What's in a Specification?

HARRISON GOLDSTEIN, University at Buffalo, SUNY, USA
CAMERON MOY, Northeastern University, USA
DANIEL PATTERSON, Northeastern University, USA

Formal specifications are a critical prerequisite for formal methods, but they are not always readily available in practice. In this study, we explore how students in an early software development course understand and engage with formal specifications. Seventeen students were asked to translate informal specifications into formal ones. We find that students often supply unnecessarily approximate specifications and predominantly make three different kinds of mistakes during the translation process.

ACM Reference Format:
Harrison Goldstein, Cameron Moy, and Daniel Patterson. 2026. What's in a Specification?. In *Proceedings of the 16th PLATEAU Workshop on Programming Languages and Human-Computer Interaction (PLATEAU '26).* 7 pages. https://doi.org/10.1184/R1/31816747

1 Introduction

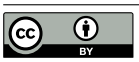
Research on software verification usually assumes that users have already discovered some *formal specification* that they want to validate their software against. These specifications can be expressed as programs or logical formulas. They should be *unambiguous* (a program should either clearly meet its specification or not), *specific* (any program that meets the specification should be "correct"), and *checkable* (it should be tractable to check if a program meets its specification given some verification technique). Of course, in practice, users do not always have such a formal specification available. Users more commonly have informal specifications like natural-language descriptions, diagrams, and input–output examples, which are critical for human-to-human communication but not necessarily helpful for verification. As software verification techniques get more powerful, it is increasingly important that users can successfully translate between informal and formal specifications. In this work, we explore the relationship between informal and formal specifications in the context of an early software development course. We evaluate students' translations from natural-language descriptions of program behavior to executable checkers of those behaviors.

Concretely, we set out to study two research questions pertaining to early-career computer science students and formal specifications:

RQ1 How do students understand what goes into a specification?
RQ2 What mistakes do students make when translating informal specifications to formal ones?

We answer these questions with a sample of 17 students from Northeastern's *Logic and Computation* course by observing their answers to a series of surveys distributed during the Spring 2025 semester. The goal of the course is to introduce students to formal reasoning about programs and, in particular, the art of writing formal specifications. Most students take the course in their first year, having taken a first-semester programming class and a first-semester discrete mathematics class. Concurrently, many students take a second-semester programming class in Java.

Authors' Contact Information: Harrison Goldstein, University at Buffalo, SUNY, Buffalo, NY, USA; Cameron Moy, Northeastern University, Boston, MA, USA; Daniel Patterson, Northeastern University, Boston, MA, USA.



PLATEAU '26, Pittsburgh, PA

https://doi.org/10.1184/R1/31816747

Publication date: March 2026.

Our results show that students often miss the mark on *specificity*, proposing under-approximate specifications like type declarations and unit tests instead of full formal properties. We also categorize mistakes that students make, ranging from simple misreadings of the informal specifications to complex logical issues that stem from overcomplicated algorithms. In summary, we find that computer science courses need to teach formal specification explicitly and carefully if we hope for students to be able to correctly apply formal methods.

To be clear, these results are unlikely to be very general: they depend significantly on the way that specification is taught at Northeastern and on the choice and phrasing of study tasks. Nevertheless, these results may inspire future studies on this topic and provide results that future researchers can use as a baseline.

2 Methods

To answer the research questions, we produced a series of small problem sets that were distributed as online surveys. Each problem set had three problems, labelled Q1, Q2, and Q3 ("Easy", "Medium", and "Hard" respectively).[1] Each problem consisted of an informal description of some simple functions and their intended behavior. The goal for students was to translate the informal description into a formal specification. A description of each problem appears in Appendix A.

Problem sets were distributed in three phases. Phase 1 (P1) was in the second week of the semester, after students had been introduced to specifications but taught little of how to build them. Phase 2 (P2) was in the eighth and ninth week of class, after all basic techniques had been covered but before the course's most challenging examples. Phase 3 (P3) was at the end of the semester.

We recruited 28 students from *Logic and Computation*. One of the authors, who was not part of the course staff, visited each section and advertised the research study. Students earned a gift card for their participation. Over the duration of the study, 11 participants dropped out (3 after phase 1 and 8 after phase 2), leaving 17 participants total who participated in phase 3. Students were distributed across all four years, with the vast majority being first and second year students.

In addition to grading each answer for correctness, we performed a lightweight thematic analysis of responses guided by our research questions. Each of the authors graded three questions (one in each of the phases and one of each difficulty) and noted repeated patterns encountered during the process. The authors then discussed the problems and themes together, which resulted in a summarized list of common themes and mistakes.

3 Results

3.1 RQ1: How Students Understand Specifications

Types as formal specifications. Many students (in problems P1Q1 and P1Q2) provided only a type signature as a formal specification. Types are certainly useful for specifying program behavior, but they were not what we asked for. All nine of our questions explicitly asked the students to define a function, not just a type, that would check the provided code's correctness.

Unit tests as formal specifications. Like types, a unit test is a weak formal specification. Tests are mostly insufficient to establish full correctness. For some problems (P1Q2, P1Q3, P2Q2) many students submitted unit tests instead of general formal specifications. These problems tended to be ones where writing the specification was difficult and students "fell back" to writing unit tests.

For example, in P2Q2, the students were asked to check that two different implementations of a queue data structure behave the same. This problem is a classic application of model-based testing [6]. The ideal solution says that *for all sequences* of queue operations, the outputs of the queue

[1] The order was randomized for participants.

operations agree (e.g., for every dequeue operation, the return value is equal for both queues). Many students picked a single interleaving of operations to check, rather than checking all sequences.

We suspect that part of the reason students wrote unit tests was that they had not yet understood how to quantify their specifications over nontrivial data. In reference to a similarly structured problem (P1Q3), a student wrote "I have no clue how to do this. I can write some check expects [unit tests] but I doubt that is what you are asking for."

Formal specifications as approximations. Both of the above forms of specification are approximations: they are technically correct, but under-specify the system and lose *precision*. Even when writing more-precise specifications, some students under-specified unnecessarily.

For example, in P1Q3, students were asked to specify the operations of a simple ATM. When specifying the `(deposit n)` operation, one student opted to check that the new balance was greater than the old balance. This specification is valid but is somewhat surprising—the student could easily have checked that the new balance was *precisely* `n` *greater* than the old balance. Instead, the student chose to approximate anyway.

3.2 RQ2: Mistakes Students Make Constructing Formal Specifications

Misreading the informal specification. One of the most common classes of errors that students made was misreading the informal specification. These were instances where the student clearly understood what they were being asked to do but seemed to get mixed up when writing the code.

For example, in P2Q1, students were asked to show that a function to compute an inverse square root was correct (i.e., `(approx-eq (inv-sq-root x) (/ 1 (sqrt x)))`) but many forgot to take the inverse of the square root (they checked `(approx-eq (inv-sq-root x) (sqrt x))`). In P3Q1, students needed to show that "parsing undoes pretty-printing." This specification might be correctly translated as `(equal? t (parse (pretty-print t)))`, but some students tested `(equal? s (pretty-print (parse s)))` instead.[2]

Approximating by accident. Even when striving for specificity, some students accidentally under-approximated their specifications. For example, in P1Q2, students were asked to check that an algorithm for randomly cycling a string (e.g., mapping `"abc"` to `"bca"`, `"cab"`, `"abc"`) always produced a valid result. Many students, with the help of a hint, recognized that this problem can be solved by checking that the new string is a substring of the original string concatenated with itself. Almost no one checked that the new string was also the *same length* as the original string.

Over-complicating the solution. Some participants submitted incorrect answers because they overcomplicated their implementation.

For example, in P3Q1 (the "parsing undoes pretty-printing" problem), one student submitted

```
(equal? s (pretty-print (parse (pretty-print (parse s)))))
```

with a comment asserting that they could check both directions ("parse undoes pretty-print" and "pretty-print undoes parse") at the same time with this scheme. If the student had tried to check the directions independently, they would have been mostly correct—at worst, their solution would have been a bit over-constrained. But this version of the specification actually fails to check either direction adequately.

Over-complicated solutions were also common in P2Q3, where students had to implement a small checker for satisfying assignments to SAT formulas. One extreme solution used a `map` inside

[2]This specification is true for some instances of parsing and pretty-printing but not all of them. A parser may discard comments and whitespace which would then not reappear after printing.

a `fold` inside a `map` inside a `fold`, without any auxiliary function definitions. Nesting all of these higher-order functions results in unreadable code that easily harbors bugs.

3.3 Evolution Over Time

We had hoped to observe students' confidence with formal specifications increase over the course of the study, but our results were not compelling. The proportion of students getting *some credit for some problem* did increase over the course of the semester (from 75% of the group in phases 1 and 2 to 100% in phase 3), but individual student trajectories were not monotonic. Perhaps clearer changes over time would have occurred if the problems themselves had stayed more similar between phases.

4 Discussion

Our observations imply valuable lessons for this course and others moving forward.

Discuss the differences between types, unit tests, contracts, and property tests. In all languages, but especially in the pedagogic language used in *Logic and Computation*, there is a spectrum of specification tools, each with different use cases. Types and unit tests are both great tools for basic checks, but they miss key aspects of correctness. Contracts and property tests capture deeper aspects of correctness, but they have strengths and weaknesses, and many student solutions showed evidence that students did not internalize when one mechanism should be used over the other.

Talk more explicitly about specification imprecision and when different approximations are valid. Approximations are a necessary part of specification, but precisely how much to approximate should be a conscious and careful choice. Educators should ensure that students are aware of when they are making an approximation and when relaxing a specification is appropriate.

One concrete way to address approximation and imprecision is adversarial thinking. Adversarial thinking [8], commonplace in the security community, could encourage students to imagine an adversary designing pathological implementations. Students' specifications should catch these programs. This mindset would help address the idea that all specifications should be approximations and encourage thinking about all of the constraints needed to guarantee correctness.

5 Related Work

Several researchers have looked at how programmers understand formal specifications. Wrenn et al. [7], for example, explore how students engage with formal specifications in the context of property-based testing (PBT). They center *relational properties* as being a particularly powerful motivating example. In a similar vein, Nelson et al. [3] study PBT in the classroom, this time focusing on how to scalably evaluate student predicates. Our study supplements these results by considering the kinds of formal specifications students write in the absence of detailed instructions about what should go into the specification. We also see examples of mistakes that show up when students directly translate informal specifications (i.e., without needing to work from scratch).

Excitingly, many courses teach specification. Some of the authors of this work recently published an experience report on teaching formal specification [2] in an earlier semester at Northeastern. There are also reports from Brown University [1], Pace University [5], and Yeditepe University [4].

6 Conclusion

For a topic that is so well-studied, the science of how programmers understand specifications still feels poorly understood. As programmers are pushed to produce code ever-faster, it is critical that they can clearly express their intent. This study is one step towards studying how students understand specifications. We hope future researchers will dive in deeper.

References

[1] Shriram Krishnamurthi, Tim Nelson, et al. 2025. Logic for Systems: A Gradual Introduction to Formal Methods. *Bulletin of the European Association for Theoretical Computer Science* (2025).

[2] Cameron Moy and Daniel Patterson. 2025. Teaching Software Specification (Experience Report). In *International Conference on Functional Programming*. doi:10.1145/3747533

[3] Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. 2021. Automated, Targeted Testing Of Property-Based Testing Predicates. *The Art, Science, and Engineering of Programming* (2021). doi:10.22152/programming-journal.org/2022/6/10

[4] Mert Ozkaya. 2019. Teaching Design-By-Contract For The Modeling And Implementation Of Software Systems. In *International Conference on Software Technologies*. doi:10.5220/0007950904990507

[5] Christelle Scharff and Sokharith Sok. 2006. From Design by Contract to Static Analysis of Java Programs: A Teaching Approach. In *Formal Methods in the Teaching Lab*.

[6] Mark Utting and Bruno Legeard. 2010. *Practical Model-Based Testing: A Tools Approach*. Elsevier.

[7] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2020. Using Relational Problems To Teach Property-Based Testing. *The Art, Science, and Engineering of Programming* (2020). doi:10.22152/programming-journal.org/2021/5/9

[8] Nick Young and Shriram Krishnamurthi. 2021. Early Post-Secondary Student Performance Of Adversarial Thinking. In *Proceedings of the 17th ACM Conference on International Computing Education Research*. doi:10.1145/3446871.3469743

A Problem Descriptions

Table 1. An overview of the problems given to students across phases.

	Phase 1	Phase 2	Phase 3
Question 1 (Easy)	Basic Diff. Testing	Fast Inv. Sq. Root	Parse-Print
Question 2 (Medium)	Cyclic Shuffle	Queue Model	Tic-Tac-Toe
Question 3 (Hard)	ATM	Check SAT	Topo. Sort

A.1 Easy, Phase 1: Comparing Function Behavior

Two functions, `simple-first-try` and `highly-optimized-final`, take positive integers as input and return booleans. They should implement the same functionality. Write a specification that checks `highly-optimized-final` is correct with respect to the behavior of `simple-first-try`.

A.2 Medium, Phase 1: Cyclic Shuffle

We have designed a function `cyclic-shuffle` that takes a string and shifts its contents a random number of places i.e., moving a character from the end to the beginning. For example, `"abc"` might turn into `"cab"` or `"bca"` or `"abc"` (because there are three characters, so up to three different shifts).
Write a function `cyclic-shuffle-prop` that takes a string input and checks that our function did the right thing on it. Hint: notice that `"abcabc"` contains all three possibilities above.

A.3 Hard, Phase 1: ATM

We've implemented the following functions that implement the behavior of an ATM:
- `clear` takes a reference to an `atm` and clears its contents, setting the balance to zero.
- `withdraw` takes an `atm` and a number `amt` and deducts `amt` from the current amount stored in the ATM. It returns the updated balance, or `'not-enough` if there is not enough money in the account.
- `deposit` adds `amt` to the `atm` and returns the updated balance.

Write a specification `atm-prop` that specifies how these three functions should behave.

A.4 Easy, Phase 2: Fast Inverse Square Root

The fast inverse square root is an infamous function in the history of video game programming that approximates the value of $\frac{1}{\sqrt{x}}$. Suppose you have a slow function `(sqrt x)`, write a specification `fast-invsqrt-prop` for `(fast-invsqrt x)` that ensures that the result is never off by more than 10% of the true result. (Note that these functions take and return floats.)

A.5 Medium, Phase 2: Queue Model-Based Testing

We have two implementations of a queue data structure. Both have the same operations:
- `enqueue-1` and `enqueue-2` each add an element to the back of the queue.
- `dequeue-1` and `dequeue-2` each remove an element from the front of the queue (returning `'nothing-to-dequeue` if the queue is empty).
- `empty-1` and `empty-2` each determine if the queue is empty. Write a specification `queues-prop` that ensures that these queues behave the same way.

A.6 Hard, Phase 2: Boolean Satisfaction

Write a function `(check-satisfying-assignment f h)` that takes a formula in 3CNF and a mapping from variables to booleans, and checks that the mapping is a satisfying assignment for the formula.

A.7 Easy, Phase 3: Parse-Print Round Trip

Parsing is the process of turning a string into some kind of data structure, and *pretty-printing* is the process of turning a data structure back into a string. It should always be the case that parsing "undoes" printing: pretty-printing a data structure and then parsing it again should result in the original value. Give a specification `parse-print-prop` that relates two functions: `(parse-t s)` that parses a string `s` and `(pretty-print-t s)` that prints a structure `t`.

A.8 Medium, Phase 3: Tic-Tac-Toe

We are implementing a game of tic-tac-toe, and we have a function `(take-turn board)` that takes a board and returns a new board by asking the player for a move. The board is represented as a list of symbols, each of which is either `'X`, `'O`, or `'E`. Importantly, `take-turn` will crash if the board is already a complete game. Write a specification `take-turn`.

A.9 Hard, Phase 3: Topological Sort

A topological sort of a directed acyclic graph (DAG) is a list of every vertex in the graph, ordered so that if a vertex `v1` comes "before" another `v2`, in the sense that `v1` has an edge to `v2`, then `v1` appears before `v2` in the list. Assume you have a DAG represented as an adjacency matrix (i.e., a list `'((v1 . v2) (v3 . v4) ...)` of edges). Write a specification `toposort-prop` for `(toposort g)`.