# Towards Better Formal Methods Visualizations

YILIANG LIANG, Carnegie Mellon University, USA
AVINASH PALLIYIL, Georgia Institute of Technology, USA
EUNSUK KANG, Carnegie Mellon University, USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA

Formal methods are mathematical techniques for specifying and verifying complex systems. While powerful, they can be difficult to understand, validate, and debug, which can limit their adoption. Visualizations can aid these processes. In this work, we present an interview study to understand how users of formal methods use visualizations in their workflow. The results of the interview study inspire the designs of Penlloy – a domain-specific visualizer for the Alloy modeling language – and our preliminary work on formalizing the notions of visual consistency.

Additional Key Words and Phrases: formal methods, Alloy specification language, visualizations, empirical study, interviews

**ACM Reference Format:**
Yiliang Liang, Avinash Palliyil, Eunsuk Kang, and Joshua Sunshine. 2025. Towards Better Formal Methods Visualizations. In *Proceedings of the 15th PLATEAU Workshop on Programming Languages and Human-Computer Interaction (PLATEAU '25).* 19 pages.

## 1 Introduction

Formal methods use rigorous logic to represent complex system behaviors and help stakeholders ensure system correctness and reliability [27]. Tools like TLA+ [36], Alloy [18], and P [5] are used to abstract complex systems into models[1] and verify the assertions and properties of these models [7]. For example, Zave [37][38] uses Alloy to understand the Chord distributed protocol, Cunha et al. [9] use Alloy to model and analyze an aircraft arrival manager, and Amazon [25] uses TLA+ to analyze "large complex real-world systems," sometimes "preventing subtle serious bugs from reaching production."

Despite their perceived power, practitioners have struggled to apply formal methods in industry: Formal models are hard to learn [32], understand [28], construct [20], and validate [15]. If stakeholders cannot learn and construct formal models, they will not be used. If they cannot understand and validate formal models, they will either not trust the assertions made by the formal methods tools, or blindly trust the results of these tools applied on an incorrect model. With the prospect of powerful program synthesizers (e.g., Spectra [23]) that generate executable artifacts from logical specifications, untrustworthy and potentially incorrect specifications can lead to the same in generated programs, which can have drastic consequences.

The difficulty of adopting formal methods often stems from their abstract nature and the use of technical notation, which can obscure their underlying meanings. Visualizations can play a crucial

[1]We use "formal models" to describe all formal artifacts such as models, specifications, and proofs.

Authors' Contact Information: Yiliang Liang, Carnegie Mellon University, Pittsburgh, PA, USA; Avinash Palliyil, Georgia Institute of Technology, Atlanta, GA, USA; Eunsuk Kang, Carnegie Mellon University, Pittsburgh, PA, USA; Joshua Sunshine, Carnegie Mellon University, Pittsburgh, PA, USA.

role in bridging this gap by providing intuitive and concrete representations that make abstract concepts more tangible [21], empowering stakeholders to engage more effectively and confidently with formal methods. In this work, we explore the role of visualizations in helping practitioners understand, validate, and debug formal models, and seek to improve formal methods visualizations through domain specificity and visual consistency. In summary, our contributions are as follows.

- We conducted an **interview study** (Sec. 3) to understand what types of visualizations are adopted (Sec. 4.1), how they are helpful (Sec. 4.2), and how they are not (Sec. 4.3);
- We designed **Penlloy, a domain-specific visualizer for the Alloy modeling tool**, improving the status quo of currently-available visualizations (Sec. 5); and
- We **formalized "visual consistency"** as a general principle for visualizing formal models, implemented under the Penlloy framework (Sec. 6).

## 2 Background and Related Work

### 2.1 Visualizing formal methods

The use of visualizations to enhance cognition of complex concepts for problem-solving has been well studied. Classically, Larkin and Simon's seminal paper [21] distinguishes between "sentential [textual] representations" and "diagrammatic [visual] representations," concluding that while informationally equivalent, because visualizations can support more efficient information search operators, "a diagram can be superior to a verbal [textual] description."

Visual formalisms like Statecharts [16] have been developed to represent complex systems. More recent and widely-used formal methods tools also have visualization capabilities, such as P [5] (through an extension Peasy [2]), B through VisB [34], and Alloy [18] through its built-in visualizer.

Within the Alloy community, Couto et al. [8] developed layout and transition managers to improve Alloy's built-in visualization. Lightning [14] allows Alloy model authors to encode "visual models" directly within Alloy, and have Alloy solve for a suitable visualization, while Sterling [12] allows Alloy users to encode visualizations in JavaScript using the D3 data visualization engine. Additionally, the Cope and Drag (CnD) tool [29] allows users to provide a light-weight and programmatic visual specification; CnD then checks whether or not the visual specification is satisfiable, and solves for an output diagram.

Our approach of domain-specific visualizations through Penlloy is unique in its use of the conceptual visualizer Penrose [35], which supports richer and more sophisticated domain-specific visualizations by directly and declaratively reasoning about geometric relationships between shapes. Users of Penlloy can specify translations from outputs of formal methods tools into sophisticated geometric constraints, enabling the system to automatically generate layouts that satisfy these constraints. This is particularly beneficial because geometric relationships between shapes are semantically meaningful and can often be translated from outputs of formal methods tools. Since there may be multiple ways to satisfy the same set of constraints, this approach also allows users to explore multiple valid visualizations of the same concepts.

### 2.2 Empirical evaluations of formal methods tools and visualizations

Advancements in formal methods have significantly expanded their capabilities and applications, but understanding their practicality requires thorough empirical evaluation. For example, Mansoor et al. [22] performed a user study to compare how novice and expert users construct and repair Alloy models. Nelson et al. [24] evaluate how students of formal methods introduce and use their own domain-specific visualizations in their models through Sterling [12], and gather retrospective opinions on the benefits and limitations of these visualizations. On the other hand, throughout our interview study, we try to systematically extract some "principles" that make good visualizations for experts in formal methods, inspiring our work on improving current visualizations and evaluating the effects of the improvements.

## 3 Interview Design

We wanted to understand, for experienced practitioners of formal methods, why visualizations are used, how they help in their work, and how they may be improved. To that end, we conducted a pre-screened semi-structured interview study with experts of formal methods. The research questions are as follows:

- **RQ1**: What kinds of visualizations do practitioners adopt?
- **RQ2**: How do practitioners use visualizations, and how are they helpful to practitioners' work?
- **RQ3**: How can visualizations be not helpful?

### 3.1 Participant recruitment and pre-screening survey

Potential participants were collected from author lists of formal methods papers, attendees of relevant workshops, and personal connections. All of these potential participants are expert users of formal methods and may require visualizations in their workflows. We rewarded each participant who completes the interview study with a gift card of value USD 25.

Each potential participant was asked to fill out a pre-screening survey about their experiences with formal methods, their experiences with producing and consuming visualizations for formal methods, and their willingness to share their formal models and visualizations throughout the interview. The pre-screening survey allowed us to prioritize participants who were willing to share with us their models and visualizations, and tailor the interviews to the participants' specific use cases.

### 3.2 Semi-structured interviews

We conducted interviews with 14 expert users of formal methods tools who met our screening criteria – have either produced or used visualizations during their work and were willing to share their visualizations throughout the interview. The interview participants come from both academia and industry, and use various formal methods tools. See Table 1 for the distribution of tools that they use, and Table 2 for descriptions of some of the exemplar participants.

| **Tools** | **Participants** |
|---|---|
| TLA+ | P4, P8, P9 |
| Alloy[2] | P2, P4, P6, P7, P8, P9, P10, P12 |
| Spin | P7, P8 |
| Lean | P13 |
| Coq | P1 |
| B-Method | P5, P14 |
| P | P4 |
| TSL synthesis[3] | P3, P11 |

Table 1. Formal methods tools and interview participants who use them.

[2]includes variants like Forge [24] and Dash+ [17].
[3]synthesis of state machines from temporal stream logic (TSL) specifications [13].

| Participant | Description |
|---|---|
| P3 | Uses **temporal-stream logic** to specify and synthesize state machines in **academic** context. |
| P4 | Uses **P** to model industrial control systems with concurrent state machines in **academic** context. |
| P6 | Uses **Alloy** to model and verify railway control systems in **industrial** context at a multinational technology conglomerate. |
| P7 | Uses **Spin and Alloy** to model, verify, and repair distributed network protocols in **both industrial and academic** contexts. |

Table 2. Descriptions of exemplar participants of the interview study.

Each interview lasted between 30 and 60 minutes and was conducted online. For RQ1, we encouraged the participants to show us (via screen sharing) their formal models, as well as any visualizations that they created or adopted alongside the formal models. We asked participants to "walk through" the visualizations, such as how they are interpreted. For RQ2, we asked questions like "for what purposes do you use this visualization" and "which parts of this diagram help you achieve these goals." For RQ3, we asked participants to elaborate on the challenges of using visualizations, as well as the types of visualizations that they believe may benefit their work. We also asked participants to brainstorm the characteristics of a "good visualization."

### 3.3 Analysis

The interviews were recorded with extensive notes taken throughout. For RQ1, we collected a gallery of visualizations that the interviewees shared with us and categorized them according to the types of information they convey. For RQ2 and RQ3, we used thematic analysis to code the transcripts.

## 4 Interview Results

### 4.1 RQ1: What kinds of visualizations are adopted?

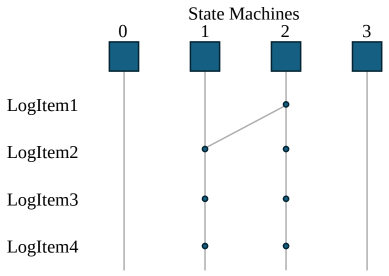
We observed a **correlation between the "semantics" of the visualizations and the "semantics" adopted by the underlying formal methods tool**. Here, "semantics" refer to the types of information conveyed. We collect some mockups of the categories of visualizations in Fig. 1.

Some formal methods tools adopt very specific semantics. For example, P [5] models concurrent message-based state machines, and TSL [13] allows users to synthesize state machines from temporal-stream logic specifications. Both tools reason about state machines, so users of these tools naturally adopt canonical visualizations of states, transitions, and traces, such as in Figures 1a and 1b.

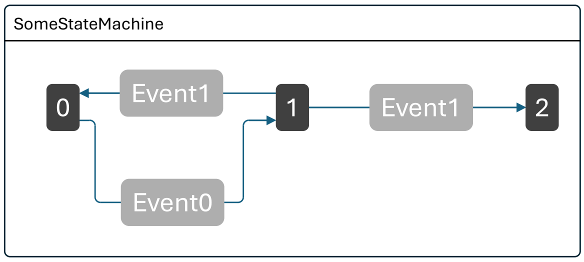
In contrast, more general-purpose tools like Alloy and TLA+ offer significant flexibility. Users specify a model as entities and relationships between entities. The model can be about anything – the tools are agnostic to the underlying natures of the models. So, visualizations often use generic "nodes-and-edges" representations (entities become nodes, and relationships become edges) to reflect the relational nature of the underlying models, such as in Fig. 1c.

### 4.2 RQ2: How are visualizations used and how are they useful?

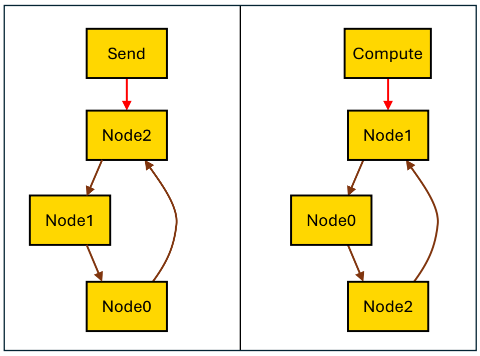
Almost all of the participants found visualizations useful in their workflow. One participant, who used Alloy to model railway systems, even exclaimed that "*without visualizations, Alloy is useless and I would have gone with a different tool*" (P6). The participants revealed three ways they use



(a) Peasy visualization [2] of an execution of concurrent state machines and messages passed between them, shared by P4.



(b) Stately's XState visualization [4] for a state machine synthesized from TSL specifications, shared by P3 and P11.



(c) The built-in visualization of Alloy for a formal model specifying the "leader election" protocol, shared by P2.

Fig. 1. Some mockups of visualizations shared by interview participants. For participant confidentiality purposes we cannot use the actual screenshots taken throughout the interviews.

visualizations: (1) to understand, (2) to validate and debug, and (3) to present and explain formal models.

*4.2.1 Understanding formal models.* Practitioners use visualizations to gradually **gain insights into model behavior**, thereby developing understanding of how complex systems work. Thirteen out of fourteen participants found that in comparison to the textual description and output of the tools, the visualizations allow them to more easily understand the model behavior.

P4 used P to model concurrent state machines in cyberphysical systems, and shared with us the workflow of loading a P trace (a log of execution of the state machines) into the Peasy visualizer [2], which generates diagrams like in Fig. 1a.

> "*When the interactions between state machines are really complicated, ... I go look for the visualization. The visualization really interesting and it helps me keep track of what are the state machines that are running ... The element that I focus the most on are the connections between the state machines, which are the hardest parts in a model.*" (P4)

Practitioners also use visualizations to **discover unexpected insights** about the system being modeled, which are often the system's "most interesting" behaviors. P10 verified distributed network protocols with Alloy and used the built-in visualizations to explore valid configurations of the model:

> "*[I use visualizations to] see scenarios of executions of my system, and be confronted with instances which are possibly not envisioned.*" (P10)

*4.2.2 Validating and debugging formal models.* Visualizations allow stakeholders to quickly **validate the models' correctness**. P8 taught formal methods using Alloy, and shared with us how the nodes-and-edges representation of Alloy instances help students detect mis-specifications.

> "*Let's say that you are trying to define a cyclic directed graph. It is really easy to specify it in Alloy. And it is really easy to see if you got it wrong, because when you look at the visualization, you'll see a loop, because your brains and eyes are really really good at spotting cycles. ... As soon as we advance to undirected graphs, you can't specify it as easily, ... but the ease of visually spotting a cycle (in case of mis-specifications) remains. For debugging, when the bug manifests in this easily recognizable way, it is so nice.*" (P8)

P3, who used Stately [4] to visualize state machines synthesized from TSL specifications [13] (like in Fig. 1b), agreed with this conclusion.

> "*Visualizations act really well as some sort of sanity check for our specifications ... [For example] visualization is extremely helpful in finding sub-graphs that are closed loops. If you enter a part of that sub-graph, you are stuck there ... this tells you that maybe you need to go back and look at your specifications.*" (P3)

Visualizations also participate in the **iterative development cycle** of models:

> "*I would build [a model] and visualize it along the way. I would start with a simple structure, visualize it, and I would say 'well that looks wrong, let's fix that, let's get that how we want it, and then move on'. I do this in an incremental loop. I wouldn't just write a long model and then visualize it. I would incrementally interweave writing models and visualizing.*" (P2)

*4.2.3 Presenting and explaining formal models.* Results of formal methods can be difficult to understand for stakeholders unfamiliar with the tool or with logic in general. Visualizations help abstract away technical details, for **presenting and explaining to these non-expert stakeholders**.

> "*Presenting formal methods is hugely important. There is a large audience of people who cannot understand Alloy text, ... but these visualizations can make perfect sense to them. They could be students who don't understand [Alloy] yet, or they could be domain experts who understand [the domain] maybe better than the formal modeler does.*" (P2)

## 4.3 RQ3: How do visualizations not help?

While most practitioners used visualizations frequently, they also found visualizations sometimes frustrating to adopt. P7, who used Alloy to verify distributed protocols, admitted that they never used the built-in visualizer for Alloy in their workflow, instead opting to hand-draw all images. In general, the participants revealed two main drawbacks with their visualizations: (1) the lack of domain specificity, and (2) the visual consistency problem.

*4.3.1 Lack of domain specificity.* Many of the existing visualizations are tailored to the semantics of the underlying notations (Sec. 4.1), but many interview participants found this one-style-fits-all strategy too limiting, especially for general-purpose formal modeling tools like Alloy.

Alloy's built-in visualizer displays instances of all models with a common visual style: objects as nodes and relations as edges (e.g., Fig. 1c). Although the built-in visualizer supports some "theme" customizations like shapes and colors, the results still adhere to the nodes-and-edges representation.

Some participants who used Alloy found this default visual style challenging to use. P6, who used Alloy to model industrial railway systems, lamented,

> "*We have a [standard] way visually [to represent our domain]. We understand how a train moves and how it visually looks, and none of this [built-in visualization] matches that [standard]. For exploratory tasks, it is challenging to use the visualizer.*" (P6)

P2, who worked closely with the Alloy community, echoed this sentiment:

> "*It would be really nice to customize [the visualization] to the domain. If [the domain] is a file system, [there should be] a file icon. I'd like other modes of representing relations other than just arrows, such as ... the containment relationship between shapes.*" (P2)

Even when the canonical nodes-and-edges representation fits well, participants also lamented about the positions of the nodes. In Fig. 1c, the network nodes have an inherent ring structure, but the visualization is agnostic to this topology. Participants found this layout frustrating:

> "*Alloy's visualizations are constructed with layers, which isn't always great when you want to represent some circular structures.*" (P10)

In summary, the participants identified the need for better **domain-specific visualizations** wherein different conceptual domains may benefit from custom, customized visualization strategies. This domain specificity can minimize the "semantic distance" between "user's model of how the system works" and the model presented by the visualizations, as explained by Dulac et al. [11]. While two participants (P5, P8) created their own domain-specific visualizations through external tools like Sterling [12] and VisB [34], the others default to using built-in tools without domain specificity.

*4.3.2 The visual consistency problem.* Participants have also lamented that some of the existing visualizations lack "visual consistency" for models with multiple states (timestamps, etc.). Chiplunkar et al. [6] describe the visual consistency problem (which they dub "continuity") as: when a visually consistent diagram evolves over multiple states, it should change "in such a way that the stakeholder can easily tell what has changed, and not more than it has to." We will formalize the meaning of visual consistency in Sec. 6.

When viewing multiple states of one instance, Alloy's built-in visualizations are not consistent. Fig. 1c shows two consecutive frames (states) side-by-side from an Alloy model specifying the "leader election" protocol. On the left, the nodes are arranged in a ring, and the *Send* event acts upon *Node2*. On the right, the nodes are arranged in the same ring structure, but the *Compute* event acts upon *Node1*. Even though the topology of the nodes remain the same, their positions change (*Node2* modes from the top to the bottom, etc.), which can be confusing for stakeholders.

> "*These graphs look isomorphic, the whole thing looks like identical structure. ... But the node I was looking at before has moved down to the bottom. I don't like the fact that this structure has been rotated in a way that I have to look carefully to understand why it has been rotated. That's weird.*" (P2)
>
> "*Currently the visualizer cannot be instructed to keep the position of certain [nodes], which is sometimes an issue.*" (P10)

## 5 Penlloy, a Domain-Specific Visualizer for Alloy

The interview study revealed a desire for *domain-specific visualizations*, wherein different conceptual domains of formal models could benefit from tailored visual abstractions. We applied this insight to the development of Penlloy, a domain-specific Alloy visualizer based on Penrose [35]. We chose to integrate our visualization tool with Alloy for multiple reasons. First, Alloy's built-in visualizer gave us a precedent to build upon for our integration of domain specificity. Additionally, the flexibility of Alloy in generating a variety of different instances, coupled with its prevalent usage by our interview participants, made it a compelling choice for our first attempt at domain specificity. Our choice of Penrose was based on its versatility in creating complex visualizations, by reasoning about geometric relationships between shapes.

In this section, we describe the design and workflow of Penlloy, apply it to a couple of different Alloy models, and present some light-weight initial expert evaluations of Penlloy.

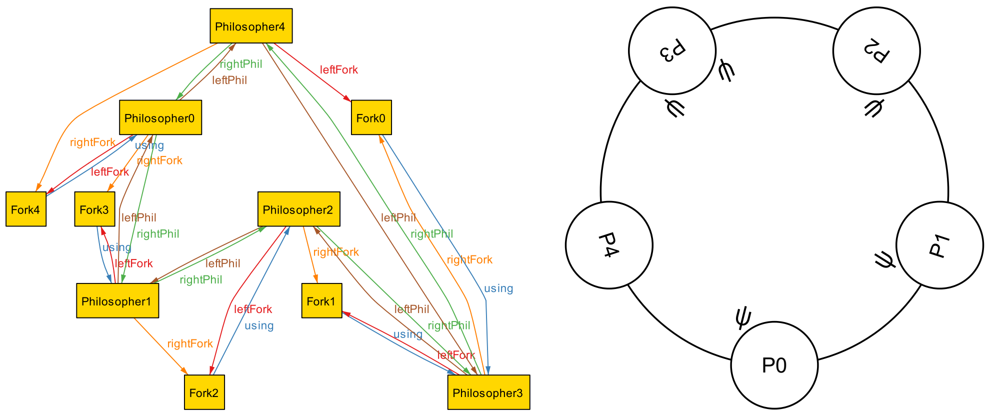
## 5.1 Tool workflow

We illustrate the workflow of Penlloy through an example of the dining philosophers problem.

*5.1.1 Motivating example: the dining philosophers problem.* The dining philosophers problem (DPP) [10] is a classical problem in the design of distributed systems. We succinctly describe the rules of DPP as follows:

- A group of philosophers sit around on a table; between every two philosophers is a fork.
- Each fork can only be held by one philosopher at once.
- A philosopher can only hold adjacent forks, and can eat only if they hold both adjacent forks.

We can encode these rules in Alloy, and ask the Alloy Analyzer to generate an instance. We use the built-in visualizer of Alloy to visualize one state of the instance (Fig. 2a), and compare it against the Penlloy visualization (Fig. 2b).

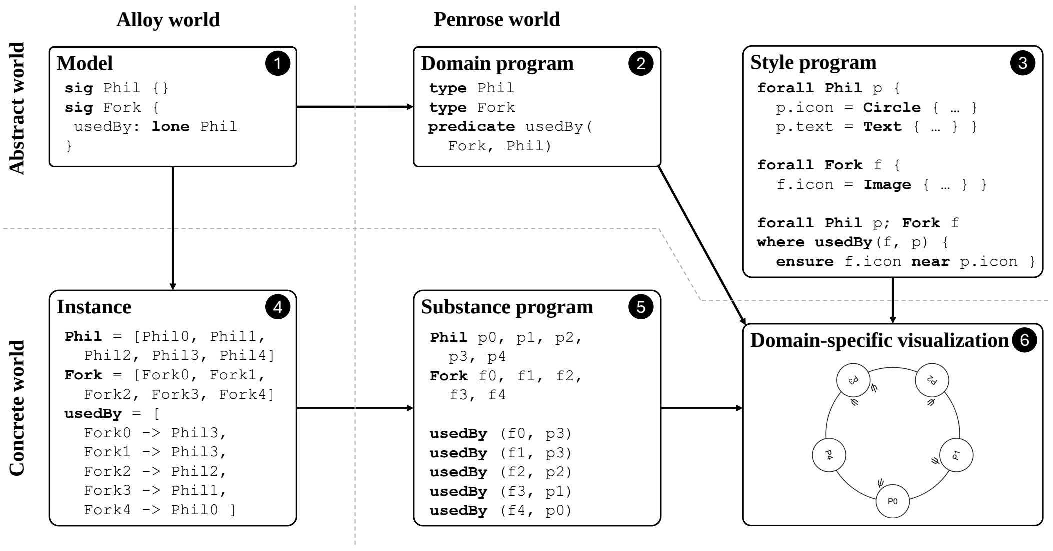


(a) The built-in visualization of Alloy (b) Penlloy

Fig. 2. A comparison of the built-in visualization of Alloy and the Penlloy visualization of a state of DPP.

In Fig. 2a, the built-in visualization of Alloy reflects the lack of domain-specificity that the interview participants have identified. The philosophers form a ring, but the built-in visualization does not reflect this topology. Relationships between the entities are expressed solely as arrows, disregarding the semantics of the relationships. On the other hand, the Penlloy visualization in Fig. 2b renders a domain-specific view of the same state. It respects the ring topology of the philosophers and the semantics of the relationships: a philosopher holding a fork translates to the fork's icon being geometrically close to the philosopher's icon.

We use the modeling problem described above to outline the design and workflow of Penlloy, illustrated in Fig. 3.



```
sig Phil {}
sig Fork {
 usedBy: lone Phil
}
```

```
type Phil
type Fork
predicate usedBy(
  Fork, Phil)
```

```
forall Phil p {
  p.icon = Circle { … }
  p.text = Text { … } }

forall Fork f {
  f.icon = Image { … } }

forall Phil p; Fork f
where usedBy(f, p) {
  ensure f.icon near p.icon }
```

```
Phil = [Phil0, Phil1,
  Phil2, Phil3, Phil4]
Fork = [Fork0, Fork1,
  Fork2, Fork3, Fork4]
usedBy = [
  Fork0 -> Phil3,
  Fork1 -> Phil3,
  Fork2 -> Phil2,
  Fork3 -> Phil1,
  Fork4 -> Phil0 ]
```

```
Phil p0, p1, p2,
  p3, p4
Fork f0, f1, f2,
  f3, f4

usedBy (f0, p3)
usedBy (f1, p3)
usedBy (f2, p2)
usedBy (f3, p1)
usedBy (f4, p0)
```

Fig. 3. Design and workflow of Penlloy.

*5.1.2 Alloy model translates into Penrose domain.* We start with the Alloy **model** of the DPP (1). For the purpose of demonstrating Penlloy, we simplify the model to extract only the most critical entities and relations: philosophers (defined as `sig Phil`), forks (`sig Fork`), and the used-by relation between forks and philosophers (`usedBy: lone Phil`). This Alloy model is an abstract representation of the system being modeled – that is, we only declare that there are philosophers, and there are forks, and that some relationship exists between them. The model does not specify which philosophers and forks are there, nor does it say anything about any forks being used by any philosophers.

Penlloy's workflow starts with translating the Alloy **model** into a Penrose **domain** program (2). This is a strightforward process: Alloy `sig`s become Penrose `type`s, and Alloy relations become Penrose `predicate`s. As expected, the **domain** program is also abstract.

*5.1.3 Penrose style program enables domain-specificity.* A Penrose **style** program enables domain-specificity by encoding how to translate elements in the **domain** program into visualizations in a domain-specific way. In this case, we provide our own **style** program (3) which encodes these rules:

- Every philosopher is rendered as a circle, with a text label.
- Every fork is rendered as an externally-sourced SVG image.
- If a fork is used by a philosopher, the shape of the fork should be near the shape of the philosopher. (Penrose allows **style** writers to specify geometric constraints and objectives on shapes, such as a shape being near another shape.)

Notice that like the Alloy **model** and the Penrose **domain**, the Penrose **style** is also abstract. The **style** program does not refer to any concrete philosophers and forks – it simply encodes rules that should triggered (e.g., the shape of the fork should be near the shape of the philosopher) *if* its condition is satisfied (e.g., *if* a fork is used by a philosopher).

Other rules such as the layout of the philosophers into a circle are omitted here for demonstration purposes. It took the authors around 15 minutes to create a suitable **style** program for this model.

*5.1.4 Alloy generates concrete instances which translate into Penrose substances.* The **model**, **domain** and **style** are all abstract descriptions of the visualization problem. To make the problem more concrete and solvable, we ask Alloy to generate a concrete **instance** (❹) of the model. The instance can either be an example instance (some valid configuration of the model, generated by the `run` command in Alloy) or a counterexample instance (configuration of the model that violates some properties, generated by the `check` command). Either ways, the instance encodes the exact philosophers (`Phil0`, ..., `Phil4`) and forks (`Fork0`, ..., `Fork4`) in the system, as well as exactly which forks are used by which philosophers (e.g., `Fork1` is used by `Phil3`).

Penlloy then takes Alloy's generated **instance** and translates each instance into a Penrose **substance** program (❺). This is a straightforward process that mostly involves line-by-line translation.

*5.1.5 Penrose takes the domain, substance, and style programs and generates a visualization.* To generate a diagram, the Penrose [35] engine requires the **domain** (❷), **style** (❸), and **substance** (❺) programs. Specifically, it exhaustively applies every rule in the **style** program to the **substance** program, compiles the applicable constraints between shapes into an optimization problem, and solves that optimization problem to give us a domain-specific diagram (❻).

*5.1.6 Penlloy visualizations are reusable.* Both Alloy **instances** and Penrose **substances** are concretizations of Alloy **models** and Penrose **domains**. Hence, this translation preserves the separation between the abstract world (**model** and **domain**) and the concrete world (**instance** and **substance**).

Since the Penrose **style** program also lives in the abstract world, Penlloy enables reusable visualizations: we could ask Alloy to generate a new **instance** of the same **model**, convert that new instance into a new **substance**, and use the same **style** program to visualize that new instance.

*5.1.7 Penlloy can explore different visual configurations of the same instance.* Recall that Penrose compiles the diagramming problem into a set of geometric constraints which it tries to satisfy. Often, there may be different valid configurations of the diagram that all satisfy the same set of constraints. Penrose allows users to explore these different configurations through a "Resample Diagram" feature, which Penlloy trivially supports as well. This is beneficial because even when two diagrams of the same instance are topologically equivalent, one visual configuration may be more informative for stakeholders than the other one.

## 5.2 Example: the River Crossing Puzzle

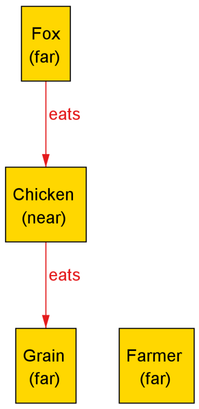
In the previous sections, we applied the Penlloy technique to the dining philosophers problem. Here, we use Penlloy visualize another model – the river-crossing puzzle [30] – to demonstrate the applicability of the Penlloy technique across different domains.

*5.2.1 Puzzle overview.* The river-crossing puzzle involves

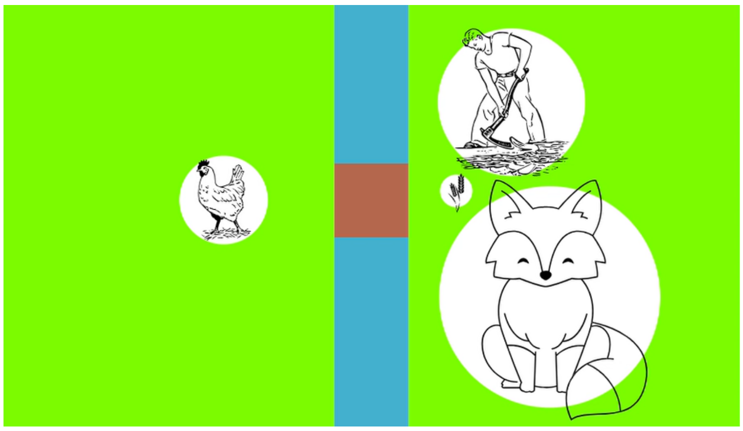
- a river with two sides, "near" and "far"; and
- four objects: a "farmer," a "fox," a "chicken," and a "grain."

Initially, all four objects are at the near side. When evolving the puzzle from one state to the next, the farmer can move through between the two sides and can optionally bring one other object (the fox, chicken, or grain) along with them. The caveat is, if left unsupervised by the farmer, then the fox would eat the chicken and the chicken would eat the grain. The goal of the puzzle is to devise a sequence of states and steps so that all four objects end up at the far side.

*5.2.2 Alloy representation of the puzzle.* To encode the above rules into a formal model in Alloy, that model would generally consist of these parts:



(a) Alloy's built-in visualization



(b) Penlloy

Fig. 4. A comparison of Alloy's built-in visualization and Penlloy's domain-specific visualization on one state of the river-crossing puzzle.

- Representations of concepts in the puzzle –
  - The farmer, the fox, the chicken, and the grain all need to be defined; and
  - A state consists of a near set of objects and a far set of objects. In the initial state, for example, the near set has the farmer, the fox, the chicken, and the grain, while the far set is empty.
- Representations of the relationship between concepts in the puzzle –
  - a relation encoding whether or not a object would eat another object; and
  - a set of rules guiding state transitions: given a current state, which properties should the next state satisfy?

This model captures the concepts and the behaviors of the underlying system (puzzle). For the concrete model code and explanation in Alloy, see the official Alloy tutorial page [3].

*5.2.3 Visualizing instances of the puzzle.* An instance of this model may consist of a sequence of states, each state encoding concrete information about which objects are on the near side and which objects are on the far side. To visualize this instance, we would visualize one state at a time. Fig. 4a shows the result of visualizing a state using Alloy's built-in visualization.

As discussed before, Alloy's built-in visualization does not exhibit domain specificity. The farmer, fox, chicken, and grain are all rendered as rectangles; most importantly, each object's property of being on the near or the far side of the river is rendered simply as attribute texts in the object's node. At a glance, it can be hard to tell exactly what is going on in the state. Consumers of this visualization needs to carefully and mentally relate the attribute words to their semantic meaning, which can be hard when done across many states of many instances.

Penlloy can give us domain-specificity in the river-crossing puzzle. See Fig. 4b for a very crude Penlloy visualization of the same state, with a **style** program (Fig. 5) that took less than 15 minutes for the authors to write. The *style* program encodes these visual rules:

- Each object is rendered as an icon mimicking the real-world object, encoded in Fig. 5a.
- Objects on the "near" side of the river being rendered on the left side of the canvas, and the "far" side on the right, encoded in Fig. 5b.

```
forall Farmer f {
    f.icon = Image {
        href: "farmer-image.svg"
    }
}
...
forall Grain g {
    g.icon = Image {
        href: "grain-image.svg"
    }
}
```

(a) Penrose **style** code that assigns each object a graphical icon.

```
forall Object o
where Near(o) {
    x = o.center[0]
    ensure x < 0
    // left side of canvas
}
forall Object o
where Far(o) {
    x = o.center[0]
    ensure x > 0
    // right side of canvas
}
```

(b) Penrose **style** code for placements of objects on the "near" and "far" sides of the river.

Fig. 5. Selections from the Penlloy **style** program that generates the domain-specific visualization for the river-crossing puzzle, in Fig. 4b.

There are additional, not-shown rules in the **style** program which, for example, sets up the scene (green grass, blue river, brown bridge) and ensure that the icons do not overlap. In general, Penlloy enables versatile domain-specific visualizations for these instances.

## 5.3 Implementation

We extended Alloy 6 with features that export the **models** and **instances** in an intermediate format[4]. We implemented the translators[5] in TypeScript from the intermediate format into **domain** and **substance**, which are hooked to a React-based web-IDE[6] (Fig. 6) that allows users to write a **style** program and generates domain-specific visualizations in real-time. We will soon provide detailed documentations on exactly how to set up and use Penlloy.

## 5.4 Initial Expert Feedback on Penlloy

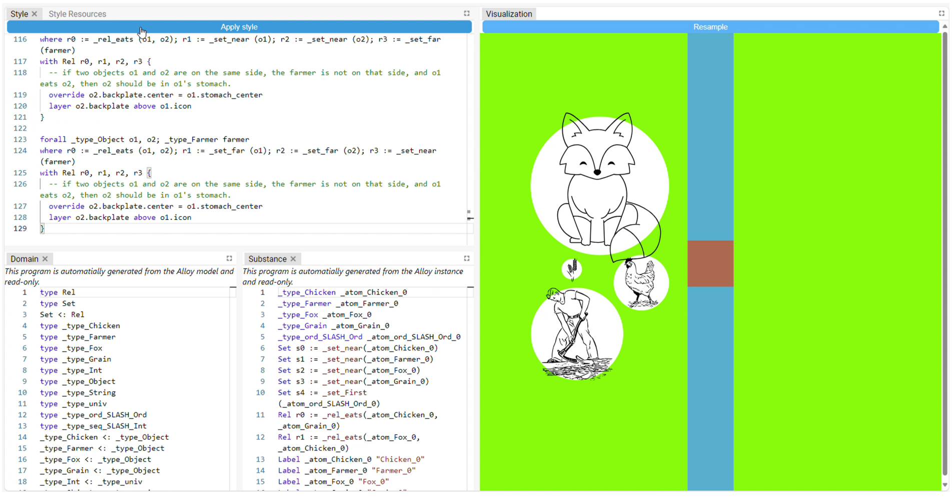
We reached out to 8 of our interview participants who used Alloy, and gave them each a demonstration of Penlloy applied on the river-crossing puzzle. We asked these formal methods experts for feedback on Penlloy-generated domain-specific visualizations.

*5.4.1 Experts find Penlloy's visualizations and customization capabilities helpful.* Most participants found Penlloy's visualizations potentially useful, especially compared to Alloy's built-in visualizations. Referring to the domain-specific visualizations of the river-crossing puzzle, one participant said,

> "*The part where an object is [rendered] left or right of the river, I think is useful and easy to understand.*" (P10)

More generally, participants liked how, compared to Alloy's built-in visualizations, Penlloy allows the full "*user, domain-expert, or human guidance*" (P8) to go into the generic **style** program.

[4]GitHub repository: https://github.com/penrose/penlloy-alloy-6
[5]GitHub repository: https://github.com/penrose/penlloy-client
[6]GitHub repository: https://github.com/penrose/penlloy-ide



```
where r0 := _rel_eats (o1, o2); r1 := _set_near (o1); r2 := _set_near (o2); r3 := _set_far
(farmer)
with Rel r0, r1, r2, r3 {
  -- if two objects o1 and o2 are on the same side, the farmer is not on that side, and o1
eats o2, then o2 should be in o1's stomach.
  override o2.backplate.center = o1.stomach_center
  layer o2.backplate above o1.icon
}

forall _type_Object o1, o2; _type_Farmer farmer
where r0 := _rel_eats (o1, o2); r1 := _set_far (o1); r2 := _set_far (o2); r3 := _set_near
(farmer)
with Rel r0, r1, r2, r3 {
  -- if two objects o1 and o2 are on the same side, the farmer is not on that side, and o1
eats o2, then o2 should be in o1's stomach.
  override o2.backplate.center = o1.stomach_center
  layer o2.backplate above o1.icon
}
```

*This program is automatially generated from the Alloy model and read-only.*

```
type Rel
type Set
Set <: Rel
type _type_Chicken
type _type_Farmer
type _type_Fox
type _type_Grain
type _type_Int
type _type_Object
type _type_String
type _type_univ
type _type_ord_SLASH_Ord
type _type_seq_SLASH_Int
_type_Chicken <: _type_Object
_type_Farmer <: _type_Object
_type_Fox <: _type_Object
_type_Grain <: _type_Object
_type_Int <: _type_univ
```

*This program is automatially generated from the Alloy instance and read-only.*

```
_type_Chicken _atom_Chicken_0
_type_Farmer _atom_Farmer_0
_type_Fox _atom_Fox_0
_type_Grain _atom_Grain_0
_type_ord_SLASH_Ord _atom_ord_SLASH_Ord_0
Set s0 := _set_near(_atom_Chicken_0)
Set s1 := _set_near(_atom_Farmer_0)
Set s2 := _set_near(_atom_Fox_0)
Set s3 := _set_near(_atom_Grain_0)
Set s4 := _set_First
(_atom_ord_SLASH_Ord_0)
Rel r0 := _rel_eats(_atom_Chicken_0,
_atom_Grain_0)
Rel r1 := _rel_eats(_atom_Fox_0,
_atom_Chicken_0)
Label _atom_Chicken_0 "Chicken_0"
Label _atom_Farmer_0 "Farmer_0"
Label _atom_Fox_0 "Fox_0"
```

Fig. 6. A screenshot of the Penlloy web-IDE that shows the automatically-generated **domain** and **substance** programs, and allows users to write and edit the **style** program. The generated visualization for the river-crossing model is on the right-hand side.

> "*I think having this kind of freedom [to customize] is great. Being able to extend [Alloy] to this kind of things [Penlloy-enabled domain-specific visualizations] is very useful.*" (P2)

*5.4.2 Experts want a balance between sophisticated and easy-to-make diagrams.* Penlloy, like other domain-specific visualization tools such as Sterling [12], requires users to manually provide a **style** program that specifies how elements of Alloy translate visually to visualizations. When shown with this **style** program, some participants feared that these custom visualizations would be too time-consuming to make, compared to the benefits they may get out from domain-specific visualizations:

> "*I think Penlloy would give me too much work. I don't need [something that Penlloy provides] because I mainly use Alloy to model things that are a bit abstract. ... I wouldn't want to have to spend a lot of time making the visualizations.*" (P10)

The same participant, however, did suggest a need to find a balance between the full customization of Penlloy and the lack of customization of Alloy's built-in visualizations:

> "*What you just showed me – it is perhaps too sophisticated to what I would want. But the current [built-in] visualizer is perhaps not sophisticated enough. I want a middle ground where I wouldn't need to enter too much information or to program ... and still get more sophistication.*" (P10)

*5.4.3 Experts warn against over-customization.* Participants warned that over-customization of visualizations can be unnecessary and even confusing. Referring to the photo-realistic icons of the farmer, fox, chicken, and grain in Fig. 4b, the participant discussed icons like those are not really necessary:

> "*For the visualizations I use, the entities can absolutely remain rectangles or ellipses. If I model a system for fire alarms, etc., I don't need ... the objects to be represented as pictures of alarms in the real world.*" (P10)

Another participant echoed this sentiment, adding that these elements can be distracting:

"*These drawings are what Edward Tufte would call 'chart junk' [31]. It's just very elaborate, very distracting. I was perfectly fine with text labels of the fox, the chicken, the grain, and the farmer. These icons are a lot of details that adds absolutely nothing to the comprehension.*" (P7)

*5.4.4 Takeaways.* These light-weight expert evaluations of Penlloy suggested:

- Experts appreciate that Penlloy enables domain-specific, customizable visualizations that go beyond Alloy's built-in capabilities.
- Experts want a balance between sophisticated and easy-to-make visualizations, which suggests the need for some ways to automatically generate the **style** program from some light-weight user inputs like a GUI panel.
- Compared to the domain-specificity of "making the entities look like real-world objects" (e.g., the fox's icon actually looks like a fox), experts are more interested in the domain-specificity through salient representations of properties and relations (e.g., the fox being on the near side of the river translates to the fox's icon being on the left side of the canvas). Therefore, the automatic generation of **style** should be more tailored to visualizing these properties and relations.

## 6 Addressing the Visual Consistency Problem

In Sec. 4.3.2, interview participants identified the lack of visual consistency as a frustrating issue with existing visualizations. This section outlines some of our early attempts to address this issue.

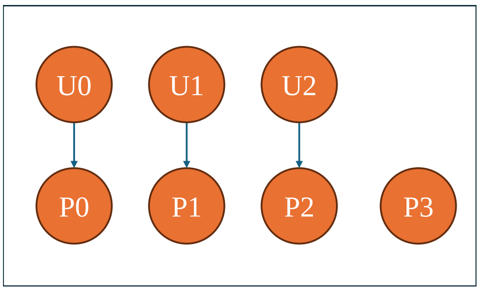
### 6.1 Categorizing visual consistency

Recall that the visual consistency problem refers to the fact that when visualizing models with multiple, evolving states, the visualizations do not maintain the positions of the entities throughout the states, thereby creating confusions.

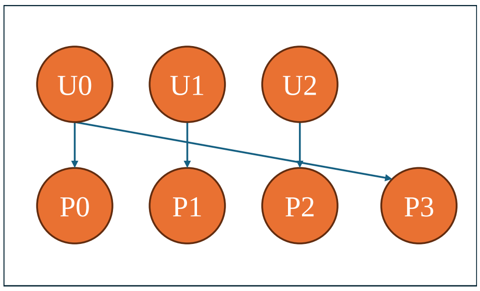
The simplest way to alleviate this problem is to enforce that the shapes remains frozen between visualizations of different states. We call this **hard positional consistency**. Compare Figures 7a and 7b to see an example. Here, hard positional consistency requires that all of the displayed nodes (*U0*, *U1*, *U2*, *P0*, *P1*, *P2*, *P3*) remain frozen as we go from State 0 and State 1, with an extremely long edge being added between *U0* and *P3*. This long edge may not be visually pleasing.

Hard positional consistency can fail to produce satisfactory layout because it is too strong. It is easy to start with a state that yields a good layout; as the state evolves, hard positional consistency can enforce a layout that is no longer suitable. An alternative notion of consistency is **soft positional consistency**, which simply moves consistency from a strongly-enforced constraint (which the visualizer must satisfy) to a weakly-enforced objective (which the visualizer should try to satisfy if possible). That is, soft positional consistency may sacrifice positional consistency to preserve the overall aesthetics of the diagram, but it should keep the violations small. To see it in action, compare Figures 7a and 7c. Notice how while *U0*, *U1*, *U2*, and *P0* remain fixed, *P1* and *P2* moved slightly to the right to make space for *P3*.

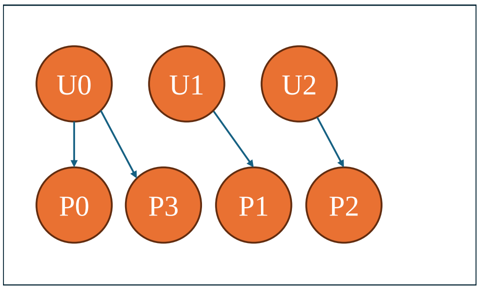
Finally, we consider **relative consistency**. Whereas positional consistency requires positions of all nodes to remain frozen with respect to the previous frame, relative consistency considers the *relative* positions between nodes connected with edges. In the example of Fig. 7, relative consistency would require the vectors between *U0* and *P0*, between *U1* and *P1*, and between *U2* and *P2* to remain frozen. To see it in action, compare Figures 7a and 7d.



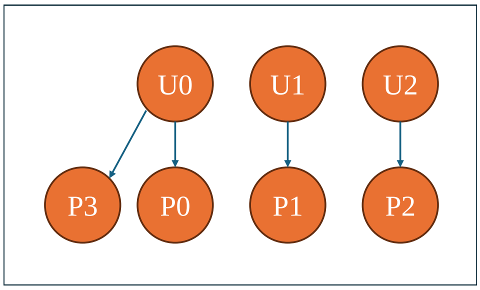
(a) State 0



(b) State 1, under **hard positional consistency**.



(c) State 1, under **soft positional consistency**.



(d) State 1, under **hard relative consistency**.

Fig. 7. Visualizations of a simple social network formal model in Alloy, with users (denoted with *U?*) and photos (denoted with *P?*), where blue arrows represent "ownership" relationships. We show the first two states of an instance, under different notions of consistency. State 0 is the initial state. In State 1, photo *P3* is added into the social network under user *U0*.

## 6.2 Formalizing visual consistency in Penlloy

A diagram $\mathcal{D}$ essentially stores the positions of all nodes, among other things. Let $\mathcal{D}$ : Node $\to \mathbb{R}^2$ be the mapping from nodes to their positions in the diagram $\mathcal{D}$. Furthermore, let nodes($\mathcal{D}$) be the set of all nodes in $\mathcal{D}$ and let edges($\mathcal{D}$) be the set of all edges (tuples of nodes) in $\mathcal{D}$. Then, between two diagrams $\mathcal{D}$ and $\mathcal{D}'$, the positional and relative consistency metrics are as follows:

$$\text{positional}(\mathcal{D}, \mathcal{D}') = \sum_{\substack{n \in \text{nodes}(\mathcal{D}) \\ \cap \text{nodes}(\mathcal{D}')}} \|\mathcal{D}(n) - \mathcal{D}'(n)\|^2, \quad \text{and}$$

$$\text{relative}(\mathcal{D}, \mathcal{D}') = \sum_{\substack{(n_1, n_2) \in \text{edges}(\mathcal{D}) \\ \cap \text{edges}(\mathcal{D}')}} \|(\mathcal{D}(n_2) - \mathcal{D}(n_1)) - (\mathcal{D}'(n_2) - \mathcal{D}'(n_1))\|^2 .$$

A consistency metric is zero when this type of consistency is satisfied. We can also try using different measures of distance, such as the non-squared norm.

To apply visual consistency in Penlloy, recall that the Penrose **style** program allows users to specify constraints and objectives on shapes in the diagram. In general, Penrose layout engine solves the following optimization problem [26]:

$$\min_{\vec{p} \in \mathbb{R}^m} \sum_{i=1}^{k} \mathcal{E}_i(\vec{p}) \quad \text{s.t.} \quad \sum_{i=1}^{l} \mathcal{P}_i(\vec{p}) = 0$$

where $\vec{p}$ is the vector of the values of all parameters in the diagram (shape positions, sizes, colors, etc.), $\mathcal{E}_i$s encode the the "energies" of the objective functions, and $\mathcal{P}_i$s are the non-negative penalty functions for the constraints. In other words, the Penrose layout engine tries to find a configuration of diagram parameters that satisfies all constraints ($\sum_i \mathcal{P}_i(\vec{p}) = 0$), while, as much as possible, also satisfying the objectives (minimizing $\sum_i \mathcal{E}_i(\vec{p})$).

We can extend the Penrose formulation of the optimization problem to work with consistency. Namely, hard consistency is

$$\min_{\vec{p}\in\mathbb{R}^m} \sum_{i=1}^{k} \mathcal{E}_i(\vec{p}) \quad \text{s.t.} \quad \mathsf{consistency}(\mathcal{D}, \mathcal{D}') = 0, \sum_{i=1}^{l} \mathcal{P}_i(\vec{p}) = 0$$

and soft consistency is

$$\min_{\vec{p}\in\mathbb{R}^m} \mathsf{consistency}(\mathcal{D}, \mathcal{D}') + \sum_{i=1}^{k} \mathcal{E}_i(\vec{p}) \quad \text{s.t.} \quad \sum_{i=1}^{l} \mathcal{P}_i(\vec{p}) = 0$$

where

$$\mathsf{consistency}(\mathcal{D}, \mathcal{D}') = \begin{cases} \mathsf{positional}(\mathcal{D}, \mathcal{D}') & \text{for positional consistency} \\ \mathsf{relative}(\mathcal{D}, \mathcal{D}') & \text{for relative consistency} \end{cases},$$

$\mathcal{D}'$ is the diagram in the previous frame, and $\mathcal{D}$ is the diagram in the current frame where $\vec{p}$ is from.

We have implemented a research prototype that brings these formulations of visual consistency into Bloom [1], a variant of Penrose that allows users to define more sophisticated constraints and objectives. We plan to incorporate this work into the Penlloy framework.

## 7 Conclusions and Future Work

This paper aims to improve visualizations of formal methods. We described our interview study on how formal methods visualizations are used and how they can be improved. Through the study, we concluded that (1) formal methods practitioners use or create **different types of diagrams depending on the underlying tool semantics**, (2) practitioners use visualizations to **understand, validate, and present formal models**, and (3) the **lack of domain specificity** and **the visual consistency problem** hinder these goals.

To address the lack of domain specificity, we designed **Penlloy, a domain-specific visualizer for Alloy models**. Penlloy leverages the correspondence between the Alloy world and the Penrose world to generate domain-specific visualizations. To address the visual consistency problem, we **formalized visual consistency** as optimization problems solvable under the Penlloy framework.

This work has laid the foundations for potential future research, such as:

- **Visual principles beyond consistency**. Visual consistency is but one of the important visual principles that can help make visualizations easier to use. Interviewees also hinted towards other potentially useful principles. For example, the "hierarchy" principle states that visualizations should intuitively reflect the hierarchical nature of models. It would be interesting to apply these and other visual principles in the Penlloy framework.
- **Empirical evaluations of visualization principles**. While the interview study has shed light on the desire for visual principles like consistency and hierarchy, we still do not yet know the effects of these principles. In particular, to what extent do these principles help formal methods practitioners understand and debug formal models, and how? It would therefore be interesting to empirically evaluate the benefits and potential drawbacks of these principles through human-subject experiments.

- **Synthesis of domain-specific visualizations from light-weight user inputs**. Current works in domain-specific visualizations of Alloy require authors of the visualizations to provide some sort of "visual instructions program": Penlloy users need to provide a **style** program; Sterling [12] users need to provide a JavaScript program using the D3 library; and CnD [29] users need to provide a similar instructions program in their own DSL. Visualizations of other generic modeling tools like B [33] require similar instruction programs. These programs encode how elements of Alloy instances translate into visual elements.
  However, as hinted by the expert feedback in Sec. 5.4.2, designing, writing, and refining the visual instructions programs can be too heavy-weight and burdensome, and require high degrees of familiarity with both the underlying language and visual design. Instead, formal methods users want a light-weight methodology to specify visualizations. We hence envision a framework that synthesizes Penlloy-**style**-like visual instructions programs from light-weight user inputs, such as a sketch of the desirable visualization for an instance. Such a synthesis may also take into account the desirable visualization principles like consistency and hierarchy. Kapur et al. [19] have investigated the use of diffusion to synthesize such programs from hand-drawn geometric sketches, a technique that may be useful in our case.
- **Beyond formal methods**. As of now, Penlloy still focuses on a single aspect (Alloy) of a single field (formal methods) of engineering, but one can imagine the Penlloy technique (mapping tool outputs to visual rules) to extend to the broader field of visualizing and understanding computer-generated outputs. Such outputs may come from not only model-finding tools like Alloy, but also other fields like program analysis (e.g., visualizing dataflow analysis outputs) and software testing (e.g., visualizing failing examples of fuzz-testing results). In general, the less "formal" the field, the more it relies on the stakeholders' intuitive understanding of its output. More effective visualizations of these outputs can help stakeholders build trust on these fields.

Formal methods visualizations can increase formal methods adoption, which may lead to more reliable systems. This paper advances this goal by paving the way for more effective visualizations.

## 8 Acknowledgments

We thank (in alphabetical order of last name) Shardul Chiplunkar, Daniel Jackson, Tim Nelson, Siddhartha Prasad, and Pamela Zave for their deep and insightful discussions on the visualization of formal methods and the design of our tools. We are also grateful to Tim Nelson and Marsha Chechik for providing valuable feedback on this work. Additionally, our collaboration with Alcino Cunha during the International Summer School on Formal Specifications and Analysis for Software Engineers (SPECS) 2023 was instrumental in kicking off this idea on improving formal methods visualizations through Penlloy.

This work was supported in part by NSF awards 2119007, 2144860, 2150217, 2319317, and 2346174.

## References

[1] [n. d.]. Bloom: Optimization-Driven Interactive Diagramming. https://penrose.cs.cmu.edu/blog/bloom
[2] [n. d.]. Peasy. https://p-org.github.io/peasy-ide-vscode/
[3] [n. d.]. River Crossing. https://alloytools.org/tutorials/online/frame-RC-1.html
[4] [n. d.]. Stately XState Visualizer. https://stately.ai/viz
[5] Vivek and Gupta, Ethan Jackson, Shaz Qadeer, and Sriram and Rajamani. 2012. *P: Safe Asynchronous Event-Driven Programming*. Technical Report MSR-TR-2012-116. https://www.microsoft.com/en-us/research/publication/p-safe-asynchronous-event-driven-programming/
[6] Shardul Chiplunkar and Clément Pit-Claudel. 2023. Diagrammatic notations for interactive theorem proving. doi:10.5075/epfl-SYSTEMF-305144

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (apr 1986), 244–263. doi:10.1145/5397.5399

[8] Rui Couto, José C. Campos, Nuno Macedo, and Alcino Cunha. 2018. Improving the Visualization of Alloy Instances. *Electronic Proceedings in Theoretical Computer Science* 284 (Nov. 2018), 37–52. doi:10.4204/eptcs.284.4

[9] Alcino Cunha, Nuno Macedo, and Eunsuk Kang. 2023. Task Model Design and Analysis with Alloy. In *Rigorous State-Based Methods*, Uwe Glässer, Jose Creissac Campos, Dominique Méry, and Philippe Palanque (Eds.). Springer Nature Switzerland, Cham, 303–320.

[10] E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Informatica* 1, 2 (01 Jun 1971), 115–138. doi:10.1007/BF00289519

[11] N. Dulac, T. Viguier, N. Leveson, and M.-A. Storey. 2002. On the use of visualization in formal requirements specification. In *Proceedings IEEE Joint International Conference on Requirements Engineering*. 71–80. doi:10.1109/ICRE.2002.1048507

[12] Tristan Dyer and John Baugh. 2021. Sterling: A Web-Based Visualizer for Relational Modeling Languages. In *Rigorous State-Based Methods*, Alexander Raschke and Dominique Méry (Eds.). Springer International Publishing, Cham, 99–104.

[13] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Temporal Stream Logic: Synthesis Beyond the Bools. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 609–629.

[14] Loïc Gammaitoni and Pierre Kelsen. 2014. Domain-Specific Visualization of Alloy Instances. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Yamine Ait Ameur and Klaus-Dieter Schewe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 324–327.

[15] Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. 2022. Little Tricky Logic: Misconceptions in the Understanding of LTL. *The Art, Science, and Engineering of Programming* 7, 2 (Oct. 2022). doi:10.22152/programming-journal.org/2023/7/7

[16] David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274. doi:10.1016/0167-6423(87)90035-9

[17] Tamjid Hossain and Nancy A. Day. 2021. Dash+: Extending Alloy with Hierarchical States and Replicated Processes for Modelling Transition Systems. In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*. 21–29. doi:10.1109/REW53955.2021.00010

[18] Daniel Jackson. 2016. *Software Abstractions, revised edition: Logic, Language, and Analysis*. MIT Press.

[19] Shreyas Kapur, Erik Jenner, and Stuart Russell. 2024. Diffusion On Syntax Trees For Program Synthesis. *arXiv preprint arXiv:2405.20519* (2024).

[20] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. 2018. Formally verified software in the real world. *Commun. ACM* 61, 10 (Sept. 2018), 68–77. doi:10.1145/3230627

[21] Jill H. Larkin and Herbert A. Simon. 1987. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science* 11, 1 (1987), 65–100. doi:10.1016/S0364-0213(87)80026-5

[22] Niloofar Mansoor, Hamid Bagheri, Eunsuk Kang, and Bonita Sharif. 2023. An Empirical Study Assessing Software Modeling in Alloy. In *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 44–54. doi:10.1109/FormaliSE58978.2023.00013

[23] Shahar Maoz and Jan Oliver Ringert. 2021. Spectra: a specification language for reactive systems. *Software and Systems Modeling* 20, 5 (April 2021), 1553–1586. doi:10.1007/s10270-021-00868-z

[24] 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. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 116 (April 2024), 29 pages. doi:10.1145/3649833

[25] Chris Newcombe. 2014. Why amazon chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 25–39.

[26] Wode Ni. 2024. Authoring Conceptual Diagrams by Codifying Visual Representations. (12 2024). doi:10.1184/R1/27969048.v1

[27] Gerard O'Regan. 2006. *Mathematical Approaches to Software Quality*. Springer London. doi:10.1007/1-84628-435-9

[28] Gerard O'Regan. 2023. *Overview of Formal Methods*. Springer Nature Switzerland, Cham, 255–276. doi:10.1007/978-3-031-26212-8_16

[29] Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi. 2024. Grounded Language Design for Lightweight Diagramming for Formal Methods. arXiv:2412.03310 [cs.CL] https://arxiv.org/abs/2412.03310

[30] Ian Pressman and David Singmaster. 1989. The jealous husbands and The missionaries and cannibals. *The Mathematical Gazette* 73, 464 (1989), 73–81. doi:10.2307/3619658

[31] Edward R. Tufte. 2001. *The Visual Display of Quantitative Information* (second ed.). Graphics Press, Cheshire, Connecticut. https://www.edwardtufte.com/tufte/books_vdqi

[32] Benjamin Tyler. 2023. *Formal Methods Adoption in Industry: An Experience Report.* Springer International Publishing, Cham, 152–161. doi:10.1007/978-3-031-43678-9_5

[33] Fabian Vu and Michael Leuschel. 2023. Validation of Formal Models by Interactive Simulation. In *Rigorous State-Based Methods*, Uwe Glässer, Jose Creissac Campos, Dominique Méry, and Philippe Palanque (Eds.). Springer Nature Switzerland, Cham, 59–69.

[34] Michelle Werth and Michael Leuschel. 2020. VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics. In *Rigorous State-Based Methods*, Alexander Raschke, Dominique Méry, and Frank Houdek (Eds.). Springer International Publishing, Cham, 260–265.

[35] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Jonathan Sunshine, and Keenan Crane. 2020. Penrose: From Mathematical Notation to Beautiful Diagrams. *ACM Trans. Graph.* 39, 4 (2020).

[36] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*. Springer-Verlag, Berlin, Heidelberg, 54–66.

[37] Pamela Zave. 2012. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.* 42, 2 (March 2012), 49–57. doi:10.1145/2185376.2185383

[38] Pamela Zave. 2017. Reasoning About Identifier Spaces: How to Make Chord Correct. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1144–1156. doi:10.1109/TSE.2017.2655056