# 3. Motivation

complete

## 3.1. A Toy Example

We envision the workflow of someone using LFORGE to be as follows:

1. The user specifies a model in interest, or they already have a completed model in Forge that they would like to formalize. Within Forge, the user writes relevant predicates and uses Forge's SAT backend to quickly check if they are satisfiable (and what satisfying instances look like), or if they are not satisfiable (that is, the negation is a theorem). Users might isolate specific predicates that they would like to prove in detail.

2. In a new Lean file, the user imports the LFORGE module and pastes their Forge specification in, verbatim. If they started the process working in the LFORGE subset of Forge, no changes need to be made. If they are working with an existing Forge file, our tool will prompt the user to make any modifications necessary to keep it compliant with the subset of syntax, including potential type annotations (see section 5.4). After this, all sigs, fields, and predicates are available in Lean.

3. Optionally, a user might want to make *additional* claims or write predicates using Lean's syntax. They have the option to do so here (see Mixed Execution, section 6.1).

4. Finally, the user will identify important predicates within their specification that they want to prove generally. They do so using the interactive theorem prover in Lean.

We start by providing a small example that might represent a Forge specification and the desired generated code in Lean as some motivation. Bertrand Russell, in illustrating Russel's paradox on sets, poses the following paradox: "Let the barber be 'one who shaves all those, and those only, who do not shave themselves.' The question is, does the barber shave himself?" [13, p. 101].

We might want to construct a formal model for this village for which the barber shaves all those who do not shave themselves, and use Forge to prototype and quickly check properties regarding this model:

```
sig Person {
  shaver: one Person
}

pred shavesThemselves[p: Person] {
  p = p.shaver
}

pred existsBarber {
  some barber : Person | all p : Person | {
    not shavesThemselves[p] <=> p.shaver = barber
  }
}
```

Attempting to run this model for `existsBarber` will yield an `Unsatisfiable`. After all, if the barber doesn't shave themselves, yet they shave all those who don't, they they ought to shave themselves. To prove this statement (and not merely rely on the fact that Forge was not able to find a satisfiable instance within its bounds), we need to transition to a theorem prover.

LFORGE aids in porting the entire specification into Lean, producing a set of equivalent Lean definitions:

```
opaque Person : Type
opaque shaver : Person → Person

def shavesThemselves : Person → Prop :=
  fun p ↦ shaver p p

def existsBarber : Prop :=
  ∃ (barber : Person), ∀ (p : Person), ¬shavesThemselves p ↔ shaver p barber
```

At this point, we might want to continue to provide a mathematical proof, as we observed in Forge, that there cannot possibly be a barber in this town. That is,

```
theorem no_barber : ¬ existsBarber := by ...
```

The conclusion of this example, including the Lean proof of our property, is provided in section 6.2. Note that this was not possible working solely in Forge, which only checks for the existence of examples or counterexamples within the bounds that it knows.

While this was a simple example, it serves as a demonstration of what is possible under this dual framework. One might provide a specification of a protocol and prove that no vulnerabilities exist (or that it has all the desired properties). Lean would serve invaluable in proving the correctness of properties about any real-life system described and prototyped in the Forge specification language. All this stays true to the goal of providing better formal methods tools that are more universally applicable and practical.

Furthermore, the pedagogical implications of such a program is also worth noting. Forge, designed to be a pedagogical language, seeks to teach formal methods gradually and introduce students to formal methods tools enabling them to work better in the real world and industry [11]. Students who are interested in formal methods often take Logic for Systems in the Computer Science department at Brown, the course which Forge was developed for and taught in. Students continue on to take Formal Proof and Verification, a course that teaches the use of proof assistants via Lean. Both courses have open-ended research-style final projects that encourages students to explore and formalize topics that they are interested in. We believe LFORGE also provides an invaluable opportunity for students who have background with both formal methods tools (or who might merely want to explore beyond) to bridge their knowledge between automated reasoning and formalization via proof. While Forge instructs us that a specification or predicate is true, Lean reveals *why* the specification is true. Oftentimes, this makes it difficult to debug incorrect specifications without additional tools like visualizations [11]. By attempting to construct proofs of model properties, stu-

dents are compelled to think about their modeling choices and justify each statement, seamlessly translating between the statements they are making and proofs of their correctness [1].

# 6. Results and Examples

## 6.1. Forge as a Lean DSL

One of the crucial benefits of working with Lean 4 as a metaprogramming language and a target for our translation is the rich support for DSL implementation and integration. Lean and its accompanying Language Server Protocol (LSP)[18] are designed to have highly flexible and extensible user interfaces that expose useful APIs for implementers of DSLs and custom UI to utilize [8, 9]. Furthermore, Lean's extensible syntax and macro system are simple yet remarkably powerful [14, 12].

It is as such that we justify framing our implementation of Forge in Lean as an honest-to-goodness domain-specific language (DSL). We do not treat user experience of our tool as an afterthought, nor do we skimp over ensuring that LForge has a set of developer aids just as capable as those found in Forge or Lean themselves. As mentioned in section 4.1, the fact that we can interact with Lean's implementation means that many of Lean's 'IDE-like' features are exposed to us and available for us to use in the Forge DSL without much overhead.

The following are some (non-exhaustive) examples of the user experience and interface of Forge within Lean.

### Syntax Highlighting

Superficially, by defining our syntax as Lean objects and isolating our keywords (we piggyback off Lean's lexer), we get syntax highlighting of Forge code 'for free', on par with Forge's native solutions. In fig. 2, the syntax of a Forge specification is color-coded.

### Types on Hover

Lean exposes an `addTermInfo'` method that allows us to attach declared names to pieces of syntax (nodes in Lean's concrete syntax tree), including custom syntax like ours for Forge. As such, we can annotate relevant pieces of syntax within our Forge specification to reflect names and types that are within scope. When the user hovers their mouse over a piece of syntax corresponding to a Forge expression, a hovering tooltip will display with the type of the expression. In fig. 2, the tooltip shows the type of a Forge predicate defined earlier in the file.

### Documentation

As a pedagogical language, Forge has a focus on usability, learnability, and helpful feedback [11], especially when its parent language Alloy is far more permissive with errors. We follow in the same vein in reporting errors and missing features, and in addition, we include documentation on Forge's syntax within Forge's on-hover features.

[18] This is the language server that processes Lean code and communicates with the code editor or integrated development environment. In this case, we use VS Code.

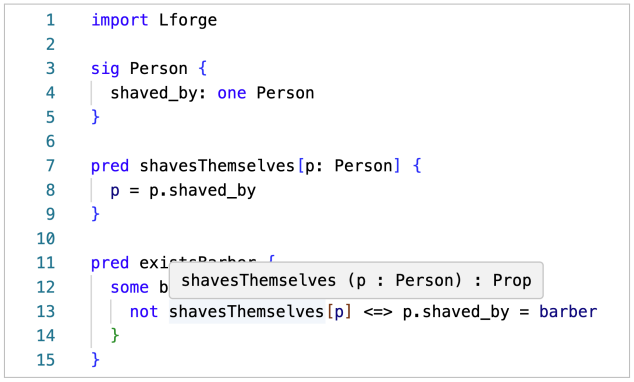


Figure 2. Tooltips containing type information are available on hover. Forge syntax is automatically highlighted without any extra work.

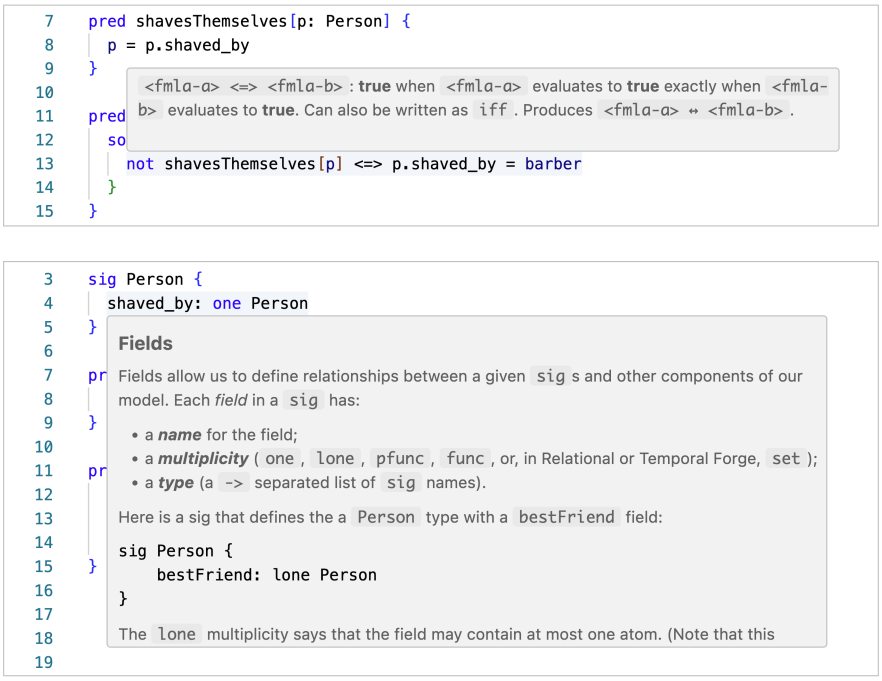


Figure 3. We can define our syntax definitions to print with custom documentation text for users new to using Forge syntax.

Forge documentation is included via docstrings that are placed inline with our syntax objects (see section 4.2), which is automatically included by Lean's LSP to display on the front end. Figure 3 showcases docstrings of varying verbosity for operators as well as declaration syntax.

Additionally, we need to be clear and verbose about language features that are not supported in LFORGE. Since LFORGE includes a subset of relational Forge determined by compatibility with Lean's semantics, we prompt users attempting to use unsupported language features with clarification and a request to redefine their statements. Figure 4 showcases an example of a prompt that `lone` sig quantifier is unsupported and potentially ambiguous.

### Error Checking

Compared to Forge or Racket, Lean (and consequently, LFORGE) provides a markedly better experience with error messages and prompting users when there are errors present in their source program. Since Lean runs in the background as an LSP, users immediately get immediate feedback on whether their source code parses and 'compiles'.[19]

Lean's error locality system allows its error monad to refer to any piece of syntax object to potentially throw an error. This allows us to prompt errors as soon and as granular as possible. Figure 5 illustrates error reporting at the level of specific identifiers.

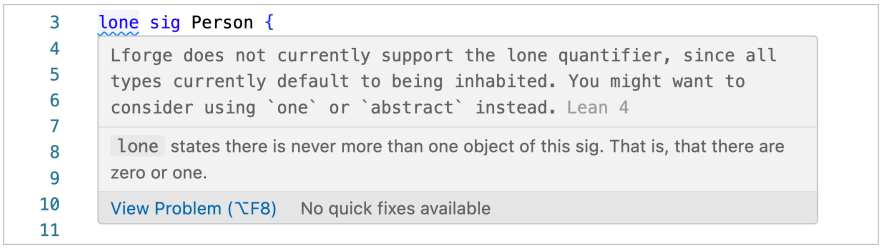


Figure 4. We can define custom error messages with our implementation to prompt users to change their specifications if a piece of syntax is ambiguous or not supported.

### Types

Lean's dependent type system is both a blessing and a curse when it comes to the task of translating a language with a foreign type system into Lean. While the type system is a highly optimized algorithm that attempts to resolve type coercions, type classes, and reductions [2], it at often delicate and temperamental, especially when we are working at such a low level of emitting Lean `expr`s, which happens *after* type unification (see section 4.3). We discuss some of the downsides of such a strict type system in section 5.4, and introduced LFORGE features that circumvent Lean's restrictions and play into Lean's type system.

Here, we discuss some of the merits of implementing a DSL designed around Lean's extensive type system. One of the side effects of translating Forge into Lean is that we inherit Lean's powerful

[19] Forge translations in Lean are not executable, so they provide their value in being interactive with the proof system.

type unification and checking system. This allows us, at specification-time, to check for type errors within the specification. Alloy, on the other hand, is purposefully untyped [4] and only reports type errors at runtime when the successful evaluation of expressions results in the empty expression [3]. This proves difficult to debug and unwieldy for users to understand, as [11] observes. For students who are newly learning the idea of relations, sets, and units, lacking instantaneous feedback on the validity of types and expressions is immensely useful.

Since expressions in our Forge DSL need to translate to typed terms in Lean, we necessarily have to specify types (or use Lean metavariables awaiting unification in place of types) to the Lean expressions emitted. Fortunately, much of this process is abstracted away by the type inference system in place in Lean. For example, to make an application, say, a set union, we don't need to specify the type of set that are being used in the operands and `mkAppM` will complete that for us. However, if we specified two sets of different types, Lean would raise an error. This results in a type-checking and inference system that is as powerful as Lean's with minimal overhead.

Since the Lean LSP provides type linting, our Forge DSL inherits this feature as well. In fig. 5, we pass the `Board` and `Player` arguments to `winRow` in the wrong order, which causes a type error. Lean can identify that the first input to `winRow`, `p`, has the wrong type and displays an appropriate error message.

```
58  pred winner[b: Board, p: Player] {
59    p != None
60    winRow[p, b]
61      application type mismatch
62        winRow p
63      argument
64        p
65      has type
66        Player : Type
67      but is expected to have type
68        Board : Type Lean 4
69
70      winRow (b : Board) (p : Player) : Prop
71      View Problem (⌥F8)    Quick Fix... (⌘.)
72  }
```

Figure 5. A Lean error message indicating a type mismatch in our Forge expression.

### Mixed Execution

Our embedding of Forge within Lean, especially our choice to map Forge structures (sigs, fields, predicates) to corresponding Lean concepts (types, relations, functions) as faithfully as possible means that a Lean file with Forge specifications supports mixed execution, an embedding model explored and supported by similar tools that merge the Alloy specification into other imperative programming languages [6, 7, 5].

All Forge specifications, once inserted into a Lean file, will generate appropriate definitions directly into the Lean environment. Using Lean syntax, we can further interact with these definitions from Forge, perhaps writing predicates or definitions that build on these. This interaction goes the opposite way as well: where Forge expects an expression, predicate, or function, declarations from Lean can be used seamlessly. This provides a frictionless user experience and allows the user to add additional constraints and rules, written in Lean, to a preexisting Forge model. Furthermore, this alleviates the tension incurred of creating the perfect translation: we are aware of the fact that only a subset of Forge is implemented due to the technical restrictions of both platforms. However, with appropriate error reporting (see above Error Checking), users can be prompted to extend their Forge specifications with additional Lean rules. Mixed execution of Forge and Lean means that Forge specifications are now extensible using a much broader functional programming language.

The following is an example of mixed execution of Forge and Lean using Lforge. The full specification of this example, a model for mutual exclusion of processes, is discussed in section 6.3.

```
1  import Lforge
2
3  abstract sig Location {}
4  one sig Uninterested, Waiting, InCS extends Location {}
5
6  sig Process {}
7
8  sig State {
9    loc: func Process -> Location,
10   flags: set Process
11 }
12
13 def flags_good (s : State) :=
14   ∀ (p : Process), loc s p = InCS ∨ loc s p = Waiting → flags s p
15
16 pred good[s: State] {
17   flags_good[s]
18   lone {p: Process | s.loc[p] = InCS}
19 }
```

While the majority of this specification is in Forge, we are working in Lean using Lforge (line 1). We define relevant sigs and fields as a Forge specification. On lines 13-14, we use the types defined in Forge to write a predicate in Lean that states that all processes waiting or in a critical state have a flag raised. In line 17, we've switched back to specifying in Forge but can continue to utilize the `flags_good` predicates we wrote above in Lean. There are no walls or abstractions between the two languages, and Forge is indeed a first-class citizen in the Lean environment.

Since this interoperability works across imports and modules, we envision projects where sections of specifications can be written in Forge and other relevant sections in Lean, allowing users to interoperate between the two. This could also introduce possibilities for users accustomed to the two distinct languages or formalization techniques to collaborate on a shared specification.

## 6.2. A Toy Example, *Continued*

We revisit our toy example from section 3.1 (the barber who "shaves all those, and only those, who do not shave themselves"). Recall that we had introduced a Forge specification for this problem, as well as an equivalent Lean specification of the paradox. We concluded earlier that while it was insightful for Forge to produce a result that this specification was `Unsatisfiable`, we might still desire a general proof of this fact outside of Forge's finite and restricted search bounds.

Here's an example of what the last part of the modeling workflow—writing and completing the proof—would look like in Lean's interactive tactic mode:

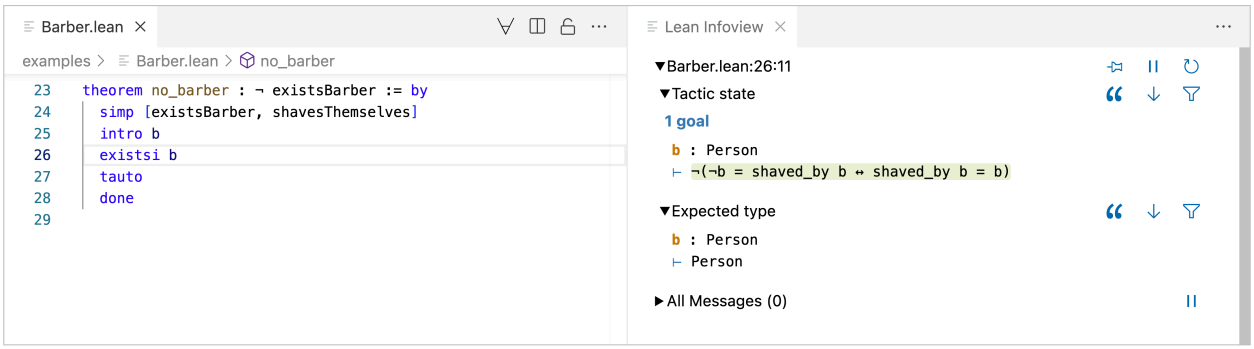


Figure 6. The proof of the nonexistence of a barber in the barber paradox, in Lean. The interactive proof state is on the right with the proof source on the left.

On line 25, we use the `simp` tactic (or we can also use `simp only` or `rw`) to rewrite our Forge-defined predicates `existsBarber` and `shavesThemselves`. Due to the simplicity of the example, the goal can be closed using the `tauto` (tautology) tactic which repeatedly breaks down assumptions and splits goals with with logical connectives until it can close the goal. Full tactic states at each step of this proof are provided in appendix B.

While simple, this example demonstrates the expressiveness of Forge programs embedded in Lean and the relative ease with which some proofs of translated properties can be executed. The following section, section 6.3, presents a more elaborative example of LFORGE.

## 6.3. A Mutual-Exclusion Protocol

Here, we present a more comprehensive example that showcases more of the functionalities of LFORGE and hopefully motivates real-world use cases of our tool. We model a basic mutual exclusion (mutex) protocol based on one of the examples presented in CSCI 1710 Logic for Systems [10]. In the course, the example is posed with 2 competing processes over a mutex. Empowered with Forge within LFORGE, we expand the model to include any number of processes.

The example model contains `State` sigs that encapsulate the state of the entire system. Processes can have several states, `Uninterested`, `Waiting` (interested but not in critical state), `InCS` (in critical state). Each state contains a set of processes that have a flag raised demonstrating they are

potentially interested in mutex. State transitions are modeled using predicates, which are of the form

```
pred transition[pre: State, p: Process, post: State] { ... }
```

Processes have 4 transitions:

1. `raise`: they can transitions from `Uninterested` to `Waiting` by raising their flag;
2. `enter`: they can transition from `Waiting` to `InCS` provided they are the only flag raised;
3. `lower`: if there is more than one flag raised, they can transition from `Waiting` back to `Uninterested` by lowering the flag;
4. `leave`: they can transition from `InCS` to `Uninterested` when processes are done.

We write a `good` predicate that states an invariant of our model that we wish to be true. In our case, the `good` predicate stipulates no two processes can be in the critical state (have acquired the lock) on the mutex at the same time, and any process that is waiting or in a critical state has a 'flag' raised. We define an `init` state predicate that states all processes are in a state of `Uninterested` and no flags are raised. A predicate titled `properties` (users sometimes use the convention `traces`) encapsulates all properties of our system: that the initial state is good and for all pairs $\langle$`pre`, `post`$\rangle$ for which there is a transition between, `pre` is good implies `post` is good. This is to say, `good` is an invariant property given our transition rules.

To test our model and that it indeed has such desired properties, we can first run Forge on the test `properties is theorem` to check that Forge cannot solve for counterexamples within its specified bounds. Then, as a next step, we can declare a theorem that states `properties` in Lean and prove our theorem.

To prove `properties`, we can split up our property into the base case (proving that the `init` state is `good`) and that each of the 4 transitions preserves `properties`. We prove each transition separately in lemmas.

An example of the tactic state during one of the proofs of one such lemma is showcased in fig. 7. We note that the tactic state in fig. 7 represents a typical Lean proof state and what a user might encounter while using LFORGE and Lean to prove specification properties, as opposed to our toy proof in fig. 6.

For a rough reference of length, our specification is roughly 70 lines long and our proof is roughly 250 lines long, which is standard for each. Neither the specification nor proof were more involved than had they been solely in Forge or Lean respectively. The full source of this example is referenced in appendix C.

## 6.4. Further Examples

We provide three further examples (without proofs), to illustrate LFORGE's translation capabilities. Said examples translate fully using LFORGE without any type errors.

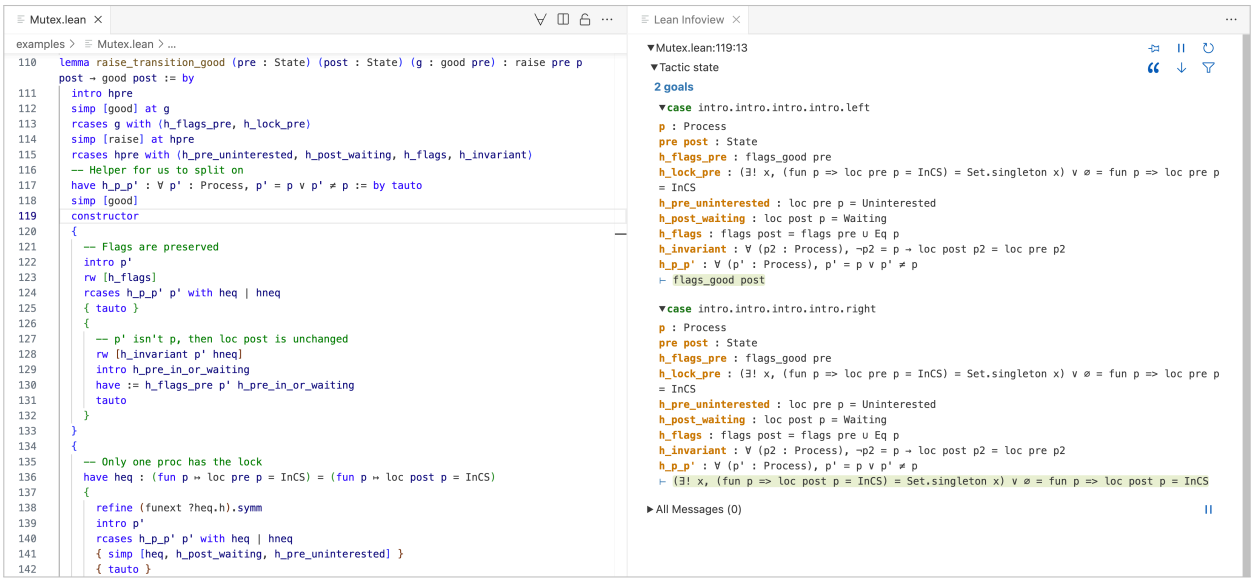


Figure 7. The Lean tactic state at one line of our proof that the `raise` transition is sound.

From the Logic for Systems course [10], we adapt the first Forge assignment on family trees, as well as the in-class example with a Tic-Tac-Toe board. Our examples are minimally modified specifications from the course and demonstrate LFORGE's capabilities of working with existing Forge specifications with little-to-no modification.

Additionally, inspired by an ongoing research project that uses Forge to model distributed systems algorithms [15], we also model the two-phase atomic commitment protocol. Our example is an entirely new Forge specification that takes inspiration from this existing research. This serves as a more complex example of a system that might be valuable to be modeled in Forge and proven in Lean.

The sources of said examples are referenced and specified further in appendix D.

# Bibliography


[1] Jeremy Avigad. "Learning logic and proof with an interactive theorem prover". In: *Proof technology in mathematics research and teaching* (2019), pp. 277–290.

[2] Leonardo De Moura et al. "The Lean theorem prover (system description)". In: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer International Publishing. 2015, pp. 378–388.

[3] Jonathan Edwards, Daniel Jackson, and Emina Torlak. "A type system for object models". In: *ACM SIGSOFT Software Engineering Notes* 29.6 (2004), pp. 189–199.

[4] Daniel Jackson. "Alloy: a language and tool for exploring software designs". In: *Communications of the ACM* 62.9 (2019), pp. 66–76.

[5] Aleksandar Milicevic et al. "Advancing declarative programming". PhD thesis. Massachusetts Institute of Technology, 2015.

[6] Aleksandar Milicevic et al. "Executable specifications for Java programs". MA thesis. Massachusetts Institute of Technology, 2010.

[7] Aleksandar Milicevic, Ido Efrati, and Daniel Jackson. "$\alpha$ Rby—an embedding of Alloy in Ruby". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings 4*. Springer. 2014, pp. 56–71.

[8] Leonardo de Moura and Sebastian Ullrich. "The lean 4 theorem prover and programming language". In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer. 2021, pp. 625–635.

[9] Wojciech Nawrocki, Edward W Ayers, and Gabriel Ebner. "An extensible user interface for Lean 4". In: *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023.

[10] Tim Nelson. *Logic for Systems*. 2024. URL: https://csci1710.github.io/book/.

[11] Tim Nelson et al. "Forge: A Tool and Language for Teaching Formal Methods". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (2024), pp. 1–31.

[12] Arthur Paulino et al. *Metaprogramming in Lean 4*. 2024. URL: https://leanprover-community.github.io/lean4-metaprogramming-book/.

[13] Bertrand Russell. *The philosophy of logical atomism*. Routledge, 2009.

[14] Sebastian Ullrich and Leonardo De Moura. "Beyond notations: Hygienic macro expansion for theorem proving languages". In: *Logical Methods in Computer Science* 18 (2022).

[15] Jinliang Wang and Tim Nelson. *Forge for Distributed Systems*. https://github.com/jinlang226/Forge-for-Distributed-System. 2024.

# Appendices

## A. Data Availability

The source code for LFORGE, including all mentioned examples and proofs, is available at this repository: https://github.com/jchen/lforge. The source code at the time of this thesis being submitted is tagged `thesis`. LFORGE is available as a package and can be included as a dependency using Lake.

Check tag.

## B. Barber Paradox Proof

The proof of the barber paradox annotated with the Lean tactic state after each tactic/step is provided below. This is also provided at this path in the code repository: `examples/Barber.lean`.

check filepath

```
theorem no_barber : ¬ existsBarber := by
  /-
  ⊢ ¬existsBarber
  -/
  simp [existsBarber, shavesThemselves]
  /-
  ⊢ ∀ (x : Person), ∃ x_1, ¬(¬x_1 = shaved_by x_1 ↔ shaved_by x_1 = x)
  -/
  intro b
  /-
  b : Person
  ⊢ ∃ x, ¬(¬x = shaved_by x ↔ shaved_by x = b)
  -/
  existsi b
  /-
  b : Person
  ⊢ ¬(¬b = shaved_by b ↔ shaved_by b = b)
  -/
  tauto
  done
```

## C. Mutual Exclusion Protocol Specification & Proofs

The Forge specification and Lean proofs for the mutual exclusion protocol described in section 6.3 are provided at this path in the code repository: `examples/Mutex.lean`. The specification spans lines 14–85 and proofs span lines 89–336.

The Forge specification for this example is taken from [10] with slight modifications for more than two processes. We contribute the entirety of the Lean proof of correctness of this protocol.

## D. Additional Examples

Additional examples are also provided in the `examples` directory.

Two examples, Tic-Tac-Toe and 'Grandpa', are based on course content from Logic for Systems [10] with minimal modifications. They serve solely to illustrate the capabilities of LFORGE on translating existing programs, and we do not claim to make additional contributions to these

models. Tic-Tac-Toe is at the following path: `examples/Board.lean`, and the 'Grandpa' is at the following path: `examples/Grandpa.lean`.

The specification regarding the two-phase atomic commitment protocol is inspired by [15] but does not use any code directly from the repository. This serves as an example of a more complex distributed system protocol that is translatable.