Lightweight Diagramming for Lightweight Formal Methods: A Grounded Language Design

Siddhartha Prasad
Brown University

Ben Greenman
University of Utah

Tim Nelson
Brown University

Shriram Krishnamurthi
Brown University

Abstract

Tools such as Alloy enable users to incrementally define, explore, verify, and diagnose specifications for complex systems. A critical component of these tools is a visualizer that lets users graphically explore generated models. As we show, however, a default visualizer that knows nothing about the domain can be unhelpful and can even actively violate presentational and cognitive principles. At the other extreme, full-blown custom visualization requires significant effort as well as knowledge that a tool user might not possess. Custom visualizations can also exhibit bad (even silent) failures.

This paper charts a middle ground between the extremes of default and fully-customizable visualization. We capture *essential* domain information for *lightweight* diagramming, embodying this in a language. To identify key elements of lightweight diagrams, we ground the language design in both the cognitive science research on diagrams and in a corpus of 58 custom visualizations. We distill from these sources a small set of orthogonal primitives, and use the primitives to guide a diagramming language called Cope-and-Drag (`CnD`). We evaluate it on sample tasks, three user studies, and performance, and find that short `CnD` specifications consistently improve model comprehension over the Alloy default. `CnD` thus defines a new point in the design space of diagramming: a language that is lightweight, effective, and driven by sound principles.

2012 ACM Subject Classification Human-centered computing → Visualization toolkits; Software and its engineering → Constraint and logic languages; Software and its engineering → Specification languages

Keywords and phrases formal methods, diagramming, visualization, language design

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.11

Supplementary Material *Software (Source Code)*: `https://github.com/sidprasad/copeanddrag/tree/ecoop-2025`
Software (Interactive Diagrams): `https://doi.org/10.5281/zenodo.15278550`

Funding This work is partially supported by US NSF grant DGE-2208731.

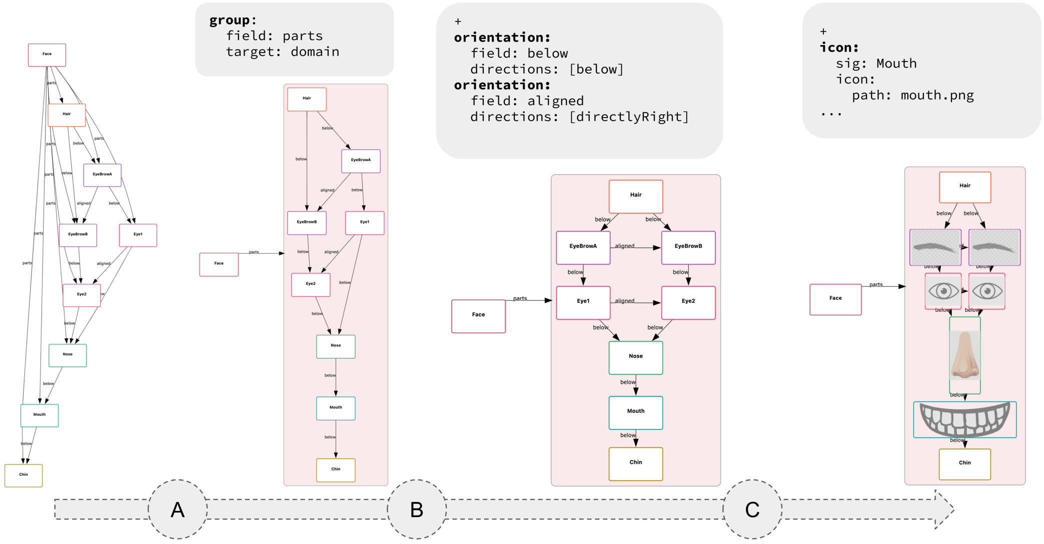
Acknowledgements We thank Tristan Dyer for sharing his insights on Sterling. We are also deeply grateful to Rob Goldstone for gently educating us about cognitive science. We thank Skyler Austen and Ji Won Chung for their feedback on study design and supplementary material.

1 Introduction

In *lightweight formal methods*, proposed by Jackson and Wing [15] and realized in tools such as Alloy [16], users build formal models iteratively, exploring the design space as they go.

Alloy specs consist of sig[nature]s (sets of atoms), fields (relations between sigs), and descriptions of system behavior. Whereas traditional verification tools are inert without

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 11; pp. 11:1–11:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



```
group:
  field: parts
  target: domain
```

```
+
orientation:
  field: below
  directions: [below]
orientation:
  field: aligned
  directions: [directlyRight]
```

```
+
icon:
  sig: Mouth
  icon:
    path: mouth.png
...
```

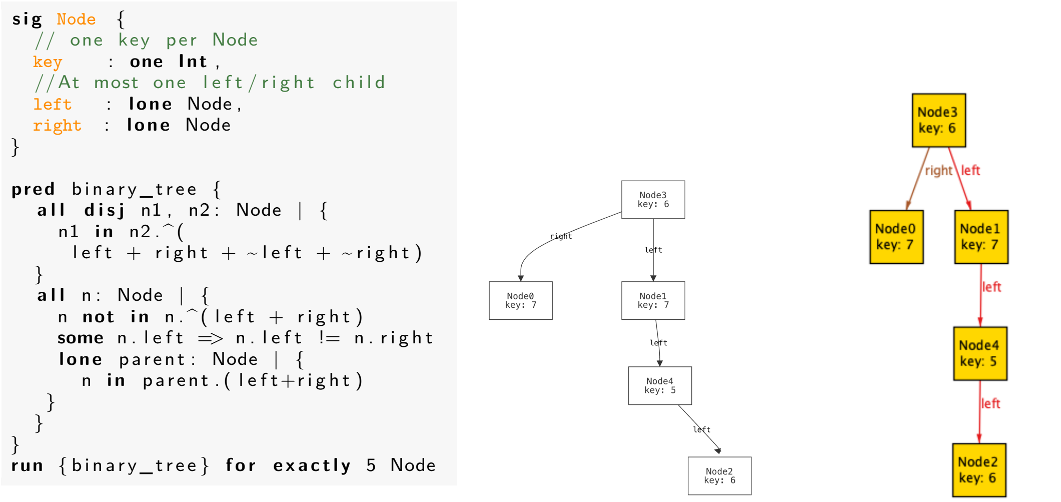
Figure 1 Cope-and-Drag is a language built to support lightweight diagramming for lightweight formal methods. A small set of orthogonal primitives allows users to incrementally refine model-finder output into more effective diagrams. This figure shows how to incrementally refine an instance describing a face using (A) grouping, (B) orientation, and (C) icons.

properties, Alloy users can begin reasoning with *just* system descriptions. Alloy uses a (SAT) solver to automatically generate instances that satisfy the spec, presenting them most commonly visually (e.g., Figure 2). This makes using Alloy as close an analog to programming as most formal methods tools get. Unlike a user of a proof assistant (where one must at least somewhat manually construct a proof) or model checker (which requires properties to verify), an Alloy user can enter a spec (akin to a program) and just click “Execute”, and see a sequence of examples. This modality, in fact, goes beyond that of a typical programming language, as the user doesn’t need to provide specific inputs to see output.

This immediacy and ability to work in the absence of properties has two benefits. First, users can spot ways in which the output does not conform to their intent and improve their specification. Second, seeing these violations suggests desirable and undesirable *properties*, which can then be formalized and verified—addressing the important and often neglected question of where properties come from. Thus, realizing lightweightness hinges heavily on the quality of visualizations.

Alloy and its recent sibling tool Forge [32] visualize relations as graphs. Alloy comes with a default GUI tool [16]. Forge uses Sterling [11] to render interactive graphs in a web browser (Sterling can visualize Alloy instances as well). Both the Alloy default and Sterling are generic, in that they do not embody any knowledge of the domain being modeled, which makes them largely indistinguishable from the perspective of our work: e.g., see Figure 2c and Figure 2b. While obtaining these visualizations requires zero effort from users, research shows that users struggle to make sense of them as models grow in size and complexity [24].

Sterling also provides support for custom JavaScript visualizations. They can be narrowly crafted to a domain, and can be quite visually attractive (e.g., [32, Figure 11]; Figure 11b). By being programmable (in addition to being interactive), they can be applied repeatedly and consistently to all output. However, these custom visualizations suffer from a dual problem. As we discuss in Section 2.4, users must: know the details of drawing libraries, which are orthogonal to formal methods; expend a great deal of effort; and not inadvertently



```
sig Node {
  // one key per Node
  key    : one Int,
  //At most one left/right child
  left   : lone Node,
  right  : lone Node
}

pred binary_tree {
  all disj n1, n2: Node | {
    n1 in n2.^(
      left + right + ~left + ~right)
  }
  all n: Node | {
    n not in n.^(left + right)
    some n.left => n.left != n.right
    lone parent: Node | {
      n in parent.(left+right)
    }
  }
}
run {binary_tree} for exactly 5 Node
```

(a) Alloy spec. **(b)** Sterling output. **(c)** Alloy Visualizer output.

Figure 2 Alloy spec of a binary tree, without any properties, alongside a generated instance (visualized by Alloy Visualizer). `Node` is a sig that defines a set of atoms, while fields `left`, `right`, and `key` describe relations between nodes. Predicate `binary_tree` imposes constraints on these fields.

run afoul of visual design principles. Furthermore, custom visualizations suffer from a critical mode of *silent failure* when the specification and visualization disagree on model invariants. Silent failures, we argue, are especially dangerous for lightweight formal methods (Section 4).

In contrast, we want customization to not only be programmable but also: be lightweight; avoid complex library over-dependence; embody good visual design principles; and prevent failure modes. By embodying these in a programming language, we can be confident that *all* programs written in that language enjoy these desirable traits. Thus, this paper views the visualization task as a *language design problem.*

We therefore present a new language, Cope-and-Drag (`CnD`), for this purpose. `CnD` is driven both top-down, by cognitive science principles of visualization, and bottom-up, by manually distilling ideas from dozens of actual visualizations (Section 2). The resulting language is small, requires little annotation, and can be used incrementally (Section 3). `CnD` is designed to avoid the critical failure mode instead of silently rendering a misleading diagram (Section 4). We show through experiments that the features of the language result in visualizations that are more effective than the default visualizer (Section 5). `CnD` is embedded in an open-source tool that serves as a substitute visualizer for Alloy and Forge, available at `https://github.com/sidprasad/copeanddrag/tree/ecoop-2025`.

While this work falls in the realm of recent language designs for visual output, we show why this work is novel (Section 6) and particularly well-suited to the important domain of lightweight formal methods (Sections 3.3 and 4). Furthermore, this work or its extensions can also apply to other formal methods tools that employ diagramming (Section 6.3).

Vocabulary

- A **specification** (or, in the lightweight spirit, *spec*) is a formal description of a system's behavior (e.g., Alloy or Forge code).
- An **instance** is a concrete assignment of values to variables that satisfies the constraints and properties defined by the spec (e.g., a model generated by SAT, lifted to the domain).

- A **bad-instance** is an instance that fails to satisfy part of the *implicit* spec that the author is *trying* to write. A bad instance reflects a discrepancy between what the explicit spec currently represents and what the author intends it to represent. We use the colloquial word "bad" intentionally, since it is human judgment that identifies such a discrepancy. The discovery and extirpation of bad-instances while driving towards the implicit spec is a key part of the lightweight formal methods process.
- A **default-graph** is a directed-graph visualization produced by the Alloy Visualizer [16] or Sterling [11] (without customization). These graphs are agnostic to the domain being modeled. (While there are differences between the two tools, discussed in Section 6.1, for the purposes of this paper, Alloy and Sterling graphs are effectively the same, so we refer to them by the same general label.)
- A **triad** is a three-part figure that consists of default-graph output, `CnD` output, and the corresponding `CnD` source code. We use triads extensively in the supplement.

Supplemental Material

A powerful aspect of `CnD` diagrams is their interactive nature. While we include screenshots in this paper, we also provide interactive versions in supplementary material. Any diagram name that appears in a blue box, as in **name**, has an interactive variant available through a Docker image. Instructions for running the Docker are in the supplement readme (`supplement.pdf`). Instructions on accessing the exact survey instruments used in Section 5.2 are available in the supplement as well.

2 Diagramming Language Design

Languages can be designed from many perspectives, such as personal taste, carefully subjecting every feature to a user study, or programmer opinion, which can be broadly expressed or narrowly targeted. While these all have their validity, we focus on *grounded* design: designs that are derived from, and can be substantiated by, external artifacts.

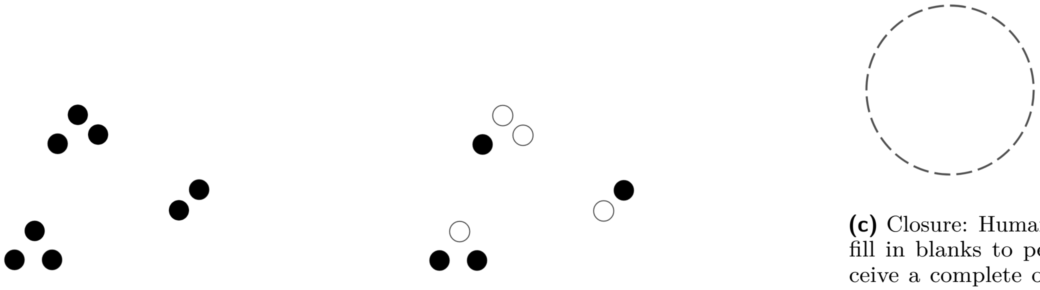
There are roughly two ways to proceed. One is bottom-up: start with a large collection of desired artifacts, cluster their commonalities and differences, identify their principal dimensions, and use those to inform the design. Another is top-down: start with general principles established by experts (usually based on prior grounded efforts) and concretize them into operations. Because these approaches complement each other well, we use both methods. We start top-down, distilling key principles from the diagramming literature (Section 2.1), and then examine a corpus of concrete examples in the context of these principles (Section 2.2). We then align these perspectives to derive a set of diagramming primitives that are both theoretically grounded and practically applicable in the context of lightweight formal methods (Section 2.3).

2.1 Top Down

We begin our language design by drawing on established principles in cognitive science, visualization, and diagramming.

2.1.1 Visual Principles

A diagram is a visual representation that uses shapes and symbols to convey information or ideas. In this role as an information presentation tool, effective diagrams rely on perceptual



(a) Proximity: Our minds perceive a connection between objects in close spatial proximity.

(b) Similarity: Objects sharing the same color, shape, or size are perceived as related.

(c) Closure: Humans fill in blanks to perceive a complete object. While this diagram consists of a set of disconnected dashes, a viewer perceives a circle.

Figure 3 Examples of the Gestalt principles of proximity, similarity, and closure.

properties common in information visualization, the most relevant (and best established) of which relate to pre-attentive processing and the Gestalt principles [5].

Pre-Attentive Processing: Pre-attentive processing is a cognitive function that allows humans to quickly detect certain visual features (e.g., color, shape, and size) of objects without conscious effort. This rapid processing can both help and hinder understanding.

The Stroop effect, for instance, demonstrates that the brain often automatically processes text, such as in word recognition, faster than stimuli such as color [42] and spatiality [48]. In the context of diagrams, this means that understanding is hindered when visual elements conflict with textual content (e.g., laying out a binary tree's `left` children to the right and `right` children to the left). Other common features that might cause undesirable pre-attentive processing include irrelevant visual elements [44], unnecessarily overlapping lines [38], or ambiguous labeling [5, 22]. Effective diagrams, therefore, use visual features to highlight important information while being mindful of extraneous cognitive load [43]. For example, Figure 5 leverages different colors to help viewers quickly distinguish between Alloy identifiers and `CnD` keywords.

Gestalt Principles: The Gestalt principles [17, 40, 47] describe how humans perceive and organize visual information, grouping objects that are close together (proximity), share attributes (similarity), form complete shapes (closure), follow a continuous path (continuity), stand out from a background (figure-ground), appear balanced (symmetry), or move together (common fate).[1] These principles play a crucial role in visual design, user interface design, and data visualization, helping to create more intuitive and easily comprehensible visual representations. Figure 3 illustrates some of these principles.

The Gestalt principles so accurately describe human perception, and leveraging pre-attentive processing is so inherently effective, that they both underpin many diagramming conventions, even when their influence is not explicitly acknowledged. For a more detailed description of how these principles apply to visualization, see [5, page 8].

[1] While these principles were originally formulated by Wertheimer in 1912 [47], we authors are more familiar with Spillman's English translation [40] and Koffka's framing in the field of psychology [17].

2.1.2 Diagramming Principles

While visualizations focus on displaying information, diagrams act as cognitive tools that facilitate deeper comprehension and analysis of complex systems [20, 22, 45]. A diagram's effectiveness stems not only from its visual appeal, but also from its ability to organize information in a way that facilitates efficient cognitive processes [20]. These processes depend on the task one is trying to perform.

Subway maps, for example, leverage the observation that semantic similarity is intuitively tied to spatial proximity [14]. By grouping together stations that are functionally related (e.g., transfer points or lines serving the same neighborhoods), they serve as effective diagrams for planning public transportation routes. However, this layout is poorly suited for understanding the actual geographic distances between stations, as the map distorts spatial relationships in favor of simplifying the navigation task.

Thus, while a "good" diagram must be well laid out, it must also relate well to the *problem* it is designed to address [5, 22]. Diagrams are of help when they:

1. Leverage spatial metaphors and natural correspondences in the source domain [20, 45].
2. Convey patterns and relations in the source domain in spatial terms [14, 20].
3. Group together all information that will be used together [14, 20].

2.1.3 Core Diagramming Operations

From the above literature, we identify that users require the following operations in order to create effective diagrams:

Relative Positioning: Users should be able to specify the relative positions of diagram elements. This allows diagram elements to be spatially arranged to reflect the underlying structure of the domain, relating to the Gestalt principles of continuity, closure, and proximity. Placing graph elements consistently in relation to others can imply diagrammatic flow, groupings, and even underlying shapes. We identify two key aspects of relative positioning:

1. *Directional relationships* between diagram elements. For example, specifying that elements should be placed above, below, to the left, or to the right of one another.
2. *Angular flow* to describe element positions. For example, specifying that elements should be arranged in a circle, or that they should form a shape.

Grouping: Users should be able to group related diagram elements together. Leveraging the Gestalt principle of proximity, grouping related elements together can help users to better understand the relationships between them. This also allows diagrammers to group together elements that will be used together, reducing extraneous cognitive load. Grouping relationships in interactive diagrams can also help users reason about higher-level structures and relationships, leveraging the Gestalt principle of common fate.

Styling: Users should be able to control some visual aspects of how diagram elements are rendered. We identify two ways in which users might want to style their diagrams: theming and iconography. Theming involves influencing the aesthetic properties of diagram elements, such as the color, size, and shape of elements. This allows users to create diagrams that tap into the Gestalt principles of similarity and figure-ground.

Iconography, on the other hand, involves the use of pictorial elements in diagrams that themselves carry semantic meaning, e.g., representing an eye with an image instead of the word "eye" (Figure 1). Icons and symbols in diagrams can tap into pre-attentive processing, allowing users to quickly understand what a diagram element represents without needing to examine the diagram in detail.

With these primitives, diagrammers should be able to create artifacts that:

- Focus on important parts of the problem.
- Are tools for knowledge presentation. That is, they offer insight into instances of the spec being explored.
- Act as tools for knowledge discovery. That is, they show key aspects of bad-instances of the spec.

2.2 Bottom Up

While the top-down approach provides a theoretical foundation for our language design, much of the literature draws on abstract principles that may not be directly applicable to the formal methods domain.

To study how these principles manifest in practice, we contacted the Forge [32] team, who kindly gave us access to anonymized data on visualizations that students created in their upper-level undergraduate course on formal methods, as well as the results of a follow-up survey on students' visualization experiences. Students had given permission for this use.[2]

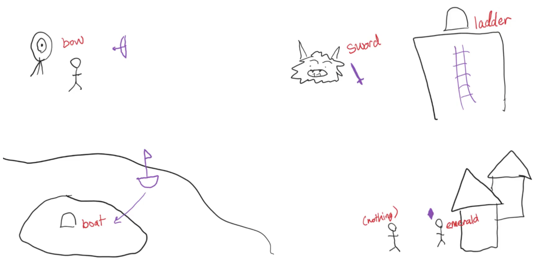


Figure 4 A student-drawn diagram illustrating the layout of their spec. This layout relies on relative positioning, grouped elements, and iconography.

The dataset contained anonymized retrospective comments on 58 final group projects (21 from 2022, 21 from 2023, and 16 from 2024), most of which were written in Forge, and visualized either as default-graphs or using D3 [3] to customize Sterling. As a result, we were able to gain insight into how two methods of visualization work for users who have roughly a semester of experience with formal modeling.

Projects covered a wide range of domains, from games like Solitaire to cryptographic protocols and group theory. Each project involved a code submission (a spec and optional visualization) as well as a recorded presentation or written summary explaining the domain being modeled, the design decisions made, and an optional demonstration of the visualization.

Most projects (47 total, 81 %) involved some kind of custom visualization (typically relying on Sterling's D3 integration). We examined these projects to determine how the principles we derived from the cognitive science literature were reflected in the visualizations students created. We also included projects in our analysis that did not include a custom visualization, but where authors were especially clear about wanting a custom visual component even if they did not have one, e.g., by presenting a hand-drawn diagram (e.g., Figure 4) or by manually rearranging nodes of the Sterling default-graph in their presentation.

In our analysis, we found that:

- 44 projects (75 %) involved custom visualizations that had clear representation salience and minimal visual clutter. Each atom in the model was represented by at most one

[2] Some of these visualizations are shown in figure 11 of [32].

element in the visualization, and each element in the visualization corresponded to an important aspect of the model.

- 41 visualizations (70 %) communicated semantic relationships between elements via their relative orientation. Examples include the grid layout of cells when visualizing the games Minesweeper and Battleships, and the relative positions of nodes in red black trees.
- 19 visualizations (33 %) grouped nodes together inside explicit shapes to imply relatedness. Examples include piles of cards in the card game Solitaire, player hands in the game Chopsticks, and sets of elements in group theory.
- 4 visualizations (7 %) conveyed spec semantics by laying out diagram elements to imply shapes or directionality. Examples include the cyclic order of player turns in the card game Coup and travel paths in a map routing algorithm.

The Forge team also shared anonymized results of a survey. Students explained that they created custom visualizations due to the large, unstructured instances presented by the default-graphs. Specific issues included: too many edges and nodes, complicated states, and overlapping components. Their custom visualizations hid unnecessary information, conveyed key aspects of the source domain, and could even act as useful debugging tools.

While the decisions made by students may not fully represent those of experienced formal methods practitioners, they do provide valuable insight into some key aspects of the modeling process. While working on open-ended projects, students had to carry out real steps a practitioner would take: choose a domain they were interested in and knowledgeable about, formally model it, debug the models, and effectively communicate their output.

2.3 Top-Down and Bottom-Up Alignment

Our analysis of student projects suggests that diagramming principles from the cognitive science literature (Section 2.1), such as reducing cognitive load and highlighting important information, are very much applicable in the context of lightweight formal methods. Across a diverse range of domains, student visualizations preserved domain-specific spatial relations, grouped together related elements, and encoded spec semantics in terms of spatial layout and directionality. They also draw attention to a key additional requirement in the context of formal methods: support for bad-instances. We discuss this in further detail in Section 4.

2.4 The Pain Barrier to Custom Visualization (with D3)

We have motivated the value of custom visualizations. However, they come at a heavy price. Not only do users need familiarity with JavaScript, but D3—one of the most influential online visualization libraries [2]—is notorious for its steep learning curve, API verbosity, and low-level operations [2, 28]. It is also worth noting that both JavaScript and D3 are entirely orthogonal learning objectives to someone learning or performing formal methods.

The nature of D3 also means that Sterling-with-D3 scripts are often enormous relative to the specs they are built to visualize. For example, the code for the river-crossing visualization in the Sterling demo [10] is *four times* longer than the spec (148 lines vs. 35 lines). It can also be difficult during maintenance and debugging to understand which parts of their D3 code correspond to which parts of the spec. Furthermore, a typical custom visualization is an “all-or-nothing” proposition: until enough of the visualization has been written, the user gets no benefit at all. This critically violates the “pay as you go” flavor that is characteristic of lightweight formal methods. It also makes it difficult to iterate, debug, and maintain custom visualizations, discouraging the use of D3 for exploratory diagramming tasks.

Program ::= Constraint* Directive*

Constraint ::= CyclicConstraint
  | OrientationConstraint
  | GroupingConstraint

CyclicConstraint ::= Field FlowDirection

OrientationConstraint ::= Field Direction$^+$
  | Sig Direction$^+$

GroupingConstraint ::= Field Target

FlowDirection ::= clockwise | counterclockwise

Direction ::= above | below | left | right
  | directlyAbove | directlyBelow
  | directlyLeft | directlyRight

Target ::= domain | range

Directive ::= PictorialDirective
  | ThemingDirective

PictorialDirective ::= Sig Icon

ThemingDirective ::=
  Attribute | Color
  | Projection | VisibilityFlag

Icon ::= path height width

Attribute ::= Field

Color ::= SigName color

Projection ::= Sig

VisibilityFlag ::= hideDisconnected
  | hideDisconnectedBuiltIns

Figure 5 Abstract syntax of Cope-and-Drag. Identifiers from the source Alloy spec are in orange, while CnD-specific keywords are in blue.

In Section 2.2, we have already indicated that some students used paper drawings or manually dragged boxes to simulate what they wanted. In the same survey in which students praised the value of custom visualizations, 37 respondents (44%) did not use JavaScript/D3 *at all.* These students primarily identified the complexity of D3 and the need to use JavaScript as a significant barrier to building custom visualizations, reporting difficulty learning JavaScript and D3, having enough time, starting, and finishing, corroborating past findings [2].

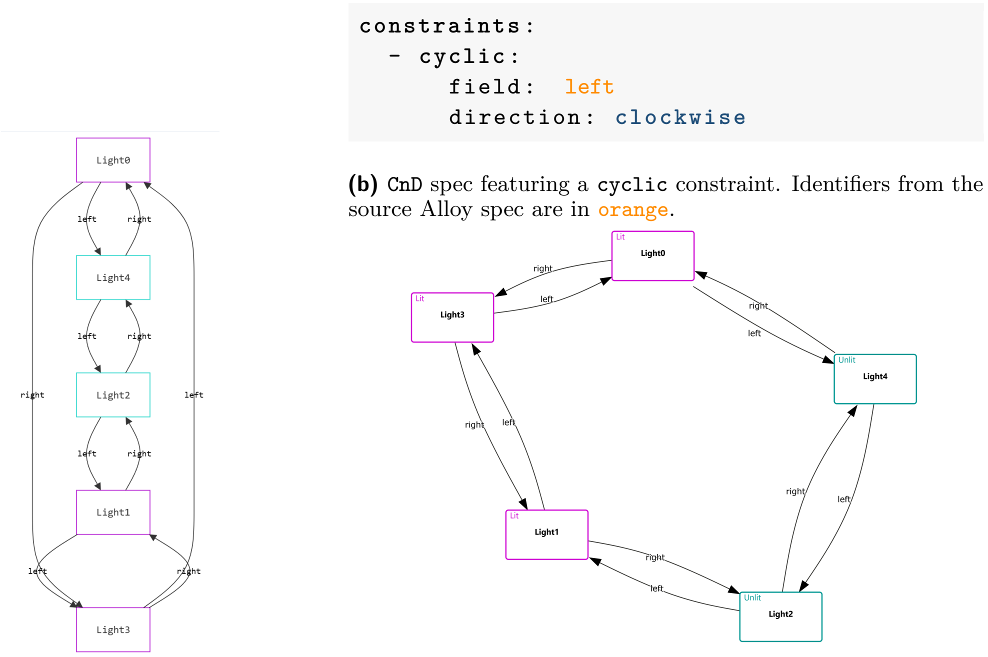
We thus see that the potential for custom visualization is undermined by the current tooling, which is onerous and often too daunting for users. (We discuss other drawing systems and languages in Section 6.) This observation demands a better, lightweight way of capturing the *essential* elements of diagrams that draws on the above principles and examples.

3 The Cope-and-Drag System

Cope-and-Drag (`CnD`) is a lightweight diagramming language designed for use with Alloy-like languages. The tool's name is inspired by the cope-and-drag casting process in metallurgy, reflecting its adaptable approach to diagram creation. `CnD` implements the diagramming operations identified in Section 2, and applies them to directed graphs with the goal of improving upon default-graphs. These generated diagrams support user interactions *while maintaining consistency with the diagram spec.* Thus, `CnD` not only facilitates diagram creation, but also supports *faithful* interactive exploration of instances being visualized.

3.1 `CnD` Primitives

`CnD` primitives operate upon Alloy/Forge sigs (which are essentially types) and their fields. These primitives fall into two categories: directives and constraints. Directives are used to specify the visual representation of a sig or field. Constraints are used to specify the relationships between visual representations. The resultant diagrams are interactive, allowing users to drag nodes around and explore the spec, while ensuring that the visual representation remains consistent with the specified constraints. For instance, if a constraint specifies that `A` must be to the left of `B`, a user will not be allowed to drag `B` to the left of `A` (**ab**). In cases



```
constraints:
  - cyclic:
      field:  left
      direction: clockwise
```

(a) Default-graph.

(b) `CnD` spec featuring a `cyclic` constraint. Identifiers from the source Alloy spec are in orange.

(c) `CnD` diagram.

Figure 6 ring-lights triad, contrasting a default-graph with `CnD` for an instance of a puzzle involving 5 lightbulbs arranged in a ring. The default misleadingly positions the nodes in a line. Supplement Figure 1 presents a larger version of the diagrams.

where constraints cannot be satisfied, no diagram is produced (Section 3.2).

Each `CnD` primitive is designed to fulfill a specific requirement identified in Section 2.

Relative Positioning: Cyclic and Orientation Constraints: Cyclic and Orientation constraints are used to specify the *relative positioning* (Section 2.3) of diagram elements.

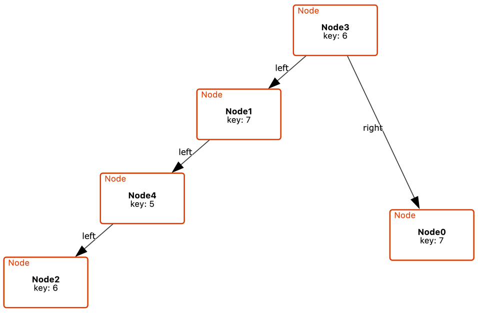
Cyclic constraints lay out atoms related by a sig field along the perimeter of a notional circle, suggesting they form a shape. If a field does not define a complete cycle, each connected subset is arranged on the circle's perimeter based on a depth-first exploration of the field. Atoms in the relation can be positioned either clockwise (the default) or counterclockwise. Figure 6 presents a triad. The `CnD` diagram arranges nodes in a circle, in accordance with the spec, because the `CnD` program includes a `cyclic` constraint. The default-graph, lacking this domain-specific information, arranges nodes in a line. A user would need to carefully read the edge labels or the underlying spec to discover that the nodes have a cyclic relationship.

Orientation constraints specify the placement of atoms in a diagram instance relative to one another. These constraints can be applied between sig types (e.g., all atoms of type `A` must be to the left of those of type `B`) or along sig fields (e.g., atoms related by the `child` field must be placed downward). For example, the orientation constraints in Figure 7a are at the heart of a `CnD` program for binary tree instances produced by Figure 2. Supplement Figure 2 presents a triad with the full `CnD` spec and a default-graph. The default produces a Stroop effect, putting the left child on the right and vice versa. Users have no way to prevent this effect.

Grouping Constraints: Grouping constraints specify that atoms related by a sig field should be grouped together spatially. Groups can be created based on either the range (default) or domain of the sig field. Grouping goes beyond merely associating atoms: `CnD`

```
- orientation:
    field:  left
    directions: [left, below]
- orientation:
    field:  right
    directions: [right, below]
```

(a) CnD spec



(b) CnD diagram.

Figure 7 bt CnD spec and diagram for binary tree instances produced by Figure 2.

replaces the individual edges between the group source and target atoms with a single edge, and places grouped atoms within a bounding box. Groups cannot intersect unless one is entirely subsumed by another.[3]

Figure 8 shows the CnD-rendered state of a river-crossing puzzle, using groups to indicate shores. This picture significantly reduces the number of edges; it also shows which entities are on the same shore, thereby avoiding confusion because near and far shore entities are interleaved in the default-graph version. A careful look at the default Sterling visualization of a similar river crossing spec in the Forge paper [32, Figure 5, left] shows that it too exhibits a Stroop effect.

In general, grouping constraints allow users to focus on higher-level structures and relationships without being overwhelmed by individual connections.

Directives: Directives allow users to control the visual aspects of how atoms of a particular sig type are rendered.

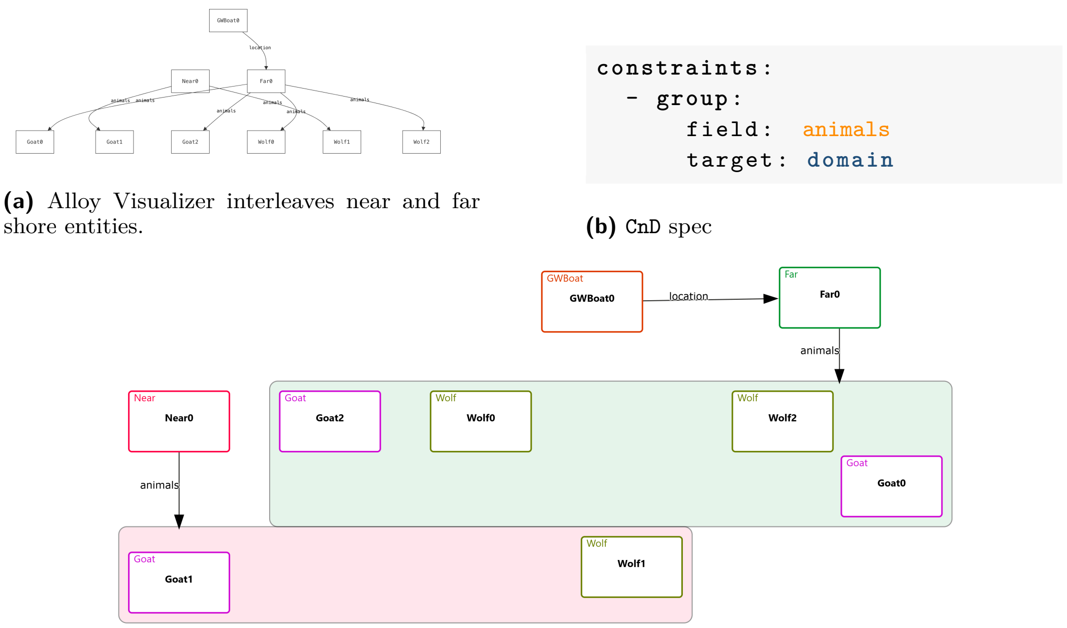
Pictorial directives specify that atoms of a particular sig must be represented by a specific icon (with or without a textual label as well). These icons help to visually situate the output in the domain being modeled, making the output less uniformly textual and exploiting pre-attentive processing. Two examples are the facial features in Figure 1 and the fruit in Figure 14.

CnD also supports a subset of the theming directives made available by the Alloy Visualizer and Sterling. Attribute directives display a field relation as a label on atoms of a particular sig type. Sig color directives associate specific colors with atoms of a certain sig type. Projection directives allow diagrams to focus on the atoms of a particular sig and their directly connected relations, hiding all other elements. Users can also hide disconnected atoms, depending on the provenance of their sig type, by using the `hideDisconnected` and `hideDisconnectedBuiltIns` visibility flags.

3.2 When Constraints Cannot be Satisfied

All CnD constraints are hard constraints, and thus *must* be satisfied for a diagram to be produced. Constraints might not be satisfied for one of two reasons:

[3] The requirement that groups not intersect is a technical limitation of the WebCola library [8] used by CnD. Overlapping groups affect the computation of bounding boxes and convex hulls. Group bounding boxes, as determined by WebCola, are also not guaranteed to be minimal, and can sometimes be larger than necessary.



(a) Alloy Visualizer interleaves near and far shore entities.

```
constraints:
  - group:
      field:  animals
      target: domain
```

(b) `CnD` spec

(c) `CnD` visualization groups together by the shore they are on.

Figure 8 rc Alloy Visualizer and `CnD` representations of a river-crossing puzzle instance.

1. A single `CnD` constraint definition could be inherently unsatisfiable. For example a constraint that requires the same field to be laid out both leftwards and rightwards:

   ```
       - orientation: {field:  next , directions: [right, left]}
   ```

   This represents a bug in the `CnD` spec, and can be identified statically. In this case, `CnD` produces an error message in terms of the constraints that could not be satisfied.

   ```
   Inconsistent orientation constraint:
   Field next cannot be laid out with directions: right, left.
   ```

   Checking for inconsistencies is a relatively simple process: `CnD` just needs to ensure that the same field is not laid out in conflicting directions.
2. Some constraints may be individually satisfiable, but become contradictory when laying out a specific instance. This is akin to a dynamic error, as it depends on the structure of the instance being visualized. For example, consider the constraints:

   ```
       - orientation: {sigs : [ X ,  Y ], directions: [below]}
       - orientation: {sigs : [ Y ,  Z ], directions: [below]}
       - orientation: {sigs : [ Z ,  X ], directions: [below]}
   ```

   If an instance's elements form a cyclic dependency, the constraints become unsatisfiable. However, an instance may lack atoms of, say, `Z`, in which case the constraints are satisfiable. `CnD` identifies these inconsistencies by treating layout as a linear optimization problem. Constraint primitives are incrementally added to the Cassowary linear constraint solver [1] to ensure that a satisfying layout exists. `CnD` stops at the first indication that constraints cannot be satisfied, and provides an error message in terms of the currently added constraints and instance atoms (e.g., Figure 10).

In both these cases, CnD does *not* produce a diagram. Instead, it provides an error message explaining that the constraints could not be met. If all constraints can be satisfied, CnD relies on the WebCola library [8, 9] to produce a diagram.

3.3 The Lightweightness of CnD

Having described the design of CnD, we now briefly justify calling it "lightweight".

Critically, CnD *refines* a default-graph output. Thus, the user can apply it incrementally: the empty CnD program still produces output, indeed, the exact same output as Sterling. (We discuss this design principle in more detail in Section 8.) Users can thus decide which aspects of the output most need refinement (just as they already do with Alloy Visualizer directives) and apply these rules incrementally. Furthermore, it is easy to turn a rule on or off, to see whether it has the desired effect. This incremental input refinement does not, however, imply incremental output refinement. Adding a new rule to a CnD program a new rule may change the layout substantially. Enforcing minimal output changes requires implicit constraints, which can obscure bad instances (Section 4) or introduce unintended Stroop effects (Section 2.1).

CnD programs also tend to be very brief. There is only a small number of primitives (constraints and directives) in the language (Section 3.1). Because these primitives apply to sigs and fields, there is an upper-bound on how many we can write, and this is independent of the size of the *operational* part of the spec.

CnD output is also "lightweight" in another sense: the language offers few bells-and-whistles, and the output is not necessarily pretty. We view the path from a CnD program to a full-fledged custom visualization as akin to that from an Alloy or Forge spec to a correct-by-construction spec verified by a proof assistant. The goal is not to produce diagrams that are attractive, but rather ones that are functionally useful and do not confuse, mislead, or otherwise abuse cognitive principles.

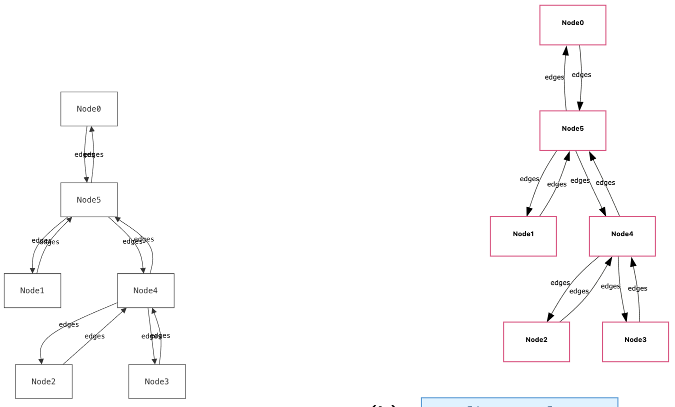
3.4 Where do CnD Programs Belong?

A design question is where to write a CnD program. Is it part of the spec, or does it live separately from the spec? This partially depends on whether it even makes sense to have more than one CnD program for a spec.

We argue that it is not only possible but also sensible to have multiple views on instances of the same spec. As an example, suppose a user is modeling a self-stabilizing protocol for a distributed system. Depending on the goal, they might want to use a ring to show network topology (e.g., Figure 12b), a DAG to examine hierarchies, or groups to explain permissions. No one view is more privileged than the other. They might even switch between these as they explore and debug different aspects.

In our current implementation, the CnD program resides in the visualizer. This is for three more reasons beyond the one given above. First, it is easy to explore commands incrementally even for a given instance (which follows Ma'ayan et al.'s recommendation [22]). Second, the same visualizer works with both Alloy and Forge (and any other tools that use the same protocol), providing portability of these ideas. Finally, it saves us the effort of modifying the implementations of Alloy and Forge.

That said, CnD programs are not completely separate from the spec; they are really a *spatial refinement* of the spec. Therefore, it would be meaningful to extend the spec languages to include (and name) each of these views. The "static" checking that CnD provides (Section 3.2) is then arguably part of the act of checking the spec itself. The user interface



(a) Default-graph.

(b) undirected-tree `CnD`, from an empty program.

Figure 9 Default Alloy and default `CnD` outputs for an undirected tree spec.

could then provide a menu of these pre-defined views, which were presumably chosen by the domain expert, while enabling the user of the spec to write their own views as well (as they currently can).

3.5 Visual Details

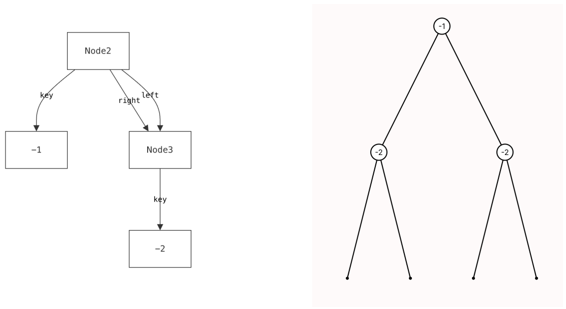
Whenever possible, `CnD` diagrams also encode a set of visual principles designed to reduce cognitive load and enhance user comprehension. Graph edges are laid out in a way that minimizes crossings and with as few bends as possible [38]. To reduce edge ambiguity, `CnD` also ensures that arrowheads are not incident on the same point of a node. We see an example of this in Figure 9, whose `CnD` program is empty, so the `CnD` output embodies only the above principles relative to the default-graph: observe the absence of overlapping edge labels. In addition, in `CnD` output, atoms of each sig are assigned a unique color by default, making salient the type of each node in the diagram. This is not visible in Figure 9 because all the atoms belong to the same sig, but we see it in several outputs such as Figures 1, 12a, and 12c (whose `CnD` programs do not include any coloring directives).

4 Bad-Instances: When Custom Visualizations Can Fail Silently

As we have discussed in Section 1, unlike tools like verifiers—which are inert in the absence of a property—lightweight formal methods can consume *just* a spec and show concrete instances of it. (Researchers have shown how to leverage non-instances as well [12, 26].) This helps bootstrap the formalization process by surfacing latent properties.

In practice, the modeling process is a series of incremental efforts that produces several incorrect specs before reaching an acceptable spec. Incorrect specs are the source of what we dub *bad-instances*: instances that do not align with the informal spec that the user is trying to formally model. It is critical for a visualizer to faithfully render bad-instances, since the user needs to see them and realize that the spec is incorrect.

Faithful rendering is not an issue with default-graphs: for all their weaknesses described earlier in this paper, to their credit, instances and bad-instances alike are displayed as directed graphs, with no information lost or hidden (except as stated in an explicit user directive). In contrast, a custom visualization needs to be sensitive to bad-instances and not misrepresent



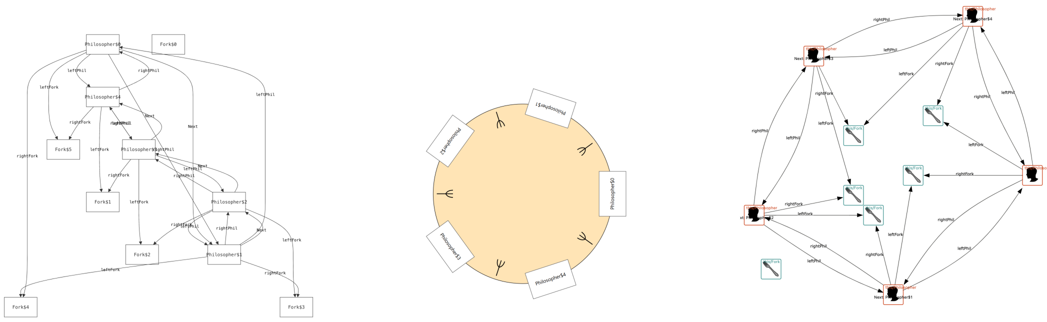
(a) Default-graph. **(b)** Sterling-with-D3.

```
Error: The instance being visualized
is inconsistent with layout constraints.

Constraint:
ENSURE: Node3 is to the left of
        Node2
conflicts with the following constraints:
ENSURE: Node2 is above Node3
ENSURE: Node2 is to the left of
        Node3
ENSURE: Node2 is above Node3
```

(c) bt-dag CnD reports an error.

Figure 10 Outputs for a bad-instance of a binary search tree spec. The instance is a DAG with two nodes. The custom Sterling visualization misleadingly draws a tree.



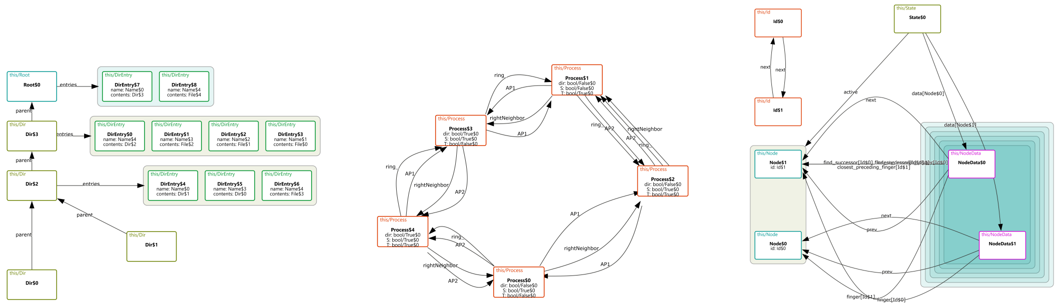
(a) Default-graph (6 forks). **(b)** Sterling-with-D3 (5 forks). **(c)** phil-inv CnD (6 forks).

Figure 11 Outputs for a bad instance of a dining-philosophers spec. The instance has 6 forks but only 5 philosophers. Only CnD shows the extra fork and a roughly circular layout.

them or, worse, accidentally suppress the way in which they are bad (so the user may never discover that the spec is flawed). Nelson et al. [32] discuss this in their study of custom Sterling-with-D3 visualizations. They find that as users modify their spec, domain-specific visualizations would behave unexpectedly, leading to confusion and frustration for users.

This problem is not limited to students. Consider these two examples writen by people who are expert in both formal modeling and Sterling:

- A common underconstraint when modeling a binary search tree is to forget to enforce that a node's left and right children are distinct. Figure 10 shows the result of applying an expert-authored visualization to a bad instance: a DAG. The custom visualizer is not robust to this possibility and thus masks an important failure. The default-graph reveals the underlying structure, while CnD does even better, displaying an error message about the inconsistency between the bad-instance and diagramming constraints.
- An expert-authored visualization of the dining philosophers problem from the Sterling website [10] also fails silently when presented with bad-instances. A common underconstraint when modeling the dining philosophers problem is not ensuring that the number of forks equals the number of philosophers. As shown in Figure 11, the expert-written visualization presents the bad-instance as an instance, hiding the presence of an extra fork. The default-graph hides no information, but the lack of semantically meaningful layout means that the sixth fork is not particularly salient. The CnD diagram, in contrast,



(a) filesystem. **(b)** ring-orientation. **(c)** chord.

Figure 12 CnD diagrams for three example specs.

arranges the domain in a meaningful way, and adds icons to distinguish philosophers from forks, making the extra fork easier to see.

Of course, an attentive visualization author can protect against some of these issues with checks, preconditions, and assertions. Besides being onerous and easy to overlook, these suffer from a more fundamental problem: they are limited by the author's assumptions of what can go wrong. That is, they can only handle *known* unknowns: faulty specs that the author thinks of. However, these would probably be turned into properties in the first place. Problems that the author does not think about (*unknown* unknowns)—the very kind for which lightweight formal methods are ideal—are, by definition, impossible to account for, and may well result in misleading or deceptive visualizations. This problem is especially salient in educational contexts, where it is well-documented that experts (if they provide visualizations) suffer from blind spots [29, 30] about what mistakes students might make.

CnD strikes a happy medium between generic and custom visualization. Because it only refines the generic output, it does not hide information. Because the refinements encode domain information, bad instances either fail noisily (Figure 10) or produce a structured diagram (Figure 11). We report on a user study involving bad-instances in Section 5.2.3.

5 Evaluation

We now evaluate CnD. Our evaluation has three parts. In Section 5.1, we examine its ability to generate visualizations for new domains. In Section 5.2, we provide empirical data on the effectiveness of the generated diagrams. Finally, in Section 5.3, we discuss CnD's performance when generating various diagrams.

5.1 Examples

It is unsurprising that CnD can capture well the demands of the examples in Section 2.2, given that these were effectively the "training set." We instead need a distinct "test set." For that, we use examples from the Alloy Models repository, a publicly available community-maintained collection of specs (`github.com/AlloyTools/models`). It is unclear how to *comprehensively* study these; instead, we chose a sample of specs for domains that were sufficiently different from one another and that we understood well (for which we could therefore meaningfully write CnD programs) and assess their effect.

Filesystem: Figure 12a shows a generic filesystem rendered in CnD. It uses an orientation constraint to lay out directories hierarchically, and a combination of grouping and orientation constraints to lay out entries in the same directory. Entry contents and names are shown as attributes. Supplement Figure 4 presents a triad contrasting the default-graph and the CnD program that refined the graph.

Ring Orientation: Figure 12b shows how we can visualize instances of an Alloy spec for the self-stabilization problem in distributed systems [7] for a ring of processes. A cyclic constraint is used to convey the circular topology of processes, while boolean flags are shown as attributes. Supplement Figure 5 presents a triad.

Chord: Figure 12c visualizes an instance of the Chord [41] distributed hash table protocol. This diagram groups active nodes together, and lays out the previous node seen by a node to its direct left. Finally, it uses grouping constraints to show the results of the find_successor and find_predecessor operations. (The multiple subsuming rectangles in this diagram are a limitation of the WebCola library used by CnD, and are used to indicate subsumed groups. One could imagine alternate implementations that collapse these groups into a single rectangle.) Supplement Figure 7 presents a triad.

5.2 User Studies

We now present user studies that examine the effectiveness of CnD:

Q1. Do CnD specs help users understand the instances being explored (Section 5.2.1)?

Q2. Can pictorial directives further enhance spec understanding (Section 5.2.2)?

Q3. Do CnD diagrams make salient key aspects of bad-instances (Section 5.2.3)?

We use the default-graph as a baseline for comparison, for three reasons. First, it is what a user would get by default. Second, it is a fixed entity (up to the choice of specific visualizations), whereas the set of possible custom visualizations is limitless and our choices might be biased. Finally, as noted in Section 4, it does not suppress problems with bad-instances.

For all three studies, we recruited participants from the Prolific platform's pre-defined pool of users identified as having "computer programming skills". Each participant was compensated USD 5 for about 15 minutes of time, and no individual took part in more than one study. Our study was deemed to not be human subjects research by our review board; we nevertheless took reasonable safeguards to protect the participants. The full versions of the study diagrams are available in the supplement.

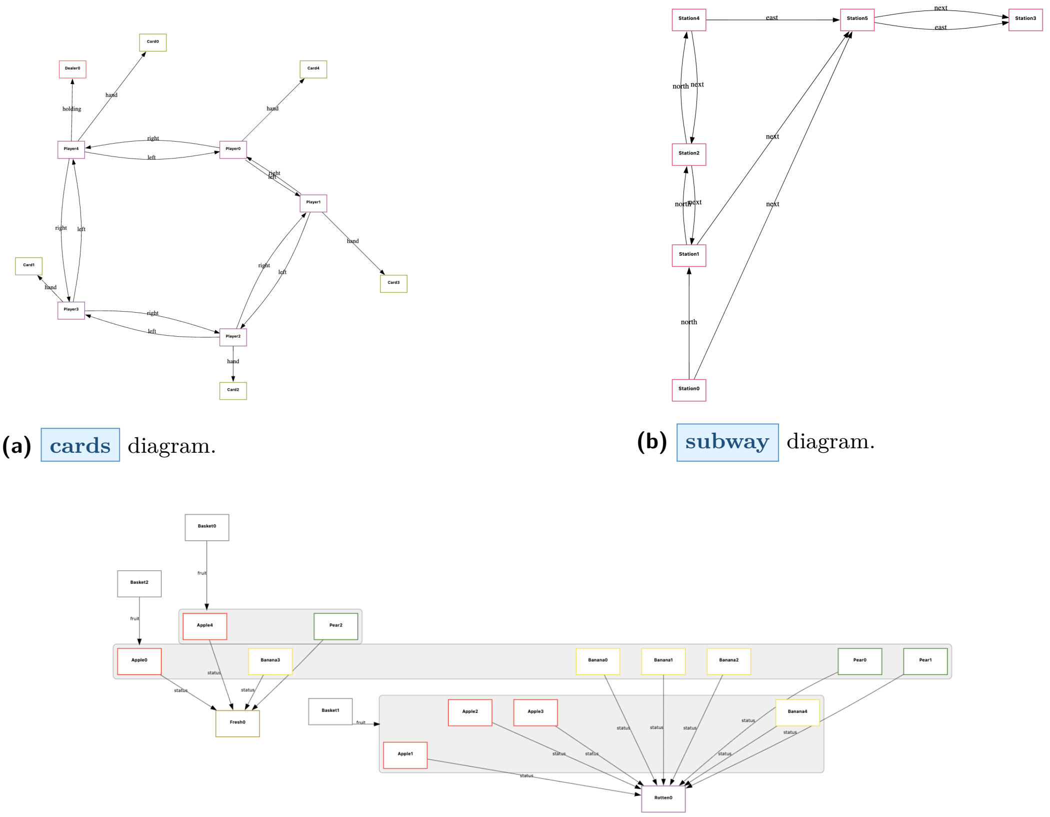
5.2.1 Instance understanding

In order to better understand how CnD diagrams offer insight into spec instances, we designed three specs, each of which lends itself well to one kind of CnD constraint.

- Cards (Figure 13a): This scenario tested the efficacy of cyclic constraints by asking participants questions about the movement of player roles across rounds of a card game.
- Subway (Figure 13b): This scenario tested the efficacy of orientation constraints by asking participants questions about the relative geographical positions of subway stations.
- Fruit (Figure 13c): This scenario tested the efficacy of grouping constraints by asking participants questions about the relative distribution of fruit across baskets.

Users were shown one instance of the spec. For each spec, we developed 2–3 questions that would exercise the user's understanding of the instance they were shown. The specs avoided details that would enable a user to answer the questions based on prior knowledge rather than the instance itself.

Each participant ($n = 38$) was randomly assigned 2 scenarios, one visualized using the default-graph and the other using CnD with *only* the associated constraint. Participants



(a) cards diagram.

(b) subway diagram.

(c) fruit diagram.

Figure 13 CnD outputs for the three Q1 user study scenarios (Section 5.2.1).

shown CnD diagrams were significantly more likely to get questions correct than those shown default-graph visualizations (62.75 % vs 48.04 % correct, $Z = 2.11$, $p < 0.05$), with a small-to-medium effect size ($d = 0.26$). Participants took approximately the same amount of time to complete the tasks (mean time on task: 105.3s for CnD, 100.46s for default-graph, $t = -0.81$, $p = 0.22$).

5.2.2 Pictorial directives

In order to test the efficacy of pictorial directives, we conducted a second study extending the Fruit scenario described in Section 5.2.1. While the scenario remained unchanged, participants (n = 30) were assigned one of two variations of the fruit diagram: one with CnD's default node representation (Figure 13c) and one with images of fruits on certain nodes (Figure 14). (Supplement Figure 8 presents both diagrams together.)

Participants shown the diagram with pictorial directives answered questions correctly more often, and faster than those shown the diagram without them (86.67 % vs 73.33 % correct, mean time-on-task: 56.76s vs 68.55s). However, these were not statistically significant at a 95 % confidence level (correctness: $Z = 1.60$, $p = 0.11$, time-on task: $t = -0.96$, $p = 0.34$).

Participants were then shown the fruit scenario with three different diagram styles: default-graph, default CnD, and CnD with pictorial directives, and asked to rank them in order of preference. Participants tended to rank the CnD with icons diagram highest (mean rank

Table 1 Scenarios used to study the effectiveness of `CnD` constraints on spec understanding.

Scenario	Constraint	Correct Answer Percent		Mean Time on Task	
		default-graph	`CnD`	default-graph	`CnD`
cards	Cyclic	28 %	36 %	161.96s	128.81s
subway	Orientation	31 %	56 %	84.02s	121.73s
fruit	Grouping	69 %	74 %	60.15s	70.32s

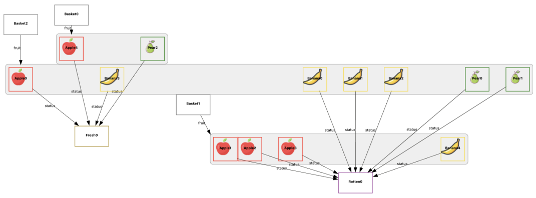


Figure 14 fruit-icons diagram for the Q2 user study (Section 5.2.2).

1.45), followed by the `CnD` with default nodes (mean rank 2.00), and then the default-graph (mean rank 2.57). This difference in ranking was statistically significant (Friedman $Q = 19.27$, $p = 6.55e - 5$). A pairwise comparison showed that the ranking for `CnD` with icons was significantly higher than that of the default-graph ($p = 3.4e - 5$). Participants identified that the `CnD` with icons diagram conveyed the same information as `CnD` with default nodes, but still showed a slight preference for the former ($p = 0.07$). Their feedback praised the "clarity" of the "structured layout", and that the icons "made relationships more visible and easy to understand" and their absence made it "harder to visualize the elements quickly", while default-graph had "complex" layout and made it "difficult to find what they are looking for".

5.2.3 Bad-Instances

To identify *bad* instances, the participant must have some intuitive sense of what constitutes a *good* instance. To avoid the confound of training (which some participants may do only half-heartedly), we picked two situations that users are likely to be able to implicitly judge: the game Tic-Tac-Toe and a human face (Figure 1 (B), *excluding* the icons (C), which we felt would introduce a confound). For each scenario, we constructed a valid and invalid instance and visualized them using the default-graph and `CnD`. The Tic-Tac-Toe instances are shown in Figure 15, while the face instances (similar to Figure 1) are in Supplement Figure 10.

Out of n = 40 participants, half of them saw `CnD` diagrams for Tic-Tac-Toe and default-graph diagrams for the Face scenario, while the other half saw the reverse. For each scenario, participants were first shown a diagram of a good instance, asked to judge its validity, and then asked a follow-up question about the scenario that required them to parse the diagram's structure. This helped build familiarity with the model, scenario, and diagramming style. Participants were then shown a diagram of a bad-instance, and asked to judge its validity. (To be clear, participants did not *know* they were being shown instances in this order: they were simply passing judgment on two diagrams. But we showed the valid instance first to avoid confusing them.)

We divide responses into two groups:

- Validity judgments: The participant asserted that the bad-instance diagram was either valid or invalid. Participants who said it was invalid were asked to explain why, while

Table 2 Scenarios used to study the understanding of bad-instances.

Scenario	Good Instance	Bad Instance
Tic-Tac-Toe	3x3 grid of Xs and Os representing a Tic-Tac-Toe board (**ttt**).	9 node graph marked with Xs and Os that do not form a grid (**ttt-inv**).
Face	Facial features arranged to match a human face (**face**).	Facial features arranged to align one eyebrow with the eyes (**face-inv**).

those who asserted it was valid were asked the same follow-up question as before.

- Uncertain responses: The participant was unsure about the diagram's validity. These participants were asked why they were uncertain.

Uncertain Responses: It is tricky to interpret uncertainty. If a user is spot-checking a spec, uncertainty may arouse their suspicions, leading them to investigate further. On the other hand, if a user is browsing a spec, they may simply move on to the next instance. However, we did not examine this in depth because relatively few participants expressed uncertainty about validity. Only 5 participants (12.5 %, mean time = 37.84s) were unsure about the validity of CnD diagrams, while 3 participants (7.5 %, mean time = 29.52s) were not sure about the validity of default-graphs.

Validity Judgments: Participants were significantly more likely to identify bad-instances and correctly explain *why* they were bad when shown CnD diagrams than when shown the default-graphs (71.43 % vs 43.24 %, $p = 0.03$, $\chi^2 = 4.73$). Participants also made these validity judgments faster with CnD (mean of 58.06s vs. 69.82s), but this difference was not statistically significant ($p = 0.37$, $U = 568.00$).

5.2.4 Threats to Validity

Several factors may limit the generalizability and interpretation of our findings. While study participants had prior programming experience, this does not imply familiarity with formal methods, Alloy, or instance visualization. Effects on understanding could differ for users with more experience in these areas [24]. We believe, however, that this wide pool of participants might help indicate CnD's utility across several kinds of programmers.

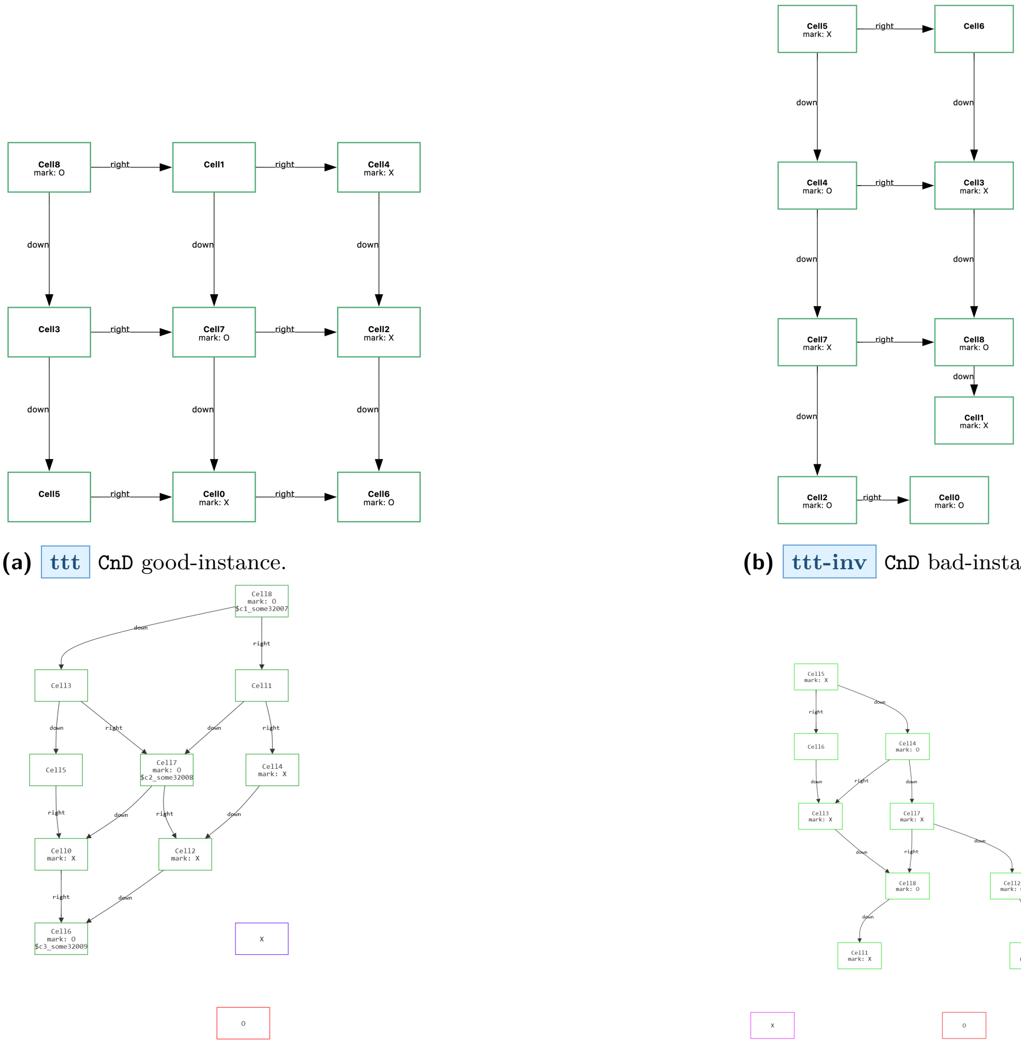
While we attempted to control for domain expertise by selecting scenarios that were either novel (Sections 5.2.1 and 5.2.2) or universally familiar (Section 5.2.3), different kinds of expertise could affect understanding. For instance, the ability to interpret a diagram might be different if the interpreter is also the author of the spec.

The relatively simple and well-understood nature of study scenarios might also limit how well our results generalize to "real world" formal methods scenarios. A bad instance, for example, might be less salient when dealing with more complex or less familiar domain.

Finally, as with any online study, there are inherent threats related to participant engagement and motivation, which could influence the outcomes. We believe that these limitations do not invalidate our findings, but they do warrant caution in interpreting the results as directly applicable to experienced Alloy users in real-world settings.

5.3 Performance

The Cassowary solver used by CnD to determine constraint satisfaction is guaranteed to find a solution if one exists [1]. This means that for any set of constraints that admit a valid visualization, CnD will always produce a solution. To assess performance, we measured the average time (over 50 runs) taken to generate diagrams for all 20 scenarios referenced in



(a) ttt CnD good-instance.

(b) ttt-inv CnD bad-instance.

(c) default-graph good-instance.

(d) default-graph bad-instance.

Figure 15 Good and bad Tic-Tac-Toe instances from the Q3 bad-instance study (Section 5.2.3).

this paper and supplementary material. All measurements were performed on a 2.30 GHz processor (AMD Ryzen 7) with 16GB memory. With the exception of chord, face, and face-inv, which took 0.98s, 1.03s, and 1.06s, all the others took half a second or less. All times are within range for interactive use. Constraint solving took under 50ms in each case; the rest of the time was spent on graph layout. Supplement Section 4 presents full numbers.

6 Related Work

CnD sits between default visualizations, which lack domain-specific semantics, and programmatic tools, which require users to define every aspect of the diagram from scratch:

1. Generic visualizers support all instances (Section 6.1). While they are flexible and broadly applicable, their lack of domain-specific insight can lead to confusing visualizations (Section 3.1).
2. Drawing and diagramming tools offer users fine-grained control over visualizations,

enabling them to craft detailed domain-specific diagrams (Section 6.2). This control demands significant effort. Users have to manage the nuts-and-bolts of diagram creation, dealing not only with domain-level constructs, but also actual "drawing" primitives (e.g., lines, shapes, points).

6.1 Default-graph Visualizers

The Alloy Visualizer [16] is the default visualizer for Alloy instances. It generates directed graph visualizations of the spec instance, with nodes representing atoms and edges representing relations between them. Users are provided a range of theming options, such as being able to change the color or shape of graph nodes or hide nodes via projection. When Alloy's magic-layout feature [39] is enabled, it can even adjust its theming and layout heuristically. Users, nevertheless, struggle to make sense of these as specs grow in size and complexity [24].

Sterling [11] is a web-based variant of the Alloy Visualizer that is designed to make large instances more comprehensible. Nodes and edges are laid out hierarchically, in an effort to reduce cognitive load, and layouts are consistent across instances, making the differences between them more salient.

We group both these tools together as default-graphs—generic visualizers that do not encode any domain-specific knowledge about the spec being visualized. Although both Alloy and Sterling provide numerous theming options, users cannot exercise control over *layout* decisions. On the other hand, the simple node-edge representation makes clear how domain constructs map to visual elements (*representation salience*), preserves domain terminology (*vocabulary correspondence*), and remains robust to bad-instances.

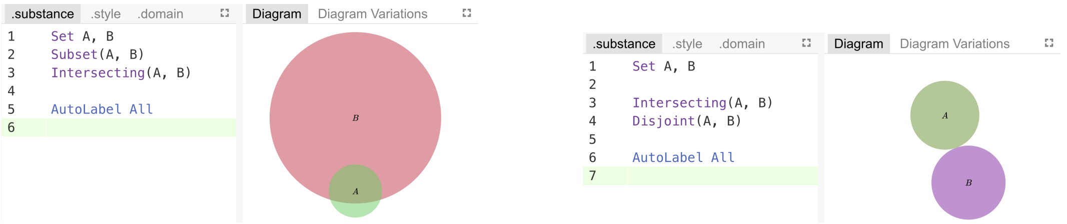
Of the prior work on Alloy visualization, `CnD` is most related to Couto et al. [6], which proposes *layout managers* for Alloy. For example, one might say that parking spaces should be arranged in a linear order with corresponding cars above. This work covers a larger set of primitives than `CnD`, such as grid and tree layouts. However, orientation appears to be defined by a single relation for each sig, which prevents expressing, e.g., the bidirectional layout of Figure 9. It is also unclear how layout managers compose (Figure 1) and how conflicts (Section 3.2) would be resolved. These works are thus complementary, with Couto et al. focused on expressive breadth, and `CnD` on composition and bad-instance discovery.

6.2 Drawing and Diagramming Tools

Sterling-with-D3: In addition to its default layout, Sterling also allows users to define custom visualizations using the D3 visualization library [3]. D3 is a powerful, expressive JavaScript library that gives users a great deal of control over the appearance of their diagrams. This enables diagrams that use a wide range of domain-specific visual encodings to represent relationships between entities. This complexity, however, comes at a cost. We observe barriers to adoption of D3 in Section 2.4.

Penrose: Penrose [50] is a domain-specific language designed to build diagrams of mathematical objects. The language's design is informed by both Gestalt principles (Section 2.1.1) and the diagramming tool requirements described by Ma'ayan et al. [22].

Penrose introduces a structured approach to the diagramming process by requiring users to separate the content of their diagram from how it is drawn. Diagrams, therefore, are constructed as three sub-programs: domain, substance, and style. The domain specifies the rules of the mathematical object being diagrammed, the substance specifies the instance of the object being visualized, and the style specifies how the object should be visualized.



```
Set A, B
Subset(A, B)
Intersecting(A, B)

AutoLabel All
```

(a) Penrose diagram of a bad-instance. Set A is a subset of set B in the spec but not in the visualization because the style file assumes that intersecting sets cannot subsume one another.

```
Set A, B

Intersecting(A, B)
Disjoint(A, B)

AutoLabel All
```

(b) Penrose diagram of a nonsensical instance in which sets A and B are declared to be intersecting and disjoint. This produces output with no indication of an error.

Figure 16 Two misleading Penrose diagrams.

This separation of concerns makes Penrose a very powerful diagramming language, with the ability to generate diagrams across a wide range of domains.

Penrose substances and domains can be thought of as analogous to specs in formal modeling languages. A Penrose domain is similar to an Alloy spec, while a Penrose substance corresponds to an instance of that spec. In contrast, “style files” are more akin to programs. Users must describe how diagram elements should be constructed from rendering primitives (e.g., lines, shapes, points) before they can write higher-level constraints on how these elements should be laid out. Given the choice of rendering primitives, a user may even have to specify the order in which higher-level constraints are applied.

This represents a heavy up-front cost. A user must have already built the diagram from the ground up before they can begin to add layout constraints relevant to their domain.

Penrose’s heavy up-front cost disincentivizes its use as an exploratory tool. The Penrose directed graphs demo,[4] for instance, involves nearly 150 lines of style code, including 13 constraints, 4 constraint application phases, and complex visualization calculations (e.g., for the angle of incidence of arrowheads on nodes). Much of this would have to be re-written if the shape of nodes were to change from circles to, say, squares.

After spending significant time on the nuts-and-bolts of rendering, a Penrose user might plausibly still find themselves looking at a misleading visualization for one of two reasons:

1. Since users have to build visualizations from scratch, Penrose diagrams may be brittle to bad-instances (Section 4). We show an example of this in Figure 16a, based on the Penrose Euler Diagrams demo.[5]
2. Penrose’s numerical optimization problem is designed to always try to find a solution. This means that the system can, misleadingly, generate diagrams even when some user-specified constraints are not satisfied (Figure 16b).

BlueFish: Bluefish [37] is a declarative diagramming framework inspired by the principles of component-based UI frameworks. The core Bluefish primitive is the *relation*, which is used to capture semantic associations between diagram elements. Bluefish is particularly well suited to diagramming because these relations are composable and extensible. For example, relations are a natural way to define a diagram where elements overlap with one another.

However, Bluefish relations are defined in terms of concrete diagram elements, rather than abstracting over their types. As a result, an author who wishes to *dynamically* populate

[4] https://github.com/penrose/penrose/tree/main/packages/examples/src/box-arrow-diagram
[5] https://penrose.cs.cmu.edu/try/?examples=set-theory-domain%2Ftree-euler

relations must fall back on Bluefish's host language (JavaScript). This is especially pertinent in the context of model-finding tools, like Alloy and Forge, where the exact elements appearing in an instance (or bad-instance) are not known in advance, and where the author may be unfamiliar with JavaScript.

6.3 Diagramming for Formal Methods

We note that multiple other formal methods tools also provide some form of diagramming. While in some cases very powerful, we are not aware of any of them providing *lightweight* diagramming in the manner of `CnD`.

Pro B: The Pro B [21] suite provides multiple tools (e.g., BMotionWeb [19] and VisB [46]) for visualizing traces. Moreover the Alloy2B [18] project can translate a rich subset of Alloy to B, meaning that Pro B can be used for some Alloy visualizations. For example, BMotionWeb [19] enables interactive, domain-specific visualizations. Creating these visualizations requires some knowledge of web programming. In VisB [46], users modify attributes of a base SVG image by linking elements of the image to aspects of the underlying spec.

Lean: The Lean [27] proof assistant provides domain-specific visualizations for structures like graphs, trees, etc. [25, 31] and for proofs as well [49]. Creating new visualizations with these libraries requires either familiarity with JavaScript (as Sterling does) or with a diagramming tool such as Penrose (Section 6.2).

GUPU: GUPU [33] is a pedagogic tool for visualizing Prolog substitution via domain-specific *viewers*. Students can use these graphical tools alongside tailored feedback and testing capabilities to better understand Prolog solutions. GUPU's viewers are written in Prolog [34], and are capable of generating sophisticated diagrams. Unlike `CnD` specs, however, GUPU viewers are heavyweight. Authors must invest significant time to get started, specifying each diagram construct, the minutiae of layout, and reconstructing the relationships between diagram elements. This is a significant barrier to entry: custom viewers presented by the GUPU authors [34] are Prolog programs that are much more complex than the Prolog programs they visualize.

6.4 Layout Tools

`CnD` builds heavily on two libraries for layout:

- WebCola [8] is a JavaScript library designed to support constraint-based graph layout [9]. WebCola constraints are specified in terms of a graph's nodes, and can be used to control node alignment, separation, and grouping. These constraints, however, are always "soft". This means that WebCola will always generate a layout, even if the constraints are not satisfied. Nevertheless, WebCola is still useful for layout, so `CnD` uses WebCola *after* constraints are deemed consistent and satisfiable (see Section 3.2).
- DAGRE is a JavaScript library for generating directed graph layouts, organizing nodes into hierarchical layers to minimize edge crossings and improve readability [36]. While DAGRE allows users to exercise some control over graph layout (e.g., specifying the direction of graph flow and node separation), users are unable to control the layout of individual nodes or edges. DAGRE's focus on hierarchical layout means that it is not well suited to more general graph layouts (e.g., cyclic graphs). Taking inspiration from Sterling, `CnD` uses DAGRE if no layout constraints are specified.

7 Limitations

Since CnD acts as a refinement on default-graphs, it inherits many of the same limitations. For instance, layout constraints may not be applicable to non-functional, non-hierarchical relations. Consider the example of an undirected tree, where each edge between nodes takes the form of a symmetric relation (Figure 9). Any orientation constraint on symmetric relations is necessarily inconsistent, so CnD cannot enforce a layout based on tree edges.

Furthermore, CnD constraints cannot add *new* information not reflected in the default-graph (though pictorial directives can). The instance in Figure 9 does not contain any information on which node is the parent of another. Thus, CnD alone cannot dictate which node is visualized as the root of the tree. Doing so without building a custom visualization requires somehow indicating the root the tree by, for instance, a field or sig in the spec.

A silver lining here, however, is that the worst-case scenario for CnD is the default-graph. As we have shown in Section 3.5, even in the absence of any CnD primitives, its output likely leads to less cluttered, more informative diagrams than the default-graph. Thus, CnD diagrams should not be less useful than default-graphs.

8 Discussion

Once we strip away the finery, the effectiveness of many custom visualizations boils down to a small number of key spatial relationships. We found that these relationships can be captured by a small set of primitives, which we encoded into a lightweight, declarative language. We discuss some of the lessons we have learned from this work below.

"Useful but not Pretty" diagrams: While many visualization tools and frameworks focus on drawing, with an emphasis on high visual quality, much of the value of a diagram comes from its ability to preserve key semantic relationships. Cope-and-Drag diagrams are often less visually appealing than those generated by D3 or Penrose, but help effectively communicate the model's structure while avoiding the pitfalls of focusing too much on aesthetic concerns.

Diagramming by Refinement: Specifications versus Programs: There is a fundamental divide between the practices of writing programs and writing specs. In personal communication, Daniel N. Jackson (the lead architect of Alloy) once explained to the last author the difference between a program and a spec: the empty program exhibits no behaviors, while the empty spec admits all behaviors. Roughly speaking, growing a program typically leads to additional behaviors, whereas growing a spec (by writing constraints) typically leads to fewer behaviors.

We believe a similar divide applies to diagramming. The Alloy Visualizer behaves like a spec: the default visualizer shows everything. As this paper has noted, this genericity and domain-independence causes problems (Section 2.2) but also has virtues (Section 4). At the other extreme are systems like D3. By providing full control, they can address some of the weaknesses we have noted (Sections 6.1 and 7). However, the user gets nothing for free: every piece of output requires programmer effort. That puts them on the program end of the program–spec split. Systems like Penrose also fall at this end, even though they provide more structure to the program and thus enable some separation of concerns.

In this typology, CnD is "spec," not "program." It consciously alters the default-graph output, with the empty CnD program leaving the output unchanged. Adding constraints and directives shapes and sometimes limits the amount of output and the number of specs or instances that can be displayed. Thinking of diagramming as spatial refinement also provides design insight, both into where a CnD-like spec can live (Section 3.4) and how errors can be handled by analogy to types (Section 3.2).

Floors and Ceilings: The Value of Lightweightness: Another useful typology is that introduced by the Scratch programming language [23], of *floors* and *ceilings.* Scratch presents itself as having a low floor and high ceiling: it's very easy to start programming but there is little limit to how sophisticated a program one can write. Without dwelling on Scratch's claims, we can view the systems we have discussed through the same lens. Default-graphs have the lowest floor of all—one needs to do nothing to obtain output—but also have a very low ceiling. Systems like D3 and Penrose have an arbitrarily high ceiling, but also an elevated floor (which in D3 is known to cause difficulties for some users; we are not aware of much evidence either way about Penrose). We would classify `CnD` as having the lowest of floors (since the program can be empty, still producing default-graph output), but also a moderate ceiling: it does some things very well, but other things (Section 7) poorly relative to a custom visualization.

Showing Erroneous Output: In Section 3.2 we discuss how `CnD` handles errors. In particular, in both cases, it refuses to produce a diagram. This avoids creating output that might mislead the user. However, it is unclear that this option is ideal. The entire premise of this paper is that seeing visual output is useful for knowledge discovery, including for spotting problems. Similarly, there are situations where producing "bad" output can quickly help a user spot why a bug in the *`CnD` program* (or, more subtly, its relationship to the spec). Providing no output at all cuts off that avenue.

For this reason, we believe it would be worthwhile to investigate a mode where we produce visual output even in the case of errors, but make salient which elements should have prevented any output from occurring. We believe there are important algorithmic, cognitive, and presentational considerations. Algorithmically, we have constraints that cannot be satisfied; we have to somehow identify constraints to break to even generate output, and then label which parts of the output are invalid. Cognitively, there are surely factors that influence which constraints to choose (which may not always be the "smallest"). Presentationally, we have to both make salient that the user is viewing an invalid instance and highlight which parts of are invalid.

Accessibility for the Visually-Impaired: Like the vast majority of data-visualization tools, the Alloy Visualizer relies primarily on modalities that are inaccessible to visually-impaired users [13]. `CnD` does not reduce the accessibility of the Alloy Visualizer. However, the encoding of essential spatial relationships in `CnD` specifications presents an opportunity to improve how visually-impaired users are able to interact with default output. In particular, the `CnD` constraints may lead to improved output: e.g., exploiting spatial audio effects for sonification [4, 35, 51].

Conclusion: `CnD` is not meant to be a last word in the diagramming conversation, but rather an interesting point in the design space. It is unclear how to make the ceiling for `CnD` much higher, at least not without "lifting" the floor. This is very difficult in the current design (which consciously modifies default-graph output) and may cause problems such as masking bad-instances. Instead, we believe there is value to having other languages that can provide other floor–ceiling ranges.

References

1 Greg J Badros, Alan Borning, and Peter J Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

2 Leilani Battle, Danni Feng, and Kelli Webber. Exploring D3 implementation challenges on Stack Overflow. In *2022 IEEE Visualization and Visual Analytics (VIS)*, pages 1–5. IEEE, 2022.

3 Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

4 Stanley J Cantrell, Bruce N Walker, and Øystein Moseng. Highcharts sonification studio: an online, open-source, extensible, and accessible data sonification tool. In *Proceedings of the International Conference on Auditory Display*, volume 2, 2021.

5 Hsuanwei Michelle Chen. Information visualization principles, techniques, and software. *Library Technology Reports*, 53(3):8–16, 2017.

6 Rui Couto, José Creissac Campos, Nuno Macedo, and Alcino Cunha. Improving the visualization of Alloy instances. In Paolo Masci, Rosemary Monahan, and Virgile Prevosto, editors, *Proceedings 4th Workshop on Formal Integrated Development Environment*, volume 284, pages 37–52, 2018. doi:10.4204/EPTCS.284.4.

7 Edsger W Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

8 Tim Dwyer. cola.js: Constraint-based layout in the browser, 2017. Accessed: 2024-12-02. URL: https://ialab.it.monash.edu/webcola/.

9 Tim Dwyer, Kim Marriott, and Michael Wybrow. Topology preserving constrained graph layout. In *Graph Drawing: 16th International Symposium, GD 2008, Heraklion, Crete, Greece, September 21-24, 2008. Revised Papers 16*, pages 230–241. Springer, 2009.

10 Tristan Dyer. Sterling JS demo, 2024. Accessed: 2024-11-12. URL: https://sterling-js.github.io/demo/.

11 Tristan Dyer and John Baugh. Sterling: A web-based visualizer for relational modeling languages. In *International Conference on Rigorous State-Based Methods*, pages 99–104. Springer, 2021.

12 Tristan Dyer, Tim Nelson, Kathi Fisler, and Shriram Krishnamurthi. Applying cognitive principles to model-finding output: the positive value of negative information. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–29, 2022.

13 Frank Elavsky, Lucas Nadolskis, and Dominik Moritz. Data navigator: an accessibility-centered data navigation toolkit. *IEEE transactions on visualization and computer graphics*, 30(1):803–813, 2023.

14 Robert Goldstone. An efficient method for obtaining similarity data. *Behavior Research Methods, Instruments, & Computers*, 26:381–386, 1994.

15 D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.

16 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2nd edition, 2012.

17 Kurt Koffka. Perception: an introduction to the Gestalt-Theorie. *Psychological bulletin*, 19(10), 1922.

18 Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. A translation from Alloy to B. In *Conference on Abstract State Machines, Alloy, B, and Z*, pages 71–86, 2018. doi:10.1007/978-3-319-91271-4_6.

19 Lukas Ladenberger and Michael Leuschel. BMotionWeb: A tool for rapid creation of formal prototypes. In *Software Engineering and Formal Methods*, pages 403–417, 2016. doi:10.1007/978-3-319-41591-8_27.

20 Jill H Larkin and Herbert A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987.

21 Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *International Symposium on Formal Methods (FM)*, 2003. doi:10.1007/978-3-540-45236-2_46.

22 Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. How domain experts create conceptual diagrams and implications for tool design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2020.

23 John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

24 Niloofar Mansoor, Hamid Bagheri, Eunsuk Kang, and Bonita Sharif. An empirical study assessing software modeling in Alloy. In *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 44–54. IEEE, 2023.

25 Lean Manual. The user-widgets system. https://lean-lang.org/lean4/doc/examples/widgets.lean.html, 2024. [Accessed Nov 19, 2024].

26 Vajih Montaghami and Derek Rayside. Bordeaux: A tool for thinking outside the box. In *Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 20*, pages 22–39. Springer, 2017.

27 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer International Publishing, 2021. doi:10.1007/978-3-030-79876-5_37.

28 Lekha Nair, Sujala Shetty, and Siddhanth Shetty. Interactive visual analytics on big data: Tableau vs D3.js. *Journal of e-Learning and Knowledge Society*, 12(4), 2016.

29 Mitchell J. Nathan, Kenneth R. Koedinger, and Martha W. Alibali. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the Third International Conference on Cognitive Science*, pages 644–648, 2001. URL: https://api.semanticscholar.org/CorpusID:14779203.

30 Mitchell J Nathan and Anthony Petrosino. Expert blind spot among preservice teachers. *American Educational Research Journal*, 40(4):905–928, 2003.

31 Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An extensible user interface for Lean 4. In Adam Naumowicz and René Thiemann, editors, *Interactive Theorem Proving*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2023/18399, doi:10.4230/LIPIcs.ITP.2023.24.

32 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. Forge: A tool and language for teaching formal methods. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):613–641, 2024.

33 Ulrich Neumerkel and Stefan Kral. Declarative program development in Prolog with GUPU. *arXiv preprint cs/0207044*, 2002.

34 Ulrich Neumerkel, Christoph Rettig, and Christian Schallart. Visualizing solutions with viewers. In *LPE*, pages 43–50, 1997.

35 Sandra Pauletto and Andy Hunt. A toolkit for interactive sonification. In *ICAD*, 2004.

36 Chris Pettitt and contributors. Dagre: A JavaScript library for directed graph layouts. https://github.com/dagrejs/dagre, 2014. Accessed: 2024-11-21.

37 Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. Bluefish: Composing diagrams with declarative relations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–21, 2024.

38 Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

39 Derek Rayside, Felix Sheng-Ho Chang, Greg Dennis, Robert Seater, and Daniel Jackson. Automatic visualization of relational logic models. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 7, 2007. `doi:10.14279/tuj.eceasst.7.94`.

40 Lothar Spillmann, editor. *On Perceived Motion and Figural Organization*. MIT Press, 2012. Includes English translations of Max Wertheimer's 1912 and 1923 papers on Gestalt psychology.

41 Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review*, 31(4):149–160, 2001.

42 J Ridley Stroop. Studies of interference in serial verbal reactions. *Journal of experimental psychology*, 18(6):643, 1935.

43 John Sweller and Paul Chandler. Evidence for cognitive load theory. *Cognition and instruction*, 8(4):351–362, 1991.

44 Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1983.

45 Barbara Tversky. Spatial schemas in depictions. In M. Gattis, editor, *Spatial schemas and abstract thought*, pages 79–112. The MIT Press, 2001.

46 Michelle Werth and Michael Leuschel. VisB: A lightweight tool to visualize formal models with SVG graphics. In *Rigorous State Based Methods*, pages 260–265, 2020. `doi:10.1007/978-3-030-48077-6_21`.

47 Max Wertheimer. Experimentelle studien über das sehen von bewegung. *Zeitschrift für Psychologie*, 61:161–265, 1912.

48 Benjamin W White. Interference in identifying attributes and attribute names. *Perception & Psychophysics*, 6:166–168, 1969.

49 David Wren. animate-lean-proofs, 2021. Accessed: 2024-11-22. URL: `https://github.com/dwrensha/animate-lean-proofs`.

50 Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)*, 39(4):144–1, 2020.

51 Jonathan Zong, Isabella Pedraza Pineros, Mengzhu Chen, Daniel Hajas, and Arvind Satyanarayan. Umwelt: Accessible structured editing of multi-modal data representations. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2024.