Teaching Software Specification (Experience Report)

CAMERON MOY, Northeastern University, USA
DANIEL PATTERSON, Northeastern University, USA

A course on software specification deserves a prominent place in the undergraduate curriculum. This report describes our experience teaching a first-year course that places software specification front and center. In support of the course, we created a pedagogic programming language with a focus on contracts and property-based testing. Assignments draw on real-world programs, from a variety of domains, that are intended to show how formal specification can increase confidence in the correctness of code. Interviews with students suggest that this approach successfully conveys how formal specification is relevant to software construction.

CCS Concepts: • **Software and its engineering → Correctness**.

Additional Key Words and Phrases: software specification, pedagogic programming languages

ACM Reference Format:
Cameron Moy and Daniel Patterson. 2025. Teaching Software Specification (Experience Report). *Proc. ACM Program. Lang.* 9, ICFP, Article 264 (August 2025), 16 pages. https://doi.org/10.1145/3747533

1 Specifying Software is a Skill

Software correctness is well-recognized as essential in many fields. Both programming languages and formal methods practitioners have explored ways of teaching it for decades. Showing that software is correct can be split into two activities: *specification* and *verification*. Specification formally characterizes (some aspect of) the intended behavior of software, whereas verification shows, using various methods, that software conforms to its specification.

While these two may seem equally important, in practice, time spent articulating a specification is often dwarfed by time spent working on verification. This imbalance can also be present in the classroom. If theorems are written by instructors, then students get little to no practice writing specifications themselves. If the theorems are relatively simple, then the complexity of the programs students reason about might never exceed single-digit line counts.

Confronted with this reality in our own proof-heavy course, we decided to turn the class on its head: rather than thinking of specifications as instructor-written problems that students solve by carrying out proofs, we made specification the primary activity, avoiding most work on proofs. Indeed, while writing correct software requires both specification and verification, arguably it is specification that is more critical. Without a specification, any proof, successful or not, is worthless.

Software specification is also a broadly applicable skill. Even projects that are never subject to formal verification can benefit from clear descriptions of functional correctness, security properties, or other behavioral invariants. For this reason, specification should be a core part of any undergraduate curriculum, and it should come as early as in the first year. This is true of our course: it is required of many students and is suggested to freshmen.

The redesigned course is based on two intertwined components. First, we employ run-time contracts and property-based testing to capture sophisticated (type) invariants and logical properties.

Authors' Contact Information: Cameron Moy, Northeastern University, Boston, USA, camoy@ccs.neu.edu; Daniel Patterson, Northeastern University, Boston, USA, dbp@dbpmail.net.

ACM 2475-1421/2025/8-ART264
https://doi.org/10.1145/3747533

Contracts [15, 24] can express dependent-type invariants, introducing students to the power of this type-based reasoning without any background on the complex topic of dependent type systems. Property-based tests (PBT) [5] also provide a relatively low-barrier means of expressing complex invariants about code. One way to understand our approach to PBT is that students practice writing theorems, but the proofs are deferred to the approximation produced by a PBT engine.

Second, we created a pedagogic functional language for the course, dubbed the Logical Student Language (LSL), that includes not only constructs for contracts and property-based testing, but also linguistic support for non-trivial programs. For example, one of our assignments has students build an idealized memory allocator and write down memory-correctness properties, which relies on state-of-the-art temporal contracts. Another assignment has them implement a distributed snapshot algorithm and express snapshot consistency as property-based tests, which relies on message-passing concurrency and an accompanying visualizer. Linguistic support allows such problems to be tractable, even for students in their second semester of programming, as we eliminate the incidental complexity that typically arises without full control over the language.

This paper contains background about our educational context, an outline of the revised course, the design of LSL, some sample assignments, what students told us in comparison to another iteration of the course taught using the Lean theorem prover [8], and how ideas from the course can be adapted to other curricula.

2 Learning From Past Experiments

For the last two decades, our institution has had a course that aims to connect logical reasoning with introductory programming courses. Historically, the course functioned as an introduction to *both* specification and verification, first using the ACL2s theorem prover [9], and more recently using the Lean theorem prover [8]. The only prerequisites for the course are a single semester of programming and discrete mathematics; it was originally envisioned, and is still often realized, as a second-semester freshman course.

Unfortunately, a phenomenon familiar to those teaching formal methods has stymied this effort: time spent on verification far surpasses time spent on specification. In an educational context, this results in the class being either too hard for students to learn, or structured such that the majority of the semester is dedicated to proofs of relatively uninteresting theorems.

Originally, the course was taught using the custom language Dracula, which is a frontend to the ACL2 theorem prover [10, 29]. This tool was used both for our class and for an upper-level course on software engineering at another institution. While the goal was theorem proving, Dracula also included a mechanism for property-based testing (called DoubleCheck). In a paper summarizing the first few years of this experiment, Page et al. [29] report that only ten to twenty percent of students ended up being able to use ACL2 via Dracula effectively. Interestingly, the authors also report that DoubleCheck, their property-based testing framework, "helps students with one of the trickiest task[s] of using logic in programming, namely, the transition from ideas to formal statements" [29]. While the authors had plans, never fully realized, to improve Dracula to address some issues, recent publications (e.g., a study by Juhošová et al. [20]) indicate that beginners still face challenges using interactive theorem provers.

More recently, when the course has been taught using Lean, students learn to write proofs about natural numbers and lists, but little more. Attempting to shift this balance seems impossible as significant practice with basic theorems over simple inductive types is necessary to build the skills needed to complete large proofs.

It is tempting to think automation might help, but after the experiments with Dracula, the course was taught for many years using ACL2s, which has world-class automation support. This does not seem to help. In particular, knowing how to drive sophisticated automatic provers, especially intermediate lemma identification, may be a more advanced skill than constructing proofs manually—even if the resulting proofs are much smaller.

While the intent of the class was always to introduce students to formal reasoning *about software*, perversely, students using Lean largely reported to us that the class had no bearing on software development whatsoever (Section 7). A frank assessment of this decades-long experiment is that at best students were introduced to the idea of mechanized proof, unconnected to the software work they do in other courses or on internship, and at worst they learn a few details about an esoteric tool they will never use again.

3 Overview of Our Course Redesign

> "This course is about formal specification and how logic can be used to help reason about software. Logic is presented from a computational perspective using property-based testing, a technique available via libraries in nearly every programming language."

Review of Propositional Logic (Week 1). The course begins with a review of propositional logic, but turned into computation: propositional formulae become boolean-valued functions, and truth tables become exhaustive sets of unit tests. This exercise introduces students to the idea that logical reasoning can be viewed as computation. Additionally, students see how translating formulae into code, and truth tables into unit tests, can uncover mistakes just by running the tests.

Atomic Data (Weeks 2–3). The second unit explores the idea of specifications as code, but in the setting of atomic data such as numbers, strings, etc. Students translate informal descriptions of how functions should work into boolean expressions that relate inputs and outputs.

Increasing Complexity (Weeks 4–8). The third unit of the class explores both data and properties of increasing sophistication. Students deal with recursive data, higher-order functions, and data abstraction via contracts. Additionally, this unit transitions the homework assignments from introductory practice to somewhat complex projects (Section 5).

Advanced Topics (Weeks 9–12). The latter part of the semester covers mutation and aliasing, which accompanies an assignment about building an idealized memory allocator (Section 5.3). Pairing the lecture content, which presents aliasing puzzles, with a large project where aliasing is a central theme, allows students to explore the concepts deeply even in a relatively short period of time. Enforcing memory invariants serves to enrich the content and helps with debugging.

This unit of the course also introduces concurrency with a simple message-passing implementation and accompanying visual debugger, all built into LSL. Students first implement a warm-up homework using message passing and, once graded, implement the Chandy–Lamport concurrent snapshotting algorithm (Section 5.4). As with the memory allocator, the project is small enough to be achievable, but complex enough that students realize that correctly implementing it is much easier if they write down invariants and ensure they are being preserved as the code runs.

Special Topics (Weeks 13–14). The last two weeks are left for special topics. Typically this means interactive theorem proving, though different instructors may choose different topics. The structure of our semester, and the placement of our last exam within the semester, means that this content is not the subject to any assessment; it serves as bonus material for students who are interested in further enrichment. We found that this is a nice place to introduce tools like Lean, as students who are interested can learn about them, but the central learning outcomes of the course are not impacted.

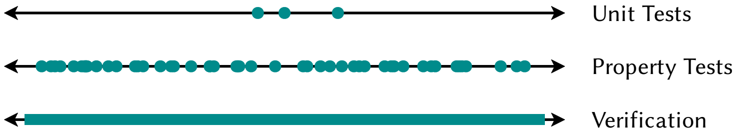

Fig. 1. Gradual Specification

4 The Logical Student Language

Creating a new programming language, especially a pedagogic one, is a substantial undertaking. The effort is worthwhile if it reduces friction, as LSL does in two ways. First, LSL is a superset of the pedagogic language used in the first-semester course on programming. Students enter our course with a full semester of experience with its syntax and semantics and can immediately acquire new skills without rehashing functional-programming basics. Second, LSL is integrated with the DrRacket [14] pedagogic IDE—also the same one used in their first course. Thus, the workflow students are already familiar with is completely unchanged. LSL also augments DrRacket to support certain learning objectives. For example, it integrates with Tyche [17], an interface for inspecting the quality of random generators, to help discover weak property-based tests.

4.1 LSL in a Nutshell

LSL is an extension of the Intermediate Student Language (ISL) from How to Design Programs (HtDP), the textbook our students use in their introductory programming course [13]. ISL is a simple pedagogic functional programming language. Traditional introductions to programming teach language features (e.g., variables, assignment, conditionals, loops) and leave program design as an implicit skill to be acquired through practice. HtDP treats program design—the craft of constructing correct and maintainable software—explicitly. The language becomes merely a vehicle through which these universal skills are taught. As such, HtDP provides a sequence of increasingly sophisticated pedagogic languages, including ISL, to maximize learning and minimize distractions [11]. For example, error messages are tailored to freshmen students by employing familiar terminology instead of technical jargon [22].

LSL adds contracts and property-based testing to ISL, which enables the *gradual* approach to specification shown in Figure 1. HtDP focuses on unit tests, or what amounts to pointwise specifications. Such tests do not cover much of the input space, but are a necessary first step. Property-based tests randomly generate inputs, enabling coverage of a greater amount of the input space. Getting the properties and generators right can take some work, but yields tangible benefits.

Our preference for contracts over types is due to three unique features of contracts. First, contracts consist of ordinary code and as such allow expressive specifications without the cognitive overhead of a sophisticated type system. In class, students regularly write specifications that are dependent, temporal, and intensional. Designing a type system to accommodate all of these properties, let alone one that is appropriate for beginners, is infeasible. Contracts allow students to freely explore the space of possible specifications, building on their existing programming intuitions instead of putting them aside. Second, contracts are a *precise* mechanism for enforcing properties. To remain decidable, type systems, or any static technique, must approximate program behavior. Since contracts monitor programs at run time, a violation guarantees that there is a true inconsistency between the specification and a program execution. Third, and relatedly, contracts supply a concrete counterexample when a program violates its specification. Each error message highlights the violated contract and comes with a *witness* to the violation. Students can use this witness to debug their program or specification. In contrast, type systems supply error messages in terms of failure to satisfy a particular syntactic discipline—easy for experienced programmers to resolve, but less so for novices.

4.2 Basic Contracts in LSL

In HtDP, students follow the Design Recipe, a checklist that includes a step for writing a comment above each function definition with its signature. Here is a representative example:

```
;; (List Integer) -> (List Natural)
(define (only-non-negative lon)
  (filter (λ (n) (or (zero? n) (positive? n))) lon))
```

Comments are inert, but their simplicity and flexibility are useful because students can write informal signatures that may be challenging to express formally (e.g., signatures with refinements or untagged unions). These specifications are perfectly reasonable, even if they thwart conventional type systems. Eventually, it becomes valuable to consider how to make such statements formal. The same example can be realized in LSL with the following contract:

```
(: only-non-negative (-> (List Integer) (List Natural)))
(define (only-non-negative lon)
  (filter (λ (n) (or (zero? n) (positive? n))) lon))
```

The `:` annotation associates a contract with the given function name, monitoring the specification at run time. These annotations mirror the signature conventions in HtDP, providing a smooth path from informal prose to formal specifications.[1] Violations of the contract raise an exception.

Without a program or test cases to exercise the functionality of `only-non-negative`, the contract serves no purpose other than documentation. One way to test if the function satisfies the contract is to use property-based testing. Every contract built into LSL, including higher-order ones, comes with a generator and shrinker [21]. Thus, students are able to run property-based tests for their specifications with a single line: `(check-contract only-non-negative)`.

The `check-contract` form generates random lists of integers based on the domain contract for the given function, calls the function repeatedly with these arguments, and ensures that the codomain contract holds. Close integration between the contract system and random-generation capabilities makes PBT convenient to use for first-year students.

If a property-based test encounters an inconsistency, it reports a concrete, shrunk counterexample as an error message:

```
discovered a counterexample
  counterexample: (only-non-negative (list -1))
  error:
    only-non-negative: contract violation
      expected: Natural
      given: -1
      blaming: anonymous-module (as server)
```

Following DrRacket's printing conventions, the counterexample can be copy-and-pasted into the REPL to reproduce the error. Note too that the error has *blame information* pointing to the party that violated the contract [15]. The `as server` parenthetical indicates that the function itself violated the contract. If a caller of the function gave an invalid argument instead, then the error would say `as client`. Additionally, DrRacket highlights the violated contract in the program's

[1] Crestani and Sperber [7] explore similar ideas, although LSL is a much larger departure from the teaching languages than the extensions described in their work.

source. Tight cooperation between LSL and the DrRacket IDE permits affordances that are especially useful for freshmen students. Such affordances are one advantage specialized pedagogic languages have over ordinary libraries.

Strengthening the signature of a function is one way to increase confidence in the correctness of code. Here is a dependent function contract that ensures the `longest-string` function returns a string whose length is greater than or equal to all others in the input list:

```
(define (longer-than-in? los)
  (λ (x) (andmap (λ (s) (>= (string-length x) (string-length s))) los)))

(: longest-string
   (Function (arguments [los (NEList String)])
             (result (AllOf String (longer-than-in? los)))))
(define (longest-string los) —— elided ——)
```

This dependent function contract has two pieces: one constraining the argument and another constraining the return value. The argument is expected to be a non-empty list of strings and is bound to the variable `los` for use in the result contract. The result is expected to be a string that satisfies `(longer-than-in? los)`, which ensures that the string is longer than any other in `los`.

A programmer can also define custom contracts, the most basic of which is an *immediate* (or *flat*) contract [15]. An immediate is a first-order check: a predicate for determining whether values satisfy the contract. Immediates stand in contrast to higher-order contracts, like function contracts, that must interpose on all future interactions with a value. LSL comes with many built-in immediates, and students can define their own:

```
(define-contract Even
  (Immediate (check (λ (x) (and (integer? x) (even? x))))
             (generate (λ (fuel) (* 2 (contract-generate Integer fuel))))
             (shrink (λ (x)
                       (let ([y (/ x 2)])
                         (if (even? y) y (sub1 y)))))))
```

This is a contract for even numbers that also includes a generator and shrinker for property-based testing. The `check` clause contains the mandatory predicate. The `generate` clause expects a function that takes `fuel`, a natural number corresponding to how hard the generator should try to construct a value, and returns a value that *must* satisfy the contract. This particular generator delegates to the generator for `Integer`, via `contract-generate`, to return an even number. The `shrink` clause takes a function that returns, if possible, a smaller value than the one given.

Contracts can also handle the kinds of data definitions seen in languages with algebraic data types. Here is the definition of a binary tree parameterized by the type of element:

```
(define-struct leaf [value])
(define-struct node [left right])
(define-contract (Tree X)
  (OneOf (Struct leaf [X]) (Struct node [(Tree X) (Tree X)])))
```

PBT works automatically: if `X` supports generation, so too will `(Tree X)`.

4.3 Enforcing Data Abstraction

Data abstraction occurs when the implementation details of a particular data type are hidden from some pieces of code but not others. Specifications about data abstraction fit well in our curriculum. First, data abstraction is easily motivated as a desirable software-engineering principle. Robust system design relies on information hiding to decrease the coupling between components [30]. Second, most programming languages offer data abstraction mechanisms—whether through access modifiers, existential types, or name mangling. Students should be familiar with the concept and understand why it matters. Finally, data abstraction provides a nice on-ramp to properties that are not just about correctness. Many important properties fall into this category, from security properties to resource constraints.

In type systems, data abstraction is enforced via universal and existential types [25]. The key difference between the two rests in *which* pieces of code must treat data abstractly. With universal types, implementation details are hidden from the server component. With existential types, implementation details are hidden from the client component. Dynamically enforced analogues, universal and existential contracts [18, 23], seal and unseal values to ensure programs treat certain values abstractly. Here is an example of an existential contract:

```
(define-struct counter-pkg (make incr get))

(define-contract Counter
  (Exists (T)
    (Struct counter-pkg [(-> T) (-> T T) (-> T Natural)])))
```

The `Counter` contract specifies the signatures of three functions contained in a `counter-pkg` structure, where `T` refers to an abstract type. There can be several packages that implement `Counter` which keep the representation of `T` hidden from clients.

Existential contracts are a *mechanism* and not the only one that may be used to achieve data abstraction. Most first-year students taking our course concurrently take a course on object-oriented programming in Java. After discussing data abstraction via existential contracts, students implement pure objects in LSL using structs with functions (i.e., methods) as fields. In such an encoding, closures are a data-hiding mechanism [26]. We then talk about these two different forms of data abstraction and their respective tradeoffs [6].

4.4 Mutation

Up to this point the course covers pure functional programming. Eventually, students must be exposed to effectful programs, and in particular, programs with mutable state. Our goal is to ensure students have a deep understanding of mutable state, are made aware of how mutation complicates reasoning about programs, and have the tools needed to manage that additional complexity.

As in most programming languages, LSL supports both variable and value mutation. Variables are mutated with the `set!` form:

```
(: n Integer)
(define n 1)
(set! n 2)
```

When a variable is mutated, the contract associated with the variable is checked again.

Value mutation is available for structs, but only when they are explicitly declared as mutable:

```
(define-mutable-struct posn [x y])
(: p (Struct posn [Integer Integer]))
(define p (posn 1 1))
(set-posn-x! p 2)
```

The contract on a struct's field is checked when that field is mutated. Making variable mutation and value mutation syntactically distinct helps make the concepts distinct in students' minds.

Table 1. This table lists the titles of every homework assignment in the course. Homework assignments that directly address skill transfer are indicated with ⋆.

WEEK	TITLE
1	Propositional Logic as Code
2	Executable Specifications for Simple Functions
3	More Executable Specifications for Simple Functions
4	Cryptography and Timing Attacks
5	Constant Folding for a Stack-Machine Compiler
6	Generating Mazes
7	SAT Solver using Four Representations of Booleans
8	⋆ Aliasing Puzzles and PBT Memos
9	Introduction to Message-Passing Concurrency
10	Manual Memory Allocator
11	Chandy–Lamport Snapshot Algorithm
12	⋆ Using Another PBT Library

4.5 Temporal Properties

Standard contracts fail to account for an important aspect introduced by mutation. Equality of immutable data is best characterized by structural equality, but equality of mutable data is best characterized by pointer equality [2]. Consequently, mutable values have a discernible identity even if their fields change. Fully reasoning about mutation necessitates reasoning about how mutable values change over time. From low-level libraries such as the Unix file API in C, to high-level libraries such as the networking abstractions in Java, temporal constraints crop up all the time.

LSL can readily express temporal properties. Consider a `fresh` function that is intended to return a different natural number every time it is called. Simply asserting that `fresh` returns a natural does not capture the key invariant about the function. Another kind of contract, known as a *trace contract* [27], is needed to enforce freshness:

```
(: ids unique-list?)
(define ids empty)

(: fresh (-> (AllOf Natural (Record ids))))
(define (fresh) —— elided ——)
```

The `(Record ids)` contract mutates the variable `ids` by appending the returned value from `fresh` onto the current value of `ids` each time the function is called. In other words, `ids` contains the *trace* of return values from `fresh`. When a variable is mutated, here `ids` via `Record`, its associated contract is checked again. Thus, the `unique-list?` predicate is checked on a sequence containing every value returned from `fresh`. This check ensures that `fresh` never yields a number that it previously returned.

The trace-contract mechanism can be used to enforce more complex properties. For example, if a protocol is specified via a state machine, then the trace predicate can run that state machine to determine if the protocol was violated.

5 A Few Homework Assignments

Table 1 shows the complete list of assignments given in the course. This section summarizes four of them to show what kinds of specifications and programs students write. Assignments come from a range of domains: games, security, low-level programming, and distributed systems.

5.1 Generating Mazes

Task. Students are asked to write and property-test transformations that convert between two maze representations and develop generators that randomly construct solvable mazes.

Rationale. This assignment has two goals. First, round-trip properties, especially between non-bijective data representations, are common and useful in real-world applications of PBT [16]. Second, naive random generation is often ineffective. Generating mazes by randomly generating lists of cells does not produce solvable mazes. Hence, students must develop smart generators that guarantee structural properties of mazes.

Details. One way to represent a maze is as a list that contains the type of a cell (e.g., empty, wall, exit) paired with a position. This *sparse* representation of a maze assumes missing cells are walls and the width (height) is inferred from the maximum x (y) coordinates.

After writing and testing functions over this data representation, it becomes apparent that it is painful to read and write mazes directly. Working with a human-friendly data representation, such as the following *dense* representation, becomes desirable:

```
'((X X P)
  (E X _)
  (_ _ _))
```

Students write conversion functions to and from the dense representation and test the round-trip property: converting a maze to and from a dense representation yields an equivalent maze. In doing so, one realizes that every maze has a unique dense representation, but not a sparse one. Although not explicitly stated in the assignment, a correct solution must define a non-trivial equality over mazes to faithfully test the round-trip property.

5.2 Cryptography and Timing Attacks

Task. Given a password-checking function that is susceptible to a timing side-channel attack, students write a property that detects this vulnerability. They then modify the function so that it is no longer vulnerable.

Rationale. Intensional properties, which reflect *how* a computation proceeds and not just what it computes, are often overlooked in discussions about software specification. Yet there are many domains where intensional properties are as essential as extensional ones. Security researchers are particularly attuned to such considerations, where side-channel attacks can leak private information. One approach to formally modeling intensional properties is to allow programs to request intensional information at run time. Once intensional aspects are reified, contracts and tests can check them like any other ordinary property.

Details. For this homework, students are given an insecure `password=?` function that returns as soon as the two input passwords differ. Thus, the time it takes `password=?` to execute is proportional to the length of the correct prefix of a given password attempt. This leak can be used to infer what the password should be in far fewer tries than what brute force guessing requires [3]. First, given a way to measure the number of character comparisons a function makes (i.e., a deterministic proxy for time), students define a property for secure password checking. The original `password=?` function should fail this property. Second, students adapt `password=?` such that it still works correctly, but also passes the timing specification. Implementing this modification requires wasting a precise amount of time on useless character comparisons.

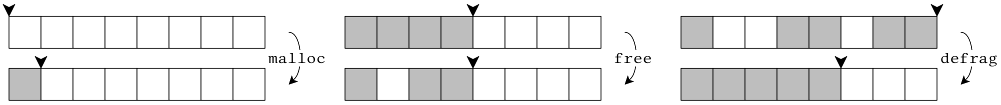

Fig. 2. Memory Operations

5.3 Manual Memory Allocator

Task. Students implement three memory-manipulating functions (`malloc`, `free`, `defrag`) for an idealized memory allocator, and write an imperative `fibonacci` procedure using these abstractions.

Rationale. Model checking tends to verify temporal properties that stipulate the order in which events should occur [1]. Such properties are important in larger systems where components must adhere to an ordered protocol. This assignment concerns temporal properties in a low-level programming context.

Details. A bump allocator splits an array of memory cells into two regions: a used portion and a free portion. The frontier between these two regions is tracked as an index into the array. Allocation increments the index, returning the address of a free cell to the user. Once the frontier index reaches the end of memory, defragmentation compacts all active cells and resets the frontier index. Figure 2 illustrates this process.

Students write a simple memory allocator following the scheme from Figure 2 and then a `fibonacci` procedure that uses their allocator. Bugs in the allocator propagate and compute nonsensical results when executing `fibonacci`. The final task is to construct a trace contract [27] that guarantees a memory cell is returned from `malloc` only if it (1) had never been allocated in the past or (2) had been allocated but was subsequently freed. Such a contract provides a simulacrum of what sophisticated tools, such as Valgrind, check about real-world low-level programs [31].

5.4 Chandy–Lamport Snapshot Algorithm

Task. Students are asked to implement the Chandy–Lamport distributed snapshot algorithm using a purely functional message-passing interface.

Rationale. Specification is useful at varying scales—from the smallest units of code to entire systems. The most complex programming assignment in our course showcases how specification works on the larger scale of a distributed system. This assignment serves as an introduction to concurrent programming for many students and hints at the difficulty of reasoning about and maintaining invariants in concurrent systems.

Details. Consider a network of banks that continuously transfer money among themselves. At any point in time, a bank might want to see a snapshot of how much money is in the network and where it is. Since no individual has a global view of the network, banks must request this information from all others in the network. The challenge is to implement an algorithm that computes a *consistent* snapshot of the network: intuitively, all money is accounted for in the snapshot. This problem is well-known in the field of distributed systems and the Chandy–Lamport algorithm is the standard solution [4].

Students implement Chandy–Lamport using a message-passing library similar to the `big-bang` interface from HtDP [12]. They must check that their implementation always provides consistent snapshots (i.e., no money is lost or gained in the snapshot compared to the starting amount). Doing so is fairly nontrivial—small mistakes in the implementation can yield subtly incorrect behavior

(1) What was the main thing that you learned from this course? What do you think the course is supposed to teach?
(2) What is the connection between logic and computation?
(3) Imagine you are working on a system, written in Java, for processing financial transactions at a credit card company. Are there ideas, skills, or techniques you learned in this class that you think would help you do a better job implementing this system?
(4) Was there anything in the course that seemed completely irrelevant, or that you could not imagine applying in any other context?
(5) Have ideas or techniques from the course been useful in other courses?
(6) Are there topics or techniques that have been useful on internships or you think will be useful on a job?

Fig. 3. Interview Questions

that leads to inconsistent snapshots. Debugging by "thinking hard" is, for most students, not productive. Describing invariants of the algorithm, writing property tests on smaller pieces of code, and unit-testing functions, all become useful for achieving a correct implementation.

6 Transferring Skills

Pedagogic languages have advantages over industrial-strength languages in the classroom, but using them risks tying the skills to the language. Students should come away from our course with the confidence to apply software specification techniques regardless of the language or tools they end up using in a job. Skill transfer must be addressed explicitly as students often cannot decouple the medium from the message.

One way to encourage skill transfer is to have assignments that go beyond the course content. Two assignments have students use their knowledge and skills outside LSL (Table 1). The first is to write two memos: one addressed to technically proficient coworkers that analyzes the strengths and weaknesses of a particular PBT library; the other is addressed to a less technically minded manager explaining why the chosen PBT library should be adopted. Any language and library is acceptable, though most students select popular PBT frameworks in Java or Python. Writing memos reinforces lessons from the course by having students explain the benefits of PBT in their own words. The second assignment has students redo several of their previous homework solutions using their chosen PBT library. This exercise helps connect concepts from class to real-world libraries, which often have a higher amount of incidental complexity than LSL.

7 Student Experiences

To help assess our approach, we interviewed students who took our class as well as students who took the previous version taught with Lean. The Lean version of the course was well liked, but we suspect this had more to do with the instructor and course logistics than with the material. The Lean course introduced students to mechanized theorem proving, proceeding from fixed data up through numbers and lists. Its final assignment was a proof of the correctness of insertion sort.

We interviewed ten students, four who took the Lean course, and six who took our version. Those who took our course finished it around six months before the interview, whereas students who took the Lean course finished it a year or more before the interview. Interviews followed guidelines from our institutional review board, answering the questions in Figure 3.

7.1 Lean Course Interviews

While these four students represent only a small window into the experience of the hundreds of students who have taken this course, the commonality of their responses is revealing. Although none reported disliking the course, none could imagine how any skills they learned had any bearing on software engineering. For a course ostensibly introducing students to the field of formal methods, with the intention of connecting mathematical reasoning to real software, this is a damning result.

Overall, students considered the purpose of the course was to teach the mechanism of formal proof: specifically, proofs on small examples. **Lean Student 2** said the course was about "learning how to correctly construct proofs... proofs regarding booleans, and then proofs regarding integers." Similarly, **Lean Student 1** reported doing proofs of "very basic things... like commutativity or associativity of multiplication" and that insertion sort "was a huge proof for us."

When pressed about the skills they were learning, or how they might use them, students reported bleak results. **Lean Student 3** explained that "even though Lean was fun, I feel like I extracted more educational value from doing the on-paper proofs in algorithms." **Lean Student 1** was not sure if the course "taught any practical applications to prove software." **Lean Student 4**, when describing their perspective on the skills they learned, said: "I like [the course]; I can see why it fits into computer science, but I didn't see why it fits into the software concentration."

The question about building a hypothetical payment processor produced universally negative results, with illuminating responses. Some students noted that they would reach for skills learned in other classes, e.g., **Lean Student 1** said they would "look towards the software side of solutions like, what are some pre-existing testing libraries... like JUnit," and **Lean Student 4** explained that "we didn't really go up to big applications of proving algorithms." Interestingly, **Lean Student 2** stated that "if I were a regular software developer making a transaction processor for credit cards, then I probably wouldn't be using [anything from the course], but if I were making the libraries that actually did the math, then I feel like I would." This gives weight to our suspicion that students thought the skills they were learning only applied to properties about booleans and numbers—**Lean Student 2** thinks that if one were to implement a *math* library, then perhaps some of the skills they learned would be useful, but since it is business logic, the skills are not relevant.

Our final question related to jobs and internships. **Lean Student 2** answered most concisely to the question of whether any skills from the class were useful to them in their job: "admittedly, no." **Lean Student 4** went into more detail, stating "there may be certain applications [for Lean]... if you need to really prove exhaustively certain applications, but... industry does not work like that... people prove programs work... largely through testing. I'm working at a finance company now, and there is nothing related to [Lean]." This demonstrates that **Lean Student 4** understood, in theory, the generalization of the simple properties to larger software systems, but did not find anything concretely useful given the vast distance between the programs shown in class and the programs encountered at work. **Lean Student 4** elaborated that the techniques used in the class were "limited to academia" or to "top engineering organizations working on cutting edge research," not, implicitly, at an ordinary finance company.

7.2 LSL Course Interviews

As with the first set of interviews, these six students offer a limited view into the experience of the several hundred that went through the class, and further, the apparent depth of their understanding of the material varied widely. For some, the main takeaway appeared to be the importance of carefully thinking about all cases in code and the value of testing. For others, it was clear that they had learned to recognize the value of extracting logical properties of systems and accepted that property-based testing was a useful way to validate those. All, though, seemed both to have understood that the course was about understanding the precise behavior of code, and all were able to recognize that the skills transferred.

These responses are in sharp contrast to responses from students in the Lean course, where the skills they learned, about the construction of formal proofs, seemed inextricably linked to the small examples through which they were demonstrated. While learning how to construct proofs may be a useful skill, it is not clear that this does anything directly to increase the likelihood that students produce reliable software.

When asked what the course was about, **LSL Student 1** responded "formalizing how programs work." Other students shared similar ideas, with **LSL Student 3** saying the "main things that I learned were thinking about how to check if a function was working correctly." **LSL Student 2** said that they "never really thought about the correctness of programs before taking [the course]." For **LSL Student 6**, the main takeaway was "the importance of testing your code."

Throughout the interviews, details emerged about what the students got out of the course. **LSL Student 4** shared that the course helped them "be very intentional with [their] programs." For **LSL Student 5**, the idealized memory allocator assignment (Section 5.3) made a particularly strong impression. **LSL Student 5** said that learning about mutation and aliasing "would definitely be useful for pretty much any job" and that hearing about C now conjures a "mental image of how `malloc` and `free` work."

Most students were able to connect skills learned in the class to our hypothetical scenario involving credit card transactions. **LSL Student 2** mentioned checking "if money is credited into your account that amount cannot be negative," and **LSL Student 1** mentioned making sure "people can't enter zero for their credit card number." **LSL Student 6** thought about how such a system would likely have a fraud detection mechanism and could "give the [fraud detection] system a bunch of inputs to test it and make sure it actually catches the fraudulent ones… or most of the fraudulent ones." Other students provided vague answers. For instance, **LSL Student 5** told us that in such a system, skills from the class could help with "making sure all the functions that you have do what you want them to do."

Although it was not a direct response to a question, a final interesting statement from **LSL Student 6** was that: "for every job, not just CS, but in general, being able to make sure the work you're doing is correct, and does what you actually wanted to do is pretty important." That this student made such an observation, when talking about the class, is precisely our goal.

8 Software Specification for All

This report describes our implementation and experience teaching a course dedicated to software specification. While some specifics may be particular to our institution, there are plenty of broad lessons for others to consider.

Teach software specification explicitly. Software specification and verification are distinct skills. Courses that focus on verification tend to have specifications that are simple and thus the skill of constructing them is not even worth mention. Software specification in real life is often challenging and hence worthy of explicit instruction.

Consider where specification fits into existing curricula. We have the luxury of an entire course devoted to this topic. At other institutions, such modules may have to be integrated into an existing curriculum. There are plenty of points where this makes sense. In a software-engineering course, PBT fits well as a supplementary testing technique alongside unit tests. In a course on typed functional programming, contracts can complement types and may be used to strengthen type signatures in the absence of full dependent types. Courses on object-oriented programming could benefit from a thorough treatment of data abstraction and its enforcement. Courses on imperative programming could benefit from a thorough treatment of mutation and how to enforce temporal constraints. A software security class might discuss how to formalize the notion of a side channel and detect it at run time. A systems class might discuss correctness properties of a memory allocator and how automated tools can check them. In short, specification is relevant across many domains and deserves to be emphasized across the curriculum.

Use real-world examples and larger programs. The interviews confirm that students came away from the Lean version of the course without seeing its relevance to software construction. This attitude seemed to be primarily due to the kinds of specifications encountered in the course, which barely went beyond simple mathematical properties. Without specifications from actual software systems, students see no connection to programming. Assignments that draw from specifications of real-world systems provide a remedy. Even though the programs are simplified, and the programming language is not realistic, the specifications can still be thought-provoking.

Provide a gradual path from unit tests to verification. Scaling up specifications demands techniques that scale too. Contracts and property-based testing are a powerful combination that is both reasonably lightweight (hence appropriate for freshmen) and expressive. In our context, using a theorem prover to verify that the Chandy–Lamport algorithm results in a consistent snapshot would be infeasible. As a waypoint between unit tests and full verification, contracts and PBT not only make the transition smoother but are independently useful techniques to master.

Avoid distractions by using the right tools. Pedagogic tools remove barriers that get in the way of learning skills. Thus, some infrastructure is needed to deliver a seamless student experience. Many of the assignments in this course would not be possible without dedicated linguistic support.

LSL works due to our institution's curriculum, which uses DrRacket and the teaching languages from HtDP. Similar infrastructure may be created for other educational environments. For example, others have adapted the HtDP curriculum to statically typed languages, such as OCaml, by porting features of the teaching languages as ordinary libraries [32]. Likewise, the functionality of LSL could be packaged as a library or framework in any host language—especially those with metaprogramming facilities.[2] Nearly every language has a library, or often several, inspired by QuickCheck [5]. High-quality libraries implementing software contracts are less common, but plenty of languages have some level of support. For instance, Clojure [19] has a popular contract library called `clojure.spec`. A lightweight wrapper around these libraries could easily make them suitable for beginning students. While using a fit-for-purpose teaching language enables the highest level of control, many of the benefits can be achieved, with far less instructor effort, using well-designed libraries.

Our hope is that these lessons inspire others to contemplate how software specification fits into their undergraduate curricula. Specification is not a purely academic subject, but a practical skill. Students deserve an education that makes it relevant to software development.

Acknowledgments

This work was supported by NSF grant SHF 2116372. Michael Ballantyne, Ryan Culpepper, Matthias Felleisen, Eden Gugsa, Ben Lerner, and the anonymous ICFP reviewers provided valuable feedback. Lisa Oakley assisted with recruiting students for interviews. Luke Jianu and Andrey Piterkin contributed substantially to the implementation of LSL. Finally, the course would not have been possible without our co-instructors, teaching assistants, and students. Thank you all.

[2] Indeed, one of our students made a partial LSL clone with Python decorators as a hobby project.

Data Availability Statement

An up-to-date version of the software and course material associated with this paper can be found at: https://doi.org/10.5281/zenodo.15653660. An archived version is also available [28].

References

[1] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[2] Henry G. Baker. 1993. Equal Rights for Functional Objects or, The More Things Change, The More They Are The Same. *ACM SIGPLAN OOPS Messenger* (1993). doi:10.1145/165593.165596

[3] David Brumley and Dan Boneh. 2005. Remote Timing Attacks are Practical. In *USENIX Security Symposium (SSYM)*.

[4] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *Transactions on Computer Systems* (1985). doi:10.1145/214451.214456

[5] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/351240.351266

[6] William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/1640089.1640133

[7] Marcus Crestani and Michael Sperber. 2010. Experience Report: Growing Programming Languages For Beginning Students. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/1932681.1863576

[8] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Conference on Automated Deduction (CADE)*. doi:10.1007/978-3-319-21401-6_26

[9] Peter C Dillinger, Panagiotis Manolios, Daron Vroon, and J Strother Moore. 2007. ACL2s: "The ACL2 Sedan". In *User Interfaces for Theorem Provers (UITP)*. doi:10.1016/j.entcs.2006.09.018

[10] Carl Eastlund, Dale Vaillancourt, and Matthias Felleisen. 2007. ACL2 for Freshmen: First Experiences. In *ACL2 Workshop*.

[11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The Structure and Interpretation of the Computer Science Curriculum. *Journal of Functional Programming (JFP)* (2004). doi:10.1017/s0956796804005076

[12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009. A Functional I/O System or, Fun for Freshman Kids. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/1596550.1596561

[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs*. MIT Press.

[14] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming (JFP)* (2002). doi:10.1017/S0956796801004208

[15] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/581478.581484

[16] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing In Practice. In *International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3639581

[17] Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. 2023. Tyche: In Situ Analysis Of Random Testing Effectiveness. In *User Interface Software and Technology (UIST)*. doi:10.1145/3586182.3615788

[18] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-Parametric Polymorphic Contracts. In *Dynamic Languages Symposium (DLS)*. doi:10.1145/1297081.1297089

[19] Rich Hickey. 2020. A History of Clojure. In *History of Programming Languages (HOPL)*. doi:10.1145/3386321

[20] Sára Juhošová, Andy Zaidman, and Jesper Cockx. 2025. Pinpointing the Learning Obstacles of an Interactive Theorem Prover. In *International Conference on Program Comprehension (ICPC)*. doi:10.1109/ICPC66645.2025.00024

[21] Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random Testing For Higher-Order, Stateful Programs. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/1869459.1869505

[22] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions With Error Messages. In *Onward!* doi:10.1145/2048237.2048241

[23] Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices!. In *European Symposium on Programming (ESOP)*. doi:10.1007/978-3-540-78739-6_2

[24] Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice Hall.

[25] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *Transactions on Programming Languages and Systems (TOPLAS)* (1988). doi:10.1145/44501.45065

[26] James H. Morris. 1973. Protection in Programming Languages. *Communications of the ACM (CACM)* (1973). doi:10.1145/361932.361937

[27] Cameron Moy and Matthias Felleisen. 2023. Trace Contracts. *Journal of Functional Programming (JFP)* (2023). doi:10.1017/S0956796823000096

[28] Cameron Moy and Daniel Patterson. 2025. Artifact: Teaching Software Specification (Experience Report). doi:10.5281/zenodo.15653661

[29] Rex Page, Carl Eastlund, and Matthias Felleisen. 2008. Functional Programming and Theorem Proving for Undergraduates: A Progress Report. In *Functional and Declarative Programming in Education (FDPE)*. doi:10.1145/1411260.1411264

[30] D. L. Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM (CACM)* (1972). doi:10.1145/361598.361623

[31] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference (ATC)*.

[32] Chihiro Uehara and Kenichi Asai. 2015. Cross Validation of the Universe Teachpack of Racket in OCaml. In *Trends in Functional Programming in Education (TFPIE)*.

Received 2025-02-27