. All student response data was collected automatically.

6.3.2. Final Exam

The final exam included a logical reasoning question that required students to complete DbC activities similar to those encountered with the online tool. Students had access to the tool leading up to the exam but not during the exam.

6.4. Results

6.4.1. ERQ 3.1-DbC Basics: Automated Analysis of Difficulties

Analysis of Lesson 1 Responses. Seventy-one students attempted Lesson 1 (Listing 2); sixty-four were successful (90%), moving on to subsequent activities. The remaining seven are candidates for targeted help. Table 8 shows the distribution of student attempts at solving the lesson.

Of the 439 student responses, 86 unique response types emerged. Three of these unique responses (3%) were correct; the remaining 83 (97%) were incorrect. We analyzed the incorrect responses for frequency of appearance and for the type of error causing the problem. Table 9 shows the top 10 most frequently given responses, which are incorrect, except for the responses that are highlighted in green.

Table 8
Student-Response Distribution (Lesson 1)

No. of Attempts	No. of Students	%
1	8/71	11.3%
2 ~ 5	32/71	45.1%
6 ~ 10	17/71	23.9%
11 ~ 15	10/71	14.1%
16 ~ 20	4/71	5.6%
> 20	0/71	0%

Table 9
Top 10 Unique Responses for Lesson 1

No.	Responses	Occurrence	%
1	Confirm S = K	49/439	11%
2	Confirm S = <K>;	43/439	10%
3	Confirm S = <#K> o #S;	37/439	8%
4	Confirm S = <K> o #S;	27/439	6%
5	Confirm S = <#K>;	26/439	6%
6	Confirm S = #K;	19/439	4%
7	Confirm S = K o #S;	19/439	4%
8	Confirm S = /* expression */;	18/439	4%
9	Confirm S = #S;	13/439	3%
10	Confirm S = #S o K;	12/439	3%

Table 10
Classification of Lesson 1 Difficulties

Difficulty	Occurrence	%
Input Values	31/83	37%
Stringification	32/83	39%
String Concatenation	10/83	12%
String Length	2/83	2%
Operation Contracts	9/83	11%
Under-Specification	12/83	14%
Variables	4/83	5%
Syntax and Other	16/83	19%

Example Semantic Difficulty: Neglecting Input Values. The second-most common incorrect response was $S = \langle K \rangle$. (The correct answer is $S = \langle \#K \rangle$.) Across unique responses, the post-conditional value of K appeared in 31 instances (37%), signaling a learning difficulty. Again, the answer is incorrect only because the `Push` specification does not guarantee that the input entry K is left unchanged. `Push` may change K , so the correct answer is $S = \langle \#K \rangle$. This difficulty could reflect a misunderstanding of the “remembered” value notation or a misunderstanding of (or inattention to) the given specification of `Push`. So while a difficulty has been spotted, it is not clear what misunderstanding has caused it to arise, a topic addressed further in our qualitative analysis (Section 6.4.2).

Classifying Learning Difficulties. Table 10 summarizes our classification of difficulty types across the 83 unique incorrect responses for Lesson 1. *Since a single response may contain more than one difficulty, the percentage column does not add up to 100% – but does reflect the frequency with which the error occurred in the 83 responses.*

This fine-grain classification of obstacles is adequate for the first lesson and makes some interventions obvious. However, further refinement is needed for the more challenging activities.

6.4.2. Validation through Qualitative Analysis

The qualitative analysis to address *ERQ 3.1* and to identify the misunderstandings underlying observed learning obstacles is based on task-based interviews of nine volunteers. The answers recorded by students were then classified based on the difficulty identified in Table 3.

Overcoming Misunderstandings. Table 11 shows the progress of Student No. 3 (one of the nine volunteers), which is consistent with learning. She changed her answer twice before submitting, and with each change, moved closer to the correct answer.

After entering her first answer, she referred to the supporting material on the screen, which inspired the change to the second answer based on the post-condition. On a second pass through the material, student No. 3 recognized the need to stringify K (place it inside $\langle \rangle$) and was able to explain the purpose behind this action. When questioned

Table 11
Student No. 3 Responses for Lesson 1

No.	Response	Tool Response
1	Confirm $S = \#S \circ K;$	
2	Confirm $S = K \circ \#S;$	
3	Confirm $S = \langle K \rangle \circ \#S;$	Incorrect
4	Confirm $S = = \langle \#K \rangle \circ \#S;$	Correct

about the inclusion of the # symbol after the second failed attempt, No. 3 responded “... initially I wasn’t thinking I needed to include that, because we didn’t change K , so I was thinking K was already its original value... We change K because we use K throughout the operation, and so we have to just prove that it is the original that is being added to the stack due to... [Push specification].”

This task-based interview shows that a potential intervention could be as simple as recommending to a student that she use the references after a failed attempt, or after a fixed amount of time has been spent on the lesson without a submission. This particular student worked for four minutes before the first submission.

Lingering Misunderstandings. While use of the reference material can assist in guiding students to the correct answer, it does not guarantee an accurate understanding of the material. Consider Student No. 8’s responses for Lesson 1, shown in Table 12. Student No. 8 appears to demonstrate the same growth as No. 3 for this lesson.

When No. 8 was asked why he included the # symbol, he responded, “I want to be able to confirm that K didn’t change between when it was pushed onto the stack and when you’re confirming it.” According to this answer, it would appear that he does not understand how an element may be affected by being pushed onto a `Stack`. This suspicion was further confirmed when he reiterated this idea after Lesson 2: “You want to show that those values didn’t get changed, they were the original values that were pushed on.” Without this task-based interview, it would not have been possible to capture this particular misunderstanding since the answers being submitted were correct.

Summary Analysis. In an automated analysis, the two students above are likely indistinguishable with respect to the difficulty concerning input values, whereas the interventions suggested by the interviews are quite different.

Table 12
Student No. 8 Responses for Lesson 1

No.	Response	Tool Response
1	Confirm $S = K;$	Incorrect
2	Confirm $S = \langle K \rangle \circ \#S;$	Incorrect
3	Confirm $S = \langle \#K \rangle \circ \#S;$	Correct

For most students, learning occurred and some misunderstandings disappeared as they progressed from the first to the second lesson. This is less apparent in the automated analysis.

Finally, while the online tool only collects student response data when they make a submission for checking correctness, in the interviews it is seen that seven of the nine participants changed their answers multiple times before ever submitting. Much of students' thought processes may not be visible in the responses collected automatically.

6.4.3. ERQ 3.2-DbC Persistence: Persistence of Difficulties on Final Exam

The final exam was administered to a class of 43 students. During the exam, students did not have access to the tool. Part 1 of the logical reasoning exam question closely resembled Lesson 2, the difference being that students were working with a preemptable queue rather than a stack. 86% of students received full credit for part 1. Those that received partial credit appeared to confuse the `Enqueue()` and `Inject()` operations.

Part 2 of the logical reasoning question resembled a combination of Activities 3 through 6 from the tool. Students were required to develop an operation's pre- and post-conditions to reflect the behavior of the code provided in the question. 84% of students received full credit for the pre-condition, and those that did not receive credit did not provide an answer. The post-condition proved to be more difficult for students, resulting in 60% of students receiving full credit, 21% receiving partial credit only, and 19% not receiving any credit.

Based on student exam performance, students were able to demonstrate their learning, thereby answering the question of which difficulties persisted on a final exam when students did not have access to the tool. The persistence of semantic difficulties seen through the analysis of tool lessons is also seen to a degree on the exam. For example, students incorrectly specified the necessary preconditions for a given code segment. One confounding factor in analyzing exam answers is that manual grading might not have been as rigorous as the tool.

7. Learning to Develop Loop Invariants

7.1. Introduction

This section focuses on a more challenging aspect of learning to reason about code involving data abstractions and contracts. To formally reason about code involving loops that manipulate a data abstraction, with or without the aid of a tool, loop invariants are necessary. This section focuses on difficulties beginning students face when learning to develop a loop invariant.

7.2. Educational Research Questions

Towards helping students write invariants for loops, we considered the following specific educational research questions (*ERQs*) in our research. The reasoning tool that aided students in developing the invariants collected data as they performed invariant lessons.

ERQ 4.1-Loop Invariant Basics: What common difficulties do students face, specifically as it concerns developing loop invariants?

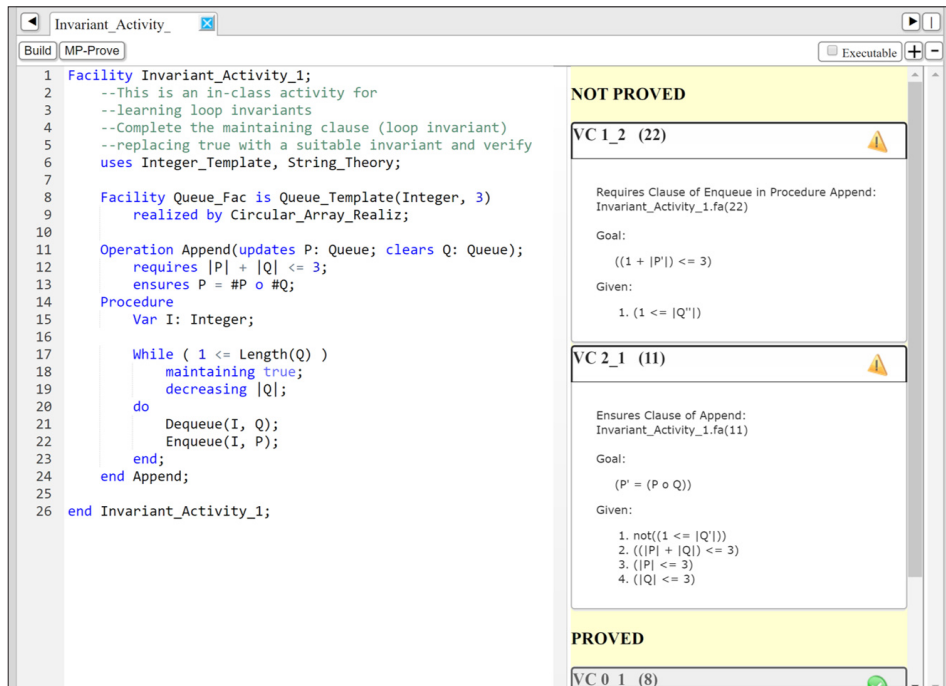
ERQ 4.2-Loop Invariant Understanding: With respect to developing loop invariants, a) what do student responses reveal about their level of understanding of the concepts and b) how suitable are their responses for identifying actionable items for intervention?

We answer both questions based on data collected as third year undergraduate software engineering students performed activities using an online verification system and developed loop invariants. ERQ 4.2 is answered using a qualitative analysis of written responses to determine if the responses show holistic, partial, or no understanding. Additionally, for ERQ 4.2, we analyze responses from a paper medium and an online medium. Obviously, the latter is more amenable to automation. The results are based on an analysis of nearly 250 submissions over three semesters, from 105 groups comprising two or three students, with a grand total of 272 students having given consent to use their data.

7.3. Online Verifier

The online system used in this experiment is backed by a formal verification engine (Sitaraman *et al.*, 2011). Using the verifier requires understanding and use of design-by-contract (DbC) assertions (Meyer, 1992). In DbC, the **requires** clause acts as a pre-condition meaning that it is the responsibility of the caller of an operation. The **ensures** clause is the corresponding post-condition that tells the caller what to expect from the operation after the call and tells the called operation's implementer what the operation must guarantee.

Fig. 4 provides a snapshot of the online verifier. When a user clicks the MP-Prove button to verify, the verifier generates and displays the verification conditions (VCs). VCs are assertions that are necessary and sufficient to prove code correctness. They arise for a variety of reasons including: that the code's **ensures** clause is met, that the **requires** clauses of all called operations are met, and that a programmer-supplied loop invariant is truly an invariant. For each VC, the verifier shows why it arises and if it is proved. Every VC needs to be proved for the code to be correct. For the example shown in Fig. 4, two of the VCs fail to prove. It turned out for some of the submitted invariants that initially failed to verify, the students were able to more

Fig. 4. Verifier Feedback Using `true` as Invariant.

quickly (i.e., with fewer attempts) arrive at a proper invariant as compared to some of the other submitted incorrect invariants. In other words, not all failures to verify show the same level of misunderstanding. We return to this topic in detail in a later subsection.

The use of mathematical strings to model a queue abstractly enables the queue's specification to use string notations and the verifier to use results from a theory of strings to prove code correctness. This functionality is critical to the formation and use of a loop invariant, which serves as an internal contract necessary for verifying the correctness of operations containing loops.

7.4. Experimental Methods

7.4.1. Experimental Overview

The experiment was conducted in a required third year course on software engineering in which students completed a set of activities on invariants using the online verifier in a class period. All invariant attempts collected and analyzed in this section are self-reported. Data used for analysis in this section was collected from a total of 272 students over three semesters: F3 (101 students), S3 (86), and F4 (85). Students worked on the activities in self-selected groups of two or three, totaling 105 groups.

7.4.2. An In-class Student Invariant Activity

For this activity, students were given a formal specification and code for an `Append` operation, Fig. 4, whose specified goal is to append one queue onto the end of another queue. Only an invariant for proving correctness is missing. Whereas classical loop invariant activities involving arrays, for example, involve the use of quantifiers, these activities are set up to factor out that additional complexity.

In the description of queues on which the `Append` operation is based, mathematical strings are used to model a queue abstractly and to capture the importance of ordering. Importantly, this mathematical modeling has nothing to do with how queues might be represented and implemented, such as using arrays, vectors, or linked lists.

When conceptualized abstractly as a string, a queue (Q) containing the following entries, \heartsuit, \clubsuit would be seen as $Q = \langle \heartsuit, \clubsuit \rangle$. When `Enqueue` is called with Q and \diamond , abstractly it is adding the diamond to the right side of the string, resulting in $Q = \langle \heartsuit, \clubsuit, \diamond \rangle$. The removal of an entry by `Dequeue` conceptually will remove an entry from left side of the string, resulting in $Q = \langle \clubsuit, \diamond \rangle$. Together they uphold the First-In-First-Out nature of a queue.

The caller is responsible for the **requires** clause where the combining of the two queues, P and Q , will not cause the modified queue P to violate the length constraint of 3. Here, the bars surrounding a queue variable (e.g., P) denotes the string length operator. The **ensures** clause $P = \#P \circ \#Q$ states that the value of P at the end of the operation is the concatenation of the input value of P , with the input value of Q . Q is cleared, meaning that it is empty after the call to the operation.

One way to accomplish the task of appending two queues is to use a **while** loop to `Dequeue` one element from Q and `Enqueue` it to the end of P . The code is straightforward. The novel elements of this code are the introduction of a **maintaining** clause that lets a programmer specify a *loop invariant* and a **decreasing** clause that lets them specify a progress metric that is used to prove termination. These assertions (i.e., maintaining clause and decreasing clause) are automatically checked by the verifier to be legitimate before it uses them in proving code correctness. The verifier is sound (Cook et al., 2012a; Sitaraman et al., 2011).

In this example, students need to replace the assignment's default invariant `true` with a correct invariant – an assertion that will hold true at the beginning and end of every iteration, and with this particular implementation, is sufficient to guarantee that the **ensures** clause is met after the loop when `Append` terminates. This task

Table 13

Example Trace Using Data Abstraction to Discover and Check an Invariant

Iteration	P	Q	Check Invariant $P \circ Q = \#P \circ \#Q$
0	$\langle 1 \rangle$	$\langle 2, 3 \rangle$	$\langle 1 \rangle \circ \langle 2, 3 \rangle = \langle 1 \rangle \circ \langle 2, 3 \rangle$
1	$\langle 1, 2 \rangle$	$\langle 3 \rangle$	$\langle 1, 2 \rangle \circ \langle 3 \rangle = \langle 1 \rangle \circ \langle 2, 3 \rangle$
2	$\langle 1, 2, 3 \rangle$	$\langle \rangle$	$\langle 1, 2, 3 \rangle = \langle 1 \rangle \circ \langle 2, 3 \rangle$

requires identifying the relationship between *input* values #P and #Q and the *current* values of P and Q, which vary from iteration to iteration. An example trace is shown in Table 13 to illustrate how a student might discover an (intended) invariant.

7.5. Results

7.5.1. ERQ 4.1-Loop Invariant Basics: Building A Catalog of Difficulties

We have employed an iterative process to develop a catalog of student difficulties with respect to learning to reason about loop invariants. The process was complicated for multiple reasons. Due to the various forms of data collection, all data had to be converted to a digital format to allow for classification. In doing so, notes were included such as the number of attempts made. Furthermore, since the research involved collecting student explanations on different types of response media, the researcher had to make some judgment calls in order to make all data compatible for analysis. A second researcher then reviewed the transcripts, verifying the data obtained and the decisions made. This researcher then proceeded to use the F3 data as the foundation, grouping similar incorrect answers together. These categories were subsequently used to label the submitted responses from S3 and F4. The occurrence of incorrect answers that belonged to these categories across multiple semesters presented promising results for the classification.

A Catalog and Frequency of Difficulties. This initial classification was then shared with a cohort of experts to receive feedback and was subsequently revised to address potential needs. The grouping of similar incorrect answers was a good start for identifying problem areas, but it was found to be inadequate. This led to a final iteration for developing activity-specific categories and this is what is reported in the catalog of difficulties in Table 14. Data (comprising incorrect answers) from all three semesters were re-categorized using the catalog. While F4 has the most groups participating (represented by *n* in Table 14), the format of the collected results provided a confounding factor which is explored in Section 7.5.2 and could explain why fewer difficulties were recorded.

Verifier Feedback and Discussion. We found that when students submitted either the Invariant *Total Size is Conserved* or *Requires Clause* (marked in Table 14 by an asterisk), they were more likely to get the correct answer on the next attempt. Of the two responses, the former made sense, since conservation of total size is an invariant, but just not a sufficient one. The reason for the latter, if any, is not obvious. This led to an examination of the online verifier’s feedback.

Fig. 4 shows the feedback students get for verifying with the default invariant `true`. The first failed VC returned indicates that the **requires** clause for a call within the loop to `Enqueue` is not met. It is the second failed VC that concerns the **ensures** clause of the `Append` operation. If we assume that students follow traditional debugging techniques they would normally begin with resolving the first unproved VC. Upon further evaluation, we see that the invariant for conservation of size also results in satisfying the **requires** clause of `Enqueue`. So when the students attempt to verify the

Table 14
Catalog of Difficulties

General Category	Activity Specific	Activity Example	F3 n=35	S3 n=28	F4 n=43
Use of Loop Condition as Invariant	Loop Condition	$ Q \neq 0$	9	2	2
Use of Constraints as Invariant	Data Structure Constraints	$ P \leq 3$	4	3	0
Focus on What is Varying as Opposed to What is Invariant	$ P $ is Changing	$ P = \#P + 1$	11	2	6
Use of Irrelevant Math Operators	Use of Substring	$P = \#P \circ \text{Prt_Btwn}(0,1,Q)$	11	2	4
	Use of Reverse	$\#Q = \text{Reverse}(P) \circ Q$	5	0	2
Confusion of Data Structure Operations (e.g., Stacks vs Queues)	Incorrect Concatenation	$Q \circ P = \#P \circ \#Q$	2	1	2
Use of Requires Clause as Invariant	Requires Clause*	$ P + Q \leq 3$	8	6	7
Use of Ensures Clause as Invariant	Ensures Clause	$P = \#P \circ \#Q$	6	2	3
Ignoring Some Input Possibilities	Assumes $\#P$ is Empty	$\#Q = P \circ Q$	11	0	3
Underspecification	Total Size is Conserved*	$ P + Q = \#P + \#Q $	7	6	4
Other	21	3	5

code with either of these invariants, they notice that only the ensures clause of the code fails to prove, focusing their attention on where it needs to be focused. So the process of verification with the online tool works as might be expected.

7.5.2. ERQ 4.2-Loop Invariant Understanding: Student Conceptions

For ERQ 4.2, student responses were analyzed to determine if students were able to communicate a holistic understanding of the task at hand, and to identify any actionable information for future instruction as well as automated tutor development.

Response Medium. In the F3 and S3 experiments, a total of 62 student groups received a piece of paper at the start of the activity that contained the instructions mentioned above, and a table to use as a scaffold, as seen in Fig. 5. We found that the scaffolding encouraged students to record each attempt. Students also used the margins to perform traces such as seen in Table 13 using concrete numbers. Analysis of student responses was labor intensive and required some interpretation of what was written, making automation difficult.

For the F4 experiment, 43 student groups received the same instructions, with a free response text box for online submission as seen in Fig. 6. While easier for automated analysis, a reasonable question is what impact the online medium has on student response.

Level of Understanding. When the medium for response changed, we observed that student responses appeared to shift from explaining what individual pieces of invariant

Activity 1		
Steps or your thought process or how you used the feedback to modify your answer	Invariant	Success? Yes or No
The length of P + Q should remain constant at every step.	$ P + Q =$ $ #P + #Q $	No
P should add I from Q while Q removes I.	$P \circ LQ = LQ \circ Q$	No
At every stage of the loop, the current state of P + Q should equal the original P + the original contents of Q.	$P \circ Q = \#P \circ \#Q$	Yes

Fig. 5. PaperResponse.

Our thought process was that this one will empty q and put each value into P during the loop. We started off by using a tracing approach and started by saying it maintains that $\#Q = Q \circ P$, but then we realized this was only true in the case when P was empty so we came to the correct conclusion that maintaining $\#P \circ \#Q = P \circ Q$ and this proved everything

Fig. 6. Online Response.

attempts meant to a reflective analysis of their work, often explaining why a sufficient invariant worked. Fig. 6 demonstrates this shift in focus for the response. Rather than stating what “should” be happening “now”, this response reflects upon attempts made and explaining why they did not work.

To evaluate this observation, student responses were analyzed for the level of understanding communicated. We identified three levels of understanding; *None*, *Partial*, and *Holistic*. Figures 5 and 6 demonstrate what would be considered holistic understanding for each response medium.

We conducted an analysis of the significance of medium on observed student understanding. Due to the validity conditions for the Chi-Square test not being met (not every option has at least 10 observations), simulations of the MAD statistic for 100,000 shuffles were run to determine an approximate p-value. The higher proportion of student responses displaying holistic understanding in the online medium is significant with a p-value of 0.0095.

Table 15
Completed Additional Activities for F4 (Online)

Understanding	Count	Activity 2	Activity 3
Holistic	11	9/11 = 81.2%	9/11 = 81%
Partial	24	21/24 = 87.5%	19/24 = 79.2%
None	8	4/8 = 50%	3/8 = 37.5%

Table 15 illustrates that students who showed some understanding for Activity 1 made good progress on subsequent, slightly more complex activities, also involving queues. The importance of intervention during the first activity for students who need it is clear.

8. Discussion and Conclusions

The overarching goal of this research is to help students learn abstract reasoning at various levels with the aid of tools. Together, the tools help students practice abstraction and instructors identify and understand student difficulties. We have integrated symbolic reasoning into the curriculum for a software development course, and then built upon that foundation for reasoning with data abstractions in a third year software engineering course.

8.1. Symbolic Reasoning

Our research in Section 4 has shown it is possible to teach symbolic reasoning to a majority of students (at the 80% or a B grade level). Classroom instruction does have a statistically significant impact. One potential threat to validity of the results is that each semester had slight variations in the experimental conditions. Using Mann-Whitney U-test on the incoming student GPA, we found that the S1 and F2 student populations were comparable. While both S1 and F2 were taught by the same instructor, the S1 semester was the first time this instructor taught this course. This could indicate that the instructor was better equipped in the F2 semester than in the S1 semester to teach these concepts, and why students performed well on this particular exam. Upon further examination, the overall average course grade for the F2 semester is actually worse than the S1 semester, indicating that students did not receive an advantage due to teacher experience.

A follow-on study to the initial work reported in this section is an exploration of an automated way to analyze student vocalizations as they perform tool lessons. Initial results from the analysis from this follow-on study are promising Almazova *et al.* (2021).

8.2. Conditional Statements

The research presented in Section 5 examines student ability to translate the skills demonstrated in Section 4 and apply it to a more challenging application. After working with the online tool, students performed better on questions that involved the tracing of conditional statements. Students also reported being more confident with their ability to trace code with conditional statements using symbolic values.

One potential confounding factor for experiments, such as this one, is that only the performance of students who have completed both a pre-and post-quiz are considered. In this process, students who perform poorly and most likely needed the additional assistance, may be omitted due to not completing either the pre-or post-quiz. It is also possible that for the students whose performance improved, the improvement comes simply from additional practice through tool lessons, but not from the supporting system and feedback. A potential follow-up study could try to account for these factors with appropriate control and treatment groups.

Future work will study the preferred method of student response when asked to analyze and explain the purpose of code with conditional statements. Since earlier multiple choice questions, such as the ones discussed in Section 5, present students with three options for the types of reasoning answers, it will be interesting to look for trends as to the preferred method of student answers and impact of their chosen method on the correctness of their answers.

8.3. Learning Design-by-Contract Assertions

The research presented in Section 6 has helped identify common difficulties for students in learning to understand formal DbC assertions and trace symbolically over code involving data abstractions. While students have syntactic and semantic difficulties, two kinds of semantic difficulties need to be addressed: Those involving mathematical modeling used in describing contracts for operations and those in understanding how and why input values need to be distinguished. Such semantic difficulties are programming and specification-language neutral, and educators need to understand them in order to develop suitable interventions.

This research also confirms the importance of using a qualitative analysis to complement quantitative analysis. While quantitative analysis based on automated data collection is beneficial for developing tutors and interventions, qualitative analysis provides insights behind student misunderstandings that give rise to learning difficulties.

8.4. Learning to Develop Loop Invariants

In Section 7 we analyzed explanations of student reasoning to identify their difficulties. A catalog has been constructed to identify places to focus subsequent lessons. Analysis of the paper and online versions of student responses allow us to reach a qualitative conclusion that the medium impacts the response, and both kinds of responses are of interest. We have found more responses in the online medium to show a holistic understanding through a subjective analysis. However, that does not mean that those using the online medium lacked such an understanding. Rather, this is what we are able to say from the responses. The online medium, which makes automation easier, is an effective option for collecting actionable information as well. A threat to

validity is that our results depend on students accurately reporting their attempts and reasons.

We have developed the process of identifying difficulties in learning loop invariants in such a way that it is a useful starting point to generalize and possibly guide the design of other systems for helping students to learn formal topics, such as discrete structures and automata theory.

8.5. *Conclusions*

We have found that students are able to successfully learn how to do symbolic reasoning, given a sequence of assignment statements. Going beyond the basics, we have found that students are also able to learn to reason symbolically about code involving conditional statements. Using that knowledge, they are successful in learning how to use formal specification of data abstractions. They are able to progress further and develop loop invariants for code involving data abstractions. In every case, the tools have aided in student learning and in helping pinpoint difficulties at a fine-grain level.

Data that continues to be collected with our tools will help us focus our future work on providing targeted feedback. The next version of the tool, a more general human-centric reasoning system, aims to take on the role of an intelligent tutor by providing tailored feedback to students and creating individualized learning experiences. Additionally, a variety of data will be processed in real time to assist instructors with identifying students that may need help, or specific sub-concepts that may need additional instruction. We need students to understand not only specific technical details but also the larger purpose of abstraction and reasoning. Ultimately, the tutor's aim is not to replace an instructor, but to be of assistance to everyone involved. Teasing out the benefits of using a tool in conjunction with instruction is among our current educational research questions. Exploring the use of the tool and the benefits of related symbolic reasoning activities at a diverse set of institutions are among our other ongoing research efforts.

Acknowledgments

We thank all members of the RESOLVE Software Research Group (RSRG) at Clemson University and The Ohio State University for their helpful comments throughout the course of this work.

Funding

This research is funded in part by US National Science Foundation grants DUE-1914667, DUE-1914820, DUE-1915088, and DUE-1915334.

References

- Aleven, V., McLaren, B.M., Sewall, J. (2009). Scaling up programming by demonstration for intelligent tutoring systems development: An open-access web site for middle school mathematics learning. *IEEE transactions on learning technologies*, 2(2), 64–78.
- Almazova, N., Hallstrom, J., Fowler, M., Hollingsworth, J., Kraemer, E., Sitaraman, M., Washington, G. (2021). Automated Analysis of Student Verbalizations in Online Learning Environments. In: *International Symposium on Emerging Technology for Education*. SETE Springer, Cham.
- Azevedo, R., Bernard, R.M. (1995). A meta-analysis of the effects of feedback in computer-based instruction. *Journal of Educational Computing Research*, 13(2), 111–127.
- Bhattacharya, P., Guo, M., Tao, L., Wu, B., Qian, K., Palmer, E.K. (2011). A cloud-based cyberlearning environment for introductory computing programming education. In: *2011 IEEE 11th International Conference on Advanced Learning Technologies*, pp. 12–13. IEEE.
- Bucci, P., Long, T.J., Weide, B.W. (2001). Do We Really Teach Abstraction? In: *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '01. ACM, New York, NY, USA, pp. 26–30. 1-58113-329-4. <https://doi.org/10.1145/364447.364531>
- Carbonneau, K.J., Marley, S.C., Selig, J.P. (2013). A meta-analysis of the efficacy of teaching mathematics with concrete manipulatives. *Journal of Educational Psychology*, 105(2), 380.
- Chen, C.-Y., Pedersen, S., Murphy, K.L. (2011). Learners' perceived information overload in online learning via computer-mediated communication. *Research in Learning Technology*, 19(2).
- Cook, C.T., Harton, H., Smith, H., Sitaraman, M. (2012a). Specification engineering and modular verification using a web-integrated verifying compiler. In: *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1379–1382. IEEE.
- Cook, C.T., Drachova, S., Hallstrom, J.O., Hollingsworth, J.E., Jacobs, D.P., Krone, J., Sitaraman, M. (2012b). A systematic approach to teaching abstraction and mathematical modeling. In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pp. 357–362.
- Cook, C.T., Drachova-Strang, S.V., Sun, Y.-S., Sitaraman, M., Carver, J.C., Hollingsworth, J. (2013). Specification and reasoning in SE projects using a Web IDE. In: *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*, pp. 229–238. IEEE.
- Cook, M., Fowler, M., Hallstrom, J.O., Hollingsworth, J.E., Schwab, T., Sun, Y., Sitaraman, M. (2018). Where exactly are the difficulties in reasoning logically about code? experimentation with an online system. In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018, Larnaca, Cyprus, July 02–04, 2018*, pp. 39–44. <https://doi.org/10.1145/3197091.3197133>
- De Bock, D., Deprez, J., Van Dooren, W., Roelens, M., Verschaffel, L. (2011). Abstract or concrete examples in learning mathematics? A replication and elaboration of Kaminski, Sloutsky, and Heckler's study. *Journal for research in Mathematics Education*, 42(2), 109–126.
- Drachova, S.V., Hallstrom, J.O., Hollingsworth, J.E., Krone, J., Pak, R., Sitaraman, M. (2015). Teaching Mathematical Reasoning Principles for Software Correctness and Its Assessment. *TOCE*, 15(3), 15–11522. <https://doi.org/10.1145/2716316>
- Forišek, M. (2006). On suitability of programming competition tasks for automated testing. In: *International Workshop, Perspectives on Computer Science Competitions for (High School) Students*, pp. 25–28.
- Fowler, M. (2021). *A Human-Centric System for Symbolic Reasoning About Code*. PhD thesis, Clemson University, Clemson, SC 29634.
- Fowler, M., Cook, M., Plis, K., Schwab, T., Sun, Y., Sitaraman, M., Hallstrom, J.O., Hollingsworth, J.E. (2019). Impact of Steps, Instruction, and Motivation on Learning Symbolic Reasoning Using an Online Tool. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27–March 02, 2019*. ACM, New York, NY, USA, pp. 1039–1045. <https://doi.org/10.1145/3287324.3287401>
- Fowler, M., Kraemer, E.T., Sun, Y.-S., Sitaraman, M., Hallstrom, J.O., Hollingsworth, J.E. (2020). Tool-Aided Assessment of Difficulties in Learning Formal Design-by-Contract Assertions. In: *Proceedings of the 4th European Conference on Software Engineering Education*, pp. 52–60.
- Ginat, D. (2014). On Inductive Progress in Algorithmic Problem Solving. *Olympiads in Informatics*, 8.
- Gries, D. (1981). *The science of programming*. Springer-Verlag.
- Guo, P. <http://www.pythontutor.com/java.html#mode=edit>
- Hallstrom, J.O., Hochrine, C., Sorber, J., Sitaraman, M. (2014). An ACM 2013 exemplar course integrating

- fundamentals, languages, and software engineering. In: *Proceedings of the 45th ACM technical symposium on Computer science education*, pp. 211–216.
- Henderson, P.B. (2003). Mathematical Reasoning in Software Engineering Education. *Commun. ACM*, 46(9), 45–50. <https://doi.org/10.1145/903893.903919>
- Herman, G.L., Kaczmarczyk, L., Loui, M.C., Zilles, C. (2008). Proof by Incomplete Enumeration and Other Logical Misconceptions. In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER '08. ACM, New York, NY, USA, pp. 59–70. 978-1-60558-216-0. <https://doi.org/10.1145/1404520.1404527>
- Heym, W.D., Sivilotti, P.A.G., Bucci, P., Sitaraman, M., Plis, K., Hollingsworth, J.E., Krone, J., Sridhar, N. (2017). Integrating Components, Contracts, and Reasoning in CS Curricula with RESOLVE: Experiences at Multiple Institutions. In: *30th IEEE Conference on Software Engineering Education and Training, CSEET 2017, Savannah, GA, USA, November 7–9, 2017*, pp. 202–211. <https://doi.org/10.1109/CSEET.2017.40>
- Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., Margaria, T. (2008). Software engineering and formal methods. *Communications of the ACM*, 51(9), 54–59.
- Hinds, P.J., Patterson, M., Pfeffer, J. (2001). Bothered by abstraction: The effect of expertise on knowledge transfer and subsequent novice performance. *Journal of applied psychology*, 86(6), 1232.
- Kaminski, J.A., Sloutsky, V.M., Heckler, A.F. (2008). The advantage of abstract examples in learning math. *SCIENCE-NEW YORK THEN WASHINGTON-*, 320(5875), 454.
- Kumar, R., Rosé, C.P., Wang, Y.-C., Joshi, M., Robinson, A. (2007). Tutorial dialogue as adaptive collaborative learning support. *Frontiers in artificial intelligence and applications*, 158, 383.
- Li, C., Dong, Z., Untch, R., Chasteen, M., Reale, N. (2011). Peerspace-an online collaborative learning environment for computer science students. In: *2011 IEEE 11th International Conference on Advanced Learning Technologies*, pp. 409–411. IEEE.
- Lister, R., Fidge, C., Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin*, 41(3), 161–165.
- Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B., Thomas, L. (2004). A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In: *Working Group Reports from ITICSE on Innovation and Technology in Computer Science Education*. ITICSE-WGR '04. ACM, New York, NY, USA, pp. 119–150. <https://doi.org/10.1145/1044550.1041673>
- McCallum, W. (2008). Commentary on Kaminski et al, The Advantage of Abstract Examples in Learning Math, Science, April 2008. *Science*.
- Meyer, B. (1992). Applying “Design by Contract”. *Computer*, 25(10), 40–51. <https://doi.org/10.1109/2.161279>
- Moreno, R. (2004). Decreasing Cognitive Load for Novice Students: Effects of Explanatory versus Corrective Feedback in Discovery-Based Multimedia. *Instructional Science*, 32, 99–113. <https://doi.org/10.1023/B:TRUC.0000021811.66966.1d>
- O'Brien, C., Goldman, M., Miller, R.C. (2014). Java tutor: bootstrapping with python to learn Java. In: *Proceedings of the first ACM conference on Learning@ scale conference*, pp. 185–186.
- Paas, F., Van Gog, T., Sweller, J. (2010). Cognitive load theory: New conceptualizations, specifications, and integrated research perspectives. *Educational psychology review*, 22(2), 115–121.
- Price, T.W., Dong, Y., Lipovac, D. (2017). iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. ACM, New York, NY, USA, pp. 483–488. 978-1-4503-4698-6. <https://doi.org/10.1145/3017680.3017762>
- Priester, C., Sun, Y., Sitaraman, M. (2016). Tool-Assisted Loop Invariant Development and Analysis. In: *29th IEEE International Conference on Software Engineering Education and Training, CSEET 2016, Dallas, TX, USA, April 5–6, 2016*, pp. 66–70. <https://doi.org/10.1109/CSEET.2016.28>
- Sitaraman, M., Adcock, B.M., Avigad, J., Bronish, D., Bucci, P., Frazier, D., Friedman, H.M., Harton, H.K., Heym, W.D., Kirschenbaum, J., Krone, J., Smith, H., Weide, B.W. (2011). Building a push-button RESOLVE verifier: Progress and challenges. *FormalAsp.Comput.*, 23(5), 607–626. <https://doi.org/10.1007/s00165-010-0154-3>
- Sweller, J., Ayres, P., Kalyuga, S. (2011). *Cognitive Load Theory*. Springer New York, New York, NY.
- Wiggins, J.B., Boyer, K.E., Baikadi, A., Ezen-Can, A., Grafsgaard, J.F., Ha, E.Y., Lester, J.C., Mitchell, C.M., Wiebe, E.N. (2015). JavaTutor: An Intelligent Tutoring System That Adapts to Cognitive and Affective States During Computer Programming. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. ACM, New York, NY, USA, pp. 599–599. 978-1-4503-

- 2966-8. <https://doi.org/10.1145/2676723.2691877>
- Wouters, P., Paas, F., van Merriënboer, J.J.G. (2008). How to Optimize Learning From Animated Models: A Review of Guidelines Based on Cognitive Load. *Review of Educational Research*, 78(3), 645–675. <https://doi.org/10.3102/0034654308320320>
- Zaccai, D., Tagore, A., Hoffman, D., Kirschenbaum, J., Bainazarov, Z., Friedman, H.M., Pearl, D.K., Weide,
- B.W. (2014). Syrus: Providing Practice Problems in Discrete Mathematics with Instant Feedback. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education. SIGCSE '14*. ACM, New York, NY, USA, pp. 61–66. 978-1-4503-2605-6. <https://doi.org/10.1145/2538862.2538929>

M. Fowler is a doctoral candidate in the Human-Centered Computing program at Clemson University. She currently holds a B.S. in Computer Science and M.A. in Teaching Secondary Science. The results presented in this article are a part of her research carried out within her dissertation thesis. She currently works as a Computer Science teacher at Walhalla High School in South Carolina. Her academic interest is in computer science education.

J. Hallstrom serves as Executive Director of Florida Atlantic University's Institute for Sensing and Embedded Network Systems Engineering (I-SENSE) and is a Professor in the Department of Electrical Engineering and Computer Science. He holds a B.S. in Systems Analysis, an M.A. in Economics, and M.S. and Ph.D. degrees in Computer and In-formation Science. His work spans embedded, networked systems; software engineering; and computer science education. His work is currently supported through the National Science Foundation, NOAA (via CCU), the Knight Foundation, the City of West Palm Beach, and industry partners. He was previously supported through the NSF, DOE, EPA, USDA, NASA, industry partners, and other entities.

J. Hollingsworth is a member of the Computer Science and Software Engineering Department at Rose-Hulman Institute of Technology in Terre Haute, Indiana, where he serves as an associate professor and teaches undergraduate CS courses as well as supervises undergraduate researchers. His research has primarily focused on the scholarship of teaching and learning in the field of computer science with support from NSF DUE grants and has publications in various CS education related conferences and journals.

E. Kraemer is a Professor in the School of Computing at Clemson University. She holds a PhD from the College of Computing at Georgia Tech. Her research interests are in computer science education and human aspects of the software development process.

M. Sitaraman is a professor in the School of Computing at Clemson University. He is a principal investigator of the multi-institutional RESOLVE software engineering research and education effort that has been continuously funded by the US National Science Foundation for thirty years. Broadening participation in computing and helping students to learn how to reason correctly and soundly about the behavior of the code they write are among the goals of his group's CS education research. To date, the results have reached over 30,000 students and over 200 educators. Dr. Sitaraman is a co-editor of a Cambridge University Press book on Foundations of Component-Based Systems. His publications have appeared in ACM Transactions on Computing Education, Computer Science Education journal, and ACM SIGCSE and ACM ITiCSE conference proceedings.

Y. Sun is a lecturer in the School of Computing at Clemson University and teaches CS1, Software Development, and Introduction to Software Engineering. In addition to his teaching duties, he is also passionate about building software applications, mentoring students with their research implementations, exploring new fields in Computer Science, and developing automated grading tools to provide feedback to students while they work on a programming assignment.

J. Wang is an undergraduate Bachelor student of Rose-Hulman Institute of Technology majoring in Computer Science and Data Science. He is interested in machine learning and deep reinforcement learning.

G. Washington is an Assistant Professor at Howard University in Computer Science. At Howard, she runs the Affective Biometrics Lab and performs research on affective computing, computer science education, and biometrics. Currently, she is leading research that explores the role of how affect/emotion is displayed in imposter phenomena and in negatively impacting performance in computer science courses. She can be reached at atgloria.washington@howard.edu.