Forge: A Tool and Language for Teaching Formal Methods

TIM NELSON, Brown University, USA
BEN GREENMAN*, University of Utah, USA
SIDDHARTHA PRASAD, Brown University, USA
TRISTAN DYER*, Stashpad, USA
ETHAN BOVE, Brown University, USA
QIANFAN CHEN, Brown University, USA
CHARLES CUTTING, Brown University, USA
THOMAS DEL VECCHIO, Brown University, USA
SIDNEY LEVINE, Brown University, USA
JULIANNE RUDNER, Brown University, USA
BEN RYJIKOV, Brown University, USA
ALEXANDER VARGA, Brown University, USA
ANDREW WAGNER*, Northeastern University, USA
LUKE WEST, Brown University, USA
SHRIRAM KRISHNAMURTHI, Brown University, USA

This paper presents the design of *Forge*, a tool for teaching formal methods gradually. Forge is based on the widely-used Alloy language and analysis tool, but contains numerous improvements based on more than a decade of experience teaching Alloy to students. Although our focus has been on the classroom, many of the ideas in Forge likely also apply to training in industry.

Forge offers a *progression of languages* that improve the learning experience by only gradually increasing in expressive power. Forge supports *custom visualization* of its outputs, enabling the use of widely-understood domain-specific representations. Finally, Forge provides a variety of *testing features* to ease the transition from programming to formal modeling. We present the motivation for and design of these aspects of Forge, and then provide a substantial evaluation based on multiple years of classroom use.

*Work done while at Brown University.

Authors' addresses: Tim Nelson, Brown University, Providence, Rhode Island, USA, timothy_nelson@brown.edu; Ben Greenman, University of Utah, Salt Lake City, Utah, USA, benjaminlgreenman@gmail.com; Siddhartha Prasad, Brown University, Providence, Rhode Island, USA, siddhartha_prasad@brown.edu; Tristan Dyer, Stashpad, Raleigh, North Carolina, USA, a.tristan.dyer@gmail.com; Ethan Bove, Brown University, Providence, Rhode Island, USA, ethan_bove@brown.edu; Qianfan Chen, Brown University, Providence, Rhode Island, USA, qianfan_chen@alumni.brown.edu; Charles Cutting, Brown University, Providence, Rhode Island, USA, charles_cutting@brown.edu; Thomas Del Vecchio, Brown University, Providence, Rhode Island, USA, thomas_del_vecchio@alumni.brown.edu; Sidney LeVine, Brown University, Providence, Rhode Island, USA, sidney_levine@brown.edu; Julianne Rudner, Brown University, Providence, Rhode Island, USA, juliannerudner@gmail.com; Ben Ryjikov, Brown University, Providence, Rhode Island, USA, benjamin_ryjikov@alumni.brown.edu; Alexander Varga, Brown University, Providence, Rhode Island, USA, alexander_varga@alumni.brown.edu; Andrew Wagner, Northeastern University, Boston, Massachusetts, USA, ahwagner@ccs.neu.edu; Luke West, Brown University, Providence, Rhode Island, USA, luke_west@alumni.brown.edu; Shriram Krishnamurthi, Brown University, Providence, Rhode Island, USA, shriram@brown.edu.

XXXX-XXXX/2024/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS Concepts: • **Software and its engineering → Semantics**; Constraints; Functional languages.

Additional Key Words and Phrases: lightweight formal-methods, formal-methods education, language levels

ACM Reference Format:
Tim Nelson, Ben Greenman, Siddhartha Prasad, Tristan Dyer, Ethan Bove, Qianfan Chen, Charles Cutting, Thomas Del Vecchio, Sidney LeVine, Julianne Rudner, Ben Ryjikov, Alexander Varga, Andrew Wagner, Luke West, and Shriram Krishnamurthi. 2024. Forge: A Tool and Language for Teaching Formal Methods. 1, 1 (March 2024), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The growing applicability of formal methods (FM) demands that we create curricula and tools that serve broad audiences, not only those who were already predisposed to learning the topic. Yet, "FM" comprises many techniques and tools: proof assistants, dependently-typed languages, property-based testing, model checking, SMT solvers, and more (e.g., Brady [2013]; Claessen and Hughes [2000]; de Moura and Bjørner [2008]; Holzmann [2003]; Leino [2010]; Nipkow et al. [2002])—not all of which are good fits for students without a formal bent. In this paper we adopt a specific class of FM that we believe is well-suited to broad computer-science education, and then describe student-friendly tools that implement it.

Our inspiration is Alloy [Jackson 2012]. Alloy embodies the *lightweight* FM philosophy exposited by Jackson and Wing [1996], and relative to many other FM tools, it offers numerous advantages for introducing FM:

- It provides a rich, relational, language. Its relational nature makes it suitable for expressing object relationships [Jackson 2002; Spivey 1992], which reflect a wide variety of systems encountered in daily life. Thus, Alloy allows students to explore their intrinsic interests via formal modeling—both within and outside of computing.
- It has a syntax reminiscent of programming languages like Java, meaning that students can import some amount of object-oriented experience when starting with Alloy—learning the tool's more powerful features as they progress.
- It offers push-button automation. Therefore, it feels familiar to students coming from a background of IDEs. Just as students might click "run" in an IDE to execute their program, they can click "run" in Alloy to generate satisfying instances of their model.
- It supports multiple modes of use. In the presence of properties, it behaves as a bounded model checker [Biere et al. 1999]. However, in the absence of properties, students can still "run" their model to explore concrete instances, which may trigger a deeper understanding (including of undesired behaviors). This relates to the surprise-explain-reward model [Wilson et al. 2003] of cognition used in other tools for aiding correctness (e.g., [Jernigan et al. 2017]). This modality stands in contrast to many other verification tools, such as many model checkers, which are inert in the absence of properties.
- It gives an automatic *visualization of instances*. When students view an instance, Alloy's visualizer presents it as a labeled directed graph, which is often a vast improvement over raw textual formats. If students prefer a tabular view, Alloy offers that as well.

Consequently, Alloy is popular and widely-used in both education and practice [alloytools.org 2023a,b,c; Jackson 2019]. However, based on several years of teaching with Alloy, we have found critical weaknesses in it for the purposes of our pedagogy:

- Alloy's sophisticated language and clever syntax contain pitfalls for students.
- Alloy's visualizer can produce unhelpful, confusing, or outright misleading output.
- Alloy's support for testing is limited, which hinders some educational opportunities for students who have programming experience.

In response, we have built a system named Forge,[1] designed to be a pedagogic variant of Alloy. Like Alloy, Forge is a language and a tool, and it draws heavily on Alloy's syntax, semantics, and infrastructure. Forge has three critical innovations:

Language Levels: Forge supports a *progression* of sub-languages that build on each other and elide unnecessary complexity (section 3). These sub-languages are not merely optional fragments of one overall language, but provide their own *epistemic closure* [Findler et al. 2002] by, e.g., ensuring that error messages are given in terms of the current sub-language.

Custom Visualization: Forge broadens Alloy's default visualizations, building atop a Web-based visualizer [Dyer 2020] to include *custom visualizations* (section 4). We have built several visualizations to further epistemic closure for homework problems, and students have built their own visualizations for domains of their choosing (examples in section 7).

Testing: Forge adds linguistic support for testing throughout model development (section 5).

In addition, Forge is built atop the Racket platform [Felleisen et al. 2018], enabling additional (as yet pedagogically untested) benefits such as domain-specific modeling languages (section 9).

While our focus in this paper is on teaching FM at the tertiary education level, we conjecture that all these elements would benefit industrial users as well, especially trainees. Indeed, one of our main examples for custom visualization is from a more expert setting (section 4.3).

Outline. After a short background on our FM pedagogy (section 2), this paper presents the design of language levels (section 3), custom visualization (section 4), and testing tools (section 5) in Forge. This tour of Forge concludes with a brief discussion of its architecture and implementation (section 6). The second half of the paper contains a substantial evaluation of Forge, both quantitative and qualitative, based on multiple years of using Forge in the classroom (section 7). The paper concludes with related work (section 8) and high-level reflections (section 9).

Forge is Open Source. The project is available from:

https://forge-fm.org/

This paper also comes with an artifact that includes the models and visualizations we reference herein, along with copies of Forge and its documentation [Nelson et al. 2024].

2 BACKGROUND: TEACHING FM GRADUALLY

There are many potential ways to teach FM. We espouse a *gradual* approach, which starts by meeting students where they are conceptually. Most notably, this includes the many students who are deeply uninterested in formalism, but are becoming experts in other fields such as distributed systems or security. Our goal is not to convert these students into FM fans, but rather to equip them with basic training before they go on to build tomorrow's digital infrastructure. We want all students to learn how to apply formal approaches to their own areas of interest and expertise.

Gradual FM comprises four core ideas:

- *Teach with tools.* Software tools lower the barrier to entry (since the tool can actively guide the student, unlike paper and pencil) and provide a familiar "programming"-esque environment in which to work. If the tool also provides rapid feedback, so much the better.
- *Embrace lightweight* FM. Drawing inspiration from Jackson and Wing [1996], we use languages with limited expressive power to enable quick, low-cost formal analysis. As a result, we focus on modeling partial systems, precisely stating properties, and revealing bugs, rather than on proving that no bugs exist.

[1]The name is of course intended as a tribute to Alloy. It should not be confused with the Alloy-based but unrelated, defunct Forge tool [Dennis et al. 2006].

- *Be systems-oriented.* While some students get excited modeling a programming language or other abstract topics, we find it is far more common for students to embrace concrete goals such as modeling a distributed hash table or a search algorithm. Gradual FM focuses on the latter, on systems-oriented topics.
- *Leverage programming intuitions.* We assume that students will have experience with programming and unit testing but perhaps no other common background—not even a course in discrete math. This setting is especially suited to modeling with objects and functions. Likewise, testing strategies that students have seen should be used to teach (e.g., using unit testing to understand satisfiability), not just be disparaged for their incompleteness.

For all the reasons mentioned in section 1—including its accessible syntax, push-button automation, visualization, etc.—Alloy is a good starting point for implementing a gradual FM pedagogy.

Experience context: The experiences reported in this paper were run in the context of such a gradual course on FM that we have taught for a decade. The course, Logic for Systems (L4S), began developing a pedagogy in 2013 and migrated from Alloy to an early version of Forge in 2019. Later years extended Forge with language levels (2021), temporal operators (2021), and tools for custom visualization (2022).

L4S is offered in the computer-science department at Brown, a selective private research university. The only prerequisite for enrollment is an introductory programming course. Our students came in with a variety of coursework experience. Some had taken only a one-semester accelerated introductory course using Pyret because they had prior programming experience. Some had completed a year-long introductory sequence using Java and one of Pyret, Racket, or ReasonML. The majority were upper-level students with additional experience, and a few were graduate students. Many students had some industrial internship experience as well. However, virtually none of them had prior formal-methods coursework. We suspect they also got no exposure to FM in industry. There were 92 students enrolled in 2022 and 64 students in 2023.

3 A PROGRESSION OF LANGUAGES

Alloy includes a powerful language, but, when it comes to teaching, one full-featured language is less ideal than a step-by-step progression of languages. We start with a critique of Alloy (section 3.1); this motivates the languages in Forge (section 3.2). We defer evaluation to section 7.

3.1 A Critique of Alloy's Language

Over the years, we have observed several issues that Alloy presents to learners. We stress that, in general, we find Alloy's language very effective for software modeling; we want programmers to learn how to use it. Rather, our observations are about identifying ways to improve the language *with pedagogy in mind.*

3.1.1 Critique: Functions vs. Relations. While on the surface Alloy has a Java-esque syntax, its semantics is based on relations. Alloy makes clever use of relational operators to support an apparently object-oriented style of modeling. For example, Alloy's join operator is written as `.` (dot) and thus *looks* like a field access, as in `p.father`. Early on this pun gets students comfortable working with Alloy. However, at some point, students realize that the semantics is not what they expected based on syntactic recall (which seems to happen no matter what we teach in class): "backwards accesses" such as `father.p` are perfectly sensible joins.

Another issue arises from the fact that all Alloy expressions evaluate to sets. In a model of family trees, we might write that there exists some person whose father is their only friend:

```
some p: Person | p.friends = p.father
```

Since both `p.friends` and `p.father` are expressions, and expressions evaluate to sets, the equality makes sense semantically. But to a student whose prior context is a language like Java, where the type `Person` is different from `Set<Person>`, this leads to confusion: shouldn't the formula produce a type error, since `p.father` is always a person, not a set *of* persons?

3.1.2 Critique: Syntactic Sugar. Alloy's language also contains syntactic choices that are convenient to expert modelers but problematic for students. We list two subtle examples below.

Non-compositionality: The following shorthand constraint, taken from Jackson [2012, ch. 6], says that the `keys` relation is a partial function between the set `Key` and the set `Room`:

```
keys in Room lone -> Key
```

If a student tries to experiment with the subexpression `Room lone -> Key` to understand the impact of **`lone`**, they will find no impact; the subexpression is equivalent to `Room -> Key`. This is because the meaning of **`lone`** applies to the **`in`** operator—even though **`lone`** appears syntactically within the cross-product (`->`) expression.

Quantification: Alongside the standard first-order **`all`** (∀) and **`some`** (∃) quantifiers, Alloy provides **`no`** and **`one`** as well, which are just shorthand for more complex formulas. However, some properties of basic quantifiers do not apply to these advanced ones. For instance, **`some`** distributes over disjunction but **`one`** does not. Alloy also allows multiple variables to be grouped under the same quantifier, but advanced quantifiers can behave unexpectedly:

`no x, y: A | ...`	is *not* equivalent to	`no x: A | no y: A | ...`
"there is no pair (x, y) such that..."	$\neq$	*"there is no x such that there is no y such that..."*

3.2 Language Levels

Problems like these present us with a quandary: for experts, or at least for compatibility, we need the full power of the language. But for learners, we want to begin with a simple language and gradually introduce useful features. Textbooks often follow exactly such a progression, but tools do not always mirror it. Therefore, students can either be tripped up by richer semantics than they expected, or can stumble on keywords (e.g., from documentation, StackOverflow, or even generative AI tools) that do not do what they wanted. These issues with the professional-grade language lead to student frustration, adding to the high cognitive load imposed by learning about specification, and often result in erroneous conclusions, which are dangerous in verification.

A solution proposed by multiple teams over the years is to use *language levels* [du Boulay et al. 1999; Findler et al. 2002; Holt and Wortman 1974]. That is, instead of just a single language, present a *progression of sub-languages* that grows with student instruction and accomplishment. This process has been implemented for the book *How to Design Programs* [Felleisen et al. 2001] in the DrRacket IDE. Other textbooks and languages, e.g., Grace [Homer et al. 2014], have also exploited this idea.

Readers familiar with Alloy may be reminded of its three modes of use Jackson [2012]: navigation, predicate calculus, and relational calculus. These correspond roughly to language levels in that they use a fragment of the full syntax, but nothing prevents students from straying outside a language boundary. For example, if a student accidentally reverses the arguments to the dot (`.`) operator:

```
mother.Ani = Barbara
```

An early language level, in which dot represents field access, should report an error—even though this "error" is a perfectly valid and well-typed expression in the full Alloy language. Moreover, if a student makes a type error, such as asking for the `population` (a property of cities) of a person:

```
Ani.population
```

An early language level should not give the standard Alloy message: "the join operation here always yields an empty set" [Edwards et al. 2004]. This is a reasonable error only if the student knows

Language Feature	Froglet	Rel. Forge	Temporal Forge	Alloy 6
Partial-function fields, **`some`**, **`all`** quantifiers	✓	✓	✓	✓
Dedicated testing constructs (section 5)	✓	✓	✓	✗
Unrestricted relation fields	✗	✓	✓	✓
Unrestricted relational operators	✗	✓	✓	✓
`one`, **`lone`**, **`no`** quantifiers	✗	✓	✓	✓
Scriptability (Section 6)	✗	✓	✓	✗
LTL operators, infinite-trace semantics	✗	✗	✓	✓
Non-compositional shorthand	✗	✗	✗	✓
Global constraints, **`sig`**-specific constraints	✗	✗	✗	✓

Fig. 1. Language-level comparison. The three language levels form a gradual progression toward the expressive power of Alloy 6, while adding teaching-focused features such as explicit testing constructs (section 5).

what a join is! Instead, the first Forge language says "sig does not have such a field." The key is that all feedback—such as error messages—must use only concepts that have been introduced so far [Findler et al. 2002], otherwise the epistemic closure of the language is lost.

Forge provides students with three separate language levels. Each closely resembles the Alloy language, but with targeted restrictions and improvements:

(1) **Froglet** is a restriction of the relational Alloy language to *partial functions*, rather than relations. Sets and most relational operators are absent; the `.` operator behaves like field access. Top-level expressions evaluate to singletons. Error messages are in terms of objects, fields, etc. Froglet also provides a `reachable` helper predicate that can express basic reachability constraints without the transitive-closure operator. Lacking recourse to relational operators, students must use explicit quantification—reinforcing their facility with basic concepts.
(2) **Relational Forge** adds sets and relational operators such as union and transpose. At this point in the curriculum our students are familiar with basic concepts and are prepared to focus more deeply on the underlying relational nature of the language. This level also introduces shorthand quantifiers such as **`one`** and **`no`**. By now, students often want to model weighted graphs and other concepts that are much easier to represent with arbitrary-arity relations than boolean-valued functions; relations are thus easy to motivate.
(3) **Temporal Forge** introduces time and temporal logic operators, in the style of Alloy 6 [alloytools.org 2023; Macedo et al. 2016]. This is a tricky step because students are progressing from finite-length traces to infinite ones. Subtle issues invariably arise, such as the question of how to model a deadlock state within an infinite trace. However, the staging of languages means that students have already become familiar with relations, and can therefore focus on understanding temporal logic without additional cognitive burden.

Students are told for each assignment which level to use (or, on larger projects, which levels they may choose from). As is standard in Racket languages, the language is chosen in the first line of each source file [Felleisen et al. 2018]. Thus, these rules can even be enforced in a grading system. In our experience, it does not take very long at all for students to understand language levels.

Figure 1 shows a comparison of features between Forge language levels and Alloy 6. Alloy has features, such as global constraints, that no Forge language supports but that expert users (including the authors) find valuable. Figures 2 to 4 show one example model for each Forge level: fig. 2 demonstrates quantification in Froglet, fig. 3 uses transitive closure in Relational Forge, and fig. 4 specifies an increasing counter in Temporal Forge. For space, we show the most pertinent parts of each example; the full files are available in the artifact.

```
abstract sig Player {}
one sig X, O extends Player {}
sig Board { board: pfunc (Int -> Int) -> Player }
pred wellformed {
  all b: Board | all row, col: Int | {
    (row < 0 or row > 2 or col < 0 or col > 2) implies
      no b.board[row][col] } }
```

Fig. 2. Froglet example: boards in tic-tac-toe. There are exactly two players: `X` and `O`. Each board contains a partial function (`pfunc`) that maps row-column pairs to player names. The `wellformed` predicate constrains play to valid row and column indices.

```
sig Node { edges: set Node -> Int }
pred wellformed { all disj n1, n2: Node | lone n1.edges[n2] }
pred connected { all disj n1, n2: Node | n2 in n1.^(edges.Int) }
```

Fig. 3. Relational Forge example: weighted directed graphs. Use of the join operator in the `connected` predicate allows easy removal of weights from the edge relation for use with transitive closure.

```
sig Counter {var value: one Int}
pred someTrace { Counter.value = 0 and
  always { Counter.value' = add[Counter.value, 1] } }
```

Fig. 4. Temporal Forge example: an increasing counter. The syntax is quite similar to Alloy 6; following linear temporal logic (LTL), expressions are evaluated with respect to an implicit time index; priming (`'`) references the next state. Thus, `someTrace` says that the counter initially holds 0 and advances by 1 in every step.

4 CUSTOM VISUALIZATION

Forge integrates the Sterling visualizer [Dyer 2020]. At first glance, this simply replaces Alloy's default visualization style with a modern Web-based interface. However, Forge also leverages Sterling's support for *custom visualizations* to both provide domain-specific visualizations for homework problems and to enable students to create their own visualizations. Custom visualizations are written in JavaScript and can either manipulate the Sterling page directly, use the D3 data-visualization library [Observable 2023], or leverage a number of built-in quick-prototyping features. Refer to our artifact or the Forge documentation for code examples.

Custom visualizations are much more than pretty pictures. They serve a valuable pedagogic purpose by avoiding confusion and lowering the cognitive load to understanding. Visualizations are especially important if FM tools are to reach students outside of FM—easily-comprehensible counterexamples can be a powerful teaching tool in security or networking classes. Industrial modeling, where instances may be huge and tied to specific domains, can also benefit from carefully-chosen visualizations. Indeed, one of the examples below (section 4.3) is derived from a challenge posed by an expert Alloy user [Zave 2020], who said of the default Alloy visualization: "I never use it, believing (rightly or wrongly) from past experience that my models are too complex." She further related [personal communication, used with permission] that she draws pictures by hand to debug models. Custom visualization can automate such drawings and make them interactive.

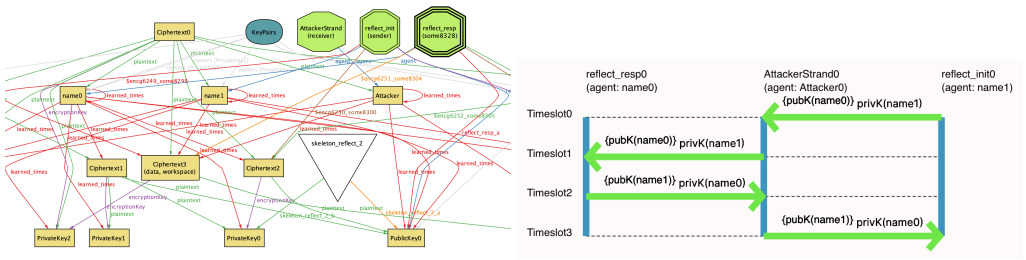


Fig. 5. Visualizing the example protocol. Left: Alloy with projection. Right: custom visualization via a sequence diagram. The middle agent in the diagram represents the medium of communication which, following Dolev and Yao [1983], is synonymous with the attacker.

To motivate custom visualizations in a pedagogic setting, we use three examples: cryptographic protocol executions (section 4.1), river-crossing puzzles (section 4.2), and network reachability (section 4.3). Each highlights issues with Alloy's default visualization and shows how our custom version mitigates the problems. Crucially, none of the issues are solved by Alloy's (useful and flexible) theming feature, which allows users to adjust presentation settings and project out some detail (e.g., showing only one system state at a time). We present results on student use in section 7.2.

4.1 Example: Crypto Protocols

Figure 5 visualizes a protocol execution instance between an initiator and a responder, with an attacker in between. The protocol has two steps: first, the initiator sends the responder's name, encrypted with the responder's public key; second, the responder replies with the initiator's name, encrypted with the initiator's public key.

The left subfigure shows Alloy's default visualization with the "magic layout" feature applied and projected to one state. Even though projection allows a viewer to step through the protocol, it is unclear what, if anything, a viewer can learn from this without much tedious study. (The default table-based output is no better; there are 19 rows in one `learned_times` relation alone.) One might imagine filtering this flood of information via theming, e.g., by hiding encrypted terms that aren't used in a given state, but because of the model's structure this is not currently supported in Alloy.

In contrast, the right half of fig. 5 shows a custom visualization of the same instance in Forge, expanding on prior work with protocols in Sterling [Siegel et al. 2021]. It captures the standard imagery of protocol message-passing in the "Alice and Bob" style [Caleiro et al. 2006; McCarthy and Krishnamurthi 2008], with each of the four transitions represented as horizontal arrows. This *domain-specific format* allows a user to see the full execution of the protocol at once, while simultaneously eliding low-level detail that is at best unnecessary for the user to see.

4.2 Example: River Crossing

Figure 6 shows the default and a custom visualization for pairs of states in a river-crossing problem. Observe how easy it is to see the difference between the two states in the custom visualization, and how hard it is in the default; a reader can very easily miss all the differences—or even that there *are* any differences. Furthermore, the layout of atoms in the default visualization is based on a generic graph layout and therefore does not reflect the actual location of objects. In the first state, the farmer is separated from the fox although they are on the near side of the river, and the chicken and fox are together despite being on opposite sides. The custom display shows the actual *river*, which is implicit in the model, and puts atoms on the correct sides of it.

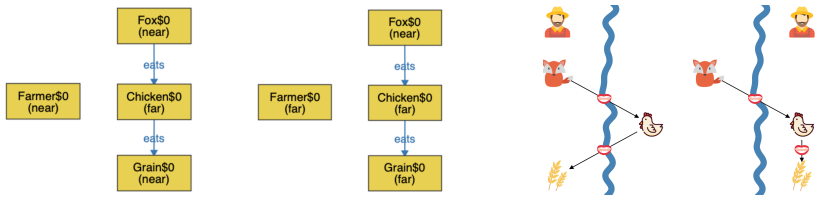


Fig. 6. River crossing states. Left: default. Right: custom.

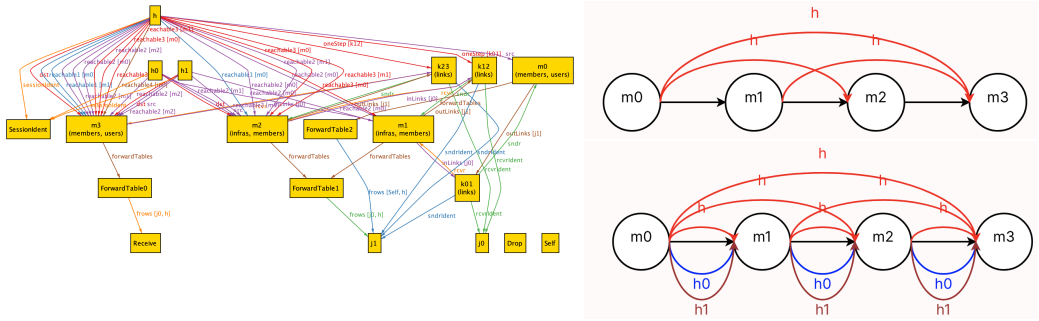


Fig. 7. Different definitions of reachability in a single instance. Left: default visualization (showing all definitions). Right: custom visualization for two different definitions; a clickable menu allows the viewer to adjust which definition should be shown. Line colors vary to show different packet headers being processed differently (not currently possible via Alloy's theming options).

4.3 Example: Network Reachability

Figure 7 visualizes a single instance for a model that contrasts four different definitions of network reachability. Because Alloy's default visualizer produces directed graphs, one might think that it should do well; however, it does not.[2] This example highlights two issues:

- The reachability relations contain tuples of the form (*hdr, src, dst*): a header that determines how the packet is forwarded, a source, and a destination. Alloy assigns the first element *hdr* to be the edge's source, the third element *dst* to be the edge's sink, and the middle element *src* (the actual source node) to be the edge label.
- Since the model represents physical links as discrete atoms, they are objects in the default visualization. Each link has a sender and receiver field, which are displayed as edges. But we would do much better to collapse each link object into a single direct edge.

These issues could perhaps be fixed by expanding Alloy's theming feature to let the user control edge-label assignment and collapse nodes into edges (rather than breaking the connection when links are hidden). However, when a student discovers a lack of theming power in their visualization, they are unlikely to spend the effort of researching and modifying Alloy's code to support their vision. Instead, our experience shows that they will struggle to debug their model with the default visualizer. Another fix might be to change the basic structure of the model, perhaps by reordering packet tuples. Indeed, we ourselves do this sometimes, but the refactoring effort is significant. More

[2]During a live lab session at a *formal-methods* summer school where this exact instance appeared, one of the students was heard to say, "Oh, no, I don't like this. Kill it with fire." of the generic visualization (which appeared before the custom visualization loaded). We believe this illustrates well the emotional experience of even a formal-methods-interested student when first faced with such visual output.

importantly, elements of a model are usually there for a reason and sweeping changes often have unforeseen consequences (especially when the modeler is not yet experienced with debugging). Custom visualizations enable modifications while avoiding the need to refactor the model.

5 TESTING: FROM ROUGH IDEAS TO RELIABLE MODELS

For students without experience in formal modeling, unit testing may be the only experience they have approaching FM; after all, a unit test is a point-wise specification. Thus, rather than disparaging unit tests for their incompleteness, we would do well to see them for what they are: a first step on the road to FM. Explicit support for different kinds of testing in a student's model can help anchor their progression from expected input-output pairs to properties to pre- and post-conditions and beyond. We begin with some critique of testing in Alloy (section 5.1) and then describe the design of testing in Forge (section 5.2). Section 7.3 reports preliminary evaluation.

5.1 Alloy Critique: Testing

Testing in Alloy uses annotations on **`run`** and **`check`** commands. For example, a command telling Alloy to produce instances in which someone is married can be converted into a test by adding `expect 1` for "should be satisfiable" or `expect 0` for "should be unsatisfiable":

```
run {some p: Person | some p.spouse} expect 1
```

This lightweight idiom is useful, but has some notable flaws:

- `expect` serves two different purposes in Alloy. Not only does it label satisfiability expectations, it also doubles (as of version 6.1.0) as a way to configure symmetry breaking per **`run`**: `expect 1` disables it and `expect 0` enables it. Converting a **`run`** into a test can therefore alter its output, since symmetry breaking often eliminates many isomorphic instances.
- Alloy lacks a way to separate test cases into suites within a single file. Alloy can execute a single run or *all* runs, but often students are working on a specific area of their model and want to run some tests without waiting for all tests to finish (which may take minutes).
- Writing unit tests with concrete instances can help students understand assignments and reinforce the semantics of satisfiability. Unfortunately, Alloy lacks explicit syntax and optimizations for this. The main challenge is that, to define a non-trivial instance in Alloy, one usually needs to name individual atoms. This means either adding global constants via **`one sig`** declarations, or naming the atoms locally with disjoint quantification (**`some disj`** `x1, x2 ..`). E.g., to define an instance of a family-tree model, we could write:

  ```
  one sig Ani, Barbara, Cedric extends Person {}
  Ani.mother = Barbara and Ani.father = Cedric and ...
  ```

  or, alternatively:

  ```
  some disj a, b, c: Person | a.mother = b and a.father = c and ...
  ```

  Since *all* tests in a module must instantiate *all* **`one sig`** definitions, the global approach can result in confusing overlap between tests in the same file. If every test is in its own file, the global approach can work well, but at the added organizational cost of multiple files. The quantification method is more compact, but can suffer from poor performance. There has been work on adding better unit-test support to Alloy, such as AUnit [Sullivan et al. 2018], but these are not currently merged into the official tool.

Finally, testing can be good for more than just validating an existing model. In the programming domain, tools like CodeWrite [Denny et al. 2019] and Examplar [Wrenn 2022] have shown that *executable examples* are a useful way to detect early misconceptions and provide students with immediate feedback—before they have written any code. They do this by running student tests

```
1  pred fullFirstRow {(Board.board[0]).X = (0+1+2)}
2  pred someMoveTaken {some Board.board}
3  ----- New Constructs: -----
4  inst wellformed_instance {
5      Board = `Board0
6      Player = `X + `O
7      X = `X   O = `O
8      `Board0.board = (1, 1) -> `X + (1, 2) -> `O
9  }
10 example moveMiddleFirst is {wellformed} for wellformed_instance
11 test expect { {someMoveTaken} for wellformed_instance is sat }
12 test suite for winning {
13     assert fullFirstRow is sufficient for winning for 1 Board
14     assert someMoveTaken is necessary for winning for 1 Board}
```

Fig. 8. Testing constructs in Forge based on the tic-tac-toe model in fig. 2. The **example** checks that a specific board is considered wellformed. The **test** confirms that moves can be taken on this board. The **assert**ions check that a specific pattern of moves implies a winner exists, and that no player can win until some moves are taken. The `winning` predicate, not shown due to space, encodes the 3-in-a-row criteria.

against a hidden, known-correct (*wheat*[3]) implementation. Ideally, we would be able to borrow this idea from programming education to help students in a formal modeling context, but Alloy is not built to run tests against a hidden model.

5.2 Design of Testing in Forge

Forge languages support a number of testing constructs (fig. 8):

Examples (line 4) are concrete instances that should satisfy a given formula. Educationally, these correspond to unit tests and tend to be used early on in the modeling process ("Here is what I think should happen. Does it look right?"). For reasons related to how examples are implemented (section 6), the example construct is currently only available in Froglet and Relational Forge, not in Temporal Forge.

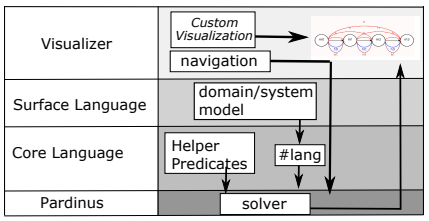
Assertions (lines 11–12) are normative statements about implication between predicates. They are extremely constrained by design, and can only express whether one predicate is *necessary* or *sufficient* for another. Educationally, these (roughly) correspond to property-based tests. ("Is an undirected tree always a directed tree?")

Test Blocks (line 9) are a less limited form of assertions that state a given run will be either satisfiable or not. They are useful in cases where assertions do not suffice, such as when a student wants to check that a property is non-vacuous. ("Can my transition system even run?") *Test blocks most closely resemble the style of testing in Alloy.*

Suites (line 10) group tests within the same file. Each suite comes with a label (typically the name of a predicate); Forge warns the user if this label does not appear in the enclosed tests.

Testing early. Forge provides the ability to run all of these testing constructs against hidden known-correct models that course staff can distribute with assignment handouts. If a student's test flags the known-correct solution as erroneous, it is likely that their understanding of the problem is invalid. In either case, students receive an instance illustrating the mismatch.

[3]As opposed to incorrect (*chaff*) implementations. We adopt this terminology, although we use only wheats.



(a) Forge Architecture

```
#lang forge/core
(set-option! 'problem_type 'temporal)
(sig Counter)
(relation value (Counter Int) #:is-var "var")
(pred someTrace (and
  (= (join Counter value) (int 0)))
  (always (= (join Counter (prime value))
             (add (join Counter value) (int 1)))))
(run someTrace_run #:preds [someTrace])
(display someTrace_run)
```

(b) Core Forge version of a temporal program (fig. 4)

Fig. 9. Forge consists of several layers. Its languages compile to a common core.

6 ARCHITECTURE AND IMPLEMENTATION

Forge is built atop Racket [Felleisen et al. 2015], with a slightly-modified Pardinus backend [Macedo et al. 2022] and modified Sterling for visualizing instances [Dyer and Baugh 2021]. Racket provides a toolkit for building languages. Forge takes particular advantage of its tools for IDE support and custom error messages. Pardinus is a cutting-edge solver. Forge was an early adopter, and Alloy later integrated Pardinus in Alloy 6. Sterling displays instances found by Pardinus in an interactive Web page. Forge adds helpers to ease the creation of custom visualizations.

Figure 9a shows the architecture in more detail. Each language level contains its own surface syntax, error messages, and expander that maps its syntax to a *core Forge* language. For instance, fig. 9b shows the core-language version of the Temporal Forge program from fig. 4. The core language is implemented as a Racket library; students do not need to engage with this core syntax unless they wish to.

Semantically, the core and surface versions are identical. However, the core language provides a convenient target for Racket macros and other Forge sub-languages. Because of this, Forge is *scriptable*: students can write Racket programs to do anything from counting instances to performing counterexample guided synthesis. We have also used this facility to create *domain-specific* modeling languages (section 9). However, we have not yet built and deployed enough of these languages to fully understand their pedagogic implications.

Performance. Because Forge is built on the same solver engine as Alloy 6, its performance is comparable to Alloy's in most respects. However, there are differences:

- When run, Forge models execute as Racket programs—meaning that there is a 1–2 second delay while the Racket process loads. We plan to fix this issue for 2024 by changing the architecture to use a persistent worker process, much like Alloy does with Java.
- Forge provides support for explicit partial instances (similar to Montaghami and Rayside [2012]), which can greatly improve the solving speed on some problems. However, partial instances cannot yet be specified at the per-state granularity in Temporal Forge (hence why **`example`** is unavailable there, since **`example`** is implemented via partial instances). This limitation is because partial instances are embodied in the upper and lower bounds passed to the solver, which apply to *all* states in a solution trace. Thus, while partial instances can still be used for optimization in Temporal Forge, fine-grained control is not yet possible.

7 EVALUATION

We evaluate Forge along the three core dimensions presented in this paper: language levels (section 7.1), custom visualization (section 7.2), and testing (section 7.3). The formal evaluation is based

on two semesters—Spring 2022 (92 students) and Spring 2023 (64 students)—of teaching with Forge in L4s. We remark that we have deployed versions of Forge in *five* semesters—which have enabled us to refine the above ideas—and several smaller events (e.g., summer schools), but do not have rigorous data from those experiences. The remarks in section 2 about our students' backgrounds apply to our survey participants.

Since our survey instruments collected a variety of data, we used several methods to conduct the analysis. For quantitative data, we built tables and graphs like those shown below. For qualitative data, three authors worked together to categorize responses using standard practices. One author proposed initial labels for the data, the other authors reviewed the codes, and the team revised the labels if needed to reach agreement. Our artifact contains all the survey instruments used in the evaluation.

7.1 Evaluation: Language Levels

Having created language levels and deployed them in the classroom, the obvious question is: are they effective? Just in terms of course structure, splitting the language was beneficial. In particular, Froglet let the L4s course (which has no discrete math prerequisite) replace a first assignment that simply drilled relational operators with a realistic modeling exercise. But here we look deeper.

To determine the concrete impacts on students, we investigate the following research questions:

RQ1. To what extent are students able to model topics of interest in the early, less expressive language levels?
RQ2. What preconceptions do students have *before* moving to a new language level?
RQ3. What difficulties do students face *after* transitioning from one language level to another?

7.1.1 RQ1: Expressive Power. There are three types of modeling assignment in L4s: homeworks, labs, and projects. While homeworks and labs are tailor-made to fit the current language level, projects are free-form. Students form groups of two or three and model any domain of their choosing. They receive guidance and feedback on proposed topics, but the specifics are entirely their own. Naturally, their choices may conflict with the design for the current language levels.

There are two projects, a midterm and a final. The midterm takes place a few days after the switch from Froglet to Relational Forge; Temporal Forge is not yet available. The final takes place after the switch to Temporal Forge. Students were encouraged to use Froglet on the midterm, but could switch to Relational Forge after meeting with a staff member. Students could use any language on the final. During the weeks-long final project, students met twice with course staff to report their progress and possibly receive suggestions; at these meetings, course staff may have recommended a certain language.

Chosen topics are naturally colored by the modeling experience students have had so far, but we believe that the breadth of topics and students' ability to execute their vision in a given language still gives us a valuable perspective on the expressive power of the early language levels.

Results. Figure 10 details the languages students used on their projects. Midterm projects were mostly done in Froglet (59 of 77 across both years) and included diverse topics such as: board games, fast-food orders, chemical reactions, sports, algorithms, and data structures. For groups who chose Relational Forge, the sticking point with Froglet was the lack of support for sets. For example, a group modeling Dijkstra's shortest-path algorithm found it convenient to store the (weighted) edge relation as a field of type `set Node -> Node -> Int` rather than emulating the relation as a partial boolean-valued function.

Most final projects used either Relational Forge (26 of 59 across both years) or Temporal Forge (also 26 of 59 across both years). Final project submissions were always more technically demanding

2022	Midterm	Final
total	45	33
Froglet	36	0
Relational	8	18
Temporal	N/A	13
SMT	1	2

2023	Midterm	Final
total	32	26
Froglet	23	2
Relational	9	8
Temporal	N/A	13
SMT	0	3

Fig. 10. Language levels that students chose for their projects. SMT projects used an SMT solver such as Z3 [de Moura and Bjørner 2008] instead of Forge.

than the midterms. Projects included OS memory management, grammar trees, and air traffic control; fig. 13, which appears in section 7.2, gives a longer list of final project topics.

A handful of groups (1 midterm and 5 finals across both years) used an SMT solver for their projects. Usually these projects needed features that neither Forge nor Alloy provide (e.g., soft constraints). In one case (Minesweeper in 2023), the group switched for performance reasons caused by the fact that Forge (like Alloy) uses only bit-vector arithmetic, which can be slower than what SMT provides via theory solvers.

Interpretation. Froglet seems to suffice for most students' first self-directed foray into modeling. Most midterm projects did not need relations, and all but 17 (22 %) of the others could encode relations as boolean-valued functions in Froglet. Froglet did not suffice for weighted graph algorithms. We believe this is an acceptable limitation because students had experience with Relational Forge at this point in the semester and could switch to it; still, it may be worthwhile to consider adding library functions for weighted graphs (analogous to the Froglet `reachable` helper). Anecdotally, the variety and scope of midterm projects compare favorably to the years before Forge was in use. Froglet may have enabled more-ambitious projects earlier by simplifying the language.

The final projects used a combination of Relational Forge (44 % overall) and Temporal Forge (also 44 % overall), with the few remaining using Froglet or SMT. We attribute the even split to two competing factors: the familiarity of Relational Forge and the expressiveness of Temporal Forge. Students were inclined toward Relational Forge because they had more experience with it at the time when project proposals were due. Temporal Forge was a compelling option because of its convenience for modeling systems that change over time—a common theme of the course.

7.1.2 RQ2: Preconceptions. Moving from one language level to the next requires students to adapt to major semantic changes. With Relational Forge, sets become the sole datatype in the language. With Temporal Forge, instances become infinite-trace lassos. These shifts can present a roadblock to learners if done poorly.

To ease the transitions between language levels, we used surveys and in-class discussions to identify preconceptions and address them *before* the class changed languages. For the surveys, students worked in pairs to assess possible behaviors for a semantic construct. E.g., the question in fig. 11 asks students currently using Froglet to rate possible outcomes for a join (`.`) in Relational Forge. Ratings used two dimensions (adapted from Tunnell Wilson et al. [2018]): is the outcome something you *expect* or not (e.g., an error might be surprising), and is the outcome something you *like* or not (e.g., an error might be helpful despite at first being surprising). The advantage of separating these dimensions is in forcing respondents to separate their knowledge of the language (*expected*) from their opinions about it (*liked*). We also asked students to write in an outcome if they preferred one that was not listed.

Prompt: `run { some c: Course | c in c.prereqs }`

Q. Rate each of the following outcomes:	Expected and Like	Expected but Dislike	Unexpected but Like	Unexpected and Dislike
SAT	○	○	○	○
UNSAT	○	○	○	○
Error: no such field `c.prereqs`	○	○	○	○
Error: course cannot be its own prereq	○	○	○	○

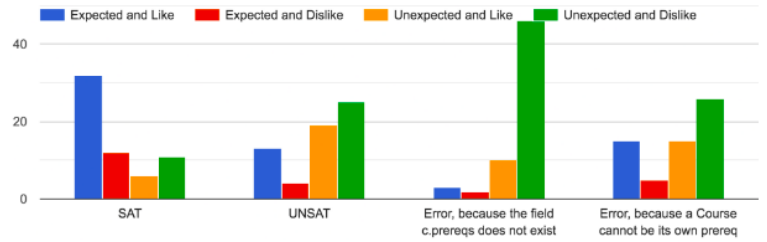


Fig. 11. Example pre-Relational-Forge survey question and responses from 2023.

Figure 11 presents one question from the Relational Forge survey and the responses from 2023. Forge returns `satisfiable` (SAT) for this prompt. The responses show that most students *expect* this outcome, but roughly a third of them *dislike* it and instead prefer an error. Students who do not expect a SAT outcome can use this example to learn why Relational Forge behaves the way it does.

Results: Froglet to Relational Forge. For the Froglet-to-Relational-Forge transition, we obtained 77 responses from 2022 and 61 from 2023. The questions focused on three features that are present in Relational Forge but inexpressible in Froglet: can a singleton be contained in another singleton (e.g., `person in c.instructor`); can a singleton be equal to a set (e.g., `person = c.TAs`); and what behavior should reversing the dot operator produce (e.g., `CS2 in prereqs.CS1`)? Across both years, for each of the above questions, students preferred some error behavior the most (e.g., expected and liked "Error: Cannot compare a person and a set."). Free-form comments indicated a strong call for type errors ("set[X] is a different type than X itself.") from 43 (56 %) in 2022 and 20 (33 %) in 2023. Flipped join attracted many negative comments, from 21 (27 %) in 2022 and 6 (10 %) in 2023, some of which were quite strong ("What a travesty that would be.") and led to lively and productive discussions about why the relational semantics is useful for lightweight FM.

Interpretation. The survey responses highlight ways that modeling with relations is more of a leap than modeling with functions for students used to programming with objects, with many *unexpected* and *disliked* behaviors relative to that prior experience. Survey comments informed followup lecture content to clarify points of confusion. More generally, the survey revealed a preference for noisy failures over silent ones (error rather than `unsat`), which motivates future work on all Forge languages (section 9).

Results: Relational Forge to Temporal Forge. For the Relational-to-Temporal transition, we obtained 71 responses from 2022 and 59 responses from 2023. The survey focused on the meaning of four temporal operators: `eventually`, `always`, `until`, and `next_state` (in LTL: F, G, U, X). Students had no prior exposure to these operators and the survey did not introduce them; its purpose was to identify potential upcoming misconceptions stemming from the English-language names for the operators. Each survey question presented a formula, a trace describing a system, and

several possible outcomes of matching the trace against the formula. We adapted this format from Greenman et al. [2023]. Traces were given in graphical form, not via Forge syntax.

We found similar responses and misconceptions to what the prior authors uncovered; e.g., students can be confused about the meaning of a formula without a temporal operator attached, often prepending an implicit `always`. We also observed confusion about whether `next_state` applies to one state or to several. Saarinen [2021] observed this issue but Greenman et al. [2023] did not; perhaps the issue is common among beginners but easy to correct.

One surprising outcome is an apparent mixup about functions versus formulas. For example, on the formula `next_state{next_state{eventually{Red}}}`, a student commented: "I feel like it doesn't really make sense to call `next_state` on `eventually` because `eventually` should *already evaluate to true or false*, and there is no actual formula for `next_state`." (emphasis ours). In fact there is no "call", and the `eventually` subformula does not evaluate to a boolean before the `next_state` operator is evaluated. This mixup is surprising because students had spent almost two months working in Froglet and Relational Forge, where quantifiers are evaluated similarly to temporal operators. It would therefore seem that claims of conceptual transfer from first-order quantification to *temporal* quantification are ill-founded in some cases; students may default to their programming intuitions even in a formal logic context.

7.1.3 RQ3: Difficulties Using New Language Levels. After each language change, we asked students to complete in-class surveys about stumbling blocks they encountered with each new language compared to the previous language level. Our goal was to identify specific issues qualitatively.

Results: Relational Forge. We obtained 54 Relational Forge responses. While a few students preferred working in Froglet ("The change to everything being treated as a set really threw me off."), most were happy with Relational Forge. One student wrote that Relational Forge is "a lot more useful and intuitive"; a few others appreciated its set-based semantics ("the fact that the underlying machinery in Forge is sets makes it all the better since I come from a more mathematical background."). Yet, there were also many reports of trouble with relational operators ("it took me a while to really understand some of the operators like the dot operator and transitive closure operator and that everything was a set"; "especially `->` and `^` were tough [...] getting problems with arity is hard to picture.").

One survey question asked which language students would use now if they could redo their midterm project. Whereas only nine midterm groups used Relational Forge (fig. 10), 29 respondents chose Relational Forge. These numbers suggest that students were very comfortable with Relational Forge after the transition. Another six chose a mix of languages and eight students did not have a strong opinion.

Results: Temporal Forge. We received 32 Temporal Forge responses. Student opinions of Temporal Forge were strongly positive overall: 17 responses preferred working in the temporal language ("Writing transition statements in normal Forge was a confusing mess. On the other hand, these transitions were much easier to write (and then later read!) in Temporal Forge."; "Having to specify a 'State' sig for (almost) every model was tedious, and I appreciate how Temporal Forge handles state for you."). If they could revisit their midterm project, 13 responding students (nearly 41 %) would have used Temporal Forge. This is encouraging with regard to the value of Temporal Forge, and the temporal fragment of Alloy 6, in teaching contexts.

Students did raise some concerns about the conceptual load of infinite traces ("I think understanding the counterexamples is harder, because it could involve infinite states and loops, and we'd need to go through all the states to find the actual contradicting state."; "The most confusing concept for me in Temporal Forge was understanding lasso traces; I was particularly confused with how this

prevented us from modeling deadlocks.") and how the expression evaluator worked in the presence of implicit traces ("Harder to use the evaluator because you have to use next state."). One student reported confusion with the priming operator, which delays the interpretation of a sub-expression until the next state: "It is counter intuitive to me that accessing a field of a Sig in the future does not actually return that Sig's field. E.g. `Person'.Name != Person.Name'`". This comment highlights a way that relations and temporal operators can combine awkwardly—especially when programming intuitions are applied.

Interpretation. Concepts like transitive closure remain challenging even halfway through the semester, pointing to the potential value of built-in helpers like `reachable` even in less restricted languages. Responses did not give a clear answer to whether the *exact* restrictions of Froglet are ideal, but it seems clear that some restrictions on the language in the beginning are helpful. In terms of pedagogy, it could be worthwhile to reveal the set-based nature of Forge earlier, while the linguistic restrictions of Froglet are still in effect.

Students found Temporal Forge appealing, but unsurprisingly reported that transitioning to an infinite-trace semantics was difficult. The semantics of priming, especially, were reported as causing confusion. Syntactic checks to spot mis-use of priming and other operators (similar to current checks for join) would be helpful in both Temporal Forge and Alloy.

7.2 Evaluation: Custom Visualization

We have three research questions about the impact of custom visualization on students:

RQ1. What benefits and frustrations impact students as *consumers* of custom visualizations written by others?
RQ2. What benefits and frustrations impact students as *producers* of their own visualizations?
RQ3. What custom visualizations do students actually create, and what sorts of tools would ease their creation in the future?

The first question gives us insight into the value of custom visualizations, separate from the overhead of creating one. This is vital for assignments and presentations in a formal-methods course, and enables the use of FM tools like Forge in other contexts such as security or distributed systems. The remaining two questions help us understand how the process of authoring custom visualizations could be improved.

7.2.1 Experimental Setup. We had two experimental phases. In both 2022 and 2023, students received example visualizations for four modeling activities: the n-queens problem, a pair of river-crossing puzzles, the dining-philosophers problem, and tic-tac-toe games. The course staff provided instruction on building visualizations with demos in class, a tutorial document, and office hours.

In 2022, students were required to produce a custom visualization on their final project (or request an exemption; only 3 of the 27 groups did so). After the semester ended, we sent out a brief survey to collect feedback with which to improve the visualizer. In 2023, we added a library to Sterling called D3FX that helps with the visualization of tables and other common patterns. The final and midterm projects suggested—*but did not require*—that students make their own visualizations, which allowed us to evaluate patterns of use without coercion. Students completed a required reflection after finishing their work. They also completed surveys after using each of our four example visualizations.

7.2.2 Formative Results. Students submitted 24 custom visualizations in 2022, covering a variety of domains including card games, group theory, graph algorithms, and election systems. We found several common elements among the visualizations:

- **Domain-specific layouts**, like the sequence diagrams in section 4.1, were common. Binary trees, sports-field and board-game layouts, and even group theory have well-understood visual idioms that students tended to use (or were frustrated by difficulties creating).
- **Positional factors**, like those in river-crossing puzzles (section 4.2) were common. Students modeling binary trees needed the left and right children of a node to be positioned properly, those modeling sports fields had a layout in mind, and so on.
- Some students added **interactive** elements, such as custom buttons to explore a trace.
- Finally, **tables** appeared in a variety of contexts. Students made custom tables to see a game's full progression over time, to organize a weekly schedule, or to arrange a panel of buttons.

After the semester ended, we sent out an exit survey and received six responses. When asked for (non-exclusive) reasons for frustration or lack of success, students selected "limited time" (5 responses), "lack of JavaScript experience" (3 responses), and "issues with D3 or other data visualization framework" (1 response). We received three free-form responses, which asked for improved documentation and reported bugs in visualizations. Lack of JavaScript experience was also a major factor at office hours, pointing to the need to simplify the task of creating visualizations.

7.2.3 RQ1: Benefits and frustrations as a consumer. For experiences with *using* the custom visualizations students were given, we conducted in-class surveys immediately after assignment deadlines. We present representative quotes and takeaways below.

Benefits:

- **Untangling the default:**
  Some students found the default visualization challenging and time-consuming to read ("The [default graph] was very messy and hard to read, so a custom one was pretty much required."). It is worth noting that Sterling's default is an improvement over Alloy's (see Dyer [2020]). The custom visualizations were especially valuable on assignments ("This visualization made it much easier to actually understand the instance.") even if the design was crude ("Although it was not very pretty to look at it did the job."). *Takeaway: Students find custom visualizations to be helpful enough to merit their inclusion in assignments.*
- **Domain-specific idioms:** Many students appreciated domain-specific idioms. For example, being able to visualize an entire trace at once was useful ("The frame by frame linear visualization of different states was very helpful and something that is missing from Forge's default visualizer."). They reported that this helped to keep track of different objects ("easier to track movements of individual [goats and wolves]."), but wished for a clearer presentation of state-to-state changes ("I would appreciate arrows or some kind of emphasis on what changed in the transition between states."). *Takeaway: Students find domain-specific display formats and other non-standard visual idioms useful, when they are appropriate. When presenting multiple states at once, it is useful to highlight changes between each.*
- **Debugging or manual validation:** Some students used the custom visualizer for debugging on homeworks ("Our implementation initially failed, by observing the visualization we know it would get to a deadlock."). Others reported that they were able to use the default table view to debug, only reaching for the custom visualizer at the end of the assignment ("I didn't use it for debugging really but for verification of my results."). *Takeaway: Custom visualizations are often (but not always) useful for debugging. Even when they are not, they can still aid manual validation.*

Frustrations:

- **Unnecessary customization:** Some students reported that they continued to use the default, especially on simple assignments ("I think with these [river-crossing] models the

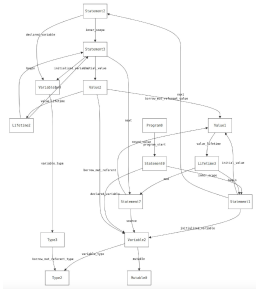
visualization isn't completely necessary. I didn't personally use it.") *Takeaway: Not everyone will find custom visualizations useful.*

- **Misleading names:** Many of the frustrations students encountered were due to specific issues with the visualizers that we provided. Object labels were sometimes ambiguous ("hard to check if it was a certain animal's crossing pattern that was causing the issue, or if we were magically making a new animal."). This led to some students finding the river-crossing visualizer "basically useless." Students were also confused when the same label ("R") was used for two different things ("I was a bit confused with the use of R in the visualization. [...] it represented that they were holding their right chopstick, but for the chopstick sigs, it meant that a chopstick was requested."). Finally, a lack of naming was sometimes also an issue ("it did not label WHICH goat/wolf was moving."). *Takeaway: ensure consistent labeling of objects across state visualizations, and avoid ambiguity in labeling.*
- **Implicit ordering:** Implicit ordering in object names caused some students confusion ("I initially thought that the ordering of the smiths represented their order in the table, but it does not. I think putting the Smiths in order would make the visualization even more helpful."). *Takeaway: ensure that any ordering on objects, including the ordering on names (e.g.,* `Person0`, `Person1`*), is clearly reflected in the visualization. Moreover, important spatial information, such as the circular ordering on seats at a round table, should be explicit.*
- **Display bugs:** Some students experienced issues with the visualization environment itself. Text was sometimes cut off or scrolling made difficult ("the time text display is cut off by default, leading me to think that something was broken with my time variable."; "Sometimes the visualization will be too big to fit on the page."). The visualization UI itself could also be frustrating ("I wasn't aware that after we click next, we need to re-click `Run` every time to see the script applied.") *Takeaway: while the UI was functional, further quality-of-life development is needed.*
- **Lack of robustness to deviation:** Finally, and most importantly, a visualization must be robust against buggy models. Course staff built visualizations with reference to a known-correct model, but students applied these visualizations to their own models. Naturally, students visualized imperfect models, revealing lack of robustness in our visualizer ("before my model was working, the arrows were pointing in the wrong direction."; "I think it would be useful to omit the arrows until a model is correct."; "I don't trust it as a way to verify that my code is correct."). E.g., our reference model disallowed a boat from crossing the river without any occupants, and so our visualizer produced confusing results if a student's model did not do the same. *Takeaway: visualizations should be robust against incomplete models and other issues that may arise, and must not be misleading in such cases.*

7.2.4 RQ2: Benefits and frustrations as producer. The retrospective surveys also uncovered useful information about producing visualizations for the midterm and final projects.

Benefit:

- **Building own visualizations:** Students who built custom visualizations echoed the positive sentiments in section 7.2.3 ("It was just better for being able to picture how our model was doing. It was very difficult to tell if it was behaving properly in tabular form."; "The visualization uses images of cards, which is so much more intuitive"; "I made the visualization mainly to help with debugging the Forge model and for my own personal purposes."). The group who produced the ambitious Rust model in fig. 12 found visualization instrumental to their success: "Without custom visualization, I don't think we could have done this." *Takeaway: Students found it worthwhile to produce their own visualizations.*



```
let mut v2: Box<i32>;          Statement0
{                              Statement0
  v2 = Box::new(1);            Statement1 Value1
  let v1: &mut Box<i32>;       Statement2
  {                            Statement2
    v1 = &mut v2;              Statement3 Value2
    drop(v2);                  Statement7
  }                            Statement2
}                              Statement0

Value1 lives from Statement1 to Statement7
Value2 lives from Statement3 to Statement3
```

Fig. 12. Visualizing an instance from student-authored final project on modeling ownership in Rust. Default (left) versus custom visualization (right, Raw D3). Used with permission.

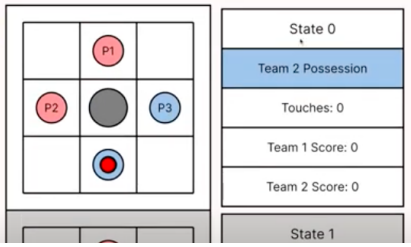
Frustrations:

- **Time investment:** By far the most common issue reported (21 of 85 responses) was the time involved in making a custom visualization ("It seemed really complicated and we preferred to spend more time trying to understand our model better and making sure it worked through tests than just visuals."; "ultimately we ran out of time/chose to spend it on other things like testing and debugging."). One respondent who made a custom visualization said they would revert to the Forge default in the future. *Takeaway: Reduce the time needed to make a visualization.*
- **JavaScript:** Even with our 2023 improvements and D3FX helper library (section 7.2.1), students had trouble programming their visualizers in JavaScript. Some had trouble uncovering the types of environment variables ("it made references to unbound variables that seemingly shouldn't exist but were injected by Forge, and it was difficult to know what the types of these variables would be."). Others asked for further documentation ("we ran out of time and thought there wasn't enough resources for us to learn / figure out how to create one."). One student reported that visualizers were "Impossible to debug." At least one group started their visualizer using the helper library and then switched to other options later in the project ("when using D3FX, we ran into some issues with alignment [...]. Switching to plain D3 gave us control."). *Takeaway: JavaScript (which students were not required to know) is a barrier for students and teaching it should not be a priority in a formal-methods course. Forge should adjust how simple visualizations are written so that even a helper library is unnecessary.*

7.2.5 RQ3: Kinds of visualizations. Figure 13 presents a showcase of 2023 final project topics and associated visualizations. There were 26 projects in total and 18 created some amount of custom visualization. Three groups used the Z3 [de Moura and Bjørner 2008] SMT solver, and so did not interact with Sterling; another 5 used the default table or graph view exclusively. Of the 18 groups that did create a custom visualization, 6 used the helper library (D3FX) we provided. Another 6 used D3 directly (Raw D3). On inspection, 5 of the Raw D3 visualizations were adapted from similar staff-provided examples that used D3. The other Raw D3 group created an ambitious visualization of borrowing in Rust that needed fine-grained control for overlapping colored regions (fig. 12), which D3FX did not provide. The final 6 groups (labeled as `div`-based) created their custom visualizations by manipulating the Sterling page directly. These were adapted from a staff-provided example.

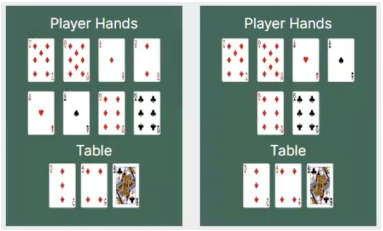
For the midterm, there were more groups (because the project teams were smaller) and fewer groups made custom visualizations. There were 32 groups and 11 made visualizations: 3 of these used D3FX, 4 used raw D3, and 4 a `div`-based visualizer. Required reflection surveys for both

Title	Visualization
2048	D3FX
Air Traffic Control	D3FX
Among Us	**Default** Table
Chopsticks Game	**Default** Table
Chopsticks Game (1)	D3FX
Chromatic Sudoku	Raw D3
Connect4	Raw D3
Gomoku	`div`-based
Grammar Modeling	Raw D3 & **Default** Graph
k-Means Clustering	Z3 Library
Minesweeper	`div`-based
Minesweeper (1)	Z3 Library
Minesweeper (2)	D3FX
OS Memory Mgmt.	**Default** Graph
Online Payment Sys.	**Default** Table
Polyomino Packing	Z3 Library
RPG Randomizer	**Default** Graph
Runway Traffic	`div`-based
Rust Lifetimes	Raw D3
Skyscrapers	Raw D3
Spikeball	D3FX
Sudoku	Raw D3
TA Queue	`div`-based
Texas Hold'em	`div`-based
Triple Triad	`div`-based
Uno	D3FX



(a) Spikeball (D3FX)

(b) Uno™ (D3FX) and Triple Triad (`div`-based)



(c) Texas Hold'em (`div`-based)

Fig. 13. Showcase of 2023 final projects. Left: Project titles and visualization modes. Right: Examples of D3FX and `div`-based-based custom visualizations. All used with permission.

projects generally confirmed the list of features from 2022, except for one group that wanted to create animations, which our helper library did not support.

7.3 Formative Evaluation: Testing

In the Spring of 2023, we gathered formative data to answer three questions:

RQ1. How did students use the ability to write tests that query a correct (*wheat*) solution?
RQ2. What difficulties or frustrations did students experience around querying correct solutions?
RQ3. To what extent did students use each of the Forge testing constructs?

We do not claim to measure learning efficacy or even fully answer the above questions. Rather, our aim in this section is to give an initial report on issues found to drive further development. Indeed, we found several issues with how students used testing constructs, in particular the assertions.

Experimental Setup. We provided students with hidden, known-correct (wheat) solutions for 2 homework assignments: RIVER (three river-crossing logic puzzles) and MEMORY (two algorithms for automated memory management). Students were able to switch between running tests on their own model and running them versus the wheat solution. Our use of wheats is guided by prior work on Examplar [Wrenn 2022] with one notable departure: we supplied only one wheat because, from our perspective, models are not overconstrained by implementation details in the same way that

programs are [Wrenn et al. 2018]. We also examined patterns of testing in students' midterm (MID) and final project (FIN) submissions, since on projects students neither had access to wheats nor any specific guidance on testing beyond the expectation that they test.

Forge provides an *opt-in* mechanism for logging models, test files, and solver results. Fewer than 20 students opted in for RIVER and MEMORY; for these, we collected information on how they queried the wheat. This let us observe work-in-progress attempts at testing. However, due to the opt-in nature of logging (which we implemented out of respect for students' privacy), we have no guarantee that they kept logging enabled consistently. Moreover, this view is shallow: we have no insight into how students reacted to the feedback from the system, only the details of their next query (if any).

7.3.1 RQ1: Use of wheats. Roughly 90% of students' submissions for RIVER and MEMORY referenced the wheat. Logs contained 760 runs for RIVER and 1168 runs for MEMORY where student tests disagreed with the wheat. On RIVER, most wheat-failing tests were examples, not assertions (144 assertions, 19%). In contrast, students wrote many assertions for MEMORY (894 assertions, 77%).

Interpretation. The data show that students did use the wheat to check their understanding of the problem. Our data suggest that wheat use is common. However, students who opted out of logging have used wheats differently. We believe that the increase in `assert` use between RIVER and MEMORY is due to the course's evolution from tests to properties, which are naturally expressed as assertions rather than examples.

7.3.2 RQ2: Wheat frustrations. For both RIVER and MEMORY, we encouraged students to submit critiques when they believed the wheat was flawed. This both supported students' thinking critically about modeling choices and gave us visibility into unexpected complications. We received 10 responses for RIVER and 5 responses for MEMORY. As a representative example, the river-crossing puzzles in RIVER involved using a flashlight or boat, sometimes under a time limit. Students wrote and tested a predicate for well-formed states. Reports revealed:

- Some students were surprised that the wheat allowed a *negative* timer value. This was deliberate, since Forge integers overflow. The report prompted a review of the topic.
- Questions were raised about whether a *solution* state predicate needed to use the location of the flashlight at all, given that the win condition was entirely in terms of people crossing. This was a good critique of our wheat, which was arguably over-constrained.

Interpretation. While wheats sometimes reinforce core concepts, they can also put artificial restrictions on students. Because a major goal of these assignments is giving students practice with breaking down and specifying properties, providing an exact breakdown in the handout would be self-defeating. Yet, casting the wheat as a *unique* correct solution is often not appropriate—as Wrenn [2022] notes for programming. It is also vital that a wheat not *over*-constrain. One way to address these issues is through trial-and-error debugging. Developing systematic methods is an important topic for future work.

7.3.3 RQ3: How did students write tests? Figure 14 reports *how many groups* used each testing construct and *how many tests they wrote* with each construct on RIVER, MEMORY, the midterm (MID), and the final (FIN). On the RIVER and MEMORY assignments, students were required to write some tests using assertions. They went beyond the requirements to write more examples, but generally did the minimum with all other constructs. RIVER and MEMORY required students to write their tests in a separate file; fig. 14 includes only these tests to avoid double-counting or misjudging student intent. Although students were told to do some testing on the projects, we did not mandate

	# projects using constructs				avg(median) uses of constructs			
	MID	FIN	RIVER	MEMORY	MID	FIN	RIVER	MEMORY
total # submissions	32	26	62	59				
w/ `test suite`	19	20	N/A	N/A	N/A	N/A	N/A	N/A
w/ `test expect`	19	20	1	84	5 (6)	14 (8)	0 (0)	4 (4)
w/ `example`	30	7	62	55	21 (18)	21 (15)	19 (13)	8 (6)
w/ `necessary`	2	2	6	14	2 (1)	1 (1)	8 (7)	5 (5)
w/ `sufficient`	0	0	4	15	1 (1)	0 (0)	8 (7)	5 (5)

Fig. 14. 2023 Forge testing by assignment. Because students were given a stencil for RIVER and MEMORY that contained suite blocks for each predicate, those assignments cannot be used to measure test-suite adoption.

use of `example`, `test expect`, `test suite`, etc. Most groups used `test expect` extensively. The `example` construct was common on the midterm but not the final.

Interpretation. Students used examples heavily on RIVER and MEMORY; RIVER prompted them to do so. Students tended to favor `test expect` far more on their midterm and final projects; in the latter case, it may be because `example` was unsupported in Temporal Forge—yet more than half of groups used `test expect` on the midterm, before Temporal Forge was introduced. There is clear value to the flexible form, especially when models are complex and examples are onerous to write. With a better interface (section 9), we expect more examples. In contrast, students did not use `necessary` and `sufficient` assertions on their projects; those forms were clearly unwieldy.

8 RELATED WORK

Forge is based on Alloy 6 and the Pardinus [Macedo et al. 2022] engine. The temporal features in Alloy 6 were first developed in the Electrum tool [Macedo et al. 2016]. Forge differentiates itself from these though its support for language levels (section 3), custom visualizations (section 4), and broader testing constructs (section 5). Sterling [Dyer 2020] is the foundation for Forge's visualizations, but with extensions prompted by our formative experiment in 2022 (section 7.2).

Language levels are not new, especially for teaching [du Boulay et al. 1999; Felleisen et al. 2001; Findler et al. 2002; Gilsing et al. 2022; Holt and Wortman 1974; Homer et al. 2014]. Likewise, others have integrated Racket and solvers—e.g., Rosette [Torlak and Bodik 2014], a tool for building solver-aided languages. Andersen et al. [2020] extend Racket with live, domain-specific IDEs. These efforts are either unrelated to FM, neglect issues like visualization, or do not focus on pedagogy.

Many others have worked on visualization in Alloy. A representative sample of this work includes magic layout [Rayside et al. 2007], the visualizer for DynAlloy [Bendersky et al. 2013; Regis et al. 2017], and the Web-based visualizer in Alloy4Fun [Macedo et al. 2021]. Couto et al. [2018] propose a broader range of potential layout options for Alloy, including grids and linear orderings. None of these works enable a truly *custom* visualization like those seen in section 4. Only Alloy4Fun is based on data from a live course.

The full Forge system most closely resembles the excellent yet little known and, we would argue, under-recognized GUPU pedagogic Prolog system [Neumerkel and Kral 2002; Neumerkel et al. 1997]. GUPU presents students with a fragment of Prolog and modified syntax that were built with teaching in mind, along with testing capabilities and "viewers" that translate answer substitutions to graphical form. GUPU even allows students to query a hidden solution and gives tailored feedback based on their tests. The main difference lies in the language: GUPU focuses on *one* sublanguage for declarative *programming*, whereas Forge gives a *progression* of languages for lightweight *modeling* atop a platform that enables domain-specific modeling languages. Another

interesting difference is that in GUPU, viewers are themselves written in Prolog, whereas in Forge, they are written in JavaScript (section 9). It is unclear which approach is better, although some of our students noted JavaScript as a barrier (section 7.2.4).

AT(P) [Beierle et al. 2004a] provides a rich set of testing constructs for Prolog, as well as automated assessment and feedback on preliminary tests. AT(P) includes analogues to `example`, `necessary`, etc. but also forms that Forge does not currently support, such as solution counting. AT(P) does not provide a progression of languages or custom visualization, although the underlying framework [Beierle et al. 2003, 2004b], can support other languages (e.g., Scheme).

AUnit [Sullivan et al. 2018] extends Alloy with structured testing and related features such as fault localization [Wang et al. 2020] and automated repair [Wang et al. 2018]. AUnit separates tests from other commands and supports examples, resolving some of our critiques in section 5. However, AUnit does not enable running tests versus hidden known-good models, which means it is not an immediate fit for our needs.

9 DISCUSSION

We close with a few general observations and a list of planned future directions for Forge.

Key Takeaways. From a linguistic point of view, we believe there are two key principles to take away from this work:

(1) In general, decompose your complex language into a sequence of increasingly sophisticated layers. We believe this principle can apply to many languages used in formal methods, which have a variety of complex operators, clever shorthands, and so on. These features are valuable to experts but can be confusing to beginners.
(2) In the context of relational modeling languages such as Alloy, consider starting with functional specifications and delaying relational (and temporal) operators until students have basic skills and are ready for problems that motivate these operators.

From a presentation perspective, we believe the evidence shown for custom visualizations—in terms of highlighting salient information (figs. 5 and 7), reducing clutter (figs. 5 and 7), and avoiding outright confusion that might almost be considered misinformation (fig. 6)—is very compelling. Moreover, because Sterling is not just Forge-centric (it speaks the same protocol as Alloy and was originally created for Alloy), visualizations created in Sterling are, in principle, usable for instances produced by other tools.

Finally, we believe there is a useful progression from testing to formal methods. This has been discussed more narrowly in prior work, e.g., for property-based testing [Wrenn et al. 2021]. We believe we have offered some useful guidelines about how to proceed here, and also linked the topic to other work in computing education. However, some care is needed. In regard to testing, this paper is a preliminary effort; a careful treatment is a topic for future work and course offerings.

Students and Telemetry. Forge includes an optional telemetry system that collects snapshots of every file submitted to the language for feedback. To opt in, students must replace a default string at the top of their assignment with an anonymous ID. In principle, these IDs let us build a timeline of file snapshots and discover, e.g., how students reacted to a particular error message. However, we have seen increasing reluctance toward logging among both students and course staff. This may be a local issue or part of a larger trend; we merely note it. Students were certainly not shy about leaving (sometimes extensive, often useful) feedback in anonymous surveys.

Towards a Testing Recipe. Students struggle with testing their models early in the development process. Even if the tool provides testing features, a methodology is needed. In the programming context, works like *How to Design Programs* [Felleisen et al. 2001] equip students with a *design*

recipe so they can proceed systematically toward correct solutions with minimal feedback from course staff. We envision a testing recipe for student models that begins with **`example`**s of how each predicate should behave, and eventually evolves into **`assert`**s that partially specify a correct solution. In 2023, we taught a variation of this recipe to students, but they did not use it—perhaps because the methodology was tied to running against hidden correct solutions (section 5) that were also involved in the grading process. We suspect that a recipe will help students structure their modeling but further work on separating the essential ideas is needed.

Instance Examples and Comprehension. Currently, the only way to write examples in Forge is through the coding interface. Students must painstakingly describe object graphs textually (fig. 8) A projectional editing interface (e.g.,[Andersen et al. 2020; Chugh et al. 2016; Hempel et al. 2018]) or a graphical interface akin to the one Crucible [Emerson and Sullivan 2023] provides would eliminate typo-level errors and give students more time to focus on quality tests. Relatedly, since the instances students receive when a test fails (section 5.2) might be large, it would be useful to categorize the failure, perhaps by leveraging the auto-grading suite for the assignment, and provide hinting in natural language alongside the instance.

Domain-Specific Modeling Languages. As mentioned in section 6, one can create *domain-specific modeling languages* that compile into Forge's core language. By way of example, Forge supports a custom language for writing cryptographic protocols, inspired by the input language of the CPSA [Doghmi et al. 2007] protocol-analysis tool, and was used to generate the example shown in section 4.1. While many domains do already have powerful formal tools (e.g., [Blanchet 2016; Doghmi et al. 2007] for protocols), Forge can provide simplified forms of their languages to give students a quick taste of what FM can do. We believe there is great value to creating these kinds of domain-specific modeling languages, where students are introduced to FM through the use of domains that they understand well or care about (whether cryptography or river puzzles or baseball). Combining domain-specific modeling languages with domain-specific visualizations would create an epistemically-closed universe where students can focus on the domain alone, and the general-purpose nature of Forge hides until needed (and never appears, in some settings).

Other Future Work. Development on Forge is ongoing. Our evaluation points to numerous areas for improvement, some of which entail a bit of engineering, and others that are research topics in their own right. For example, students objected to mixing singletons and sets in Relational Forge (section 7.1); what would the consequences be of enforcing further restriction on comparison? Students found Temporal Forge convenient for stateful models; given that relations and temporal operators seem independent, why not offer a Temporal Froglet language level? Priming expressions confused students; what sorts of static checks are possible? Can the lessons from section 7.2 and systems like GUPU [Neumerkel et al. 1997] lead to a Sterling where students do not need learn JavaScript to write custom visualizations? What is the pedagogic relationship (if any) between language levels and the theories provided by SMT-based tools? Finally, and more generally, we believe that learning modeling, how to precisely express invariants, etc. are valuable for building engineering skills outside of formal methods. However, we have no rigorous evidence for this. A longitudinal study of students in advanced computer-science courses could be informative.

Our ultimate goal is not to fragment the Alloy community. Rather, we hope that insights from our pedagogic context are useful in helping improve both tools. We also plan to incorporate pedagogically-useful additions to Alloy as they appear; for instance, records [Brunel et al. 2023] could be worthwhile in our setting.

DATA-AVAILABILITY STATEMENT

The Forge tool is Open Source and hosted on a public repository (section 1). This paper has an artifact that includes Forge, its documentation, the survey instruments, and all necessary code to support the modelling and visualization examples in the paper [Nelson et al. 2024].

ACKNOWLEDGMENTS

We are grateful to the Alloy and Electrum teams for creating excellent tooling we could build on. Our thinking on gradual FM was profoundly influenced by Daniel Jackson, whom we also thank for numerous thought-provoking conversations. Pamela Zave contributed many useful discussions, including of visualization challenges. Shriram Krishnamurthi thanks Markus Triska for pointing him to GUPU.

We owe a debt to the many students over the past decade who have taken our courses in Alloy and Forge, especially to recent students for their helpful feedback and survey contributions. All project images were used with permission; for this we thank especially Thomas Castleman and Ria Rajesh (Rust); Noah Atanda, Madison Lease, and Priyanka Solanky (Uno); Michael Donoso, Haley Flores, and Yishan Liu (Spikeball); Nick Bottone, Sebastien Jean-Pierre, and Robert Murray (Triple Triad); and Matthew Boranian and Austin Lang (Texas Hold'em). We deeply appreciate other work that students such as Lucy Reyes, Abigail Siegel, Mark Lavrentyev, **[[FILL: ANY OTHERS? (and)]]** did with Forge while they were at Brown. Anna Ohrt especially worked on enhancements to Forge's visualizer that will be deployed in the Spring of 2024, but not did feature in the tool evaluated here.

We are grateful for support from the US NSF through grants SHF-2227863, SaTC-2208731, and CCF-2030859. Ben Greenman thanks the CRA for support through the CIFellows project.

REFERENCES

alloytools.org. 2023. Alloy Analyzer Downloads. https://alloytools.org/download.html. Accessed August 27, 2023.

alloytools.org. 2023a. Case study applications of Alloy. http://alloytools.org/citations/case-studies.html. Accessed August 1, 2023.

alloytools.org. 2023b. Courses using Alloy. http://alloytools.org/citations/courses.html. Accessed August 1, 2023.

alloytools.org. 2023c. Translation into Alloy. http://alloytools.org/citations/language-translations.html. Accessed August 1, 2023.

Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *PACMPL* 4, OOPSLA (2020), 222:1–222:28. https://doi.org/10.1145/3428290

Christoph Beierle, Marjaa Kulaa, and Manfred Widera. 2003. Automatic Analysis of Programming Assignments. In *DeLFI*. GI, 144–153. https://dl.gi.de/handle/20.500.12116/15081

Christoph Beierle, Marija Kulaš, and Manfred Widera. 2004a. Partial Specifications of Program Properties. In *TeachLP*. 17 pages. https://ep.liu.se/ecp/012/ecp04012.pdf. Accessed 2024-01-24.

Christoph Beierle, Marija Kulaš, and Manfred Widera. 2004b. A Pragmatic Approach to Pre-Testing Prolog Programs. In *INAP and WLP*. Springer-Verlag, 294–308. https://doi.org/10.1007/11415763_20

Pablo Bendersky, Juan Pablo Galeotti, and Diego Garbervetsky. 2013. The DynAlloy Visualizer. In *Latin American Workshop on Formal Methods*, Vol. 139. 59–64. https://doi.org/10.4204/EPTCS.139.6

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS*. Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security* 1, 1–2 (2016), 1–135. https://doi.org/10.1561/3300000004

Edwin C. Brady. 2013. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. 2023. Adding Records to Alloy. In *ABZ*. Springer, 212–219. https://doi.org/10.1007/978-3-031-33163-3_16

Carlos Caleiro, Luca Viganò, and David Basin. 2006. On the Semantics of Alice&Bob Specifications of Security Protocols. *Theoretical Computer Science* 367, 1–2 (2006), 88–122. https://doi.org/10.1016/J.TCS.2006.08.041

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *PLDI*. ACM, 341–354. https://doi.org/10.1145/2908080.2908103

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. ACM, 268–279. https://doi.org/10.1145/357766.351266

Rui Couto, José Creissac Campos, Nuno Macedo, and Alcino Cunha. 2018. Improving the Visualization of Alloy Instances. In *F-IDE@FLoC*, Vol. 284. 37–52. https://doi.org/10.4204/EPTCS.284.4

Leonardo de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular Verification of Code with SAT. In *ISSTA*. ACM, 109–120. https://doi.org/10.1145/1146238.1146251

Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Koli Calling*. ACM, 11:1–11:10. https://doi.org/10.1145/3364510.3366170

Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. 2007. Searching for Shapes in Cryptographic Protocols. In *TACAS*. Springer, 523–537. https://doi.org/10.1007/978-3-540-71209-1_41

Danny Dolev and Andrew Chi-Chih Yao. 1983. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–207. https://doi.org/10.1109/TIT.1983.1056650

Benedict du Boulay, Tim O'Shea, and John Monk. 1999. The Black Box Inside the Glass Box. *International Journal of Human-Computer Studies* 51, 2 (1999), 265–277. https://doi.org/10.1006/IJHC.1981.0309

Andrew Tristan Dyer. 2020. *Lightweight Formal Methods in Scientific Computing*. Ph. D. Dissertation. North Carolina State University. https://repository.lib.ncsu.edu/items/95554653-06ab-4a76-adbd-846f2c9b995f. Accessed 2024-01-24.

Tristan Dyer and John Baugh. 2021. Sterling: A Web-Based Visualizer for Relational Modeling Languages. In *ABZ*. Springer, 99–104. https://doi.org/10.1007/978-3-030-77543-8_7

Jonathan Edwards, Daniel Jackson, and Emina Torlak. 2004. A Type System for Object Models. In *FSE*. ACM, 189–199. https://doi.org/10.1145/1029894.1029921

Adam G. Emerson and Allison Sullivan. 2023. Crucible: Graphical Test Cases for Alloy Models. In *ISSRE*. IEEE, 218–227. https://doi.org/10.1109/ISSRE59848.2023.00065

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press. http://www.htdp.org/

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Communications of the ACM* 61, 3 (2018), 62–71. https://doi.org/10.1145/3127323

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *SNAPL*. Schloss Dagstuhl, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming* 12, 2 (2002), 159–182. https://doi.org/10.1017/S0956796801004208

Marleen Gilsing, Jesús Pelay, and Felienne Hermans. 2022. Design, Implementation and Evaluation of the Hedy Programming Language. *Journal of Computer Languages* 73, 101158 (2022), 17 pages. https://doi.org/10.1016/J.COLA.2022.101158

Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. 2023. Little Tricky Logic: Misconceptions in the Understanding of LTL. *Programming* 7, 2 (2023), 7:1–7:37. https://doi.org/10.22152/programming-journal.org/2023/7/7

Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *ICSE*. ACM, 654–664. https://doi.org/10.1145/3180155.3180165

Richard C. Holt and David B. Wortman. 1974. A Sequence of Structured Subsets of PL/I. In *SIGCSE*. ACM, 129–132. https://doi.org/10.1145/800183.810456

Gerard J. Holzmann. 2003. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley. https://doi.org/10.5555/2029108

Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. 2014. Graceful Dialects. In *ECOOP*. Springer, 131–156. https://doi.org/10.1007/978-3-662-44202-9_6

Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 11, 2 (2002), 256–290. https://doi.org/10.1145/505145.505149

Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis* (2 ed.). MIT Press. https://doi.org/10.5555/2141100

Daniel Jackson. 2019. Alloy: A Language and Tool for Exploring Software Designs. *Communications of the ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

Daniel Jackson and Jeanette Wing. 1996. Lightweight Formal Methods. *IEEE Computer* (1996). https://people.csail.mit.edu/dnj/publications/ieee96-roundtable.html. Accessed 2024-01-24.

William Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, Amy Ko, Christopher J. Mendez, and Alannah Oleson. 2017. General principles for a Generalized Idea Garden. *Journal of Visual Languages and Computing* 39 (2017), 51–65. https://doi.org/10.1016/j.jvlc.2017.04.005

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Nuno Macedo, Julien Brunel, David Chemouil, and Alcino Cunha. 2022. Pardinus: A Temporal Relational Model Finder. *Journal of Automated Reasoning* 66, 4 (2022), 861–904. https://doi.org/10.1007/s10817-022-09642-2

Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. 2016. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *FSE*. ACM, 373–383. https://doi.org/10.1145/2950290.2950318

Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel Sozinho Ramalho, and Daniel Castro Silva. 2021. Experiences on Teaching Alloy with an Automated Assessment Platform. *Science of Computer Programming* 211, 102690 (2021), 21 pages. https://doi.org/10.1016/j.scico.2021.102690

Jay McCarthy and Shriram Krishnamurthi. 2008. Cryptographic Protocol Explication and End-Point Projection. In *ESORICS*. Springer, 533–547. https://doi.org/10.1007/978-3-540-88313-5_34

Vajih Montaghami and Derek Rayside. 2012. Extending Alloy with Partial Instances. In *ABZ*. Springer, 122–135. https://doi.org/10.1007/978-3-642-30885-7_9

Tim Nelson, Ben Greenman, Siddhartha Prasad, Tristan Dyer, Ethan Bove, Qianfan Chen, Charles Cutting, Thomas Del Vecchio, Sidney LeVine, Julianne Rudner, Benjamin Ryjikov, Alexander Varga, Andrew Wagner, Luke West, and Shriram Krishnamurthi. 2024. Artifact for Forge: A Tool and Language for Teaching Formal Methods. https://doi.org/10.5281/zenodo.10463960

Ulrich Neumerkel and Stefan Kral. 2002. Declarative Program Development in Prolog with GUPU. In *WLPE*. 77–86. https://arxiv.org/abs/cs/0207044

Ulrich Neumerkel, Christoph Rettig, and Christian Schallart. 1997. Visualizing Solutions with Viewers. In *LPE*. 43–50. https://www.complang.tuwien.ac.at/ulrich/papers/PDF/wlpe97.pdf. Accessed 2024-01-24.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag. https://doi.org/10.1007/3-540-45949-9

Observable. 2023. D3: Data-Driven Documents. https://d3js.org. Accessed August 25, 2023.

Derek Rayside, Felix Sheng-Ho Chang, Greg Dennis, Robert Seater, and Daniel Jackson. 2007. Automatic Visualization of Relational Logic Models. *Electronic Communications of the EASST* 7 (2007), 15 pages. https://doi.org/10.14279/tuj.eceasst.

7.94

Germán Regis, César Cornejo, Simón Gutiérrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti, and Marcelo Frias. 2017. DynAlloy Analyzer: A Tool for the Specification and Analysis of Alloy Models with Dynamic Behaviour. In *FSE*. ACM, 969–973. https://doi.org/10.1145/3106237.3122826

Sam Saarinen. 2021. *Query Strategies for Directed Graphical Models and their Application to Adaptive Testing*. Ph. D. Dissertation. Brown University. https://repository.library.brown.edu/studio/item/bdr:kgyft3b4/

Abigail Siegel, Mia Santomauro, Tristan Dyer, Tim Nelson, and Shriram Krishnamurthi. 2021. Prototyping Formal Methods Tools: A Protocol Analysis Case Study. In *Protocols, Strands, and Logic*. Springer, 394–413. https://doi.org/10.1007/978-3-030-91631-2_22

J. Michael Spivey. 1992. *Z Notation — A Reference Manual* (2nd ed.). Prentice Hall.

Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2018. AUnit: A Test Automation Tool for Alloy. In *ICST*. IEEE, 398–403. https://doi.org/10.1109/ICST.2018.00047

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *PLDI*. ACM, 530–541. https://doi.org/10.1145/2594291.2594340

Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: a User Study. In *DLS*. ACM, 1–12. https://doi.org/10.1145/3276945.3276947

Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated Model Repair for Alloy. In *ASE*. ACM, 577–588. https://doi.org/10.1145/3238147.3238162

Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2020. Fault Localization for Declarative Models in Alloy. In *ISSRE*. IEEE, 391–402. https://doi.org/10.1109/ISSRE5003.2020.00044

Aaron Wilson, Margaret Burnett, Laura Beckwith, Orion Granatir, Ledah Casburn, Curtis Cook, Mike Durham, and Gregg Rothermel. 2003. Harnessing Curiosity to Increase Correctness in End-User Programming. In *CHI*. ACM, 305–312. https://doi.org/10.1145/642611.642665

John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *ICER*. ACM, 51–59. https://doi.org/10.1145/3230977.3230999

John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *Programming* 5, 2 (2021), 9. https://doi.org/10.22152/programming-journal.org/2021/5/9

John Sinclair Wrenn. 2022. *Executable Examples: Empowering Students to Hone Their Problem Comprehension*. Ph. D. Dissertation. Brown University. https://cs.brown.edu/research/pubs/theses/phd/2022/wrenn.john.pdf. Accessed 2024-01-24.

Pamela Zave. 2020. Discourse Forum Reply. https://alloytools.discourse.group/t/visualization-for-alloy-what-do-you-want/111/2. Accessed August 22, 2023.