Synthesizing Visual Specifications

YILIANG LIANG, Carnegie Mellon University, USA
EUNSUK KANG, Carnegie Mellon University, USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA

Creating custom visualizations is difficult because users must specify layout rules even when they only have an example diagram in mind. We explore an alternative approach: synthesizing visual specifications from user-edited example diagrams. Focusing on relational data and Alloy instances, we outline a workflow where users iteratively correct generated diagrams, and the system infers generalizable layout constraints. We also introduce an early visual-specification language to express these inferred rules. This work sketches a first step toward example-driven visual specification synthesis.

Additional Key Words and Phrases: specifications, visualizations, program synthesis

ACM Reference Format:
Yiliang Liang, Eunsuk Kang, and Joshua Sunshine. 2026. Synthesizing Visual Specifications. In *Proceedings of the 16th PLATEAU Workshop on Programming Languages and Human-Computer Interaction (PLATEAU '26).* 12 pages.

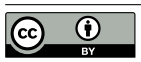
1 Introduction

Visual representations play a central role in how people interpret, explore, and communicate complex information. Across domains – ranging from physics [13] to software engineering [7], from formal methods [14, 18] to knowledge representation in HCI [11] – visualizations have long made complex concepts tangible to human stakeholders. Studies have repeatedly shown that well-designed visualizations can improve comprehension, reduce cognitive load, and support analytical reasoning [5, 8, 9, 19, 27, 28, 30]. Yet, despite their usefulness, creating effective visual representations remains a challenging and error-prone activity.

A key difficulty lies in the **design of custom visualizations** tailored to the semantics of a specific domain. Unlike general-purpose diagramming tools, domain-specific visualizations must depict the particular visualization styles unique to a given domain. Literature hints at its importance [14, 18], and indeed, practitioners have designed a variety of languages and tools (e.g., Vega-Lite [25], Charticulator [22], Penrose [34], Cope-and-Drag [20], etc.) that allows customizations through DSL programs or GUI operations. Users of these tools author **visual specifications**, which link the abstract concepts in the data to visual and spatial properties. Empirical studies show that, indeed, these customizations help stakeholders better understand the complex concepts behind them [16].

Despite their expressive power, **authoring such visual specifications is hard** in practice. While DSL- and GUI-based interfaces reduce the barriers to creating custom visualizations, they do not eliminate the underlying conceptual burden of authoring visual specifications. Users must still explicitly indicate, for example, which parts of the data maps to which visual channels.

Authors' Contact Information: Yiliang Liang, Carnegie Mellon University, Pittsburgh, PA, USA, yiliangl@andrew.cmu.edu; Eunsuk Kang, Carnegie Mellon University, Pittsburgh, PA, USA; Joshua Sunshine, Carnegie Mellon University, Pittsburgh, PA, USA.



PLATEAU '26, Pittsburgh, PA

This paper proposes an alternative approach: the **synthesis of visual specifications from user-given examples and interactions**. In general, writing specifications is difficult, but when users work through concrete examples and imagine what should happen visually to these specific cases, the task becomes easier. These concrete examples serve as "partial specifications" which a synthesizer may exploit to generate visual specifications that satisfy these examples. Prior research [15, 35, 36] has shown that example-driven synthesis of specifications is viable. In this paper, we offer a preliminary exploration of how similar ideas might extend to visual specifications. We sketch a tentative design for an example-guided synthesis workflow, applied to visualizations of formal modeling.

2 Background, Motivation, and Related Work

2.1 Visual specifications

Fig. 1 illustrates the role of visual specifications within the broader visualization pipeline. This paper uses the term "data" broadly: It includes not only conventional tabular data, but also relational structures and higher-level domain concepts. For example, in a chemistry domain, the arrangement of atoms that constitute a molecule is itself data. A visual specification mediates between such data and the visualization tools responsible for producing the final diagrammatic representation. These specifications describe the "recipes" on how elements of the data should be translated into their visual forms, and they allow users to customize visualizations.

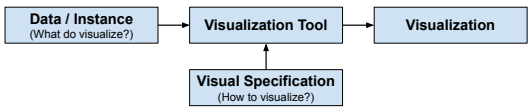


Fig. 1. A visual specification serves as the bridge between data and visualizations. It encodes how elements of the data should map to elements of the visualization. The visualization tool applies the visual specification to the data to generate the final diagrammatic representation.

2.2 Visual specification languages

A long line of research has explored how to most efficiently express a user's visualization intent through visual specifications. The foundations of this area trace back to Wilkinson's seminal *The Grammar of Graphics* book [33] which outlined the explicit fundamental components of visualizations. A number of modern systems (like Vega-Lite [25]) have since implemented Wilkinson's work into concrete visual specification languages and tools. Similarly, Wickham [31] devised a grammar for statistical visualizations, implemented by ggplot2 [32].

Tools like Lyra [23], Lyra 2 [37], and Charticulator [22], then, introduce GUIs on top of these textual languages, where users construct visualizations through direct manipulation. These tools expose interactions such as dragging a data field onto a visual channel. Under the hood, each GUI interaction updates an underlying declarative specification (e.g., a Vega-Lite program, or a structured constraint-based representation).

Many visualization tasks center on relational data, structured as entities and relations between entities, such as graphs, networks, and hierarchies. Penrose [34] creates custom visualizations for relational data through its declarative visual specification language known as *style*, which translates entities and relations in the data into shapes and geometric constraints between shapes. For example, a Penrose user can write a *style* program to the effect of "Whenever two entities x and y are related by relation R, the shape of x must contain the shape of y." The Penrose engine would collect these constraints and generate a diagram that satisfies them.

2.2.1 Visual specifications for formal modeling. Visualizations of relational data play an important role in many formal models, which represent complex systems in logic and help stakeholders

understand, explore, and verify the correctness and reliability of system design. In short, users write a model (an abstract description of the system), and formal modeling tools generate valid instances of the model that capture system behaviors. Despite their perceived utility, literature has shown that formal models can be hard to learn, understand, construct, and validate [12, 17, 29].

Grounded in the belief that visualizations make complex subjects more tangible, some formal modeling works have visualization support. For example, the Alloy modeling language [10] has a built-in instance visualizer which renders entities in the instance as nodes and relations as edges. Prior studies [14, 18] showed that these visualizations indeed help stakeholders understand, validate, and communicate their formal models.

These studies, however, also revealed that such generic visualizations are often insufficient, especially when stakeholders reason with rich, domain-specific representational conventions. One participant (P6) from prior work [14] used Alloy to model industrial railway systems, and complained about how Alloy's visualizations don't look like their industry-standard visualizations. Such a need inspired a substantial body of work on customizing visualizations for formal modeling. Alloy's own GUI-based "theme customization" feature supports simple customizations such as hiding certain types of objects and rendering certain types of objects as customized shapes. Other tools offer deeper forms of domain specificity. Penlloy [14, 18] translates Alloy models and instances into Penrose [34] programs, where users write a Penrose *style* language to customize visualizations. Sterling [6] supports domain-specific visualizations through D3-based [2] JavaScript programs, while Cope-and-Drag (CnD) [20] provides a DSL for authoring light-weight custom visualizations.

2.3 Difficulty of authoring visual specifications

These visual specification languages and systems do not, however, eliminate the conceptual burden of creating custom visualizations. Each of the tools mentioned for custom visualizations of formal models – Penlloy [14], Sterling [6], and CnD [20] – requires users to write explicit visualization logic, whether in Penrose's *style* language (Penlloy), D3/JavaScript (Sterling), or a custom DSL (CnD). Literature shows that this poses significant barriers. Nelson et al. [16] and Prasad et al. [20] reported that users valued Sterling's expressiveness but struggled with "the time involved in making a custom visualization" and the steep D3 learning curve [3], and feedback on Penlloy [14] highlighted similar concerns of creating custom visualizations "giv[ing] me too much work" (P10).

Even GUI-based visualization tools – such as Alloy's "theme customization" feature, Lyra [23], and Charticulator [22] – do not fully resolve this challenge. GUIs reduce the syntactic burden of programming, but they still require users to explicitly articulate the visualization they want. Creating a stacked layout, for example, still requires explicitly toggling options like "Stack Y" [22]. As Satyanarayan et al. [24] noted, these tools assume that users "come with specific chart design in mind" and simply need convenient ways implementing those designs. But in practice, users often don't have such a design prepared and do not know "what customizations are available and how they might best be applied to the problem at hand" [21].

Ultimately, existing systems do not eliminate the cognitive burden of inventing, specifying, and refining a visual specification. Users must still decide how to encode entities, how to depict relations, how to arrange components spatially, and how to align the visualization with domain conventions. For formal modelers, this becomes an additional layer of specification work atop already difficult-to-interpret formal models. And, crucially, in the early stages of exploration, users often do not yet know what visualization they need, making it especially challenging to articulate mappings or layout rules in advance.

These observations point to a clear research gap: Can we reduce the cognitive burden by generating visual specifications automatically from user-provided example visualizations? This paper presents some of our preliminary work on this problem.

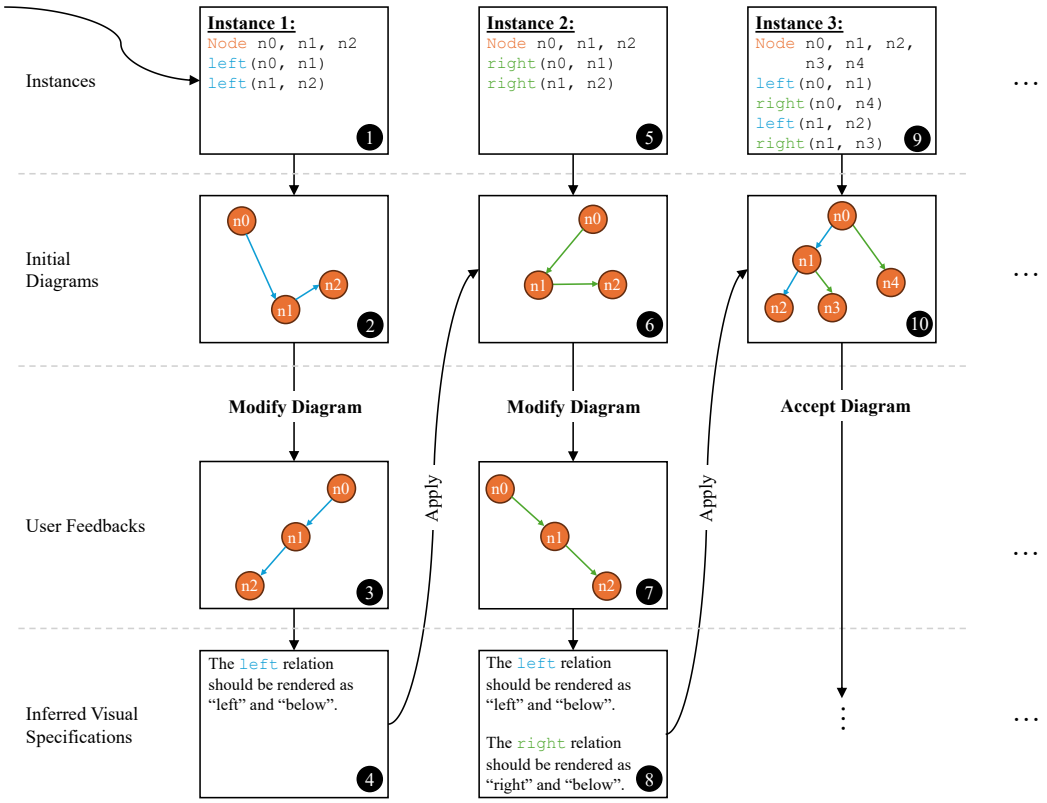


Fig. 2. Example workflow of our proposed system. Starting with an instance, the system generates an initial diagram for that instance. The user then provides feedback on that diagram, which is then used to synthesize a visual specification. The synthesized visual specification is refined with user feedback on multiple instances.

3 Example Workflow and Design Objectives

As early exploratory work, we focus on the visualization of **relational data** generated by relational formal modeling tools like Alloy, for which we aim to generate visual specification programs in the form of, specifically, **layout programs**. Layout programs determine the positions and sizes of the nodes; other stylistic properties such as colors and shape types are, for now, out of scope.

Our proposed system aims to **infer a layout program from light-weight user input in the form of user-provided example visualizations**. Users provide a few illustrative examples of how they want the visualizations to be laid out, and the system generates a layout program that generalizes these examples. Fig. 2 walks through an example workflow of our proposed system, with which we discuss some of the conceptual design objectives guiding this proposal.

3.1 Walkthrough of example workflow

We first introduce an example scenario of visualizing Alloy instances. An Alloy model specifies the types (known as sigs in Alloy) of the entities and relations that the model reasons about, while an Alloy instance instantiates the model with concrete entities and relations. Here, we use an Alloy model for a binary tree. This model consists of a sig `Node` and binary relations `left` and `right` between nodes, which encode information on the left child and right child of nodes. For example,

if $(a, b) \in$ `left`, then b is the left child of a. Instances of this model would specify the concrete entities belonging to the sig `Node` along with the concrete `left` and `right` relations, like ❶, ❺, and ❾ in Fig. 2.

3.1.1 First iteration. The first instance of the model ❶ involves three nodes (`n0`, `n1`, and `n2`) where `n0`'s left child is `n1` and `n1`'s left child is `n2`. The system first generates a layout ❷ in which all elements are placed in an arbitrary initial configuration. This is because at the beginning of the workflow, the system has not yet inferred any visual specifications. The user finds the layout ❷ unsatisfactory as it does not visually reflect the "left child" relations between nodes. Through a direct-manipulation interface, the user modifies the system-generated diagram by dragging the nodes to their desired positions, such that `n1` is to the left and below `n0`, and `n2` is to the left and below `n1`. This results in a user-provided diagram ❸.

Using the user-provided diagram ❸ and the instance ❶, the system concludes the following partial visual specification ❹: "Whenever two nodes a and b are related by the relation `left` in the instance, the shape of b should be to the left of and below the shape of a". Such a partial visual specification can then be applied onto future instances.

3.1.2 Second iteration. We now move onto the next instance ❺, which involves the same three nodes, but now, `n0`'s right child is `n1` and `n1`'s right child is `n2`. Because the previously-generated visual specification ❹ only matches on the `left` relation in the instance which does not appear in the new instance, the system once again generates an arbitrary layout ❻. Once again the user finds it unsatisfactory, and uses the direct manipulation interface to drag the nodes into their desired positions ❼, where `n1` is to the right of and below `n0` and `n2` is to the right of and below `n1`.

With this new diagram ❼ and new instance ❺, along with the previous diagram ❸ and instance ❶, the system refines the generated visual specification with a new rule in addition to the previous rule ❽: "Whenever two does a and b are related by the `right` relation in the instance, the shape of b should be to the right of and below the shape of a".

3.1.3 Third iteration. Now we move onto the third instance ❾ which is more complex, containing five nodes and both `left` and `right` relations. The system applies the previously-generated visual specification ❽ onto this new instance to generate diagram ❿, where indeed, the left children of nodes are rendered on the left and below the parent, and the right children of the nodes are rendered on the right and below the parent. The user finds this diagram satisfactory, so they accept the diagram as the user-provided diagram. And as before, the system uses the instance-diagram pairs to synthesize a visual specification which is then applied to the future instances. This cycle continues until a specified number of instances have been covered or until the user is satisfied with the visual specification that the system generates.

3.2 Design objectives

The example workflow allows us to conclude some design objectives of the proposed synthesis tool.

3.2.1 Light-weight user-provided example diagrams. An important objective is to require users to contribute only minimal, intuitive effort, far less than authoring a full visual specification. The users should not need to write explicit layout rules such as "elements of type T should be stacked vertically" or "if x and y are related by R then x should be to the right of y." Instead, users provide a set of **example instance-diagram pairs** $\{(I_i, D_i)\}_{i=1}^{N}$, where

- An instance I_i (e.g., 1, 5, and 9) is the data the user wants to visualize, generated by some automatic tools like Alloy; and
- An example diagram D_i (e.g., 3, 7, and 10) is a user-provided depiction of how they would like instance I_i to be visualized.

The goal is to generate a visual specification program P that satisfies all instance-diagram pairs.

To construct such an example diagram D_i of instance I_i, the user would start with an automatically generated diagram (2, 6, and 10) of I_i, based on both the instance I_i and the previously-generated visual specification. The user decides whether or not this diagram is satisfactory. If so, the user preserves the diagram (as in 10). If not, the user modifies the diagram into a new diagram (3 and 7) through a direct manipulation interface. Typical operations include repositioning individual elements via dragging and adjusting sizes of individual elements. This workflow provides a deliberately familiar and light-weight mechanism for authoring example diagrams, with interaction mechanisms similar to those in Adobe Illustrator.

3.2.2 Iterative development of visual specifications. Just like any specification tasks, designing a visual specification benefits from an iterative workflow. The user begins by providing an example instance-diagram pair (I_i, D_i). The system generalizes from this pair along with any previously-supplied pairs, to synthesize a provisional visual specification P_i. When a new instance I_{i+1} becomes available the system applies P_i to produce a candidate diagram D'_{i+1}. The user may accept this generated diagram or interactively refine it into the desired D_{i+1}. Such refinements impose new constraints that guide the next synthesis step.

This iterative process allows users to strategically choose which example instances to provide and in what order. For example, tools like Alloy automatically generates instances starting with the simplest ones. Beginning with simpler instances helps users articulate basic visual principles without being overwhelmed by the intricacies of more complex configurations. This incremental approach mirrors the workflows already embraced in formal modeling environments.

3.2.3 Properties of good visual specification programs. When provided with example instance-diagram pairs, the system should generate a visual specification program P that encodes the visual patterns within the example pairs. There are two objective for P to be a good program – **soundness** and **completeness**.

P is **sound** if it enforces constraints that are consistent with (i.e., not violated by) the example instance-diagram pairs. Suppose, for an instance I_i, P requires node x to be positioned to the right of node y. If in the user-provided diagram D_i node x is actually on the left of node y, then P is not sound. The soundness requirement essentially stipulates that the system should never generate a visual specification that directly conflicts with the provided examples. But soundness alone does not suffice – a program that asserts no requirement is vacuously sound, since indeed, it is consistent with the example instance-diagram pairs.

P is **complete** if it captures all intentional patterns by the user across the given examples. This requirement, however, is inherently difficult to formalize because it relies on knowledge of user intentions, and sometimes even the users themselves do not have complete knowledge of their intentions. Consequently, completeness must be interpreted not as an absolute guarantee but as a best-effort approximation – the synthesized program should generalize all example regularities while avoiding overfitting to accidental geometric artifacts or layout choices that do not reflect persistent user intentions.

One way to approximate completeness is to enumerate over a large number of sound programs with respect to the provided examples, to cover as much possible visualization requirements as

$$
\begin{aligned}
\text{Program} &::= \mathit{Clause}^* \\
\text{Clause} &::= \mathit{LayoutConstraint} \mid \mathsf{IF}\ \mathit{Selector}\ \mathsf{THEN}\ \mathit{Clause} \\
\text{Selector} &::= v : \mathit{SomeSig} \mid \mathit{SomeRel}(v_0, v_1, \cdots) \\
\text{LayoutConstraint} &::= \mathsf{Layout}\ v_1\ \mathit{BinaryLayout}\ v_2 \\
&\quad \mid \mathsf{Layout}\ v_1\ \mathit{UnaryLayout} \\
&\quad \mid \mathsf{Cycle}\ v_1\ \mathit{CyclicLayout}\ v_2 \\
\text{BinaryLayout} &::= \mathit{LeftOf} \mid \mathit{RightOf} \mid \mathit{Above} \mid \mathit{Below} \mid \mathit{HorizontallyAligned} \mid \mathit{VerticallyAligned} \\
&\quad \mid \mathit{Contains} \mid \cdots \\
\text{UnaryLayout} &::= \mathit{LeftOfCenter} \mid \mathit{RightOfCenter} \mid \mathit{AboveCenter} \mid \mathit{BelowCenter} \mid \cdots \\
\text{CyclicLayout} &::= \mathit{Clockwise} \mid \mathit{Counterclockwise}
\end{aligned}
$$

Fig. 3. A portion of the abstract syntax of our visual specification language.

possible. If the user does not intend a particular requirement, they may implicitly provide examples that violate that requirement, causing any program encoding the unintended requirement to become unsound and thus eliminated. Of course, this strategy can only succeed if the underlying specification language is sufficiently expressive.

4 Language Design

We now present some of our early work in synthesizing visual specifications for formal methods, starting with the visual specification language that we are working with. Fig. 3 contains the abstract syntax for our visual specification language, which is inspired by the set of features in CnD, identified by Prasad et al. [20] as desirable for formal modeling stakeholders.

4.1 Programs, clauses, selectors, and constraints

A program P consists of clauses. Each clause C encodes a nested list of "if ... then ..." rules, with one or more **selectors** followed by one **constraint**.

Each **selector** matches on some parts of an instance. A *sig-selector* ("v : *SomeSig*") binds the variable v to all instantiated entities of a type *SomgSig* in the instance, while a *rel-selector* ("$\mathit{SomeRel}(v_0, v_1, \cdots)$") restricts the clause to only consider the tuples of entities that satisfy the relation *SomeRel*. This design mimics the selectors in the Penrose *style* language.

At the end of each clause is a **layout constraint** that specifies how the entities matched by the selectors should be visually laid out. We support binary layout constraints that specify the positional relationship between two nodes. For example, "Layout *a LeftOf b*" requires that the shape of the node *a* is to the left of the shape of the node *b*. The constraint "Layout *a Contains b*" requires that a shape *a* contains another shape *b*, useful in visualizing hierarchical relationships. We also support some simple unary layout constraints, namely those that constrain a shape's positional relationship to the center of the canvas. Such constraints are useful for enforcing global layout conventions. For example, to specify that "all objects of type T must appear on the left side of the canvas," we can write "IF v : T THEN Layout v *LeftOfCenter*." Like CnD [20], our language also supports the layout of nodes into clockwise or counterclockwise cyclic structures, useful when visualizing nodes arranged in a cycle, such as in the leader election protocol [4].

4.2 Layout enforcement

To enforce the requirements of a clause to a diagram, our system first evaluates all the clause's selectors to compute a set of tuples to which the clause applies. Then it enforces the clause's terminal layout constraint to the shapes of each tuple.

Each layout constraint is enforced by translating it into constraints between shapes in Penrose [34] through the Bloom framework [1, 26]. For example, the constraint "Layout *a LeftOf b*" translates into Penrose constraint "$a.x < b.x$." For cyclic layout constraints such as "Cycle *a Clockwise b*," our system first gathers all node pairs to which this constraint applies. If a valid cycle exists among these node pairs, the system creates an artificial circle shape c, and generates Penrose constraints so that the nodes, in that specified order, lie along the circumference of c. In the end, the Penrose solver would attempt to generate a diagram that satisfies all constraints.

5 Synthesis

Recall that the goal of synthesis is to generate a visual layout program P that is both sound and complete with respect to all instance-diagram pairs. That is, P must both be satisfied by all the instance-diagram pairs, and capture as much of the "user intentions" behind the diagrams as possible. This section outlines some of our preliminary ideas on the synthesis step.

5.1 Measure of satisfaction

Our visual specification language allows us to measure how well a given diagram satisfies the constraints enforced by a clause. This satisfaction measure can be used to inform the synthesizer on the soundness of a clause. We only want to synthesize clauses that have a high satisfaction score. In the end, we combine these satisfying clauses together into a full visual specification program.

We measure diagram satisfaction as a numerical score $s \in [0, 1]$ with 1 indicating perfect satisfaction. Each layout constraint maps to a "scoring function" that evaluates how well the corresponding geometric relationship holds in the diagram. For example, a constraint "Layout *a LeftOf b*" maps to a variant of the logistics function whose parameters are the positions of nodes a and b, tuned so that the further towards the left a is to b, the closer the function value is to 1. As another example, a constraint "Layout *a VerticallyAligned b*" maps to a variant of the Gaussian curve, tuned such that the closer the x-coordinates ot a and b are, the closer the score is to 1.

Finally, a clause may match many tuples in many instances. To aggregate these scores we use the geometric mean. This ensures that a clause that is violated by any parts of any diagrams will have a low satisfaction score.

5.2 Preliminary ideas on the synthesis algorithm

We start with a naïve, exhaustive synthesis algorithm that generates clauses with high scores:

(1) Generate a well-formed clause C (up to a certain bound on the number of nestings).
(2) Generate a set F of concrete constraints that C enforces on the existing instances $\{I_i\}_{i=1}^{N}$.
(3) Compute the compounded satisfaction score of F on the example diagrams $\{D_i\}_{i=1}^{N}$.
(4) If the score exceeds some threshold, keep the clause C.
(5) Repeat.

In short, this algorithm exhaustively searches through all well-formed clauses (up to a certain bound) and only keeps those that are satisfied by the given diagrams. At the end, we compose the high-satisfaction clauses together to form a program P.

5.2.1 Footprints. Step (2) of the algorithm "simulates" the execution of C on the instances and yields a set F of concrete constraints that C enforces on the diagrams. We call the set F the **footprint**

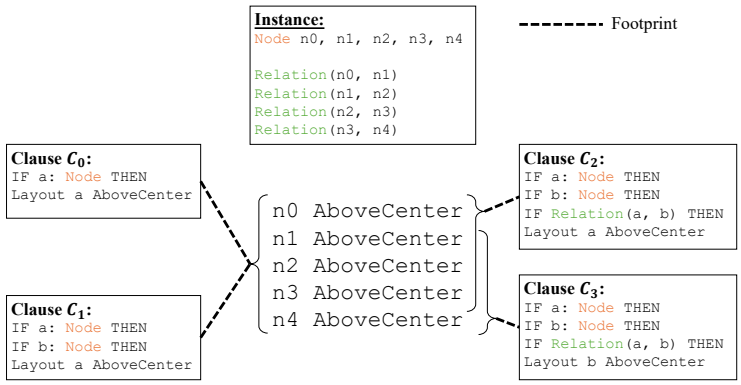


Fig. 4. An Alloy instance, four clauses, and their footprints. Observe that some clauses have equivalent footprints, and footprints of some clauses are strict subsets of those of other clauses.

of C on these instances. Metaphorically, we think of a footprint as some artifact (in this case, set of concrete constraints) left behind by the execution of a program. Fig. 4 illustrates the concept of footprint through an instance and four clauses. The top of the figure defines an instance I with five nodes and some relations, along with four syntactically different clauses C_0, C_1, C_2, and C_3 and their footprints on I.

5.2.2 Footprints as approximation of clause equivalence. As seen in Fig. 4, some clauses, namely C_0 and C_1, have equivalent footprints under a given instance. And observing C_0 and C_1, we can tell that they have equivalent behavior. What footprints give us, then, is an approximation of clause equivalence. If two clauses produce different footprints under the same instance, then they are not equivalent. Otherwise, we consider them observationally equivalent.

This allows us to categorize clauses into equivalence classes based on their footprint. For Fig. 4, we would have three equivalence classes: E_0 consists of C_0 and C_1; E_1 consists of C_2, and E_2 consists of C_3 (see the three equivalence classes in Fig. 5). Whenever the synthesis algorithm considers a new clause, it creates a new equivalence class for this clause. If this clause has the same footprint as another clause, it merges the equivalence classes. In the end, we would only take a "representative clause" from each equivalence class (e.g., shortest clause, such as C_0 from E_0) to form a visual specification program. This allows us to maintain an overall small program, whose clauses each have different behaviors.

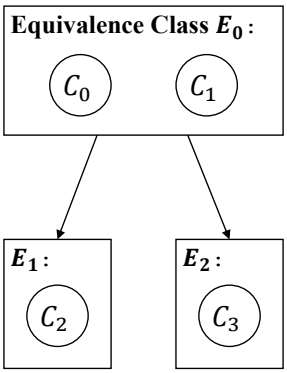


Fig. 5. We can arrange the clauses in Fig. 4 into equivalence classes based on their footprints (Sec. 5.2.2), and arrange equivalence classes into graphs based on strict subset/superset relations between footprints (Sec. 5.2.3).

5.2.3 Footprint lattices for comparing clause behavior. Additionally, as illustrated in Fig. 4, some clauses have footprints that are strict subsets of that of other clauses. For example, C_2's and C_3's footprints are both strict subsets of C_0's and C_1's. Lifting to footprints of equivalence classes, E_1's and E_2's footprints are strict subsets of E_0's. This gives us a way to compare footprints of different equivalence classes – see Fig. 5 which arranges the equivalence classes E_0, E_1, and E_2 into a graph where each edge represents a "strict subset/superset relation" in footprints.

Now, if the algorithm encounters a new clause whose footprint is not equivalent to that of another equivalence class, it would attempt to insert the new equivalence class in an appropriate position within the graph based on the strict subset/superset relations. The result is a graph consisting of a set of upper semi-lattice of equivalence classes, from which we pick a representative clause of the top equivalence class in each upper semi-lattice. In the case of Fig. 5, the final program would consist of an element of E_0. We pick from the top equivalence classes in the lattices because they are more general: They have larger footprints because they enforce stronger constraints on the diagrams of the instances, because they have more weaker and less specific selectors that match larger parts of the instances. In the end, since each clause that is picked covers a larger set of possible constraints in the diagrams, we need fewer clauses to cover the space of all possible constraints. The overall program, then, would be shorter and more interpretable.

6 Discussions, Open Questions, and Ongoing Work

This paper proposes a shift in the authoring of custom visualizations for formal modeling: moving from explicit writing of visual specifications to an example-driven synthesis workflow. We designed a prototype visual specification language focused on layout, and proposed some ideas on the synthesis of programs in this language. Many open questions arise, however, from this proposal.

Language expressiveness. Our prototype language supports a partial set of layout features to limit its complexity. We plan to evaluate the expressiveness of this language with real, stakeholder-made custom diagrams, which would inspire additional features such as softer layout constraints like *Near* and more flexible features like shape duplication and grouping. Our scope also explicitly excludes stylistic properties such as color and shape types, on which effective visualizations often rely. We must investigate whether or not our language and synthesis should support these features.

Other synthesis techniques. Our current synthesis algorithm relies on a naïve, exhaustive search of clauses up to a certain bound to maximize coverage of possible user intentions. While the footprint-based idea helps us reduce the final program size and complexity, it does not reduce the search space. We intend to explore alternative methods such as incorporating both positive and negative examples (example diagrams that users provide/accept or reject, respectively) and applying constrained specification learning techniques like [36] to generate clauses that are both satisfied by the positive examples and violated by the negative examples.

User interactions. The workflow in Sec. 3.1 makes an assumption that the users never provides "wrong" diagrams – that every diagram the users provide will satisfy whichever desired visual specification they have in mind. In early exploratory stages, however, users do not often know what visualizations they want [21]. The system's workflow should tolerate these types of "wrong" diagrams. For example, if a "wrong" diagram causes a previously-generated program to no longer be satisfied, the system may employ techniques like unsat-core to pinpoint to users exactly which parts of the diagram causes the conflict. We can also support some notion of "maximal satisfaction" of programs. Moreover, our synthesis process currently considers only the final diagrams for synthesis. Could the user's intermediate interactions – edits or adjustments made while constructing a diagram – provide useful signals for synthesis? For instance, if the users' interactions signal that they do not want a specific constraint (e.g., explicitly unaligning two nodes), can we use that information to "mask out" certain parts of the footprint lattices that contain these constraints?

Our overarching goal is to lower the barrier to creating visual specifications through example-based interaction. This paper provides an initial foundation for pursuing that vision.

References

[1] [n. d.]. Bloom: Optimization-Driven Interactive Diagramming. https://penrose.cs.cmu.edu/blog/bloom

[2] [n. d.]. D3 by Observable. https://d3js.org. [Accessed 01-12-2025].

[3] Leilani Battle, Danni Feng, and Kelli Webber. 2022. Exploring d3 implementation challenges on stack overflow. In *2022 IEEE visualization and visual analytics (VIS)*. IEEE, 1–5.

[4] Ernest Chang and Rosemary Roberts. 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM* 22, 5 (1979), 281–283.

[5] William S Cleveland and Robert McGill. 1984. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American statistical association* 79, 387 (1984), 531–554.

[6] 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.

[7] 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

[8] Christopher Healey and James Enns. 2011. Attention and visual memory in visualization and computer graphics. *IEEE transactions on visualization and computer graphics* 18, 7 (2011), 1170–1188.

[9] Mary Hegarty. 2011. The cognitive science of visual-spatial displays: Implications for design. *Topics in cognitive science* 3, 3 (2011), 446–474.

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

[11] Peiling Jiang, Jude Rayan, Steven P. Dow, and Haijun Xia. 2023. Graphologue: Exploring Large Language Model Responses with Interactive Diagrams. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. ACM, 1–20. doi:10.1145/3586183.3606737

[12] 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

[13] 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

[14] Yiliang Liang, Avinash Palliyil, Eunsuk Kang, and Joshua Sunshine. 2025. Towards Better Formal Methods Visualizations. *PLATEAU Workshop 2025* (8 2025). doi:10.1184/R1/29086949.v1

[15] Daniel Neider and Ivan Gavran. 2018. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–10.

[16] 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

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

[18] Avinash Palliyil. 2025. Improving Formal Methods Visualizations. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 224–225.

[19] Stephen E Palmer. 1992. Common region: A new principle of perceptual grouping. *Cognitive psychology* 24, 3 (1992), 436–447. doi:10.1016/0010-0285(92)90014-S

[20] 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

[21] Derek Rayside, Felix Chang, Greg Dennis, Robert Seater, and Daniel Jackson. 2007. Automatic visualization of relational logic models. *Electronic Communications of the EASST* 7 (2007).

[22] Donghao Ren, Bongshin Lee, and Matthew Brehmer. 2018. Charticulator: Interactive construction of bespoke chart layouts. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 789–799.

[23] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An interactive visualization design environment. In *Computer graphics forum*, Vol. 33. Wiley Online Library, 351–360.

[24] Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John Stasko, John Thompson, Matthew Brehmer, and Zhicheng Liu. 2019. Critical reflections on visualization authoring systems. *IEEE transactions on visualization and computer graphics* 26, 1 (2019), 461–471.

[25] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 341–350.

[26] Griffin Teller and Joshua Sunshine. 2025. BLOOM: Simplifying Interactive Diagram Development with Optimization Constraints. Plateau Workshop.

[27] D. Todorovic. 2008. Gestalt principles. *Scholarpedia* 3, 12 (2008), 5345. doi:10.4249/scholarpedia.5345 revision #91314.

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

[29] 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
[30] Colin Ware and Robert Bobrow. 2004. Motion to support rapid interactive queries on node–link diagrams. *ACM Transactions on Applied Perception (TAP)* 1, 1 (2004), 3–18. doi:10.1145/1008722.1008724
[31] Hadley Wickham. 2010. A layered grammar of graphics. *Journal of computational and graphical statistics* 19, 1 (2010), 3–28.
[32] Hadley Wickham. 2011. ggplot2. *Wiley interdisciplinary reviews: computational statistics* 3, 2 (2011), 180–185.
[33] Leland Wilkinson. 2011. The grammar of graphics. In *Handbook of computational statistics: Concepts and methods*. Springer, 375–414.
[34] 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).
[35] Andreas Zeller. 2011. Specifications for free. In *NASA Formal Methods Symposium*. Springer, 2–12.
[36] Changjian Zhang, Parv Kapoor, Ian Dardik, Leyi Cui, Romulo Meira-Goes, David Garlan, and Eunsuk Kang. 2024. Constrained ltl specification learning from examples. *arXiv preprint arXiv:2412.02905* (2024).
[37] Jonathan Zong, Dhiraj Barnwal, Rupayan Neogy, and Arvind Satyanarayan. 2020. Lyra 2: Designing interactive visualizations by demonstration. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2020), 304–314.