Tool-Aided Assessment of Difficulties in Learning Formal Design-by-Contract Assertions

Megan Fowler, Eileen T. Kraemer, Yu-Shan Sun, Murali Sitaraman*
Clemson University
Clemson, SC, USA
mefowle@clemson.edu

Jason O. Hallstrom
Florida Atlantic University
Boca Raton,, Florida, USA
jhallstrom@fau.edu

Joseph E. Hollingsworth
Rose-Hulman Institute of Technology
Terre Haute, Indiana, United States
hollings@rose-hulman.edu

ABSTRACT

Object-based development using design-by-contract (DbC) is broadly taught and practiced. Students must be able to read and write symbolic DbC assertions that are sufficiently precise and be able to use these assertions to trace program code. This paper summarizes the results of using an automated tool to pinpoint fine-grain difficulties students face in learning to symbolically trace code involving objects. The pilots were conducted in an undergraduate software engineering course. Quantitative results show that data collected by the tool can help to identify and classify learning obstacles. Qualitative findings help validate student misunderstandings underlying these difficulties. Analysis of exam questions helps understand the persistence of student learning to read and write simple assertions about code behavior. Together, these results provide directions for intervention.

CCS CONCEPTS

• **Applied computing → Computer-assisted instruction**; • **Theory of computation → Program reasoning**.

KEYWORDS

Contracts, Correctness, Objects, Online tool, Reasoning, Software Engineering, Specification, Verification, Tracing

ACM Reference Format:
Megan Fowler, Eileen T. Kraemer, Yu-Shan Sun, Murali Sitaraman, Jason O. Hallstrom, and Joseph E. Hollingsworth. 2020. Tool-Aided Assessment of Difficulties in Learning Formal Design-by-Contract Assertions. In *Proceedings of ECSEE '20: European Conference on Software Engineering Education (ECSEE'20).* ACM, New York, NY, USA, 9 pages. https://doi.org/**.****/*******.*******

1 INTRODUCTION AND MOTIVATION

The challenges students face in writing code and the pedagogical tools used to overcome them have received significant attention, e.g., [12, 21, 34]. The importance of code tracing to help beginners write and understand code has also received considerable attention [4, 7, 16, 24, 26, 27]. Unfortunately, helping students learn to trace code in the presence of objects –arguably among the most important CS topics– has received much less attention [3, 19, 25, 38]. Tracing method calls is ideally done abstractly so that students are not overwhelmed by internal data structures [1]. For example, if the string <△, ★, □> denotes the value of a stack object, and the left-most value, i.e., △ denotes the top value, after tracing over a call to `pop`, the stack's value will be <★, □>. Students should not confuse these abstract values with how the stack is represented internally (e.g., using an array or a linked structure); the focus here is on understanding the abstract behavior of operations like `pop`().

While tracing with concrete values (as in the above example) is a useful first step, ultimately, students must be able to understand code behavior involving arbitrary inputs which requires a symbolic approach. For example, if `#S` denotes an arbitrary initial value of a non-full stack, then after a call to `push`, followed by a call to `pop`, a student should be able to reason that the stack's value remains `#S`. While there is some prior work on tracing object behavior with abstractly specified design-by-contract (DbC) assertions, students must also be able to read and write formal assertions. There is pioneering work in teaching formal assertions for mathematical reasoning in CS education [6, 9–11, 14, 23, 35, 40], but few studies directly assess students' ability to read and interpret formal assertions and the specific challenges faced in writing them.

1.1 Software Engineering Context

The study presented here is part of a larger project that has involved over a thousand students at multiple institutions, over multiple years, covering a variety of undergraduate software development courses at varying levels [17, 20, 22]. This paper focuses on results from pilots in a software engineering course where students learn to read and write formal assertions. Students complete their exercises using an automated reasoning tool with a number of powerful features, assisting us in identifying specific obstacles students encounter. Our findings are based on 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.

While students in the course receive significant exposure to formal methods and develop non-trivial code based on instructor-supplied contracts (e.g., involving recursion, loops, and invariants), this paper does not cover those aspects of the course [5, 31]. Instead, using simple activities that involve tracing over calls on objects

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ECSEE'20, June 15–19, 2020, Seeon Monastery, Germany
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/**.****/*******.*******

using contracts expressed as formal assertions, this study aims to understand the basic difficulties students face in learning to reason precisely about code. 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 [15, 39].

1.2 Tool-Aided Assessment

A key aim of using the automated tool is to understand the specific learning difficulties students face when reasoning about code so that appropriate interventions may be developed [2, 8, 31]. 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.

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 (such as those mentioned in this paper) arise for beginning formal methods learners no matter what the language. At the same time, semantic difficulties, such as the one discussed in this paper concerning the distinction that between the input and output values of a parameter in an operation's postcondition, also arise across formal specification approaches.

1.3 Specific Objectives

The overarching research question is how to help students in learning to read and write formal DbC assertions. In addressing this larger question, the paper addresses the following specific questions (*ERQ*s).

ERQ 1: What common learning difficulties in reading and writing formal DbC assertions can be pinpointed with an automated tool and collected data?

ERQ 2: Which difficulties persist as students move through tool activities and later on a final exam when they do not have access to the tool?

This paper summarizes findings to these and related questions. It uses insights that qualitative analysis provides into the misunderstandings that cause the difficulties to validate the findings from automated analysis.

2 ACTIVITIES AND THE REASONING TOOL

There are many tutoring environments for beginning programmers, e.g., [13, 24, 30, 36, 38]. The tool employed in this paper shares characteristics with these efforts; it is automated and online, requires no installation, and collects student's responses as they work through activities [18, 32]). It is unique, however, in several important respects. Since the tool is concerned with helping students learn how to read and write assertions, it is not connected to a commercial programming language, nor does it check assertions using execution or answer keys. Instead, the tool uses a *verifying compiler* to prove correctness of symbolic assertions [32]. A variety of logically equivalent answers are possible for any activity. For the purposes of the research in this paper, the tool has been used less as a tutor and more of an aid to help us assess learning obstacles and develop interventions. More generally, the verifying compiler has been used extensively for a variety of tool-based reasoning activities in CS education; those details and citations have been suppressed here for the anonymous version.

For the work reported here, the tool is instrumented with six symbolic reasoning activities involving DbC assertions on objects. The activities are presented with scaffolding that includes instructions and reference materials, helping to reduce extraneous cognitive load [28, 33, 37]. A screenshot appears in Figure 1.

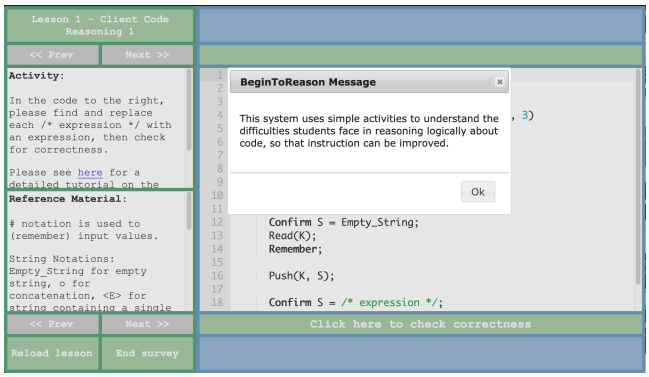

Figure 1: Online Reasoning Tool

2.1 DbC Assertion Basics

Given the focus on assertions, the code in each activity 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're used to (i.e., the distinguished argument is passed as an explicit parameter – `Push(K, S)`, instead of `s.push(k)`) [2].

Specifications rely on simple mathematical concepts, such as sets and strings, the latter being suitable for capturing the behavior of stack, queue, and list objects, where order is important. Notations used include `Empty_String`, `○` for string concatenation, `|S|` for string length, and `<E>` to denote a string containing a single entry.

Several of the activities involve `Stack` objects. 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> ○ #S`; i.e., the concatenation of the input entry (`#E`) and the input stack (`#S`).

2.2 Activities

For each activity, the **Reference Material** section (see Figure 1) reminds students of the mathematical symbols and operators that can be used to complete the exercise.

The first two activities ask students to consider formal contracts for operations, such as `Push` and `Pop`, and then reason about code involving those operations. To facilitate symbolic reasoning, as opposed to concrete values, a **`Remember`** construct is used. Students are instructed that this is not an executable statement; **`Remember`** denotes the point at which initial values are defined for variables; e.g., where `#K` is defined for some variable `K`.

While different formal approaches may differ in the details of reasoning about objects, such as in using mathematical sequences instead of mathematical strings to capture the behavior of objects such as stacks or in using a different way to capture object values at different states instead of using the # notation for input value, symbolic tracing over code with objects needs some version of the concepts presented in this paper.

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

Listing 1: Activity 1

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

To answer the question correctly, students must be able to read and understand the contracts. A correct answer is `S = <#K>`. Trivial answers, such as `S = S`, are not accepted. Other incorrect answers include the following.

- `S = <>` - Something has been pushed on the stack.
- `S = K` - Type mismatch; `S` is a string of entries; `K` is an integer.
- `S = <K>` - 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`.)

The second activity, shown in Figure 2, contains calls to `Push` and `Pop` and is slightly more involved. The student response shown in red is incorrect: After two pushes, followed by a pop, `S` will not equal `#S`. The second element of the response, `K = #J`, shown in green, is correct; `J` is pushed second, and then popped. The calls to `Push` and `Pop` are highlighted in green because the tool has proven that no overflows or underflows exist, though these details are not relevant for the present activity.

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.

```
13    Read(I);
14    Read(J);
15    Remember;
16
17    Push(I, S);
18    Push(J, S);
19    Pop(K, S);
20
21    Confirm S = #S;
22    Confirm K = #J;
```

Figure 2: Activity 2 Reasoning & Visual Feedback

3 DETAILS OF THE EXPERIMENT

The experiment was conducted in the spring of 2018, as part of a third-year CS course in software engineering. The course description includes program specification and reasoning as core topics and is required of all CS majors.

3.1 Online Tool

Seventy-one students interacted with the tool across two sections with different instructors. Students had prior experience with the tool as part of a prerequisite software development course [29]. 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.

3.2 Task-based Interviews

A diverse group of students, both academically and demographically, were asked to engage in task-based interviews involving their interactions with the online tool. Nine of the students volunteered to perform an interview during their personal time. One student had a learning disability. The sample details are as follows:

Course Grade	Gender	Ethnicity
3 A's	6 Male	5 White
4 B's	3 Female	2 African American
2 C's		1 Hispanic or Latino
		1 Asian

Participating students encountered the DbC activities for the first time during the task-based interview. Each student was asked to vocalize her thought process as she worked through the first three activities. All on-screen interactions were recorded. If a student was not vocal or did not explain why a problem was being reworked, the interviewer would give a vocal prompt. Each interview concluded when the student completed the three activities or after twenty minutes. In the latter case, students were asked to complete the activities post-interview, continuing from where they stopped.

3.3 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. The question for the first section of the course involved a "preemptable" queue that includes an `Inject` operation to insert a new entry at the front of a queue, in addition to the usual `Dequeue` and `Enqueue` operations. Contract specifications for all of the operations were included as reference material.

Here we focus on the first section of the course, which involved 43 students and two versions of the exam. The difference between the two versions involved an ordering difference between calls to `Inject` and `Enqueue` (with interspersed calls to `Dequeue`). Naturally, the correct answers differ between these two versions, but the complexity of the questions is similar.

The exam for the second section of the course involved a stack-based question, similar to those used in the online activities; student performance on this questions is excluded from the analysis.

4 ERQ 1: AUTOMATED ANALYSIS OF DIFFICULTIES

There were approximately 2000 responses generated by all of the students for the six DbC activities. Here we present the results of our analysis for the first activity, addressing *ERQ 1*.

4.1 Analysis of Activity 1 Responses

Seventy-one students attempted Activity 1 (Listing 1); sixty-four were successful (90%), moving on to subsequent activities. The remaining seven are candidates for targeted help.

Table 1 shows the distribution of student attempts at solving the activity. Some students tried again at a later date for additional practice, but these responses are not included in the analysis.

Table 1: Student-Response Distribution (Activity 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%

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 2 shows the top 10 most frequently given responses, all of which are incorrect, except the responses highlighted in green.

4.2 Example Semantic Difficulty: Neglecting Input Values

The second-most common incorrect response was `S = <K>`. (The correct answer is `S = <#K>`.) 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 = <#K>`. 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 an difficulty has been spotted, it is not clear what misunderstanding has caused it to arise, a topic addressed further in our qualitative analysis (Section 5).

Table 2: Top 10 Unique Responses for Activity 1

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

4.3 Example Syntactic Difficulty: Distinguishing Element vs. String

The most common incorrect response, `S = K`, occurred in approximately 11% of the 439 total responses. This type of error represents 32 of the 83 unique responses (39%), including, for instance, `S = #K`. Interestingly, while this type of error is common for the first activity, the difficulty mostly resolves in subsequent activities.

A discrete math instructor will appreciate the need for students to distinguish between an element `E` and a set containing `E`, i.e., {`E`}. This "stringification" difficulty (Table 3, line 2) is similar, reflecting a type mismatch, where students do not distinguish between an entry and a string containing an entry. While the underlying misunderstanding could be merely that of syntax, it is also possible that a learner has a deeper misunderstanding about strings.

4.4 Classifying Learning Difficulties

Table 4 summarizes our classification of difficulty types across the 83 unique incorrect responses for Activity 1. *Since a single response may contain more than one difficulty, the percentage column does not add up to 100% – but does reflect the percentage of time that the error occurred in the 83 responses.*

Table 3: Classification of Activity 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%

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

One aggregation of obstacles for Activity 1 concerns math notations. Here, the *Stringification*, *String Concatenation*, and *String Length* obstacles suggest a need for a better understanding of string notations. Interventions can usually occur together. The *Operation Contracts* difficulty suggests that there is a misunderstanding of one or more of the `Stack` operations' contracts. The specific response `S = #S`, which occurred ~3% (13/439) of the time, suggests that some students did not understand that the value of `S` is modified by `Push`. The *Under-Specification* difficulty reflects a student assertion that is true, but which does not capture the essence of the code. For example, the response `S = S` was entered four times for Activity 1 (4/439), stating that stack `S` at line 18 equals itself. And finally, the *Syntax and Other* category covers syntax and other problems, when the assertion includes one or more variables that are misspelled, not declared, or not applicable. In some cases, these problems are not easily classified from the given response.

5 VALIDATION THROUGH QUALITATIVE ANALYSIS

The qualitative analysis to address *ERQ 1* and to identify the misunderstandings underlying observed learning obstacles is based on task-based interviews of nine volunteers. The interviews generated two hours and 13 minutes of video. These videos were transcribed to contain both what the participant vocalized, as well as their tool interactions. The transcripts were reviewed and analyzed by two researchers independently. No demographic differences were noted.

The answers recorded by students were then classified based on the difficulty identified in Table 3. While only 4 of the difficulties appeared in the group who performed tasked-based interviews, and Two-Way ANOVA test revealed that there is not a statically significant difference between the proportions of difficulties for each group.

Table 4: Task-based Interview Activity 1 Difficulties

Difficulty	Occurrence	%
Input Values	27/44	61%
Stringification	12/44	27%
Under-Specification	1/44	2%
Variables	4/44	9%

5.1 Overcoming Misunderstandings

Table 5 shows the progress of Student No. 3, which is consistent with learning. She changed her answer twice before submitting, and with each change, moved closer to the correct answer.

Table 5: Student No. 3 Responses for Activity 1

No.	Response	Tool Response
1	**Confirm** `S = #S ○ K;`	
2	**Confirm** `S = K ○ #S;`	
3	**Confirm** `S = <K> ○ #S;`	Incorrect
4	**Confirm** `S = <#K> ○ #S;`	Correct

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` and was able to explain the purpose behind this action. When questioned 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 Activity without a submission. This particular student worked for 4 minutes before the first submission.

5.2 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 Activity 1, shown in Table 6. Student No. 8 appears to demonstrate the same growth as No. 3 for this activity.

Table 6: Student No. 8 Responses for Activity 1

No.	Response	Tool Response
1	**Confirm** `S = K;`	Incorrect
2	**Confirm** `S = <K> ○ #S;`	Incorrect
3	**Confirm** `S = <#K> ○ #S;`	Correct

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 Activity 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.

5.3 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.

For most students, learning occurred, and some misunderstandings disappeared as they progressed from the first to the second activity. 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 ERQ 2: PERSISTENCE OF DIFFICULTIES

As a prelude to discussing persistent syntactic and semantic difficulties students encounter in their initial introduction to formal contracts, we summarize the activities analyzed in this study.

6.1 Summary of All Activities

Unlike in Activities 1 and 2, where students need only understand given operation contracts, in Activities 3 through 6, they must write contracts for new operations (a **`requires`** or **`ensures`** clause) – a considerably more complex task.

```
7   Operation Mystery (updates S: Stack);
8     requires |S| >= 1;
9     ensures S = Empty_String;
10  Procedure
11    Var I: Integer;
12    Pop(I, S);
13  end Mystery;
```

Figure 3: Activity 3 Reasoning & Visual Feedback

The value of an automated tool, such as the one employed in this research to pinpoint difficulties, depends on the choice of activities. Learners will flounder if the activities are too hard or not focused. Activities 3 and 4 focus on writing requires clauses. Figure 3 shows Activity 3. This activity appears straightforward in that given the call to `Pop` in the code on a stack `S`, `Mystery` operation must include a precondition that `|S| >= 1`. While this answer is close, it is not correct. This is because the specification says that `Mystery` must ensure that stack `S` must be empty at the end. This end result is possible with a single call to `Pop` only if `|S| = 1`. This simple activity illustrates the interplay between pre- and postconditions, in general. Activity 4 involves an if statement, but it also asks the students to complete a precondition.

```
7   Operation Mystery(updates S, T: Stack);
8     requires |S| >= 1 and |T| <= 2;
9     ensures Reverse(S) o T = /* expression */;
10  Procedure
11    Var Temp: Integer;
12
13    Pop(Temp, S);
14    Push(Temp, T);
15  end Mystery;
```

Figure 4: Activity 6

Activities 5 and 6 concern writing ensures clauses. Figure 4 shows Activity 6. Here, a part of the postcondition is given, so the students need to complete only the right hand side of the equation. The correct answer is `Reverse(S) o T = Reverse(#S) o #T`. This is the effect of popping one stack and pushing on to another based on the specifications of `Pop` and `Push` given in the reference material.

6.2 Persistence of Difficulties in Activities

The six activities illustrate the fine-grain level at which various concepts can be presented to identify trouble spots, mostly automatically. Most students were able to successfully complete each of the six activities: Activity 1, 90%; Activity 2, 92%; Activity 3, 95%; Activity 4, 84%; Activity 5, 88%; and Activity 6, 85%. Table 7 summarizes the obstacles for all six activities. The first column is identical to the last column of Table 4. The additional columns are for Activities 2 through 6. Recall that since one response can reflect more than one obstacle, percentage columns do not add up to 100%. The table illustrates that whereas some syntactic difficulties, such as those having to do with string syntax disappear, others, such as ones pertaining to operation contracts persist.

One confounding factor is that while the obstacle classification is informative for Activity 1, it is insufficiently detailed for the more challenging, later activities. Math notation obstacles (rows labeled *Stringification*, *String Concatenation*, and *String Length*) were largely resolved beyond the first activity, except for *Stringification* in Activity 5. (A closer examination showed many such responses from a small number of students who were likely floundering.)

Whereas the math notation aggregation is not particularly useful for later activities, the *Operation Contracts* classification is overly broad. Further work is needed to more precisely classify the associated obstacles.

Table 7: Difficulties for All Activities

	Activities					
Difficulty	**1**	**2**	**3**	**4**	**5**	**6**
Input Values	37%	20%	24%	9%	31%	41%
Stringification	39%	13%	0%	4%	30%	5%
String Concatenation	12%	9%	0%	0%	0%	0%
String Length	2%	0%	2%	2%	1%	0%
Operation Contracts	11%	47%	76%	70%	57%	64%
Under Specification	14%	1%	0%	21%	8%	11%
Variables	5%	2%	11%	10%	19%	0%
Syntax and Other	19%	7%	17%	35%	14%	23%

The persistence seen in the table is summarized in a more aggregate form (e.g., the grouping of all string-syntax related errors) in the following illustration which shows which DbC contract-level trouble spots persist. One point of note here is the task of writing a requires clause got easier (going from activity 3 to 4), the task of writing an ensures clause (going from activity 5 to a more challenging activity 6) turned out to be harder.

Table 8: Pairwise Comparisons to Operation Contracts

Difficulty	**Dif**	**t Ratio**	**Prob > t**	**Lower 95%**	**Upper 95%**
Input Values	0.1880	2.79	0.0100	0.0491	0.3269
String Error	0.2336	3.46	0.0019	0.0947	0.3725
Under Specification	0.3103	4.6	0.0001	0.1713	0.4491
Variables	0.3212	4.76	<0.0001	0.1823	0.4601
Syntax and Other	0.1951	2.89	0.0078	0.0562	0.3340

6.3 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

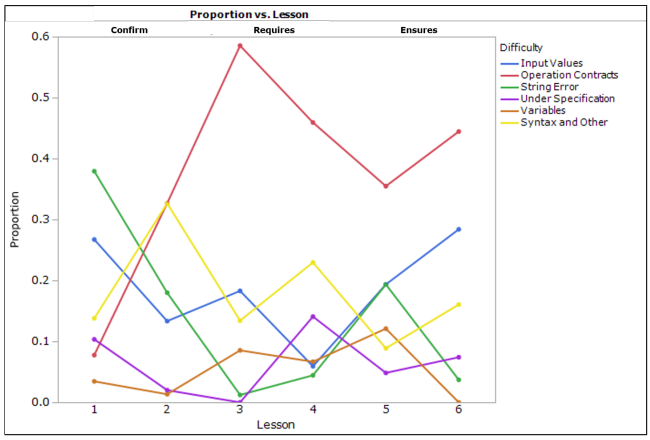

Figure 5: Persistence of Difficulties Across Six Tool Activities

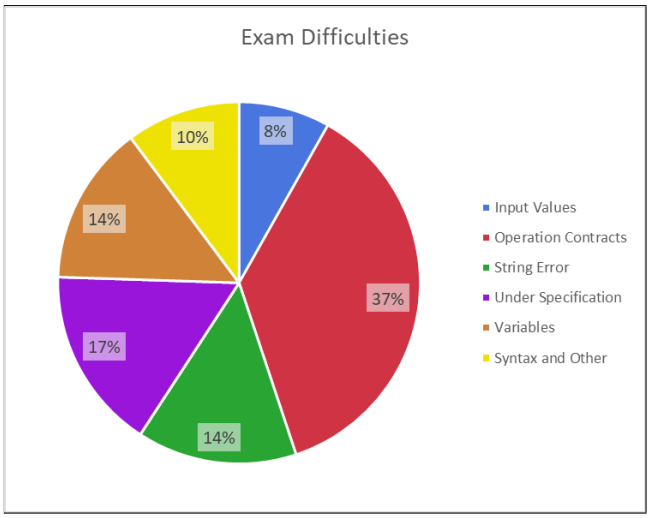

Figure 6: Persistence of Difficulties on the Exam

reasoning exam question closely resembled Activity 2 (Figure 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 (Figures 3 and 4). Students were required to develop an operation's pre- and post-conditions to reflect the behavior of its given code. 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 *ERQ 2*. The persistence of semantic difficulties seen through the analysis of tool activities is also seen to a degree in the illustration of answers on the final exam. One confounding factor is that manual grading might not have been as rigorous as the tool.

7 DIRECTIONS FOR INTERVENTIONS AND CONCLUSIONS

7.1 Interventions

Based on the fine-grain analysis, automated and manual interventions are possible. In some cases, an intervention may refer a student to reference material. In other cases, the intervention may be tailored to specific students, based on data collection and analysis.

Table 9: A Sequence of Responses from a Student

No.	Responses	Timestamp
1	**Confirm** S = <K>;	2018-04-25T18:56:20
2	**Confirm** S = <#S> ○ <K>;	2018-04-25T18:56:37
3	**Confirm** S = <K> ○ <#S>;	2018-04-25T18:56:55
4	**Confirm** S = <#K> ○ <#S>;	2018-04-25T18:57:12
5	**Confirm** S = <#S> ○ <#K>;	2018-04-25T18:57:33
6	**Confirm** S = <#K>;	2018-04-25T18:57:42

Table 9 shows a chronological sequence of responses given by a student for Activity 1. Notice that except for the missing #, the first response is identical to the last. A timely intervention that directed the student's attention to the reference material on the specification of `Push` or the notion of remembered values might have helped this student avoid floundering. At the same time, the intervening responses may indicate obstacles that may arise later.

In another case, a student working over a six-minute time span on Activity 1 came up with 16 incorrect responses and never successfully finished the activity. This student did not attempt any of the other activities. The first response, `S = Empty_String`, and subsequent responses reveal the use of a "guess and check" approach. For this student, an automated tutor could use the response sequence to document the extent of misunderstanding, which could later be used by an instructor to provide needed help.

7.2 Conclusions and Ongoing Research

The research presented in this paper has helped identify common difficulties for students in learning to understand formal DbC assertions and trace symbolically over code involving objects. While students have syntactic and semantic difficulties, the semantic ones involving mathematical modeling used in describing contracts for operations and understanding how and why input values need to be distinguished that persist are important to address. Such semantic difficulties are programming and specification-language neutral, and educators need to understand them to develop suitable interventions.

Data collection in subsequent semesters has provided additional information with regard to student performance and interaction with the tool. Rather than working independently, students completed the task in pairs or even groups of three. Early analysis of captured video has shown an increase of qualitative data collected via student verbal exchanges when engaged with a peer as opposed to working alone and self reporting their thought process. Future analysis will investigate the roles assumed by students as well as the level of content mastery achieved by each student. Concurrent

research is also being conducted regarding student ability to read and write more complex assertions, such as loop invariants. Qualitative error analysis is also in progress to assist in the development of the most effective instructional strategies in all cases.

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 obstacles. Students are able to demonstrate the ability to read and write assertions. Future directions involve development of intelligent tutors, informing instructors and providing tailored guidance for students with the most difficulties.

ACKNOWLEDGMENTS

This research is funded in parts by the U. S. National Science Foundation grants DUE-1915334, DUE-1914667, and DUE-1915088 .

REFERENCES

[1] Paolo Bucci, Timothy J. Long, and Bruce W. Weide. 2001. Do We Really Teach Abstraction?. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education* (Charlotte, North Carolina, USA) *(SIGCSE '01)*. ACM, New York, NY, USA, 26–30. https://doi.org/10.1145/364447.364531

[2] Michelle Cook, Megan Fowler, Jason O. Hallstrom, Joseph E. Hollingsworth, Tim Schwab, Yu-Shan Sun, and Murali Sitaraman. 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*. 39–44. https://doi.org/10.1145/3197091.3197133

[3] Stephen Cooper, Wanda Dann, Randy Pausch, and Randy Pausch. 2003. Teaching Objects-first in Introductory Computer Science. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Navada, USA) *(SIGCSE '03)*. ACM, New York, NY, USA, 191–195. https://doi.org/10.1145/611892.611966

[4] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 164–172.

[5] Svetlana V. Drachova, Jason O. Hallstrom, Joseph E. Hollingsworth, Joan Krone, Richard Pak, and Murali Sitaraman. 2015. Teaching Mathematical Reasoning Principles for Software Correctness and Its Assessment. *TOCE* 15, 3 (2015), 15:1–15:22. https://doi.org/10.1145/2716316

[6] Ingo Feinerer and Gernot Salzer. 2009. A comparison of tools for teaching formal software verification. *Formal Asp. Comput.* 21, 3 (2009), 293–301. https://doi.org/10.1007/s00165-008-0084-5

[7] Sue Fitzgerald, Beth Simon, and Lynda Thomas. 2005. Strategies That Students Use to Trace Code: An Analysis Based in Grounded Theory. In *Proceedings of the First International Workshop on Computing Education Research* (Seattle, WA, USA) *(ICER '05)*. ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/1089786.1089793

[8] Megan Fowler, Michelle Cook, Kevin Plis, Tim Schwab, Yu-Shan Sun, Murali Sitaraman, Jason O. Hallstrom, and Joseph E. Hollingsworth. 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, 1039–1045. https://doi.org/10.1145/3287324.3287401

[9] Susan L. Gerhart, Eric C. Hehner, and Harlan D. Mills. 1983. Teaching Formal Methods for Program Development and Verification (Panel Session). In *Proceedings of the Fourteenth SIGCSE Technical Symposium on Computer Science Education* (Orlando, Florida, USA) *(SIGCSE '83)*. ACM, New York, NY, USA, 50–. https://doi.org/10.1145/800038.801011 Moderator-Turner, A. Joe.

[10] J. Paul Gibson. 2008. Weaving a Formal Methods Education with Problem-Based Learning. In *Leveraging Applications of Formal Methods, Verification and Validation*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 460–472.

[11] Paul Gibson and Dominique Méry. 1998. Teaching Formal Methods: Lessons to Learn. In *Proceedings of the 2Nd Irish Conference on Formal Methods* (Cork, Ireland) *(IW-FM'98)*. BCS Learning & Development Ltd., Swindon, UK, 56–68. http://dl.acm.org/citation.cfm?id=2227414.2227418

[12] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2008. Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. ACM, New York, NY, USA, 256–260. https://doi.org/10.1145/1352135.1352226

[13] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. ACM, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[14] Peter B. Henderson. 2003. Mathematical Reasoning in Software Engineering Education. *Commun. ACM* 46, 9 (Sept. 2003), 45–50. https://doi.org/10.1145/903893.903919

[15] Geoffrey L. Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2008. Proof by Incomplete Enumeration and Other Logical Misconceptions. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/1404520.1404527

[16] Matthew Hertz and Maria Jump. 2013. Trace-based Teaching in Early Programming Courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. ACM, New York, NY, USA, 561–566. https://doi.org/10.1145/2445196.2445364

[17] Wayne D. Heym, Paolo A. G. Sivilotti, Paolo Bucci, Murali Sitaraman, Kevin Plis, Joseph E. Hollingsworth, Joan Krone, and Nigamanth Sridhar. 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, CSEE&T 2017, Savannah, GA, USA, November 7-9, 2017*. 202–211. https://doi.org/10.1109/CSEET.2017.40

[18] RESOLVE Online Tool Introduction. [n.d.].

[19] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. ACM, New York, NY, USA, 107–111. https://doi.org/10.1145/1734263.1734299

[20] Cazembe Kennedy and Eileen Kraemer. 2018. What Are They Thinking?: Eliciting Student Reasoning About Troublesome Concepts in Introductory Computer Science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research, (Koli Calling âĂŹ18)*. 1–10. https://doi.org/10.1145/3279720.3279728

[21] Cazembe Kennedy and Eileen T. Kraemer. 2019. Qualitative Observations of Student Reasoning: Coding in the Wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. ACM, New York, NY, USA, 224–230. https://doi.org/10.1145/3304221.3319751

[22] Eileen T. Kraemer, Murali Sitaraman, and Joseph E. Hollingsworth. 2018. An Activity-Based Undergraduate Software Engineering Course to Engage Students and Encourage Learning. In *Proceedings of the 3rd European Conference of Software Engineering Education, ECSEE 2018, Seeon Monastery, Bavaria, Germany, June 14-15, 2018*, Jürgen Mottok (Ed.). ACM, 18–25. https://doi.org/10.1145/3209087.3209100

[23] Gregory Kulczycki, Murali Sitaraman, Nigamanth Sridhar, and Bruce W. Weide. 2016. Panel: Engage in Reasoning with Tools. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, TN, USA, March 02 - 05, 2016*. ACM, New York, NY, USA, 160–161. https://doi.org/10.1145/2839509.2844657

[24] Amruth N. Kumar. 2013. A Study of the Influence of Code-tracing Problems on Code-writing Skills. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (Canterbury, England, UK) *(ITiCSE '13)*. ACM, New York, NY, USA, 183–188. https://doi.org/10.1145/2462476.2462507

[25] Danny B. Lange and Yuichi Nakamura. 1997. Object-Oriented Program Tracing and Visualization. *Computer* 30, 5 (May 1997), 63–70. https://doi.org/10.1109/2.589912

[26] Colleen M. Lewis. 2014. Exploring Variation in Students' Correct Traces of Linear Recursion. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. ACM, New York, NY, USA, 67–74. https://doi.org/10.1145/2632320.2632355

[27] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 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* (Leeds, United Kingdom) *(ITiCSE-WGR '04)*. ACM, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[28] Roxana Moreno. 2004. Decreasing Cognitive Load for Novice Students: Effects of Explanatory versus Corrective Feedback in Discovery-Based Multimedia. *Instructional Science* 32 (01 2004), 99–113. https://doi.org/10.1023/B:TRUC.0000021811.66966.1d

[29] Omitted. 2019.

[30] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. ACM, New York, NY, USA, 483–488. https:

//doi.org/10.1145/3017680.3017762

[31] Caleb Priester, Yu-Shan Sun, and Murali Sitaraman. 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*. 66–70. https://doi.org/10.1109/CSEET.2016.28

[32] Murali Sitaraman, Bruce M. Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M. Friedman, Heather K. Harton, Wayne D. Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith, and Bruce W. Weide. 2011. Building a push-button RESOLVE verifier: Progress and challenges. *Formal Asp. Comput.* 23, 5 (2011), 607–626. https://doi.org/10.1007/s00165-010-0154-3

[33] John Sweller, Paul Ayres, and Slava Kalyuga. 2011. *Cognitive Load Theory*. Springer New York, New York, NY.

[34] Allison Elliott Tew. 2010. *Assessing Fundamental Introductory Computing Concept Knowledge in a Language Independent Manner*. Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA, USA. Advisor(s) Guzdial, Mark. AAI3451304.

[35] Henry MacKay Walker. 1998. Modules to introduce assertions and loop invariants informally within CS1: experiences and observations. *SIGCSE Bulletin* 30, 2 (1998), 31–35. https://doi.org/10.1145/292422.292437

[36] Joseph B. Wiggins, Kristy Elizabeth Boyer, Alok Baikadi, Aysu Ezen-Can, Joseph F. Grafsgaard, Eun Young Ha, James C. Lester, Christopher M. Mitchell, and Eric N. Wiebe. 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* (Kansas City, Missouri, USA) *(SIGCSE '15)*. ACM, New York, NY, USA, 599–599. https://doi.org/10.1145/2676723.2691877

[37] Pieter Wouters, Fred Paas, and Jeroen J. G. van Merriënboer. 2008. How to Optimize Learning From Animated Models: A Review of Guidelines Based on Cognitive Load. *Review of Educational Research* 78, 3 (2008), 645–675. https://doi.org/10.3102/0034654308320320 arXiv:https://doi.org/10.3102/0034654308320320

[38] Benjamin Xie, Greg L. Nelson, and Andrew J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, New York, NY, USA, 344–349. https://doi.org/10.1145/3159450.3159527

[39] Diego Zaccai, Aditi Tagore, Dustin Hoffman, Jason Kirschenbaum, Zakariya Bainazarov, Harvey M. Friedman, Dennis K. Pearl, and Bruce W. Weide. 2014. Syrus: Providing Practice Problems in Discrete Mathematics with Instant Feedback. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. ACM, New York, NY, USA, 61–66. https://doi.org/10.1145/2538862.2538929

[40] Daniel Zingaro. 2008. Another Approach for Resisting Student Resistance to Formal Methods. *SIGCSE Bull.* 40, 4 (Nov. 2008), 56–57. https://doi.org/10.1145/1473195.1473220