Mocking Temporal Logic

Colin S. Gordon
csgordon@drexel.edu
Department of Computer Science
Drexel University
Philadelphia, Pennsylvania, USA

Abstract

Temporal logics cover important classes of system specifications dealing with system behavior over time. Despite the prevalence of long-running systems that accept repeated input and output, and thus the clear relevance of temporal specifications to training software engineers, temporal logics are rarely taught to undergraduates.

We motivate and describe an approach to teaching temporal specifications and temporal reasoning indirectly through teaching students about *mocking dependencies*, which is widely used in software testing of large systems (and therefore of more obvious relevance to students), less notationally intimidating to students, and still teaches similar reasoning principles. We report on 7 years of experience using this indirect approach to behavioral specifications in a software quality course.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Specification languages**; **Software verification and validation**; • **Social and professional topics** → **Software engineering education**.

Keywords: temporal logic, software specification, software testing, software engineering education

ACM Reference Format:
Colin S. Gordon. 2024. Mocking Temporal Logic. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E '24), October 24, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3689493.3689980

1 Introduction

Temporal logics have long been known to encompass key classes of specifications for how systems behave over time [17, 51] and are increasingly widely used in industry [31, 47]. Despite this, temporal logics are rarely taught at the undergraduate level in computer science and software engineering curricula. The most recent ACM/IEEE-CS/AAAI curriculum guidelines for CS mention behavioral specifications only in regard to concurrent programs [38, p. 224]. The ACM/IEEE curriculum guidelines for SE [6, 42] make no mention of behavioral specifications at all. Temporal logics build on other formal logics which *are* typically taught to undergraduates [10, 34] — propositional and first-order logic — which are already specifically perceived as challenging by undergraduates due to both notational challenges [24, 62, 63] and the challenges inherent to moving from familiarity with informal natural language to rigorous formal languages (itself motivating decades of work on using natural languages for specifications [26, 28, 59, 60]). Application of logic to software specifications is then a further learning challenge [20, 21, 25], distinct from learning the formal interpretation.

Additionally, most introductions on how to think about software, from introductory sequences to software engineering and software testing material tend to focus on single-input single-output specifications. This primes students to think about single invocations of a method or system, but leaves them less well-prepared to tackle system behavior over time. State-machine based testing is a well-established concept [11, 35], but has never been widely adopted in classrooms (likely due in part to its limited uptake in industry outside a few domains [58]). UML is still taught in many software engineering and computer science programs, and whatever its merits (its use in industry has been declining for years [50]) typically only class diagrams dealing with data and inheritance decisions are taught in any detail. In our experience, the main portions that addresses behavior over time are typically either briefly noted in passing (sequence diagrams [1, Ch. 17]) or completely ignored (state machines [1, Ch. 14]).

Yet behavioral specifications remain an important class of specifications for real systems. This naturally includes applications of formal methods, given that model checking of temporal logics is one of the most successful, and widely-deployed formal methods [18, 31] beyond type systems. But it also includes much more. A substantial fraction, if not a majority, of software written today takes the form of either long-running internet or network services or applications with graphical user interfaces. Both of these large (overlapping) categories of software are expected to continue running for extended periods of time, dealing with sequences of inputs (possibly from multiple sources simultaneously) interleaved

SPLASH-E '24, October 24, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1216-6/24/10
https://doi.org/10.1145/3689493.3689980

with actions of the system itself. This falls outside the batch-processing input-output form of programs used for the bulk of instruction in software testing. In fact, the overwhelming majority of software testing and software quality books lack any coverage of this topic, with a very few exceptions discussed in Section 4.6 [5, 48, 69, 70]. More broadly, any system whose behaviors are amenable to formalization using temporal logic *have* such temporal, behavioral specifications, regardless of whether anyone writes them down formally.

Of course, this is no surprise to practitioners building real systems, who have their own well-established ways of dealing with such behavioral specifications, which are suitable for teaching, have obvious industry-relevance, and allow us to teach some of the same specification principles. We can bridge the gap to temporal specifications and behavioral specifications more broadly (Section 2) by focusing on *mocking* in software engineering courses. Mocking is a common practice in industry [65, 68] of refactoring code to pass dependencies explicitly, in order to test code in isolation from those dependencies by using *testing-only implementations* (called *mocks*) that control for external factors. A simple example is passing a fake clock to code that checks the current time of day, so for example, behaviors that only run outside business hours in an actual deployment can be tested during business hours. This is also used for simulating errors (e.g., testing how the code handles lack of network connectivity without actually disconnecting the test machine's network access).

Mocking is relevant to this discussion because mocking can be used used to control *series* of interactions with a dependency, and then check that client code behaves correctly — much like assume-guarantee reasoning [36, 52]. Widely used mocking libraries often include their own domain-specific languages (DSLs) for defining these test-only implementations concisely, styled in a way that is simultaneously "regular" code (e.g., as an embedded DSL) but also shares many commonalities with finite-trace temporal logic specifications. While common software testing educational materials emphasize only the isolation-from-dependencies aspect of mocking (Section 4.6), there is a clear connection to temporal specifications, which we elaborate on in Section 4. They can specify that successive calls to a mock's methods return different values, or to specify certain kinds of relationships between mocked method inputs and results (including errors). Similar DSLs state assertions about how the client interacted with the mocks over time, including properties like calling certain methods in appropriate orders, appropriate numbers of times, or with particular kinds of arguments.

We put forth mocking as a sensible stepping stone to richer behavioral specifications, particularly but not exclusively temporal logics, because they separate the behavioral specifications themselves from totally new syntax; are widely used in industry and therefore of clear relevance to even the most application-focused students; have immediate utility to students in other contexts with minimal preparatory work (e.g., no need to build a rich or extensive formal model); and still effectively teach the same core specification and reasoning principles. We also argue that they avoid some of the linguistic confusion that appears when teaching temporal logics directly [29]. We are are not arguing that this approach should supplant instruction of temporal logics where it is offered, but that it may be an effective precursor, and may be a way to incorporate these principles in curricula that lack them, in a way that is more palatable to industry-focused or application-motivated students.

2 A Short Crash Course in Temporal Logic

Temporal logics are a class of modal logics, where the modal operators deal with time: applying a temporal operator to a proposition ϕ shifts the *time* at which ϕ is expected to be true. Temporal logics have a long, rich history with connections to other fields [8, 12, 17, 18, 51, 54], but are chiefly of interest in CS because they can be used to specify expected system behavior over time, and those specifications can be passed to tools (model checkers [8, 18, 31]) that automatically find cases where system models violate those descriptions of required behavior. Over the years, model checking has accumulated an impressive résumé of real-world successes [31, 47].

The most common form, (propositional) *Linear Temporal Logic* (LTL), addresses a single linear notion of time: the state of the world (or, computer system) as it evolves during a particular execution.[1] Beyond propositional logic's usual operators, it includes operators that shift the evaluation *time* at which their propositional arguments are expected to hold.

The main property is evaluated at the moment a program starts, but modalities can express claims about later parts of execution. $\Box\phi$ / "always" ϕ asserts that ϕ must be true at the current moment, *and* at *all* future moments. This is often used to state invariants (e.g., $\Box$ValidState). $\Diamond\phi$ / "eventually" ϕ asserts that ϕ is true at *some* point finitely far in the future — perhaps right now, but perhaps only once 3,000,000 steps from now. In particular, it requires ϕ to definitely become true in the future. $\bigcirc\phi$ / "next" ϕ asserts that ϕ is true in the state after the current state. Some variants include $\Diamond^{-1}\phi$ / "previously" ϕ to assert that ϕ occurred in the past, which allows more properties to be stated more concisely [43].

LTL also includes an assertion $\phi \mathbin{\mathsf{U}} \psi$ / ϕ "until" ψ, which requires that not only is ϕ eventually true (in the sense above), but at all moments in time from now until then, ϕ must be true. This is also used for liveness properties, but additionally constrains the time between now and then.[2]

These operators can combine to express subtle properties. For example, LTL specifications like $\Box(\mathsf{RequestReceived} \rightarrow \Diamond\mathsf{ResponseSent})$ (i.e., whenever a request is received, it is

[1] Other variants deal the space of possible future executions, rather than "the one that happens" [17, 23, 40].

[2] A careful reader will wonder if "until" makes "eventually" redundant; indeed, $\Diamond\phi \equiv \mathsf{True}\ \mathsf{U}\ \phi$, but beyond being directly tied to earlier temporal logics that lacked "until," "eventually" is just very convenient.

eventually responded to) are common — this captures a critical sort of property for long-running server processes.

3 Teaching Behavioral Specifications

Temporal logics are the most prominent and most well-established of a broader family of specifications, called *behavioral* specifications, which includes a wide range of type systems and program logics specifying which series of steps a program is permitted, required, or forbidden to take [4, 61, 66]. It is worth distinguishing behavioral specifications, and considering their place in undergraduate curricula, because they deal with program behaviors beyond the reach of more common (lightweight) static analyses and widely used type systems. Traditional input-output tests as covered in most undergraduate-level software engineering curricula do not directly address behaviors over time. For as long as behavioral specifications (particularly temporal specifications) have been known as useful, the verification community has worked on teaching them to students. One recurring challenge in having undergraduate students work with these specification classes is providing adequate supporting material, in the forms of reference material, a range of existing examples *applied to software specification* (since this is a distinct skill from learning specification forms themselves [24, 25]), and tool support for students to build intuition. Historically, this has limited the options to temporal logics. Relevant undergraduate-suitable books with extended case studies exist for TLA+ [41, 71], SPIN [9], and mCRL2 [30], but few other systems. Recent years have seen significant additional efforts to supplement classic textbook approaches with other material for teaching temporal logics [37, 39, 46, 55], though typically in the context of a full course devoted to formal methods broadly, temporal logics specifically. Many institutions' curricula constraints make the addition of such a course at the undergraduate level difficult. Some institutions do teach temporal logics as a topic in concurrent programming courses [9] (hence the idiosyncratic placement of the ACM/IEEE-CS/AAAI mention of temporal logics); our institution is one such place, but these courses are typically taken by a small minority of students, and the focus on small concurrent algorithms leaves many students with an impoverished view of the concept's applicability.

Other classes of behavioral specifications (beyond temporal logics) generally have either only researcher-targeted monographs [4]; no tools or unmaintained tools [1, 35]; or both [3]. Mocking is one of two exceptions: undergraduate-ready materials for the basics exist (Section 4.6, though not emphasizing the behavioral aspects), and a plethora of actively-maintained professionally-used libraries, requiring only a one-line inclusion in a dependency management tool for most languages. The other partial exception, property-based testing, is discussed in Section 6.

An additional difficulty is convincing students of the broad applicability of *any* software correctness tool they have not heard is already in widespread use throughout industry. Historically, formal methods have been most eagerly and successfully adopted outside academia in specific areas where the costs of bugs are prohibitively high (such as the infamous Intel Pendium FDIV bug [53] driving adoption of formal methods at Intel [32], or distributed systems work at Amazon [47]). Various formal methods have been successful in such conditions because the combined costs of bugs and difficulty of finding them through other means makes the additional learning curve for tools beyond standard testing seem "worth it." It is generally easy to convince students of this for those specific domains, but difficult to convince them that the same styles of specification and thinking are just as valuable for *other* domains (in other words, transfer is hard [49, 56]). This leaves many students with the impression that these techniques are valuable *only* in those specific domains, rather than as general principles. This is a drastic improvement over the common impression 20 years ago that anything beyond testing was wildly impractical, but also a far cry from teaching students to adopt general reasoning principles regardless of domain. Mocking *is* widely used in industry. And because common applications deal with isolating tests from unpredictable or difficult-to-configure dependencies, there are a wealth of easy-to-cover domains of relevance, which many upper-level students have prior experience with (web requests, general networking, database connections, etc.).

4 Mocking Temporal Specifications

The core idea of mocking is to pass the code under test a fake, testing-only implementation of some object the code depends on. This testing-only implementation, called a *mock* object, has two jobs. First, it simulates specific reactions to the client code, to trigger specific behaviors (e.g., returning an error to trigger client error handling). Second, it either directly checks that the client interacts as expected (e.g., calling methods with expected arguments and/or in appropriate orders), or records enough information for test code to do so.

Mocking requires no special support; it is fundamentally a clever refactoring. However, test suites tend to require a new mock for every test, so mocking libraries that permit mock configuration within each test, using concise descriptions of mock behavior — usually an embedded domain-specific language (DSL) with some similarities to temporal logics — are popular. These DSLs are claimed to yield more readable descriptions of how the mock should behave; in our experience (Section 5) this is at least somewhat true, in the sense that we do not observe mistakes seen in learning LTL [29].

In Java, one of the most popular tools for this, also used in our teaching (Section 5), is Mockito. Configuring a Mockito mock of Figure 1's `interface T` begins with:

```
T dep = mock(T.class);
```

```
interface T {
  boolean isValid();
  int get(int index);
}
```

Figure 1. A simple interface for our example.

This allocates a mocked object whose internals are provided by Mockito using reflection. Notice that this constructs a mock for an *interface* with no default implementations (though abstract and concrete classes can also be mocked in this way). Each method at this point returns a default value for its return type (e.g., `0` or **`null`** or **`false`**).

Now we can consider the typical form of a temporal logic verification query: env ∧ program ⇒ spec. That is, we typically are interested in the question of whether under certain environmental assumptions env, the program being examined satisfies the spec, where all three components are given in the form of temporal logic formulae. Each of these three pieces has a direct analogue in the use of mocking libraries: the program is the actual program code, and the environment and specification receive separate assume-guarantee-style treatments in the form of Java code.

4.1 The Program

Whereas typical temporal logic usage requires a model of the program (either as a temporal formula, or as a state machine like a Büchi automaton [12] that stands in direct correspondence to a temporal logic formula), the model in this case is the actual program code itself. The only change is the requirement that the code be refactored so that test cases can use the mock instead of the original dependency. For example, instead of instantiating the `T` directly, the client code would need to accept it as an argument (directly or via a factory), so that test code can provide the mock (here, `dep`) and the normal program can provide the normal dependency.

This refactoring by itself is straightforward to motivate to students. Every student understands that when testing code whose behavior depends on the time of day, you want to be able to swap out the time source for testing: no one wants to stay at work past 5pm just to run a test case for functionality triggered after close of standard US business hours, or wait for a holiday to test holiday behavior.[3] Even better, this type of refactoring often benefits not only testing, but functionality as well: the system may want to support a choice of local time or a network time server when available.

4.2 Assume: Specifying Environmental Behaviors

Method behaviors can be configured using a DSL consisting of method calls on `dep`. For example,

```
when(dep.isValid()).thenReturn(true);
```

configures `dep.isValid()` to always return **`true`**. We could also configure it to throw an exception:

```
when(dep.isValid()) // □(isValid ⇒ IllegalState)
    .thenThrow(new IllegalStateException());
```

Critically, chained calls mimic state changes:[4]

$$\texttt{//}\ (\neg\text{isValid})\ \mathrm{W}\left(\begin{array}{c}\text{isValid}\wedge\text{return(true)}\wedge\\ \bigcirc(\square(\text{isValid}\Rightarrow\text{IllegalState}))\end{array}\right)$$

```
when(dep.isValid()).thenReturn(true)
    .thenThrow(new IllegalStateException());
```

The above code configures the first call to `isValid()` to return **`true`** (*if* it occurs, which is not required — this is the purpose of the *weak* until operator), and all subsequent calls will throw the specified exception. (The LTL formula in the comment uses weak until to enforce that the *first* call returns true.) Behaviors can also be input-dependent, using both literals and the *matchers* (essentially, predicate objects) common to advanced JUnit usage:

```
when(dep.get(0)).thenReturn(1);
when(dep.get(greaterThan(0))).thenReturn(2);
when(dep.get(lessThan(0)))
    .thenThrow(new IllegalArgumentException());
```

Semantically, in the case where no specifications of method behavior overlap and disagree, the series of `when` calls can be intepreted as the conjunction of temporal logic clauses. An individual call chain of the form:

$$\text{when}(d.m(\overline{args}))\overline{.\text{then}\mathrm{X}_i(v_i)}^{n}$$

can be interpreted as a temporal logic formula requiring that from the initial state, the ith call with matching arguments returns or throws the ith value, and for all matching calls after the nth, the behavior for the nth call is used. In the case of overlap the last specification wins (though the documentation recommends avoiding this).

A natural question for instructors familiar with temporal logics is whether one can define recurring alternating or evolving behavior. Not all mocking frameworks permit this, but Mockito includes a way to provide custom code to compute a behavior for the mocked method (`thenAnswer`), rather than simply providing values or exceptions. If that custom code captures state, it can change its behavior over time, and if the state is shared with handlers for other methods it is possible to coordinate interactions of mocked methods. This use is of course less declarative.

4.3 Guarantee: Asserting Interactions

Mocking frameworks *augment* traditional testing assertions, they do not replace them. So for example, if a client is supposed to return a particular value, or value with a particular

[3] A few students have had the good fortune of running DrRacket on Halloween, and can appreciate that it was tested in advance.

[4] The *weak until* operator W is like "until" but doesn't require the second condition to happen, if the first remains true: $\phi\ \mathrm{W}\ \psi \equiv (\phi\ \mathrm{U}\ \psi) \vee (\square\phi)$

property, when run with a particular mock, those checks are performed the traditional way (e.g., with JUnit assertions).

Mockito provides the ability to check properties of how the code under test interacted with the mock in the past. Mocks record the sequence of method calls made after all configuration calls, providing a finite trace of interactions over which assertions can be made. These properties are evaluated at the *end* of the trace, so we use the past-time variant of eventually, "previously" ($\Diamond^{-1}$, at some point in the past) to indicate the temporal logic analogues.

Simple assertions can verify that certain calls occurred at some (any) point before the end of execution, like checking that `get` was called with 0 at some point:

```
verify(dep).get(0); // ◇⁻¹(get(0))
```

There are extensions to check that interactions never occurred (`verify(dep, never()).get(1)`), or occurred some number of times using matchers (the `never()` above is actually a matcher accepting 0, and matchers such as `times(5)` and `and(atLeast(2),atMost(4))` work).

By default, successive assertions are conjoined, that is

```
verify(dep).get(0); // ◇⁻¹(get(0))
verify(dep).get(1); // ∧◇⁻¹(get(1))
```

asserts the same properties as if the lines were swapped. Asserting that certain interactions took place in a particular order requires a layer of indirection:

```
InOrder o = inOrder(dep);
o.verify(dep).get(0); // ◇⁻¹(get(0)...
o.verify(dep).get(1); //    ... ∧ ◇(get(1)))
```

asserts that elements 0 and 1 were retrieved *in that order*, by writing the checks in that order. Additional setup can assert the relative order of interactions across multiple mocks.

There are two main differences between typical LTL specifications and mock specifications. First, because this is a testing framework, all specifications are necessarily over finite traces. Second, because assertions are typically checked at the end of a test case, mocking libraries are tailored for *past-time* temporal assertions — the `verify` calls are evaluated in the last state of the execution rather than the first. (Note, however, that the mock behaviors configured with `when` remain forward-time declarations, as they are established prior to (subject) program execution, and without knowing the length of the execution trace.

4.4 Putting It All Together

Figure 2 gives a longer example of a full JUnit test demonstrating the mock configuration as specifying an environment, the program (fragment) under test, and a specification of expected program behavior under that environment. Comments give corresponding (future-time) LTL specifications of the mock/environment viewed from before program execution in the "Environment" section, and corresponding

```
@Test public void checkValid() {
  // Environment
  T dep = mock(T.class);
  //□(isValid ⇒ return(true))
  when(dep.isValid()).thenReturn(true);
  // □(∀n ≠ 0. get(n) ⇒ IllegalArg)
  when(dep.get(lessThan(0))).thenThrow(
    new IllegalArgumentException());
  when(dep.get(greaterThan(0))).thenThrow(
    new IllegalArgumentException());
  // ¬get(0) W (get(0)∧...
  // ...return(3) ∧ ◯□(get(0) ⇒ return(5)))
  when(dep.get(0)).thenReturn(3).thenReturn(5);

  // Program
  Client c = new Client(dep); // pass mock
  int result = t.sumValidElements();

  // Specification
  // finalProgramResult = 3
  assertEqual(3, result);
  // ⋄⁻¹(get(0) ∧ ⋄⁻¹(isValid()))
  // equivalently ⋄⁻¹(isValid() ∧ ⋄(get(0)))
  InOrder o = inOrder(dep);
  o.verify(dep).isValid();
  o.verify(dep).get(0);
}
```

Figure 2. A longer example of env ∧ program ⇒ spec.

(past-time) LTL specifications of expected program behavior in the "Spec" section. These comments are only for clarity here regarding the association between mock configuration and verification and LTL; we never show students formal LTL notations and do not ask for their use. The mock is configured to always return true for `isValid()`, and throw an `IllegalArgumentException` when asked for elements at non-zero indices. For index 0, *if* `get(0)` is ever called, that call returns 3 and afterwards subsequent calls return 5. Expressing this in LTL is quite involved, requiring the weak until operator W. After calling the code under test (passing the mock dependency), the test checks the final result, and verifies that `isValid()` was called before `get(0)`. For space we check less than we might for a summation.

This demonstrates the typical shape of a complete test involving mocks, though some cases involve the construction of more interesting data structures (e.g., if the equivalent of `sumValidElements` took arguments). The three segments must always occur in this arrangement. The mock (environment) must be configured prior to being passed to the code under test (the constructor, or other API call where the mock is passed, might immediately interact with it). The "program"

(fragment) under test must come before the assertions, which are eagerly evaluated. The regularity of this structure, and the fact that it parallels the common "Arrange-Act-Assert" structure of input-output unit tests, seem to help students structure their tests.

4.5 Putative Advantages of the Mocking Approach to Temporal Specifications

Syntax. As suggested above, because the specifications are expressed in a subset of Java (and the students entering this course have previous experience with Java), these specifications avoid preemptively discouraging students who are wary of formal notation due to previous struggles in math, logic, or CS theory courses. (Similar arguments apply to mocking libraries embedded in other languages.)

The fact that the "syntax" for specifying assumptions (`when`) and checking guarantees (`verify`) are clearly distinct seems to alleviate confusion regarding which parts of a logical specification serve what roles. In this setting, the breakdown into env $\wedge$ program $\Rightarrow$ spec has crisp syntactic distinctions: env is anything with a `when`, spec is anything with an `assert` or `verify`, and everything else is the program being tested. In the case of Mockito specifically, the syntactic difference between specifying dependency behavior (assumptions) and client code requirements (guarantees) makes explicit which part of a specification is for what purpose.

Complementing Other Material. Unlike standard temporal logics, students see the applicability of mocks immediately, and see it as a natural complement to other software testing discussions. Many of the natural applications of mocking deal with dependencies which are *already* abstracted and passed, via dependency injection or explicit configuration: abstractions over backing stores (local, remote, varied formats, replicated or not); messaging channels (local IPC vs. remote); platform (OS) abstractions. These topics tend to arise in other classes students have already taken or are taking concurrently (networking, systems programming, courses with open-ended projects). Students who have taken a course writing sockets-based networking code or web requests are often particularly eager to pick up mocks.

Building to Temporal Logics. While mocking libraries are typically imperative under the hood, as in our earlier examples, they are largely used as declarative specifications in the style of a typical logical specification: the order of `when` clauses is immaterial, and in practice most `verify` clauses focus on checking whether or not certain interactions between client and mock occurred at all. Only the `InOrder` verification reflects an imperative nature.

This bit of imperative leakage into the specifications exists in all widely-used mocking libraries, though it is not fundamental. Samimi et al. [57] propose fully declarative mock specifications closer in flavor to temporal logic, though also note that one of the primary advantages of the established DSL style of mock (and assertion) specifications is that they are perceived as lower-overhead and less intimidating.

4.6 Teaching Materials

Most textbooks on software testing do not address mocking at all. In fact, we are aware of only 4 that discuss mocking, and these still do not discuss behavioral mocking over time in depth. So pushing students to think about changes in behavior over time requires going beyond current off-the-shelf teaching materials. Tarlinder's 2016 *Developer Testing* [69] mentions the ability to have subsequent calls to a method behave differently just once, with a trivial example, and no explanation of why this might be valuable. The latest 2020 edition of *JUnit in Action* [70] gives a simple example of this, updated from earlier editions [67]. Aniche's 2022 *Effective Software Testing* [5] does not mention this possibility, nor does Okken's updated (2022) *Python Testing with pytest* [48]. None of these books discuss verifying that interactions with mocks occur in the specified order, emphasizing the original, primary goal of mocks: isolating code under test from dependencies which may have unpredictable or environment-dependent behavior.

This standard coverage does reinforce the idea of specifying the environment as part of a specification, and gets students familiar with the basics of the mocking DSL, in a form where all environment specifications are simple "always" ($\Box$) properties and all verifications are simple "previously" ($\Diamond^{-1}$) properties. So it is an effective foundation for building to temporal properties. For Mockito, the chaining of `thenX` methods and the use of `InOrder` verification are syntactically modest additions beyond the coverage common to the three Java-centric books above; extensions for other frameworks and in other languages are similarly (syntactically) modest.

As with LTL, the two key challenges are teaching students to think about behaviors over time (rather than single-round input-output behavior), and teaching students how to formalize behaviors described in natural language in an unambiguous notation. We cover these by building atop basic non-temporal mocking material like that above.

5 Experience

We have taught mocking through this implicit temporal logic lens for the past 7 years (starting in the 2017-2018 academic year), to roughly 70 students per year, in a broad-coverage software engineering course focused on software quality, largely through the lens of testing. The course is required for software engineering students, serves as an elective for computer science students, and is taken primarily by upper-level students. Prior to teaching about mocking, we review classic closed- and open-box testing, code coverage, and in more recent years basic property-based testing. Mocks are covered at the end of a week that discusses stateful testing in general, including a high-level overview of state machine testing [11]

to come up with method call sequences that drive an object with state into interesting conditions, though students are not asked to directly apply state machine testing — they are only asked to actually work with mocks in assignments.

Approach. The introduction to mocking itself starts with a typical framing for teaching this material, focusing on isolating code from its dependencies for repeatable and efficient unit testing (e.g., replacing a dependency used to check the current time, or calls to expensive external dependencies like databases), in line with textbooks selected for the course in part based on this goal. The textbooks (initially Tarlinder [69], later Aniche [5]) broach coverage up to the use of mocks which verify a that simple set of operations occur, though not in any specific order.

During lecture, a live demo is given where various small tests pass and fail based on changes to the mock specification and what we verify, to give students a sense of what each part of the assume (Section 4.2) and guarantee (Section 4.3 portions of the test mean. The three pieces of these tests are explicitly connected to the common three-piece structure of unit tests, commonly known as "Arrange-Act-Assert" (which correspond almost perfectly to env, program, and spec, with the exception of arranging other non-mocked input values to the tested code). This is the jumping off point for one of the course's homeworks. Lecture material does not discuss temporal logic directly at all.[5] Instead, after showing small examples of mocks changing behavior over time, we solicit student examples of cases where they have encountered a dependency that returns different values during repeated interactions. Most students have previously encountered various subsets of stateful network connections (via sockets, or payment or login APIs), iterators, database connections, local file APIs, control inversion in GUIs (polling user input), low-level device APIs, and other examples.

Practice and Assessment. Each year we have given students an assignment to test a client of an FTP-like network connectivity library given only a Java interface (with Javadoc comments) for the server connection (plus the documentation for Mockito, a Java mocking library). They are instructed to test the behavior of the client code corresponding to different scenarios depending on which methods of the server connection interface return what kinds of data or errors. Each scenario requires students to configure a mock of the server connection, run the client code with that specific mock, and assert properties of the resulting execution. This setup exposes several of the features highlighted earlier:

- The connection being mocked acts as the environment
- The client under test is the system being specified

This setup avoids some challenges typical of teaching temporal logic directly. First, the configuration and assertion languages, while embedded domain-specific languages provided by Mockito, are Java code, which is less intimidating to students than LTL; no student has ever expressed *apprehension* about Mockito's syntax.[6] Second, the client-server setting is a natural setting for assume-guarantee reasoning: students recognize that in making a series of requests to a server, the server may respond differently based on the particular data available, and that testing the client together with the server and an explicit (or generated) dataset is often impractical.

Students are not given detailed introductions to Mockito, but are given template code that constructs an almost-trivial mock of the interface they need to simulate, and pointers to the specific sections of the Mockito documentation required to complete the assignment (in total, this fragment amounts to roughly 3.5 printed pages' worth of generously formatted text, most of which is example code). The assignment asks students to write 14 tests (some similar to each other). One representative test is "Test that if the connection succeeds but there is no valid file of that name, the client code calls no further methods on the connection except `closeConnection`. That is, the client code is expected to call that method exactly once, but should not call other methods after it is known the file name is invalid." Another is "Test that if the initial server connection succeeds, then if an `IOException` occurs while retrieving the file (requesting, or reading bytes, either one) the client still explicitly closes the server connection." Some tests deal with longer scenarios (10–12 client-mock interactions). Some of the tests (like the second above) are intentionally under-specified, which prompts many students to approach course staff with questions about under-specification.[7] All of the tests require students to conceive of a series of behaviors (interactions between client and server), correctly configure the mock (environment) to mimic the server side, and write appropriate `verify` calls to check that the client reacted appropriately. Thus these do test students' ability to convert high level descriptions of expected (conditional) behavior into precise behavioral specifications, in the form of mocks rather than temporal logic.

5.1 Student Reactions and Outcomes

Students generally do quite well on the assignment, with most students writing correct mocks and correct assertions for all but one or two scenarios.[8] The provided client code has multiple bugs which students are expected to identify through testing: they are told multiple scenarios will result in test failures for correct tests, but are not told which. This

[5] Some iterations of the course have momentarily mentioned temporal logics several weeks after this assignment when discussing static analysis tools, but we have never discussed this for the mocking assignment.

[6] In contrast, when we teach LTL and TLA+ in graduate courses, syntax is often a major stumbling block.

[7] This is a cross-cutting, explicitly-taught topic throughout the course; any plausible interpretation is given credit.

[8] Note that at the moment, ChatGPT is terrible at using mocks.

appears to result in students thinking through their specifications (mocks and assertions) more carefully based on students' self-reported experiences, because it reduces the inclination to automatically interpret a passing test as correct.

The course is typically taught in the fall quarter, at a university where roughly 2/3 of CS and SE students are on a spring-summer co-op cycle (i.e., those students are in class for roughly 6 months in fall and winter quarters, and spend roughly 6 months in the spring and summer quarters in full-time co-op positions. Many students choose this specific university because the co-op program (3 six-month co-ops over 5 years) offers significant industry experience before graduation, and as a result the student body skews more towards career-focused students than other CS and SE programs.

This assignment has been popular among students in the course for all 7 years it has been used, based on unsolicited direct student feedback and end-of-course evaluations. Each year a small number of students report using mocking on an earlier co-op experience, and a larger number report it coming up in interviews for upcoming co-ops or full-time roles during the course.[9] This evidence of employer interest in mocking seems to reinforce student attention for even the most application-minded students. We do not have specific evidence that time-varying mocking specifically is valued; empirical studies of mocking do not investigate that level of detail in how mocks are used [64, 65, 68] and few resources on mocking distinguish between mocks that do or do not change behavior over time.

5.2 Expected and Unexpected Benefits

As anticipated, students were not intimidated by notational challenges; learning Mockito is perceived by most students as similar to learning any other testing library. Students are able to learn the tool quite quickly. Newcombe et al. [47] report experienced engineers being productive in TLA+ in about 2 weeks of spare time; our undergraduates are given 2 weeks to learn Mockito and apply it to an abstract problem specification and generally succeed, while taking 3-4 other courses.

We initially anticipated spending significant time clarifying the detailed semantics of the specifications and assertions. In practice, we have spent almost no time on this at all in 7 years. The Mockito documentation is concise and example-driven; students effectively learn a set of specification patterns [22], which they seem to be able to apply effectively based on the documentation alone. Instead almost all (faculty and TA) office hour discussions related to the assignment are devoted to understanding the event orderings implied by the English specifications, which is one of the intended main activities for the assignment, as it is one of the core intellectual challenges in formalizing any behavioral specification.

[9]Fall term is the most common interview time for both final-year students seeking full-time roles and earlier students seeking a spring-summer co-op; nearly half of the undergraduate CS majors, and all undergraduate SE majors, are on this co-op cycle.

An unexpected benefit of emphasizing behavioral specifications via mocking rather than temporal logic is that many semantic errors seem to be preempted as well. Greenman et al. [29] report on the kinds of mistakes made in translating English into LTL, and comparing the categories of mistakes to those made by students on this assignment is enlightening: *most kinds of errors they observe are difficult or impossible to make with Mockito's specifications.* This appears to be partly incidental to Mockito's tests having different expressivity from LTL, and partly due to using a DSL in a familiar PL.

Part of the difference is likely due to working with different classes of linear-time models. Greenman et al. study mistakes with state-based LTL, where each point in time is a state of the system, so many properties can be true in a given state. Mockito's specifications deal with event-based linear-time properties, where the moments in time are not states of the system but which method of the mock is being called (with which arguments). As a result, many properties that would be expressed with LTL's *until* operator do not make sense (those describing state invariants). This immediately suppresses the most common sources of English-to-LTL misconceptions in Greenman et al.'s study. Greenman et al. also note that participants sometimes forget that ϕ U ψ requires ψ to eventually become true. As noted in Section 4, the closest common analouge in Mockito (chaining `when` configurations) corresponds to the *weak* until operator which is consistent with that expected behavior.

The next most common family of mistakes in their study deal with so-called "implicit" operator assumptions: cases where a participant in their study needed to explicitly write a temporal operator, but did not, presumably believing that certain parts of the formula would *implicitly* be shifted by an appropriate temporal operator. The most common of those were cases of assuming an implicit *F* (*finally* / *eventually*) operator. In Mockito, all properties are evaluated at the end of a finite trace with regard to earlier execution, so are *implicitly* eventually/finally (or previously) properties, removing opportunities to make this mistake. The next most common mistake in Greenman et al.'s study was assuming an implicit always/globally operator. In an event-based model, few "always" properties make sense, though some desirable properties are difficult to express (and therefore, rarely expressed in practice with mocks), such as always-eventually properties.

To some extent the way time evolves in Mockito specifications is more similar to English than LTL specifications. Notice that while the conditionals ("if A, B") in the assignment examples above do not explicitly say that the conclusion of the implication holds later than the antecedent, readers infer this. Linguists discuss this phenomenon (in almost all natural languages) through the concept of a narrative "advancing the time of discourse" [19]: simply continuing a narrative of a situation implicitly advances the time being discussed. However, in LTL, every shift in time must be explicitly marked with a modal operator. Mockito's DSL in

Java does not perfectly adhere to discourse semantics, but is closer in some ways: chaining a `thenReturn` later in a chain of calls corresponds to it happening later in execution, and `InOrder` is similar (things mentioned later are expected to happen *strictly* later). Effectively, the LTL modalities that have analogues in Mockito's Java DSL are *implicit* in the DSL, as in natural language discourse.

One class of mistakes from Greenman et al.'s study that *does* appear in student submissions for mocking assignments, and is the most common conceptual mistake in our experience, is what Greenman et al. call `BadStateIndex` mistakes: where a property is meant to be true at a certain point in a trace, but a developer instead writes a property that states its truth at a *different* point in the trace. Typically for the mocking assignment this takes the form of having a mocked method whose behavior changes over time throw an exception or return a specific value (e.g., one triggering termination of a loop) one call earlier or one call later than is appropriate for the intended property (sometimes, not always, by virtue of misaligning two parts of the specification which are always executed in pairs at runtime, such as a check for availability of more data and a request for more data if the first call is successful). This results in not triggering the intended client behavior.

5.3 Student Challenges

Most student questions to course staff are about uncovering subtleties of thinking and speaking/writing about these sequences of interactions – both the sequencing (the temporal aspect) and the two-party (assume-guarantee) aspect — on their way to deciding how to express the intended conditions with Mockito. These questions are similar to those we get when teaching TLA+ in a graduate course, so this challenge is expected and indicates overlap in the reasoning principles are being exercised by mocking (over time) and TLA+. A small number of students need reminding that Mockito can give different behaviors to subsequent calls to a mocked method, having forgotten about the in-lecture demonstration of chaining `thenReturn`s and skimmed through the documentation too fast. But a quick reminder suffices.

Among the students who do poorly, there are a few recurring themes in why and in what way they went astray. Most students who make `BadStateIndex` mistakes in Greenman et al.'s taxonomy do so only once or twice, not systematically. The most common repeated mistakes are *not* due to misunderstanding the scenarios. Some students focus so heavily on configuring the mocks via `when` that they simply forget to write the assertions at all. Fewer include appropriate `verify` calls but omit traditional JUnit assertions to check that the client returns an appropriate value for the tested scenario, having become convinced that JUnit and Mockito cannot mix (despite examples mixing them in lecture).

A less-common tooling-specific difficulty relates to implicit mock configurations. If a method's behavior is not explicitly configured for some method input, it returns the default value of the return type; for booleans, some students either simply forget to configure a needed method, or assume that unconfigured boolean-returning mock methods return **`true`** though the default is actually **`false`**. In these case, tests often pass for the wrong reason (by not testing the intended behavior). This is reversed from the corresponding situation in LTL, where omitting information about some behavior would cause verification to fail.

Some students who are accustomed to always reading the code they are working with line-by-line struggle to adapt their own debugging practices to mocks, because it is impractical (or due to dynamic bytecode rewriting, impossible) to read or step through the mock configuration or execution in detail. In contrast to tracing code down to a library, they cannot consult documentation for the behavior of a mocked call the way they would for a network library call, because they themselves configured the behavior semi-declaratively. So understanding the execution of a test without a debugger requires solid understanding the mock configuration language. It is still possible for students to step through their test code in a debugger if they do not attempt to enter the mocked methods. In effect students are forced to treat specifications abstractly, though this may differ for other mocking frameworks or other languages.

A less common point of struggle that is directly related to the assignment's intended conceptual challenges is that a small but noticeable number of students are not accustomed to speaking or writing about how the state of a system or component changes over time at a higher level than concrete scenarios with all details present (e.g., in a specific execution), though they have experience working with at least some of the examples of Section 5 in earlier courses. These students benefit from talking through smaller, simpler examples one-on-one, often file I/O examples, which students tend to have the most experience with.

5.4 Contrasting Instruction on Temporal Logic

The term after we teach this material, there is a graduate course on software quality. The graduate course is structured around reading research papers (mostly classic or highly-influential applied papers), with relevant assignments as well. When we teach that course (the contents vary by instructor), students spend 2 weeks reading and discussing papers introducing the roles of formal specification (Wing [72] and Butler and Johnson [13] one week) and temporal logic (Newcombe et al.'s discussion of TLA+ at Amazon [47], and the introductory portions of Wayne's *Practical TLA+* [71] the next). The audience for that course is predominantly masters-level students with undergraduate degrees from other universities.

Class time discusses the readings, covers the semantics of basic temporal operators (those common to both LTL TLA+ [44]), and includes demonstrations of TLA+ [41] and the associated model checker. Students are then asked in an

assignment to write TLA+ specifications for a small buggy traffic light model (itself given in PlusCal), and find those bugs. This assignment was in fact the original inspiration for the the assignment described in Section 5; a desire to cover similar types of specifications with undergraduates, without taking as much time to ease students into formal notation and without focusing on such a small target application, was the original motivation for considering mocks in the undergraduate course.

We have limited experience with a small number of students who took the undergraduate course with the mocking work proceeding to the graduate course when we have taught it (all undergraduates we have taught in this graduate course previously took the undergraduate course). Undergraduates who learned mocking of behavioral specifications in the undergraduate course anecdotally appear to pick up TLA+ more easily than graduate students who have not, based on one-on-one discussions over the years and high grades on the TLA+ assignment. However, this is *strongly* confounded by the fact that undergraduates are already required to have strong grades in all earlier courses for the department to permit them to take any graduate course, while the graduate students merely need to be in the program. So this should definitely *not* be taken as evidence that students with prior exposure to mocking necessarily learn temporal logics better or more easily than those who have not.

Likely more informative is the nature of the questions and discussions raised by students in the two contexts. Aside from syntactic questions (TLA+ syntax justifiably garners more questions than Mockito's DSL) and discussion of iterative liveness properties [2] (not expressible via mocks), the conceptual questions from students learning TLA+ and Mockito are remarkably similar, and deal primarily with questions of how to draw formal distinctions based on natural language descriptions of expected changes in behavior.

6 Alternatives

Earlier we noted that temporal specifications are only a subset of the broader class of behavioral specifications. It is worth considering what other kinds of behavioral specifications might be introduced at the undergraduate level, in the context of a full course on formal methods or temporal logic being out of reach. As noted in Section 3, *property-based testing* (PBT) is the only other approach covering some behavioral specifications that has undergraduate-ready material and working tools for hands-on experience.

PBT [15] is increasingly taught to undergraduates [73] (including in this course), but despite its long-established [16] support for testing imperative programs (which is used in industry [7, 14, 27, 33, 45]), that aspect of property-based testing is taught even more rarely than temporal logic. This style of specification does not *directly* describe overall behaviors across time. The standard PBT approach to testing stateful systems is via a form of model-based testing: a (guided) randomly generated sequence of program actions is applied to both the code under test and a more abstract model. The generation process on sequences of operations can filter to include only sequences for a particular scenario in order to check properties of the code in that situation, so does permit a form of assume-guarantee type split in the specification.

The test passes if the expected relationship holds between the final state of the real code and the final state of the model, and any results returned have the expected properties. This results in the specification being less directly stated than in input-output properties — it is essentially tailored to checking simulation, which is lower-level (and more code-intensive to implement) than typical behavioral specifications. (It also emphasizes testing a dependency in a client environment, rather than testing client code in an environment describing abstracted dependency behavior.)

We have had a small number of honors students study these techniques for their honors projects.[10] But in the context of broad-coverage software testing and quality course, even these strong students spend significant time just learning how to organize the pieces of the specification, which tends to end up taking much more code to specify. This is reasonable for checking simulation with a model, but verbose enough to be a distraction for lighter-weight properties like checking that certain methods were called in order, and such properties also require manually implementing relevant state in the model — which is precisely what mocking DSLs avoid.

7 Conclusions

We have described 7 years of experience using mocks to teach behavioral specification in an undergraduate software engineering course, rather than ignoring explicitly-behavioral specifications or reaching for classic but seemingly more (learning-)time-intensive approaches like temporal logics. Our experiences have been largely positive, in the sense that most students are able to grasp and use this class of behavioral specification — which has clear relation to Linear Temporal Logic — in a relatively short time. While there are reasons to prefer teaching temporal logics directly when possible, we believe that teaching mocking with mock behavior varying over time is a valuable alternative that fits into more undergraduate curricula, and has advantages in terms of student approachability and motivation.

Acknowledgments

This work was supported by NSF grants CCF-2220991 and CCF-1844964.

[10] Honors students at our university earn a special "honors" designation by doing extra projects in a certain number of courses during their degree. Acceptance to the program that permits this is highly competitive, so these are among the strongest students at the university.

References

[1] 2017. OMG Unified Modeling Language (OMG UML). https://www.omg.org/spec/UML/2.5.1/PDF Version 2.5.1.

[2] Bowen Alpern and Fred B Schneider. 1985. Defining liveness. *Information processing letters* 21, 4 (1985), 181–185.

[3] Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. 1999. *Type and effect systems: behaviours for concurrency*. World Scientific.

[4] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. 2016. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages* 3, 2-3 (2016), 95–230.

[5] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide*. Manning Publications.

[6] Mark Ardis, David Budgen, Gregory W Hislop, Jeff Offutt, Mark Sebern, and Willem Visser. 2015. SE 2014: Curriculum guidelines for undergraduate degree programs in software engineering. *Computer* 48, 11 (2015), 106–109.

[7] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4.

[8] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[9] Mordechai Ben-Ari. 2006. *Principles of concurrent and distributed programming*. Pearson Education.

[10] Mordechai Ben-Ari. 2012. *Mathematical logic for computer science*. Springer Science & Business Media.

[11] Robert Binder. 2000. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

[12] J. Richard Büchi. 1960. Weak Second Order Arithmetic and Finite Automata. *Zeitschrift für Math. Log. und Grundl. der Math.* (1960).

[13] Ricky Butler and Sally Johnson. 1993. Formal methods for life-critical software. In *9th Computing in Aerospace Conference*. 4516.

[14] Alessandro Cimatti, Sara Corfini, Luca Cristoforetti, Marco Di Natale, Alberto Griggio, Stefano Puri, and Stefano Tonetta. 2022. A comprehensive framework for the analysis of automotive systems. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 379–389.

[15] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[16] Koen Claessen and John Hughes. 2002. Testing monadic code with QuickCheck. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. 65–77.

[17] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263.

[18] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. 2018. *Handbook of model checking*. Vol. 10. Springer.

[19] Elizabeth Couper-Kuhlen. 1989. Foregrounding and temporal relations in narrative discourse. *Essays on tensing in English* 2 (1989), 7–29.

[20] Svetlana V. Drachova, Jason O. Hallstrom, Joseph E. Hollingsworth, Joan Krone, Rich Pak, and Murali Sitaraman. 2015. Teaching Mathematical Reasoning Principles for Software Correctness and Its Assessment. *Trans. Comput. Educ.* 15, 3, Article 15 (Aug. 2015), 22 pages. https://doi.org/10.1145/2716316

[21] Svetlana Drachova-Strang. 2013. *Teaching and assessment of mathematical principles for software correctness using a reasoning concept inventory*. Ph. D. Dissertation. Clemson University. http://tigerprints.clemson.edu/all_dissertations/1095/

[22] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*. 411–420.

[23] E Allen Emerson and Joseph Y Halpern. 1986. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)* 33, 1 (1986), 151–178.

[24] Kate Finney. 1996. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering* 22, 2 (1996), 158–159.

[25] Kate M Finney and Alex M Fedorec. 1996. An empirical study of specification readability. In *Teaching and Learning Formal Methods*. Academic Press.

[26] Norbert E Fuchs, Uta Schwertel, and Sunna Torge. 1999. Controlled natural language can replace first-order logic. In *14th IEEE International Conference on Automated Software Engineering*. IEEE, 295–298.

[27] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[28] Colin S Gordon and Sergey Matskevich. 2023. Trustworthy Formal Natural Language Specifications. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 50–70.

[29] Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. 2023. Little Tricky Logic: Misconceptions in the Understanding of LTL. *The Art, Science, and Engineering of Programming* (2023).

[30] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and analysis of communicating systems*. MIT press.

[31] Orna Grumberg and Helmut Veith. 2008. *25 years of model checking: history, achievements, perspectives*. Vol. 5000. Springer.

[32] John Harrison. 2010. Formal methods at Intel—An overview. In *Second NASA Formal Methods Symposium*, Vol. 8. 179–195.

[33] John Hughes, Benjamin C Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of dropbox: property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 135–145.

[34] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press.

[35] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. 2007. *Model-based software testing and analysis with C*. Cambridge University Press.

[36] Cliff B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (1983), 596–619.

[37] Jan Kofroň, Pavel Parízek, and Ondřej Šerỳ. 2009. On teaching formal methods: behavior models and code analysis. In *Teaching Formal Methods: Second International Conference, TFM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2*. Springer, 144–157.

[38] Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2024. *Computer Science Curricula 2023*. Association for Computing Machinery, New York, NY, USA.

[39] Markus A Kuppe. 2023. Teaching TLA+ to Engineers at Microsoft. In *Formal Methods Teaching Workshop*. Springer, 66–81.

[40] Leslie Lamport. 1980. " Sometime" is sometimes" not never" on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 174–185.

[41] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers*.

[42] Richard Joseph LeBlanc, Ann Sobel, Jorge L Diaz-Herrera, and Thomas B Hilburn. 2006. *Software engineering 2004: curriculum guidelines for undergraduate degree programs in software engineering*. IEEE Computer Society.

[43] Nicolas Markey. 2003. Temporal logic with past is exponentially more succinct. *Bulletin-European Association for Theoretical Computer Science* 79 (2003), 122–128.

[44] Stephan Merz. 2008. The specification language TLA+. *Logics of specification languages* (2008), 401–451.

[45] Wojciech Mostowski, Thomas Arts, and John Hughes. 2017. Modelling of Autosar libraries for large scale testing. *arXiv preprint arXiv:1703.06574* (2017).

[46] Tim Nelson, Ben Greenman, Siddhartha Prasad, Tristan Dyer, Ethan Bove, Qianfan Chen, Charles Cutting, Thomas Del Vecchio, Sidney LeVine, Julianne Rudner, et al. 2024. Forge: A Tool and Language for Teaching Formal Methods. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 613–641.

[47] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73.

[48] Brian Okken. 2022. *Python Testing with pytest*. Pragmatic Bookshelf.

[49] David N Perkins, Gavriel Salomon, et al. 1992. Transfer of learning. *International encyclopedia of education* 2 (1992), 6452–6457.

[50] Marian Petre. 2013. UML in practice. In *2013 35th international conference on software engineering (icse)*. IEEE, 722–731.

[51] Amir Pnueli. 1977. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*. ieee, 46–57.

[52] Amir Pnueli. 1984. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*. Springer, 123–144.

[53] Dick Price. 1995. Pentium FDIV flaw-lessons learned. *IEEE Micro* 15, 2 (1995), 86–88.

[54] Nicholas Rescher and Alasdair Urquhart. 1971. *Temporal Logic= Library of Exact Philosophy, Vol. 3*. Springer Verlag.

[55] Raven Rothkopf, Angel Leyi Cui, Hannah Tongxin Zeng, Arya Sinha, and Santolucito Mark. 2023. Towards the usability of reactive synthesis: Building blocks of temporal logic. In *Plateau Workshop*.

[56] Gavriel Salomon and David N Perkins. 1989. Rocky roads to transfer: Rethinking mechanism of a neglected phenomenon. *Educational psychologist* 24, 2 (1989), 113–142.

[57] Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. 2013. Declarative mocking. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 246–256.

[58] Ina Schieferdecker and Andreas Hoffmann. 2012. Model-based testing. *IEEE software* 29, 1 (2012), 14–18.

[59] Rolf Schwitter. 2002. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*. IEEE, 228–232.

[60] Hiroyuki Seki, Tadao Kasami, Eiji Nabika, and Takashi Matsumura. 1992. A method for translating natural language program specifications into algebraic specifications. *Systems and computers in Japan* 23, 11 (1992), 1–16.

[61] Christian Skalka, Scott Smith, and David Van Horn. 2008. Types and trace effects of higher order programs. *Journal of Functional Programming* 18, 2 (2008), 179–249.

[62] Colin Snook and Rachel Harrison. 2001. Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information and Software Technology* 43, 4 (2001), 275–283.

[63] Colin Frank Snook. 2001. *Exploring the barriers to formal specification*. Ph. D. Dissertation. University of Southampton.

[64] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 402–412.

[65] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing java systems: Why and how developers use them, and how they evolve. *Empirical Software Engineering* 24 (2019), 1461–1498.

[66] Robert E Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE transactions on software engineering* 1 (1986), 157–171.

[67] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. 2010. *JUnit in action*. Manning Publications.

[68] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: Automated test generation for database applications via mock objects. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 289–292.

[69] Alexander Tarlinder. 2016. *Developer testing: Building quality into software*. Addison-Wesley Professional.

[70] Catalin Tudose. 2020. *JUnit in action*. Manning Publications.

[71] Hillel Wayne. 2018. *Practical TLA+: Planning Driven Development*. Apress.

[72] Jeannette M Wing. 1990. A specifier's introduction to formal methods. *Computer* 23, 9 (1990), 8–22.

[73] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming* 5, 2 (2021).

Received 2024-07-05; accepted 2024-08-08