Validation of Formal Models: A Case Study

Pamela Zave[1][0000−0002−6568−2052] and Tim Nelson[2][0000−0002−9377−9943]

[1] Princeton University, Princeton NJ, USA, pamela@pamelazave.com
[2] Brown University, Providence RI, USA, tbnelson@gmail.com

Abstract. A *valid* formal model is well-defined in the portion of the real world being modeled, and relevant to the system-development project it is intended to support. This paper explains in depth what makes a formal model valid, and shows its critical importance in formal methods for system development, including program verification. The paper is based on a classroom presentation [8] of the well-known Peterson algorithm for locking in concurrent systems [23]. The paper shows how to formalize this case study in Alloy [10], and uses it to present a list of manual and automated techniques for validating formal models.

Keywords: formal methods · Alloy · Peterson lock

Dedication

This paper is dedicated to Cliff Jones, colleague, mentor, and friend. His influential invention of rely/guarantee reasoning is a well-developed and practical tool that has introduced the idea of domain modeling to many practitioners and developers of formal methods.

1 Introduction

One of the oldest sayings about computing is still one of the best: "garbage in, garbage out." This saying is equally true when the computer program is a verification tool, the input is a formal model of a computer system, and the output is verified assertions. A verified assertion is garbage if it is ill-defined or irrelevant in the real world of the computer system being modeled. Despite the excellence of today's verification tools, their output is, too often, garbage.

The process of ensuring that the input to a verification tool is not garbage, and thus will not result in garbage output, is *validation.* Validation is checking that a formal model is *valid*, meaning that it has *at least* the two properties shown in Figure 1. *Accuracy* refers to the relationship between the real world and all modeling abstractions: the abstractions must be faithful descriptions of the real world, whether they are intended to match the real world as it is now or as it should be. There are two kinds of abstractions, those in peoples' heads and those encoded in formal languages. *Comprehensibility* refers to the agreement between mental and formal abstractions: do the formal abstractions mean what the people working with them think they mean?

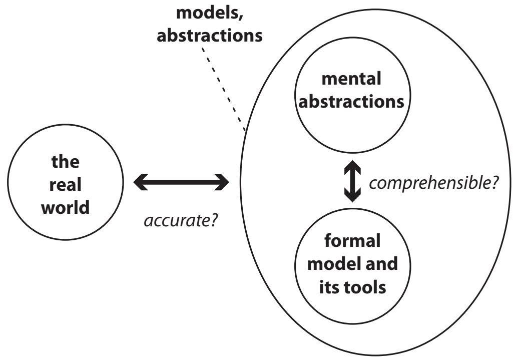

Fig. 1. The two essential questions answered by validation.

Because of the importance of mental abstractions, validation must be at least partially informal, but it can be assisted greatly by formal analysis and verification. Accuracy and comprehensibility are minimal criteria, and are often augmented with many others for an expansive definition of validity.

Informally, validation attempts to answer the questions, "Does this model mean what I think it means, and is that meaning faithful to the real world?" In considering what a formal model means, we rely on the framework summarized in §2. The crucial point of this framework is that there must be a separation between the *domain* or environment that motivates the need for a computer system, and the *system* that interacts with the domain to fulfill that need. In our experience most cases of invalid[3] formal models are due to flawed or inadequate modeling of the domain. This paper illustrates the power of good domain modeling in building good formal models.

While this seems to be a straightforward prescription for improvement, it may not be so easy to apply. Many published examples of domain/system separation are cyber-physical, where the domain is a collection of physical phenomena and the stand-alone system controls them in some way (see Jones's discussion in [16]). In contrast, most software development is concerned with adding something to an enormous base of existing code. In the latter case, the interface between domain and system lies deep in the semantics of the programming infrastructure. The domain is already formal in some sense, being constructed from digital logic, but this fact is of little use. In practice its behavior is far too complex to model completely, and much of it is unknown to the system developer.

The purpose of this paper is to provide an example of validating a formal model of a new software system designed to interact with an existing software system. We hope the example will serve multiple purposes. It provides a template for modeling some software domains, and also for modeling domain/system

[3] In the software-engineering sense of meaningless or irrelevant, not in the mathematical sense of not being a valid assertion.

boundaries found in code. The example should serve as a tutorial on formal methods, and—most importantly—as an introduction to validation.

First, §2 summarizes the framework for associating meanings with formal models. Next, §3 introduces the the case study, and the following four sections present the case study as a sequence of models. Finally, §8 summarizes the validation steps applied, and §9 discusses the overall conclusions about validation. This paper has room to show only fragments of the formal models, but the complete set of formal models can be found online [29].

2 A framework for formal modeling

The framework is drawn from three related papers on requirements engineering [12, 13, 28]. These papers introduce a method for deriving the specification of a computer system from its domain and requirements. Software-development processes today are more diverse, and various situations are constrained in different ways, so decisions must sometimes be made in a different order. Nevertheless, even though software development might not follow the *method* of these papers, the artifacts and relations they define are completely general and form a powerful framework for understanding the *meanings* of formal models.[4]

To be clear, it is necessary to distinguish between *meaning* and *formal semantics* as they are used here. Our favorite formal method is the combination of state-transition systems and temporal logic, as exemplified by Alloy, Event-B, TLA+, process algebras, and many others. In these related formalisms, a model is a state-transition system and its *semantics* is a set of traces.

Knowing the *semantics* of a model is not enough to know its *meaning,* which also includes how it corresponds to the real world. And without knowing its meaning, it is impossible to tell whether it is an accurate and comprehensible formal description of real or hypothetically real phenomena, i.e., whether it is valid. The simplest example of the difference concerns the primitive terms in the formal model. The meaning of the model depends on *designations*, which are natural-language descriptions of how instances of these primitive terms can be identified in the real world. In this paper, designations are spread throughout the text. Another, even more important, aspect of meaning will be discussed later in this section.

The artifacts of the Jackson/Zave framework are illustrated by Figure 2. A *domain* or environment is the portion of the real world, digital or not, that motivates the need for a computer system. The *system* is the computer system (hardware or software) that we are interested in, and that interacts with the domain in a purposeful way. All other terms are relative to these two.

Figure 2 is a Venn diagram over the universe of all phenomena. It shows that the domain and system usually contain different phenomena, but they always have some shared phenomena, which is their *interface.*

The *specification S* is the part of the formal model that constrains the behavior of the system. Ultimately there will be an *implementation M* that produces

[4] In the literature, the framework is commonly referred to as the Jackson/Zave model.

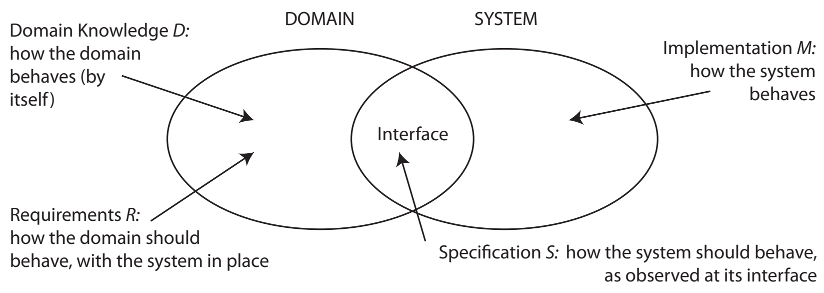
DOMAIN
SYSTEM
Domain Knowledge D: how the domain behaves (by itself)
Implementation M: how the system behaves
Interface
Requirements R: how the domain should behave, with the system in place
Specification S: how the system should behave, as observed at its interface

Fig. 2. The artifacts of a formal model. The Venn diagram represents sets of phenomena. Arrows indicate which phenomena are described or constrained by an artifact.

this behavior when enacted by one or more computers. Ideally either the specification is simpler and more comprehensible than the implementation, or the specification is efficiently executable so that S and M can be the same. The diagram illustrates one important way in which the specification can be simpler than the implementation: it is confined to the phenomena of the interface, which are part of the domain as well as part of the system.

Domain knowledge D is the part of the formal model that constrains how the domain behaves all by itself, without the influence of the system. *Requirements* R are the part of the formal model that describes how the domain should behave, with the system implemented and installed.

Semantically, a formal model is nothing more than a set of constraints on its constants and variables, whose values must satisfy all the constraints at all times. *The most important way to attach meaning to a formal model is to say, for each constraint, why it is or should be true.* In other words, what enforces or will enforce the constraint? In this framework, we attach meaning to each constraint by identifying it as domain knowledge, requirement, specification property, or implementation property. The real-world domain enforces the domain knowledge, and the implementation will enforce the specification.

In software engineering, automated reasoning was first used for the purpose of program verification, which is proving that an implementation satisfies its specification. In validation, automated reasoning has other—equally important—uses. We need to know that domain knowledge D and the specification S are consistent, because if they are not, the system cannot work as specified. Most importantly, we need to know that D and S together imply (enforce) the requirements R. This tells us that if the system is developed according to its specification, it will do the job it is intended to do. The case study will also show other ways in which verification tools can help in validation.

Even for program verification, a valid domain model may be essential. For example, rely-guarantee reasoning, pioneered by Jones [14, 15] and elaborated by Jones and many others [7], is a pillar of program refinement and verification. It is motivated by the fact that programs running concurrently with their environments often rely on assumptions about their environments (domains). Rely-guarantee reasoning shows how to incorporate parts of a formal domain model into proofs of program correctness.

For a much deeper and broader view of domain modeling than these brief definitions allow, you should read [11].

3 The case study

The subject of our case study is the well-known two-party Peterson lock used in multiprocessor programming [23]. Our presentation follows a class lecture by Herlihy [8], which presents the algorithm in a sequence of three versions, each written in Java.

In the case study, the domain consists of two user programs, each running in its own thread, and sharing a resource. The system to be developed is a lock program called by the user programs to enforce mutual exclusion with respect to accessing the resource.

The formal models in the case study are written in Alloy [10], and use the Alloy Analyzer Version 6, which incorporates linear-time temporal logic, for formal reasoning (see `alloytools.org`). Due to lack of space, this paper will explain the case study but not the Alloy language itself.

As we present the case study, we show how to address three challenges. The first challenge is converting from a program to a state-transition system, so that temporal logic can be used for specification and verification. Although there are many stylistic differences, the most important difference is that programming languages have built-in control mechanisms (e.g., method call and return, waiting by looping until a *while* condition fails) while state transitions rely exclusively on action preconditions for control.

The second challenge is to organize the model according to the Jackson/Zave framework, with a clear separation between the domain (as motivation for a software-development project) and the system (as the goal of the project). Because our example is very simple, the separation may seem grandiose and overdone, but this seems better than using an example that is more complex.

The third challenge is to validate the formal model. Herlihy's lecture tells us some things about preliminary versions of the lock. Our goal, reproducing validation in a real development project, is to discover errors without the help of an oracle. It is also important to note that the errors are realistic. Except for the known problems with preliminary versions of the lock, we committed all the modeling errors ourselves!

4 First version: Lock 1

4.1 States

In creating a state-transition model, the first thing we think about is its state. We don't know or care what the user programs do, except that in accessing the shared resource they have three states: *Uninterested* (in using the shared resource),[5] *Waiting* to use the resource, and *InCritSec* (in a critical section accessing the resource). Figure 3 shows the state of each user program and how it changes over time in a typical scenario.

In Lock 1 there is a *Lock* object with a flag for each of the two threads. The flags are represented by a set *flagRaised* containing the identifiers of all flags now raised. Figure 3 also shows the state of the lock and how it changes over time. As seen in the figure, the basic idea of Lock 1 is that user programs call *Lock* methods *lock* and *unlock* to enter and leave their critical sections. When *lock* is called it puts the identifier of the calling thread in *flagRaised*. The call returns when and only when the caller's flag is the only one raised. After using the resource, the user program calls *unlock*, which lowers its flag and returns immediately. As shown in Figure 3, if a thread is waiting for the lock, then the other thread's *unlock* will end the waiting.

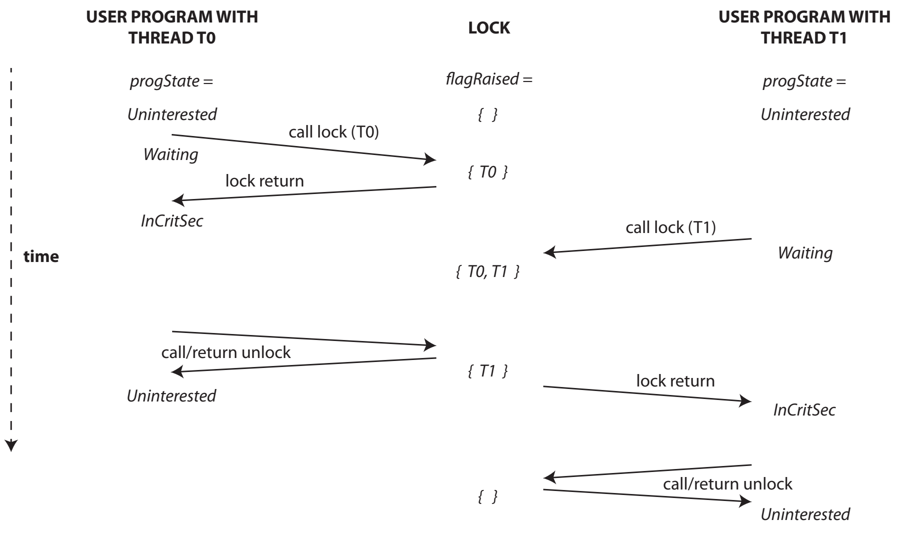

Fig. 3. A sample trace showing how Lock 1 works. Arrows show the flow of control of a thread as time passes.

On our first try, the state declarations and initial state in Alloy 6 look like this, with domain state and system state labeled:

```
sig Thread {}

abstract sig ProgState {}
```

[5] This state is often named *Idle*, but the program need not be idle—it can be busy doing something other than accessing the shared resource.

```
one sig Uninterested, Waiting, InCritSec extends ProgState {}

one sig Progs {  var progState: Thread -> one ProgState  }       -- domain

one sig Lock {  var flagRaised: set Thread  }                    -- system

pred initialState {
   # Thread = 2                                                  -- domain
   Progs.progState = Thread -> Uninterested                      -- domain
   no Lock.flagRaised                     }                      -- system
```

Members of the primitive set *Thread* serve as thread identifiers, as used by the operating system. The model has a *Progs* object containing a *progState* variable that is a function from threads to states of the user programs. For each thread, *progState* maps the thread to the current state of the program that the thread is running. Initially, all programs are *Uninterested*, and no flag is raised.

4.2 Actions

In a state-transition model, actions represent atomic changes to the model state. This means that all parts of an action are executed instantaneously, and cannot be interleaved with anything else. This assumption is obviously not true in the real world of this case study, and eventually we will need to check its validity—is it a safe approximation, or does it lead us into big mistakes? Figure 3 suggests that three actions can be considered atomic: *callLock, lockReturn,* and *unlock* (call and return together).

In the model, each action has a predicate expressing its enabling condition, and a predicate expressing its state change. Busy waiting in the lock program is replaced by the enabling condition on *lockReturn*, which says that *t*'s flag is raised and the other thread's flag is *not* raised. In these actions, control of the thread as it passes between user and lock programs remains implicit.

```
pred callLockEnabled [t: Thread] {           -- action initiated by domain
   t.(Progs.progState) = Uninterested  }
pred callLock [t: Thread] {
   callLockEnabled [t]
   Progs.progState' = Progs.progState ++ (t->Waiting)  --update prog state
   Lock.flagRaised' = Lock.flagRaised + t           }        -- raise flag

pred lockReturnEnabled [t: Thread] {         -- action initiated by system
   Lock.flagRaised = t             }
pred lockReturn [t: Thread] {
   lockReturnEnabled [t]
   flagRaised' = flagRaised                              -- no flag change
   Progs.progState' = Progs.progState ++ (t->InCritSec)  }  --update state

pred unlockEnabled [t: Thread] {             -- action initiated by domain
   t.(Progs.progState) = InCritSec  }
```

```
pred unlock [t: Thread] {
   unlockEnabled [t]
   Progs.progState' = Progs.progState ++ (t->Uninterested)  --update state
   Lock.flagRaised' = Lock.flagRaised - t                }   -- lower flag
```

Because Alloy 6 is based on linear temporal logic, it is necessary to express which traces we want as instances of the model. A trace is an instance if its initial state satisfies *initialState*, and its subsequent states are the results of the three actions.

```
pred delta {  some t: Thread |
   (  callLock [t] || lockReturn [t] || unlock [t]  )  }

pred anyTrace {  initialState && always delta  }
run anyTrace expect 1
```

Finally, there are requirements on how the domain should behave, with the lock implemented and the user programs using it. Fortunately, the requirements on a two-party lock are well known and easily expressed in temporal logic:

```
pred mutuallyExclusive {  # (Progs.progState).InCritSec <= 1  }
assert mutualExclusion {  anyTrace => always mutuallyExclusive  }
check mutualExclusion                                        -- VERIFIED

pred nonDeadlocking {  some t: Thread |
   (callLockEnabled [t] || lockReturnEnabled [t] || unlockEnabled [t])  }
assert noDeadlocks {  anyTrace => always nonDeadlocking  }
check noDeadlocks                                            -- VERIFIED

pred nonStarvation {  all t: Thread |
   t.(Progs.progState) = Waiting =>
      eventually t.(Progs.progState) = InCritSec  }
assert noStarvation {  anyTrace => always nonStarvation  }
check noStarvation                                           -- VERIFIED
```

The *expect 1* annotation on the command *run anyTrace* says that there should be traces that satisfy *anyTrace*. We run the Alloy Analyzer, and that is true. The tool also verifies the requirements (within a bounded scope, which will be discussed later), so everything looks great! We have a working lock design!

4.3 A little validation

Well, maybe, just in case, we should try a little validation. In this paper validation steps fall into three categories: (i) There are general-purpose checks, that apply to many (if not all) models, especially models of concurrent systems. (ii) There are checks that the domain and system are kept properly separate. If a constraint is partly enforced by the domain and partly by the system, it will be extremely difficult to understand and work with. (iii) There are checks on the validity of the domain model, which tend to be especially powerful. Validation steps are numbered, and listed under those numbers in Table 1.

The first three validation steps are general-purpose.

(1) In this model, every action should define the variable values in the post-state completely. This we check "manually," i.e., by inspecting the model. The Alloy style updates each of the two state relations *progState* and *flagRaised* all at once, so it is unnecessary to have separate "frame conditions" to say, "the state of the other thread does not change."

(2) Whenever a model has separate state components, and we expect their values to be correlated, it is worth checking that they are correlated. This can be done in Alloy by defining a state invariant and asking the Analyzer to verify it. When explicit proofs are being constructed (which is not necessary in the Alloy context), this is often the first step of a proof.

```
assert stateInvariant {
   anyTrace => always
   (all t: Thread |
      t.(Progs.progState) = Uninterested => t ! in Lock.flagRaised
   && t.(Progs.progState) = Waiting => t in Lock.flagRaised
   && t.(Progs.progState) = InCritSec => t in Lock.flagRaised    )  }
check stateInvariant                                          -- VERIFIED
```

(3) The third validation step is a check that control in the model works as intended. In the real world, each thread executes a sequence of actions. In the model, this translates to the expectation that in each thread, at any time, only one action is enabled. This is encoded in the following assertion:

```
assert sequentialControl {
   anyTrace => always
   (all t: Thread |
      ! (callLockEnabled [t] && lockReturnEnabled [t])
   && ! (callLockEnabled [t] && unlockEnabled [t])
   && ! (lockReturnEnabled [t] && unlockEnabled [t]) )  }
check sequentialControl                                   -- NOT VERIFIED
```

As the comment shows, the model is not valid in this respect. The Analyzer provides a counterexample showing that *lockReturnEnabled* and *unlockEnabled* can both be true at the same time, for the same thread. This is an important departure from reality, and there is no telling what mischief it might cause as versions of this model evolve.

In the real world, whenever *lockReturn* is enabled, *unlock* for the same thread cannot be enabled because the lock program has control of the thread. So the real problem here is that the domain and system are not well-enough separated, making it unclear when the domain or system can initiate an action. This means that control of the thread should be modeled explicitly. To model control, we add a third state relation, which is now the interface between domain and system—both sides can observe it, and whichever side has control can give it up. A thread is in the set *runningProg* if and only if it is running the user program rather than the lock code. Here is the new state and its initialization, new actions, and a new invariant:

```
one sig Progs {  var progState: Thread -> one ProgState,         -- domain
                 var runningProg: set Thread           }      -- interface

pred initialState { . . .
   Progs.runningProg = Thread . . . }             -- added to initialState

pred callLockEnabled [t: Thread] {               -- precondition on domain
   t.(Progs.progState) = Uninterested && t in Progs.runningProg  }
pred callLock [t: Thread] {
   callLockEnabled [t]
   Progs.progState' = Progs.progState ++ (t -> Waiting)   -- domain update
   Progs.runningProg' = Progs.runningProg - t          -- interface update
   Lock.flagRaised' = Lock.flagRaised + t            }   -- system update

pred lockReturnEnabled [t: Thread] {             -- precondition on system
   Lock.flagRaised = t && t ! in Progs.runningProg }
pred lockReturn [t: Thread] {
   lockReturnEnabled [t]
   flagRaised' = flagRaised                            -- no system change
   Progs.runningProg' = Progs.runningProg + t          -- interface update
   Progs.progState' = Progs.progState ++ (t -> InCritSec)  }  --dom update

pred unlockEnabled [t: Thread] {                 -- precondition on domain
   t.(Progs.progState) = InCritSec && t in Progs.runningProg  }
pred unlock [t: Thread] {
   unlockEnabled [t]
   runningProg' = runningProg             -- no apparent interface change,
     -- but interface actually changes twice, once after each state update
   Progs.progState' = Progs.progState ++ (t -> Uninterested)  --dom update
   Lock.flagRaised' = Lock.flagRaised - t                 }  --sys update

assert stateInvariant {
   anyTrace => always
   (all t: Thread |
      t.(Progs.progState) = Uninterested =>
         (  t in Progs.runningProg && t ! in Lock.flagRaised  )
   && t.(Progs.progState) = Waiting =>
         (  t ! in Progs.runningProg && t in Lock.flagRaised  )
   && t.(Progs.progState) = InCS =>
         (  t in Progs.runningProg && t in Lock.flagRaised    )  )  }
```

The validation check *sequentialControl* can now be verified.

(4) Having modeled thread control, we can check more thoroughly that the domain and system are kept as separate as they should be. The rules for this model are as follows: (i) Pure domain state can be read or written only when the domain has control of the thread. (ii) Pure system state can be read or written only when the system has control of the thread. (iii) Interface state can be read

or written at any time. The interface variable *runningProg* is well-defined, even though it can be changed by either system or domain, because it can only be changed by one action at a time.

This validation step is a manual check, the results of which are recorded in the comments on the new action model above. For example, *callLock* is initiated by the domain, and *callLockEnabled* reads only a domain variable and an interface variable. Within the action predicate *callLock*, the domain state changes, then the method call transfers control to the lock code, then the system state changes. The comments are important because the updates within the action are sequential only in our interpretation of the model's meaning, not in the model's formal semantics.

4.4 Domain behavior

(5, 6) The last two validation steps for Lock 1 are focused on the domain. It is always important to validate that the domain can do what it is expected to do or allowed to do, even when it is interacting with the system. We think of desired scenarios, encode them in predicates, then ask the Analyzer to instantiate the predicates. Here the first scenario is that each user program requests the lock eventually, at its own time. The second scenario is that both user programs request the lock at almost the same time, so that both *callLock* actions execute before either *lockReturn* can execute, and both users become *Waiting*.

```
pred bothProgsRequestLock [disj t0, t1: Thread]  {
   anyTrace
   eventually t0.(Progs.progState) = Waiting
   eventually t1.(Progs.progState) = Waiting     }
run bothProgsRequestLock expect 1

pred thereIsContention [disj t0, t1: Thread]  {
   anyTrace
   eventually Thread.(Progs.progState) = Waiting  }  -- both progs waiting
run thereIsContention expect 1                           -- INCONSISTENT
```

Note the way this domain knowledge and the requirements complement each other. This domain knowledge says that the programs will request the lock. The requirements say that when a program requests the lock, it will get the lock.

However, the second scenario cannot be instantiated, which is bad news—it means that, for some reason, the model is very unrealistic. But why? The actions don't look suspicious.

One part of the model that has not received much attention yet is the definition of *anyTrace*. If we look at it carefully and think about exactly what it means, it is saying that in any trace of this model, in any state, the enabling condition for one of the actions is true, and that action occurs. In other words, it is stating formally that there can be no deadlocks! The fact is that in Lock 1, contention causes a deadlock, and *thereIsContention* cannot be instantiated because an instance of this model cannot deadlock.

There is an alternative explanation having to do with the fact that Alloy 6 does not allow finite traces, but only infinite traces with finite representations ("lasso traces" terminating in loops). A deadlock stops all execution, resulting in a finite trace. Because a deadlock (in the strictest sense) results in a finite trace, no deadlock (in this strictest sense) can ever be part of the formal semantics (trace set) of an Alloy 6 model.

We think ours is a much better explanation because it is about the meaning of the model, rather than the details of some tool or temporal logic. Temporal logics with finite traces are much used in some areas of computing [5], and some model checkers allow finite traces [9].

The validation problem is fixed with the addition of a "stuttering step" *doNothing* to *delta*. This new action is enabled only when no other action is enabled. This allows a deadlocking trace to end in a loop of *doNothing* actions.

```
pred doNothingEnabled { all t: Thread | (
   ! callLockEnabled[t] && ! lockReturnEnabled[t] && ! unlockEnabled[t]) }
pred doNothing {
   doNothingEnabled && runningProg' = runningProg &&
   progState' = progState && flagRaised' = flagRaised  }

pred delta {  some t: Thread |
   (  callLock [t] || lockReturn [t] || unlock [t] || doNothing  )  }
```

Now we have a final version of Lock 1. With this version, *noDeadlocks* and *noStarvation* cannot be verified, which is the truth because Lock 1 can deadlock and starve.

5 Second version: Lock 2

Lock 2, as presented by Herlihy, has a different mechanism with a different lock state:

```
one sig Lock {  var polite: lone Thread  } --polite is one or zero threads

pred initialState { . . .
   no Lock.polite  . . .  }  -- replaces initialization of Lock.flagRaised
```

The *callLock [t]* action sets the value of *polite*[6] to *t*. The other two actions leave it unchanged. The all-important precondition to the *lockReturn* action is

```
pred lockReturnEnabled [t: Thread] {             -- precondition on system
   Lock.polite ! = t && t ! in Progs.runningProg  }
```

This means that when a client program calls *lock [t0]*, the call can return only when, after *t0*'s call, the other thread calls *lock [t1]*, thus setting the value of *polite* to *t1*. Because *unlock* does not change *polite*, it is an action of the user program only. Figure 4 shows the typical behavior of Lock 2.

[6] Presentations of this idea often call the polite thread the "victim;" we believe that "polite" more accurately conveys the intuition, while also being less alarming.

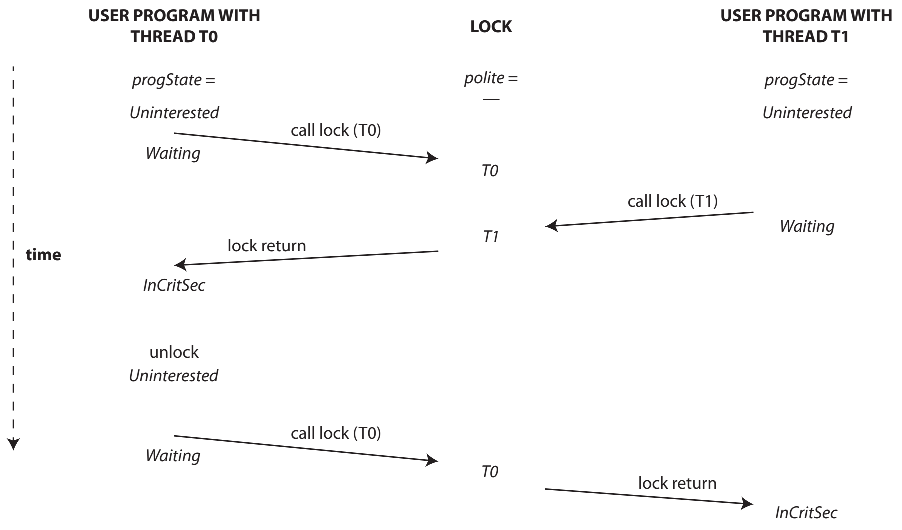

Fig. 4. A sample trace showing how Lock 2 works.

(7, 8) The model of Lock 2 satisfies all the previous validation checks. Yet it seems strange, doesn't it? This inspired us to come up with two other domain scenarios for validation. In the first scenario, one thread dies for some reason and never again requests the lock. Because there can be no contention, in a fault-tolerant system, the other thread should continue to work and access the shared resource freely. In the second scenario, there is no shared resource, so neither thread ever calls on the lock.

```
pred faultTolerantBehavior [disj t0, t1: Thread]  {
   anyTrace
   eventually always t1.(Progs.progState) = Uninterested
   always eventually t0.(Progs.progState) = InCritSec  }
run faultTolerantBehavior expect 1                         -- INCONSISTENT

pred noSharedResource [disj t0, t1: Thread]  {
   anyTrace
   always t0.(Progs.progState) = Uninterested
   always t1.(Progs.progState) = Uninterested  }
run noSharedResource expect 1                              -- INCONSISTENT
```

The Analyzer reports that *faultTolerantBehavior* is not possible with Lock 2, which pinpoints the problem with it: for a thread to continue entering a critical section, it *needs the other thread to also be interested* and take turns with it. As with the early models of Lock 1, all the requirements are verified, but because the model is not valid, verification does not mean that the requirements are actually satisfied.

The inconsistency of *noSharedResource* is different—why can't this scenario occur? The answer goes back to the discussion of stuttering steps in §4. Because our model includes only uses of the lock, if the lock is not used, then a trace must consist of nothing but stuttering steps. Yet the current definition of *doNothingEnabled* disallows stuttering when any other action is enabled. If we go to the other extreme and allow stuttering in any state, then it will not be possible to prove progress properties. The right answer for this problem is to allow stuttering when *callLock* is enabled, so that using the lock is optional, and to disallow stuttering when the locking mechanism has something to do. So the best domain model, allowing the true freedom of the domain, has the following definition, and then *noSharedResource* can be instantiated.[7]

```
pred doNothingEnabled {  all t: Thread |
   ! lockReturnEnabled[t] && ! unlockEnabled[t]  }
```

6 Requirements in Versions 1 and 2

(9) The requirements for Lock 1 can be found in §4.2. The requirements in Lock 2 look exactly the same as for Lock 1, with the qualification that *noDeadlocks* depends on *lockReturnEnabled*, which is different in the two models. The definitions of *lockReturnEnabled* are different in the two models because each is dependent on its lock implementation, which is unfortunately because the requirements then deprive designers and developers of full engineering freedom. Requirements should constrain only domain phenomena (including the domain/system interface, which is shared between them).

Here is new version of *noDeadlocks*, without the problem of dependence on the implementation, i.e., dependence on part of the system that is not also part of the domain. *noStarvation* is also rewritten, for convenience.

```
pred nonDeadlocking [t: Thread] {
   t.(Progs.progState) = Waiting =>
      eventually t.(Progs.progState) = InCritSec  }
assert noDeadlocks {
   anyTrace => some t: Thread | always nonDeadlocking [t]  }
check noDeadlocks                                          -- NOT VERIFIED

assert noStarvation {
   anyTrace => all t: Thread | always nonDeadlocking [t]  }
check noStarvation                                         -- NOT VERIFIED
```

Now *noDeadlocks* and *noStarvation* have the same form, the only difference being whether there is always progress for one thread or for both. This makes a very interesting change: formerly *noDeadlocks* was a safety property, but now

[7] You might think that basic modeling questions such as these (What is a trace? Can it be finite? Are there stuttering steps?) should have a single standard answer, but in fact the answers are problem-dependent.

it is a progress property! This is not so strange if you think about viewpoint. To an omniscient observer the property is safety—the observer can see into the implementation state, and whenever the implementation reaches a certain state, there is a deadlock. A user program (domain) is not omniscient, however, and all it observes is that it called *lock*, which is taking a very long time to return—a progress problem.

For this particular model the two forms of the *noDeadlocks* property seem to be equivalent, in that each implies the other. This is not always the case. Consider a different domain in which *callLock* forks the thread, so that a user program can continue working on other tasks while waiting for *lockReturn*. Further, imagine that the user program can time out if it has been waiting too long, and call a lock method to withdraw its request for the lock. In this case *noDeadlocks* as defined by a bad state (the safety property) would always be satisfied by Lock 1, but *noDeadlocks* as defined by progress in at least one thread would not be.

7 Version 3: The Peterson lock

In a very direct sense, Lock 3, known as the Peterson lock, is the sum of Locks 1 and 2. More specifically, it maintains all the state of both locks. The crucial precondition of *lockReturn*, which allows a client program to enter its critical section, is satisfied if either the precondition of *lockReturn* in Lock 1 is satisfied, *or* if the precondition of *lockReturn* in Lock 2 is satisfied. In this way, the Peterson lock combines the advantages of the previous two, and allows each one to make up for the disadvantages of the other. It passes all of the validation checks so far, and satisfies all the requirements!

7.1 A true specification

As with any case study, we have to think about how its ideas would hold up in a much larger, more complex real development project. In this context, a limitation of the models of Locks 1 and 2 is that they lack a proper specification, visible to the system and yet independent of the lock implementation (e.g., the difference between Locks 1 and 2). To show what a true specification might look like in this simple example, in Lock 3 the interface is extended with the system's own version of *progState*, called *lockState*. Because this variable is specification only, it need not be implemented.

```
one sig Lock {  var lockState: Thread -> one ProgState,  -- interface spec
                var flagRaised: set Thread,               -- system only
                var polite: lone Thread             }     -- system only

pred initialState { . . .
   Progs.progState = Thread -> Uninterested               -- domain only
   Lock.lockState = Thread -> Uninterested . . . }          -- interface
```

Now the system maintains *lockState* as a copy of *progState*. The *callLock* action is a typical example:

```
pred callLockEnabled [t: Thread] {               -- precondition on domain
   t.(Progs.progState) = Uninterested && t in Progs.runningProg  }
pred callLock [t: Thread] {
   callLockEnabled [t]
   Progs.progState' = Progs.progState ++ (t -> Waiting)   -- domain change
   Progs.runningProg' = Progs.runningProg - t          -- interface change
   Lock.lockState' = Lock.lockState ++ (t -> Waiting)
                                        -- system changes interface state
   Lock.flagRaised' = Lock.flagRaised + t                -- system change
   Lock.polite' = t                                  }   -- system change
```

The requirements are unchanged. However, there is now a specification that looks very similar to the requirements, except that it refers to interface phenomena only. Notice that the specification does not constrain the lock implementation in any way.

```
pred systemMutuallyExclusive {  # (Lock.lockState).InCritSec <= 1  }
assert systemMutualExclusion { anyTrace => always systemMutuallyExclusive}
check systemMutualExclusion                                    -- VERIFIED

pred systemNonDeadlocking [t: Thread] {
      t.(Lock.lockState) = Waiting
   => eventually t.(Lock.lockState) = InCritSec  }
assert systemNoDeadlocks {
   anyTrace => some t: Thread | always systemNonDeadlocking [t]  }
check systemNoDeadlocks                                        -- VERIFIED

assert systemNoStarvation {
   anyTrace => all t: Thread | always systemNonDeadlocking [t]  }
check systemNoStarvation                                       -- VERIFIED
```

In §2 we mentioned the proof obligations that are central to validation. The Lock 3 model already shows that the domain model and the specification are consistent, because the specification is verified, and the model has traces illustrating domain behaviors. The other obligation is that the specification is sufficient, meaning that if the implementation satisfies the specification and the implementation is installed in the domain, then the requirements *will* be satisfied.

(10) Because the Lock 3 model has a locking mechanism and satisfies both requirements and specification, it hides whether the specification alone is sufficient. To check this, we made another model version that stripped out the locking mechanism, leaving only the domain model, interface included. Not surprisingly, this model fails on most requirements and specification properties.[8] To check sufficiency, we then verified the following assertions:

```
assert sufficientMutexSpec {  anyTrace =>
  ((always systemMutuallyExclusive) => (always mutuallyExclusive))  }
```

[8] Because the no-lock lock returns immediately from all calls, it never deadlocks.

```
check sufficientMutexSpec                                    -- VERIFIED

assert sufficientNostarvSpec {  anyTrace =>
   (  (all t: Thread | always systemNonDeadlocking [t])
   => (all t: Thread | always nonDeadlocking [t])     )  }
check sufficientNostarvSpec                                  -- VERIFIED
```

The assertions are saying that if any trace satisfies a specification property, it also satisfies the corresponding requirement. The assertions would not be true for any model, but they are true for this one, because the domain model ensures that *progState* and *lockState* are always equal.

7.2 Atomicity

(11) With Lock 3 comes a new general-purpose validity concern, which is whether model actions can be executed atomically by the implementation. In an implementation, *flagRaised* and *polite* might be represented by three variables: a Boolean for each thread indicating whether its flag is raised, and a thread identifier for *polite*. The *callLock* action changes two of these variables. The *lockReturn* action (in its precondition) reads two of these variables (assuming it does not also read the flag Boolean for its own thread). And all three variables can be read or written concurrently by actions of the other thread. Given that multiprocessors usually provide atomicity only at the level of accesses to individual memory cells, it does not seem that these actions are guaranteed to be atomic in the implementation.

The only way to have a truly valid model in this situation is to split up actions into smaller actions that can be guaranteed atomic. Now the expanded interface comes in handy, because the set *ProgState* can be extended with a new member *Waiting2*. The extra member will help the new lock model coordinate its extra actions. Now the former *callLock* action becomes a sequence of two actions:

```
pred callLock1Enabled [t: Thread] {             -- precondition on domain
   t.(Progs.progState) = Uninterested && t in Progs.runningProg  }
pred callLock1 [t: Thread] {
   callLock1Enabled [t]
   Progs.progState' = Progs.progState ++ (t -> Waiting)   -- domain change
   Progs.runningProg' = Progs.runningProg - t          -- interface change
   Lock.lockState' = Lock.lockState ++ (t -> Waiting)  -- interface change
   Lock.flagRaised' = Lock.flagRaised + t                 -- system change
   polite' = polite                                  }   -- no sys change

pred callLock2Enabled [t: Thread] {             -- precondition on system
   t.(Lock.lockState) = Waiting && t ! in Progs.runningProg  }
pred callLock2 [t: Thread] {
   callLock2Enabled [t]
   progState' = progState                              -- no domain change
   runningProg' = runningProg                       -- no interface change
   Lock.lockState' = Lock.lockState ++ (t -> Waiting2)    --interface chng
   flagRaised' = flagRaised                              -- no system change
   Lock.polite' = t                                    }    -- system change
```

After *callLock1*, the *lockState* of the thread is *Waiting*. After *callLock2*, the *lockState* of the thread is *Waiting2*. In each action, only one of *flagRaised* and *polite* is modified.

The *lockReturn* action also becomes two actions, one with a precondition based on *flagRaised*, and one with a precondition based on *polite*. There is no intermediate state, however, because each action returns control to the user program, so only one is executed for each call. This is the equivalent of combining the two preconditions, shown below, with an *or* operator.

```
pred lockReturn1Enabled [t: Thread] {             -- precondition on system
      t.(Lock.lockState) = Waiting2 && t ! in Progs.runningProg
   && Lock.flagRaised = t                                     }

pred lockReturn2Enabled [t: Thread] {             -- precondition on system
      t.(Lock.lockState) = Waiting2 && t ! in Progs.runningProg
   && Lock.polite ! = t                                       }
```

With this change, the *stateInvariant* and *sequentialControl* assertions must also be rewritten, in obvious ways. Then all the previous assertions can still be verified, showing that the Peterson lock can handle the full concurrency of multiprocessor implementations. We should have expected this, knowing that the Peterson lock is being taught 43 years after its invention. But if this were a new locking algorithm, how else but with a valid model could we be sure that it works?

(12) In our final validation step, we turn our attention to the length of traces. By default, the Alloy Analyzer checks all traces up to 10 states and 10 actions. With previous model versions, in 6 actions both threads could go through a complete cycle, which makes 10 actions seem like a reasonable standard. In the new version of Lock 3, however, a thread cycles in 4 actions, and both can cycle in 8—too close to 10 for comfort. To be sure of more thorough coverage, the Lock 3 model now runs for 16 actions (steps), which may be excessive but is still efficient. Trace lengths are extended as follows.

```
check domainNoStarvation for 2 but 16 steps
```

8 Summary

Table 1 summarizes the validation steps in this paper, of which a majority are automated, and a majority found a problem of some kind. In reference to the three categories introduced in §4, steps 1, 2, 11, and 12 are general-purpose. Steps 4 and 9 ensure that the domain and system are properly separated, while steps 5, 6, 7, 8, and 10 check the validity of the domain model. Step 3 is also

	Validation Check	**Version**	**Manual/ Automated**	**Problem Found?**
1	every variable is always defined	1	Man	No
2	expected state invariant holds		Auto	No
3	control within a thread is sequential		Auto	Yes
4	domain and system are separate in actions		Man	No
5	in domain, both threads can get lock		Auto	No
6	in domain, there is contention		Auto	Yes
7	in domain, there is fault-tolerance	2	Auto	Yes
8	in domain, shared resource can be absent		Auto	Yes
9	requirements constrain the domain only		Man	Yes
10	specification is sufficient to satisfy requirements	3	Auto	No
11	actions are atomic		Man	Yes
12	trace length is adequate		Man	Yes

Table 1. A summary of validation steps in the case study. Early validation checks are also made on all later versions. A "problem found" can be a failure of validity, or a bug in one of the locks.

general-purpose, but in this model the problem it revealed was a problem of domain/system separation.

For users of Alloy and other modeling languages, visualization of model instances is an indispensable validation tool. The Alloy Analyzer is notable for the excellence of its visualization, and has inspired a great deal of research on improving it even further (e.g., Sterling [6, 26], Forge [21], et al. [1, 4, 18, 24]).

The validation techniques presented in this paper are complementary to visualization in every way. In short, the techniques in this paper produce *focused* and *interesting* model instances to be understood, and visualization makes it easier to understand them. The techniques in this paper provide relevant questions about the model, and visualization makes it easier to answer them.

9 The importance of domain models

In §2 we introduced the distinction between the *formal semantics* of a model and its *meaning*. Its meaning explains how the formal semantics apply to the real world. The meaning implies, for each explicit or implicit constraint in the model, that it should hold because the domain naturally causes it to hold, or because the system implementation will cause it to hold.

Despite its history, the idea of domain modeling still seems to be puzzling or controversial to many people. And we all know that software-engineering concepts are tricky to teach because they show their true value only in large, complex, long-lived software projects. With this in mind, we offer justifications that might appeal to students, followed by those that rely on the wisdom of experience.

For students, the main point is that the heart of the domain model is the four behavioral scenarios encoded in these predicates: *bothProgsRequestLock, thereIs-Contention, faultTolerantBehavior, noSharedResource*. Most directly, we know from Herlihy that there is something wrong with both Locks 1 and 2, and the exact bugs are diagnosed with these predicates alone (Steps 6 and 7). Of the four predicates, three led to the discovery of some problem, whether algorithmic bug or validation problem.

Another discussion-worthy point is the way that domain modeling converts issues of formal semantics (which can differ from language to language) to issues of meaning. There are two examples in this paper. First, there is the discussion about deadlocks in §4.4. Second, although not previously discussed, there is the technique of vacuity checking, which checks—for each implication—that the implication is not trivially true because the antecedent is unsatisfiable. In this case study, all requirements are implications with *anyTrace* as antecedent. The domain model ensures that *anyTrace* is not just satisfiable in some possibly-trivial semantic sense, but that it is satisfied by useful and common scenarios.

A colleague who teaches formal methods tells us that, "many students stop after writing and checking one or two simple predicates, believing that their model is correct." Very likely the one or two predicates were all they could think of! A final point to stress with students is that thinking about the domain gives you something real to think about—which fires the imagination and leads to much richer and more realistic models.

For more experienced audiences, we might point out that an emphasis on domain modeling is very helpful for seeing how a model might be made more general, and for encouraging generality. General models for important domains are valuable reusable artifacts [27].

More concretely, domain models are at the heart of well-known techniques such as model-based testing. A domain model can be used to enumerate sequences of operations that drive a system into a particular state or (even when state is not involved) generate tests that exercise the space of structurally-complex inputs, as in TestEra [19], Korat [20], and Whispec [25]. Domain models can even be used to aid in evaluating student testing [22]. Likewise, in property-based testing, as popularized by the QuickCheck [3] tool, the properties themselves are reliant on a (perhaps unstated) domain model. For example, properties might need to express what it means for a list of names to be sorted (by which linguistic convention?) or whether a year occurs before another (must we consider the Julian-Gregorian calendar transition?). Without the domain, we don't—and can't—know what it means for a system to be "correct."

Alloy 6 is the result of merging Alloy with Electrum [17], which adds full support for linear temporal logic to Alloy. As part of the literature on Electrum, in [2] there is a proposal for imposing more structure on actions. Do these proposed structures help support validation or force models to be more valid? In [2] there is automatic generation of frame conditions for actions, which is certainly helpful. No other aspects of their proposal would help with the validation steps in this particular case study. Nevertheless, this seems to be a promising direction for future research. Small additions to the syntax of actions might be very helpful for automating more validation checks.

From our perspective, validation is under-appreciated, and too much of what comes out of verification tools is, effectively, garbage. Some people believe that you cannot teach validation—that it is learned only in the school of hard knocks—but we do not agree. Hopefully this case study will help other teachers of formal methods think about how they would teach validation.

References

1. Bendersky, P., Galeotti, J.P., Garbervetsky, D.: The DynAlloy visualizer. In: Aguirre, N., Ribeiro, L. (eds.) Latin American Workshop on Formal Methods. vol. 139, pp. 59–64 (2013)
2. Brunel, J., Chemouil, D., Cunha, A., Hujsa, T., Macedo, N., Tawa, J.: Proposition of an action layer for Electrum. In: Proceedings of the 6th International ABZ Conference. Southampton, United Kingdom (2018)
3. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming (2000). https://doi.org/10.1145/357766.351266
4. Couto, R., Campos, J.C., Macedo, N., Cunha, A.: Improving the visualization of Alloy instances. In: Masci, P., Monahan, R., Prevosto, V. (eds.) Proceedings 4th Workshop on Formal Integrated Development Environment. vol. 284, pp. 37–52 (2018)
5. De Giacomo, G., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (2015)
6. Dyer, T., Baugh, J.: Sterling: A web-based visualizer for relational modeling languages. In: Rigorous State Based Methods (2021)
7. Hayes, I.J., Jones, C.B., Meinicke, L.M.: Specifying and reasoning about shared-variable concurrency. In: Theories of Programming and Formal Methods. Springer LNCS 14080 (2023)
8. Herlihy, M.: Multiprocessor synchronization: Mutual exclusion. `https://cs.brown.edu/courses/csci1760/lectures/lecture%202%20mutual%20exclusion%20dark.pptx`, accessed 28 October 2023.
9. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
10. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006, 2012)
11. Jackson, M.: Problem Frames. Addison-Wesley (2001)
12. Jackson, M., Zave, P.: Domain descriptions. In: Proceedings of the IEEE International Symposium on Requirements Engineering. pp. 56–64. IEEE Computer Society Press (1992)
13. Jackson, M., Zave, P.: Deriving specifications from requirements: An example. In: Proceedings of the 17th International Conference on Software Engineering. pp. 15–24. ACM Press (April 1995)
14. Jones, C.B.: Tentative steps toward a development method for interfering programs. Transactions on Programming Languages and Systems **5**(4), 596–619 (1983)
15. Jones, C.B.: Accommodating interference in the formal design of concurrent object-based programs. Formal methods in system design **8**(2), 105–122 (1996)

16. Jones, C.B.: From problem frames to HJJ and its known unknowns. In: Nuseibeh, B., Zave, P. (eds.) Software Requirements and Design: The Work of Michael Jackson, pp. 357–368. Good Friends Publishing (2010)
17. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. ACM (2016)
18. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.C.: Experiences on teaching Alloy with an automated assessment platform. Sci. Comput. Program. **211**, 102690 (2021)
19. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs (2001). https://doi.org/10.1109/ASE.2001.989787
20. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs (2007). https://doi.org/10.1109/ICSE.2007.48
21. Nelson, T., Greenman, B., Prasad, S., Dyer, T., Bove, E., Chen, Q., Cutting, C., Vecchio, T.D., LeVine, S., Rudner, J., Ryjikov, B., Varga, A., Wagner, A., West, L., Krishnamurthi, S.: Forge: A tool and language for teaching formal methods. In: Object-Oriented Programming Systems, Languages, and Applications (2024), (accepted and in preparation, to appear)
22. Nelson, T., Rivera, E., Soucie, S., Del Vecchio, T., Wrenn, J., Krishnamurthi, S.: Automated, targeted testing of property-based testing predicates. In: The Art, Science, and Engineering of Programming (2022). https://doi.org/10.22152/programming-journal.org/2022/6/10
23. Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters **12**(3), 115–116 (1981)
24. Rayside, D., Chang, F.S., Dennis, G., Seater, R., Jackson, D.: Automatic visualization of relational logic models. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **7** (2007). https://doi.org/10.14279/tuj.eceasst.7.94
25. Shao, D., Khurshid, S., Perry, D.E.: Whispec: White-box testing of libraries using declarative specifications (2007). https://doi.org/10.1145/1512762.1512764
26. Siegel, A., Santomauro, M., Dyer, T., Nelson, T., Krishnamurthi, S.: Prototyping formal methods tools: A protocol analysis case study. In: Protocols, Strands, and Logic. pp. 394–413 (2021). https://doi.org/10.1007/978-3-030-91631-2_22
27. Zave, P.: Theories of everything. In: Proceedings of the 38th International Conference on Software Engineering. IEEE (2016)
28. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology **6**(1), 1–30 (January 1997)
29. Zave, P., Nelson, T.: Three versions of a two-party lock: A case study in formal modeling. `https://pamelazave.com/peterson.zip` (2024)