Tool-Aided Loop Invariant Development: Insights into Student Conceptions and Difficulties

Megan Fowler, Eileen Kraemer,
Murali Sitaraman
Clemson University
mefowle@clemson.edu

Joseph E. Hollingsworth
Rose-Hulman Institute of Technology
hollings@rose-hulman.edu

ABSTRACT

To develop code that meets its specification and is verifiably correct, such as in a software engineering course, students must be able to understand formal contracts and annotate their code with assertions such as loop invariants. To assist in developing suitable instructor and automated tool interventions, this research aims to go beyond simple pre- and post-conditions and gain insight into student learning of loop invariants involving objects. As students develop suitable loop invariants for given code with the aid of an online system backed by a verification engine, each student attempt, either correct or incorrect, was collected and analyzed automatically, and catalogued using an iterative process to capture common difficulties. Students were also asked to explain their thought process in arriving at their answer for each submission. The collected explanations were analyzed manually and found to be useful to assess their level of understanding as well as to extract actionable information for instructors and automated tutoring systems. Qualitative conclusions include the impact of the medium.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Software and its engineering** → **Formal methods**; **Software verification**.

KEYWORDS

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

ACM Reference Format:
Megan Fowler, Eileen Kraemer,, Murali Sitaraman, and Joseph E. Hollingsworth. 2021. Tool-Aided Loop Invariant Development: Insights into Student Conceptions and Difficulties. In *26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2021), June 26-July 1, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3430665.3456351

1 INTRODUCTION AND MOTIVATION

One central aspect of software engineering is development of software that functions correctly according to its specification. The development of correct software requires students to understand formal contracts and to annotate their code with assertions such as pre-conditions, post-conditions and loop invariants to show that their code works informally or verifies formally [7, 10]. 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 [9].

Almost all modern verification tools such as those summarized in [13] expect software developers to supply suitable invariants. This is because systems aimed at discovering them automatically (e.g., [5, 17]), are limited in what they can infer. There is little pedagogical content knowledge (PCK), not only of content but also of students' prior knowledge, common difficulties and effective pedagogy[24], on teaching loop invariants. Automated collection and reporting of such student difficulties can help instructors with less experience to more quickly develop this aspect of PCK.

When students learn to write loop invariants for iterative code, they can achieve a level of understanding not possible otherwise [8]. Towards helping them achieve this goal, this paper addresses the following specific educational research questions (*ERQ*s).

ERQ 1: What common difficulties do students face, specifically as it concerns developing loop invariants?

ERQ 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?

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 2 is answered using a qualitative analysis of whether written responses show holistic, partial, or no understanding. Additionally, for ERQ 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 2 or 3 students, with a grand total of 272 students having given consent.

While post-hoc analysis of student responses guide interventions looking forward, automated analysis can facilitate immediate feedback by an instructor or a tool. The process and results of using an automated, online tool to build a catalog of difficulties and for identifying actionable information for loop invariants are more generally applicable to other topics such as discrete math and automata theory, because they also rely on gradual acquisition of skills required to wield logic and write assertions.

ITiCSE 2021, June 26-July 1, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8214-4/21/06…$15.00
https://doi.org/10.1145/3430665.3456351

2 RELATED WORK

Many papers discuss the importance of formal methods despite it not being particularly well represented in undergraduate coursework, and sometimes met with student resistance[4, 6, 9, 26, 30]. This has sparked the development of various methods to help teach students formal methods, which includes loop invariants. A previous study summarizes some of the most common student attempts in developing a sufficient loop invariant [22]. Some aspects to consider are the exercises used and making them applicable to the real world, minimizing the reliance on math, using languages that are familiar to students, and the use of tools [30].

Studies, such as the one in this paper, are concerned with semantic student difficulties concerning logical reasoning about the behavior of a piece of code. Some of this research is on the topic of student misconceptions about introductory programming constructs [23]. Whereas Tew's work focuses on the more basic constructs [28], the work of [11, 12] focuses on more advanced introductory concepts. Work has also been conducted on analyzing student ability to write correct code using iteration and recursion [16, 19].

A topic related to this paper is the use of tracing in student understanding of code[3, 18], because one approach students use to identify an invariant for a loop is to trace over the loop code multiple times. Students who cannot trace code often struggle to explain code [29]. A tool-based tracing study in [27] has analyzed collected incorrect responses to categorize various student difficulties with data structures and language-specific constructs.

Student explanations have been analyzed for various purposes. Difficulties with data structures based on think-aloud transcript analysis is the focus of [31]. The pioneering work in [1] argues that an explanation should demonstrate a high level of abstraction, correctness, and low ambiguity. The potential for automation through machine learning is explored in [21].

Many tools exist for verification purposes [6, 13, 15]. In [14], an IDE-based theorem prover is discussed as a means to help students learn how to write formal mathematical proofs of problems from theory of computation. This tool provided scaffolding for students while writing proofs and the authors note that students required little training, an experience that is similar to ours.

3 DETAILS OF THE EXPERIMENT

3.1 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. Students were instructed to "Be deliberate and document your thought process every time before you check an invariant on the system." All attempts collected and analyzed in this paper are self reported. Data used for analysis in this paper was collected from a total of 272 students over three semesters: Fall 2017 (101 students), Spring 2018 (86), and Fall 2018 (85). Students worked on the activities in self-selected groups of two or three, totaling 105 groups.

An ANOVA test (Figure 1) performed on the final course grades across all three semesters indicated *no* significant difference, so student performance in course activities in the three semesters are comparable.

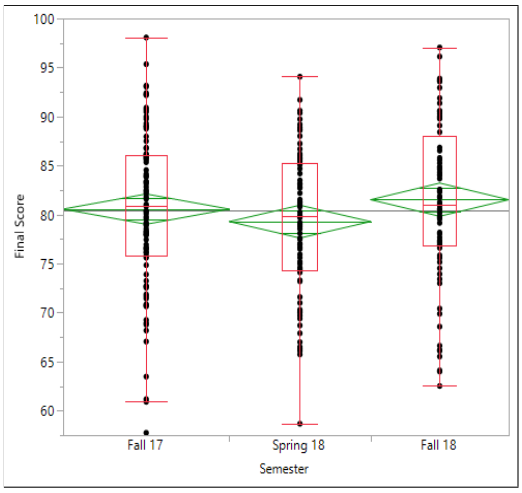


Figure 1: Oneway ANOVA Boxplot of Final Course Grades for Three Semesters

3.2 Online Verifier

The online system used in this experiment is backed by the RESOLVE formal verification engine [10, 25]. Using the verifier requires understanding and use of design-by-contract (DbC) assertions [20]. In DbC, the `requires` clause acts as a precondition meaning that it is the responsibility of the caller of an operation. The `ensures` clause is the corresponding postcondition that tells the caller what to expect from the operation and tells the called operation's implementer what the operation must guarantee.

Figure 2 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 code in Figure 2, two VCs fail to prove, as explained in Section 5.2.

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 loop invariants, which serves as an internal contract necessary to verify the correctness of operations containing loops.

3.3 An In-class Student Invariant Activity

Before working on invariant activities, students had learned the basics of using the verifier. Prior to student interaction with the verifier, the instructor led students through various introductory verification activities followed by an example verification of code involving a loop. The RESOLVE language used by the verifier allows formal specification [10]. Coding in this language has syntactic differences from popular languages, such as Java, but students at this level have little difficulty with those differences.

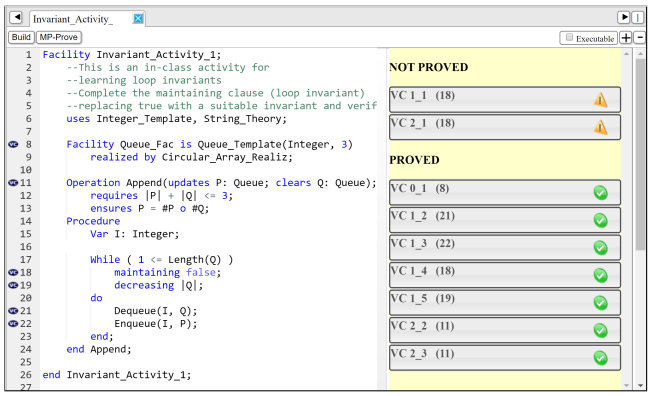


Figure 2: Verifier in Action Showing Proving and Failing Verification Conditions (VCs)

An example invariant activity used in the experiment is found in Listing 1. For this activity, students are given a formal specification and code for an Append operation whose 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.

Listing 1: Invariant Activity 1

```
Operation Append(updates P:Queue; clears Q:Queue);
    requires |P| + |Q| <= 3;
    ensures P = #P ○ #Q;
Procedure
    Var I: Integer;

    While ( 1 <= Length(Q) )
        maintaining true;
        decreasing |Q|;
    do
        Dequeue(I, Q);
        Enqueue(I, P);
    end;
end Append;
```

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. Simple string notations (e.g., concatenation, denoted by ○) are used in the specification and results about those notations (e.g., ○ is associative) are used in verification. 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, ♡, ♣ would be seen as Q = <♡, ♣>. When Enqueue is called with Q and ♢, abstractly it is adding the diamond to the right side of the string, resulting in Q = <♡, ♣, ♢>. The removal of an entry by Dequeue conceptually will remove an entry from left side of the string, resulting in Q = <♣, ♢>. Together they uphold the First-In-First-Out nature of a queue.

For this particular example, queues are declared to have a maximum length (which happens to be 3 and has no bearing on verification). 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 ○ #Q states that the value of P at the end of the operation is the concatenation of the input value of P, (denoted by #P) with the input value of Q (denoted by #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 as is shown in Listing 1. 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* (the focus of this paper) and a **decreasing** clause that lets them specify a progress metric that is used to prove termination. These assertions are automatically checked by the verifier to be legitimate before it uses them in proving code correctness. The verifier is sound [2, 25].

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 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 1 to illustrate how a student might discover an (intended) invariant.

Table 1: Example Trace to Discover and Check an Invariant

Iteration	P	Q	Check Invariant P ○ Q = #P ○ #Q
0	<1>	<2,3>	<1> ○ <2,3> = <1> ○ <2,3>
1	<1,2>	<3>	<1,2> ○ <3> = <1> ○ <2,3>
2	<1,2,3>	<>	<1,2,3> = <1> ○ <2,3>

4 ERQ 1: BUILDING A CATALOG OF DIFFICULTIES

We have employed an iterative process to develop a catalog. 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 medium, the researcher had to make some judgment calls 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 Fall 2017 data as the foundation, grouping similar answers together into the resulting categories found in Table 2. These categories were subsequently used to label the submitted responses from Spring 2018 and Fall 2018. The occurrence of these categories across multiple semesters presented promising results for the classification.

4.1 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 answers was a good start for

Table 2: Activity 1 Example Categories

Answer Category	Fall 17	Spring 18	Fall 18
|Q| != 0	9	2	2
|P| <= 3	4	3	0
|P| is Changing	11	2	6
Use of Substring	11	2	4
Use of String Reverse	5	0	2
Incorrect Concatenation	2	1	2
Requires Clause |P| + |Q| <= 3	8	6	7
Ensures Clause P = #P o #Q	6	2	3
#Q = P o Q	11	0	3
|P| + |Q| = |#P| + |#Q|	7	6	4
Other	21	3	5
Correct #P o #Q = P o Q	**31**	**27**	**38**
Total Attempts	126	54	76

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 3. Data from all three semesters were re-categorized using the catalog.

Figure 3 provides a visual breakdown of the frequency of each difficulty across each attempt for Activity 1 (Listing 1) in the Fall 2017 semester. The horizontal axis is the attempt number and the colors indicate categories. The blue color at the top corresponds to the correct invariant. With each attempt, more students arrived at the correct invariant as can be seen as we move from left to right in the figure. Almost everyone had completed the assignment by the seventh attempt. We also note that not every difficulty appeared in each attempt as seen, for example, by the disappearance of the orange bar for *|P| is Changing* after the third attempt.

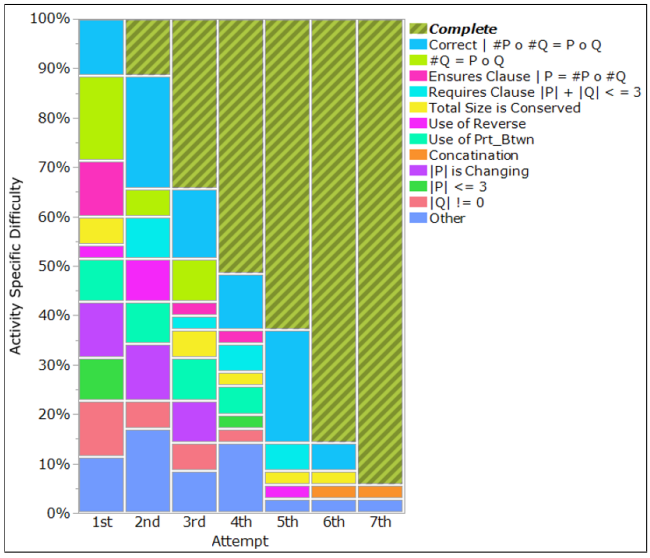


Figure 3: Frequency of Difficulties Across Fall 2017 Attempts

4.2 Verifier Feedback and Discussion

Figure 4 contains the feedback students get for verifying with the default invariant `true`.

Table 3: Catalog of Difficulties

General Category	Activity Specific	Activity Example
Use of Loop Condition as Invariant	Loop Condition	|Q| != 0
Use of Constraints as Invariant	Data Structure Constraints	|P| <= 3
Focus on What is Varying as Opposed to What is Invariant	|P| is Changing**	|P| = |#P| + 1
Use of Irrelevant Math Operators	Use of Substring	P = #P o Prt_Btwn(0,1,Q)
	Use of Reverse	#Q=Reverse(P) o Q
Confusion of Data Structure Operations (e.g., Stacks vs Queues)	Incorrect Concatenation	Q o P = #P o #Q
Use of Requires Clause as Invariant	**Requires Clause* ** **	|P| + |Q| <= 3
Use of Ensures Clause as Invariant	Ensures Clause	P = #P o #Q
Ignoring Some Input Possibilities	Assumes #P is Empty	#Q = P o Q
Underspecification	Total Size is Conserved*	|P| + |Q| = |#P| + |#Q|
Other	...	...

* Students were likely to provide a sufficient invariant on the next attempt

** Students submitted similar answers multiple times

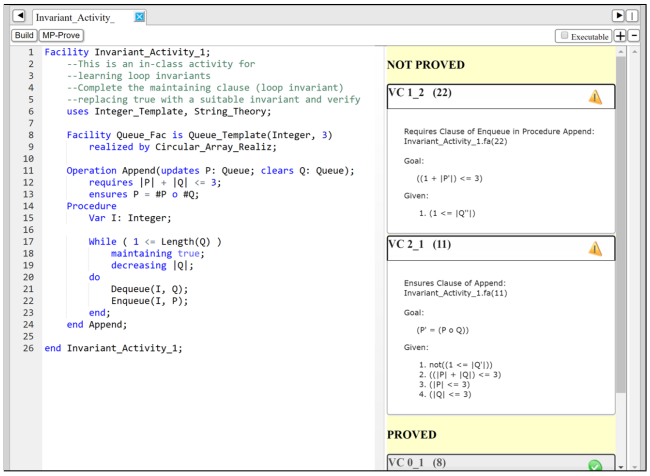


Figure 4: Verifier Feedback Using `true` as Invariant

The feedback explains why students who submitted either the Invariant *Total Size is Conserved* or *Requires Clause* (marked in Table 3 by an asterisk), they were more likely to get the correct answer on the next attempt. The first one is an invariant, but just not a sufficient one. The reason for the latter, if any, is not obvious.

The first failed VC (1_2) indicates that the requires clause for a call within the loop to `Enqueue` is not met. It is the second failed VC (2_1) 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. Previously mentioned invariants, *Total Size is Conserved* and *Requires Clause*, satisfy the requires clause of `Enqueue`. When the students attempt to verify the code with either of these invariants, only the ensures clause of the code fails to prove, focusing their attention on where it needs to be focused. So the verification process works as might be expected, and may explain why these two invariants preceded a successful attempt.

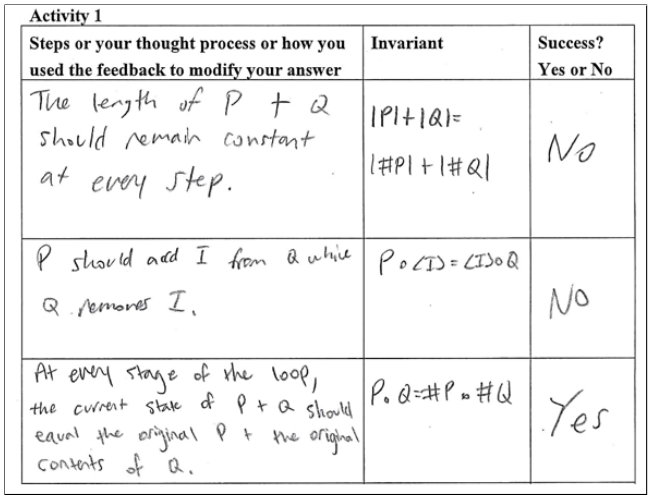
5 ERQ 2: STUDENT CONCEPTIONS

ERQ 2 focuses on analysis of student responses to determine their level of understanding and to identify any actionable information.

5.1 Response Medium

In the Fall 17 and Spring 18 experiments, a total of 62 student groups received a piece of paper at the start of the activity that contained a table to use as a scaffold, as seen in Figure 5. We found that the scaffolding encouraged students to record each attempt. Students also showed tracing as in Table 1.

For the Fall 18 experiment, 43 student groups received the same prompt as seen in Figure 5 but used a free response text box for online submission as seen in Figure 6. While easier for automated analysis, a reasonable question is what impact the online medium has on student response.



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 ∘ ⟨I⟩ = ⟨I⟩ ∘ 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 ∘ Q = #P ∘ #Q	Yes

Figure 5: Paper Response

5.2 Level of Understanding

When the medium for response changed, we observed that student responses appeared to shift from explaining what individual pieces of invariant attempts meant to a reflective analysis of their work, often explaining why a sufficient invariant worked. Figure 6 demonstrates this shift in focus for the response. Rather than

> 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 o 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 o #Q = P o Q and this proved everything

Figure 6: Online Response

stating what "should" be happening "now", this response reflects upon attempts made and explaining why they did not work. We believe the removal of the scaffolding is the reason for this change because there were no significant changes in instructors, materials, or student performance across semesters.

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.

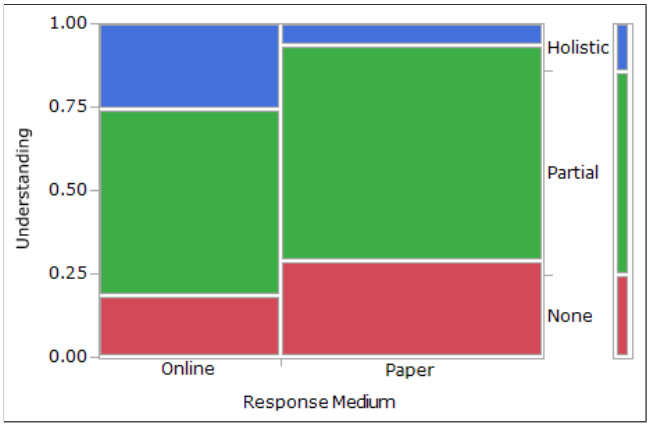


Figure 7: Proportion of Responses Showing Understanding

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 4 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.

Table 4: Completed Additional Activities for Fall 18 (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%

5.3 Actionable Information

A central reason for transitioning to online collection of student responses is the potential to collect and use actionable information to provide immediate feedback to students. If online responses showed a more holistic understanding but contained less actionable information, that would be problematic for automated feedback.

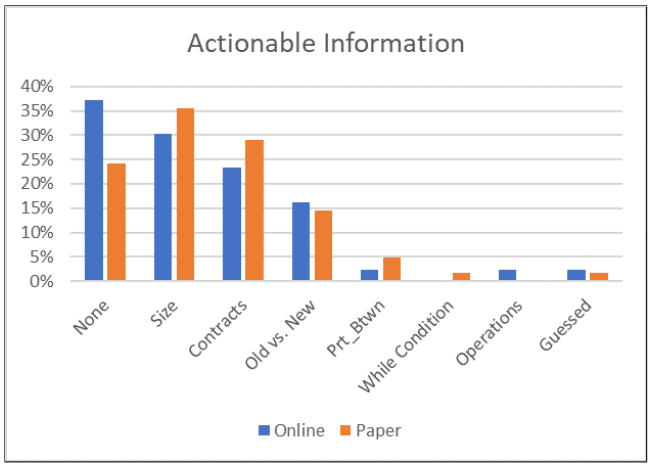


Figure 8: Actionable Information from Student Responses

The identification of actionable information for automation strongly mirrored the Catalog of Difficulties in (Table 3) under ERQ 1. Key words were identified such as *Length*, *Size*, *Requires*, *Ensures*, and even *Guess*. Key words for which similar feedback is appropriate were then consolidated, resulting in the distribution seen in Figure 8. Each bar represents the proportion of responses that contained that keyword. For example, approximately 35% of responses on paper referred to queue sizes.

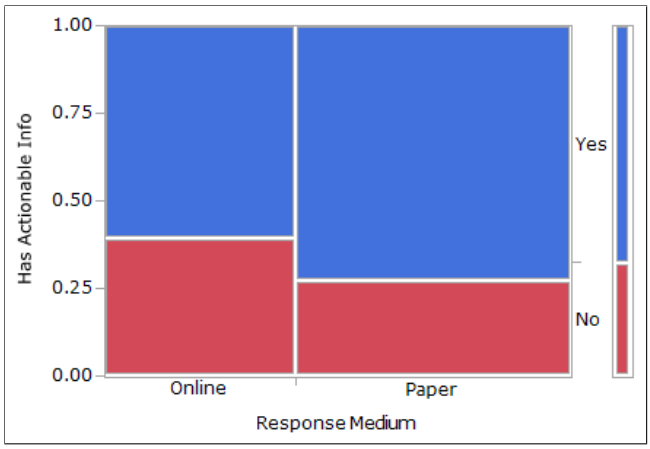


Figure 9: Proportion of Student Responses that Contained Actionable Information

The relative risk of students providing actionable information on paper medium is 1.26 times that of students reporting online. However, with a p-value of 0.192, this difference is not significant. Based on these findings, we plan to proceed with collecting student responses online to facilitate automated processing to provide tailored feedback for students.

6 DISCUSSION AND CONCLUSION

The current state of instruction on such topics as writing invariants—in the absence of automation—for developing correct iterative code relies on instructor experience to identify common student difficulties and to know how to address these difficulties. Student learning requires direct instruction or the creation of exercises that guide a student to discover previously unknown aspects of the formal verification process, discover holes in their reasoning process or to confront misconceptions.

For automated tutoring systems to be successful, they must be able to detect student difficulties and be able to respond with appropriate interventions. If students display holistic understanding of the topic, the tutor would move on to the next topic. Unable to detect a holistic understanding, the tutor would have the student continue working on the current topic. Knowing the nature of the difficulty through keywords in their explanations and classification of their answers, will assist the tutor for providing appropriate feedback, or select remedial lessons on the current topic.

In this work, we have analyzed expressions of student reasoning to identify student difficulties. By identifying these difficulties, we can provide better support for students. For example, *size* was identified to be a popular keyword (Figure 8), and five of the ten categories of difficulties involves queue sizes (Table 3). Often these students would repeatedly submit similar answers, resulting in 48% of all incorrect answers to be size-related. In this scenario, an automated tutor could identify the keyword *size* as being actionable, and then further evaluate the answer to determine the specific difficulty, allowing us to help students earlier (Table 4). For an instructor, the catalog is helpful for identifying where students may be struggling and for re-designing subsequent lessons.

Analysis of the paper and online versions allow us to reach a qualitative conclusion that the medium impacts the response, and both kinds of responses are of interest. While we have found more responses in the online medium to show a holistic understanding through a subjective analysis, that does not mean that others lacked such an understanding. Rather, this is what we are able to say from the responses. The online medium, more suitable for automation, is also an effective option for collecting actionable information. A threat to validity is that our results depend on students accurately reporting their attempts and reasons. Few studies exist that examine finer-level and higher-level student difficulties in writing formal assertions. This paper discusses a process to identify these difficulties with regard to loop invariants in a manner that can aid both automated tools and instructors. The process is done in a way to generalize and possibly guide the design of other systems for helping students learn formal topics, such as discrete structures and automata theory.

ACKNOWLEDGMENTS

This research is funded in part by NSF grants DUE-1914667 and DUE-1915088. We thank members of the RESOLVE Software Research Group (RSRG) at Clemson, Florida Atlantic, and Ohio State.

REFERENCES

[1] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. A Validated Scoring Rubric for Explain-in-Plain-English Questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 563–569.

[2] Charles T Cook, Heather Harton, Hampton Smith, and Murali Sitaraman. 2012. Specification engineering and modular verification using a web-integrated verifying compiler. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1379–1382.

[3] 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.

[4] Guido de Caso, Diego Garbervetsky, and Daniel Gorín. 2013. Integrated program verification tools in education. *Software: Practice and Experience* 43, 4 (2013), 403–418.

[5] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1-3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[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] 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 the 4th European Conference on Software Engineering Education*. 52–60.

[8] David Gries. 1981. *The science of programming*. Springer-Verlag.

[9] 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

[10] 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

[11] 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 '18)*. 1–10. https://doi.org/10.1145/3279720.3279728

[12] 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

[13] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K Rustan M Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, and Benjamin Weiß. 2011. The 1st Verified Software Competition: Experience Report, Vol. 6664. 154–168. https://doi.org/10.1007/978-3-642-21437-0_14

[14] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne, and Christoph Kreitz. 2017. Theorem Provers as a Learning Tool in Theory of Computation. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 83–92. https://doi.org/10.1145/3105726.3106184

[15] 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

[16] 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

[17] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic Loop-invariant Generation and Refinement Through Selective Sampling. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 782–792. http://dl.acm.org/citation.cfm?id=3155562.3155660

[18] 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

[19] Renée A. McCauley, Brian Hanks, Sue Fitzgerald, and Laurie Murphy. 2015. Recursion vs. Iteration: An Empirical Study of Comprehension Revisited. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Kansas City, MO, USA, March 4-7, 2015*, Adrienne Decker, Kurt Eiselt, Carl Alphonce, and Jodi Tims (Eds.). ACM, 350–355. https://doi.org/10.1145/2676723.2677227

[20] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40–51. https://doi.org/10.1109/2.161279

[21] Christopher M Nakamura, Sytil K Murphy, Michael G Christel, Scott M Stevens, and Dean A Zollman. 2016. Automated analysis of short responses in an interactive synthetic tutoring system for introductory physics. *Physical Review Physics Education Research* 12, 1 (2016), 010122.

[22] 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

[23] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1 (2017), 1:1–1:24. https://doi.org/10.1145/3077618

[24] Lee S Shulman. 1986. Those who understand: Knowledge growth in teaching. *Educational researcher* 15, 2 (1986), 4–14.

[25] 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

[26] Sotiris Skevoulis and Vladimir Makarov. 2006. Integrating formal methods tools into undergraduate computer science curriculum. In *Proceedings. Frontiers in Education. 36th Annual Conference*. IEEE, 1–6.

[27] Kristin Stephens-Martinez, An Ju, Krishna Parashar, Regina Ongowarsito, Nikunj Jain, Sreesha Venkat, and Armando Fox. 2017. Taking Advantage of Scale by Analyzing Frequent Constructed-Response, Code Tracing Wrong Answers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 56–64. https://doi.org/10.1145/3105726.3106188

[28] 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.

[29] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop*. 117–128.

[30] 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

[31] Daniel Zingaro, Cynthia Bagier Taylor, Leo Porter, Michael Clancy, Cynthia Bailey Lee, Soohyun Nam Liao, and Kevin C. Webb. 2018. Identifying Student Difficulties with Basic Data Structures. In *Proceedings of the 2018 ACM Conference on International Computing Education Research, ICER 2018, Espoo, Finland, August 13-15, 2018*, Lauri Malmi, Ari Korhonen, Robert McCartney, and Andrew Petersen (Eds.). ACM, 169–177. https://doi.org/10.1145/3230977.3231005