Designing for Semi-formal Programming with Foundation Models

Josh Pollock[1], **Ian Arawjo**[2], **Caroline Berger**[3] **and Arvind Satyanarayan**[1]

[1] *MIT, Cambridge, USA*
[2] *Université de Montréal, Montréal, Canada*
[3] *Aarhus University, Aarhus, Denmark*

Abstract

End-user programmers, such as scientists and data analysts, communicate their intent through culturally specific, semi-formal representations like formulas and wireframes. Research on end-user programming interfaces has sought to democratize programming but has required advances in program synthesis, UI design, and computer vision to support translating a representation to code. As a result, end-users must still frequently translate such representations manually. Foundation models like ChatGPT and GPT-4V dramatically lower the cost of designing new programming interfaces by offering much better code synthesis, UI generation, and visual comprehension tools. These advances enable new end-user workflows with more ubiquitous semi-formal representations. We outline the translation work programmers typically perform when translating representations into code, how foundation models help address this problem, and emerging challenges of using foundation models for programming. We posit semi-formal and notational programming as paradigmatic solutions to integrating foundation models into programming practice. Articulating a design space of semi-formal representations, we ask how we could design new semi-formal programming environments enabled through foundation models that address their emergent challenges, and sketch "proactive disambiguation" as one solution to bridging gulfs of evaluation and execution.

Keywords: End-user programming. Semi-formal programming. Notational programming.

1 Introduction

End-user programmers, from scientists to data analysts to user interface (UI) designers, use semi-formal notations — like academic papers, interface sketches, and requirements documents — to communicate their ideas. To convert their notations to running code, these users must perform *translation work*—translating their *semi-formal,* disciplinary, and cultural notations into *formal* specifications such as programs [1] (section 3). For example, while a UI composed of sliders, menus, and buttons can be sketched in a few seconds, translating this semi-formal specification to code in a Jupyter notebook requires searching for and understanding libraries like ipywidgets, then manually translating that sketch into code.

While researchers and developers have created tools for translating specific kinds of semi-formal notation (such as Wrex for data wrangling [2], FlashFill for spreadsheet formulas [3], Falx for data visualization [4], and Notate for quantum circuits [5]), supporting a new notation has often required developing a custom synthesis engine, UI design, or computer vision model, requiring research-paper level effort to engineer. However, with the recent success of *foundation models* — multimodal models "trained on broad data [...] that can be adapted to a wide range of downstream tasks" [6] — the challenges of synthesis, UI generation, and visual comprehension have been greatly reduced. With GPT4-V, for instance, code for a chart or mathematical notation can be created from just a screenshot, and a data analysis can be specified in natural language. The challenge has now shifted from engineering bespoke synthesis systems to designing malleable programming environments that empower end-users through the affordances of foundation models, a paradigm shift we call *semi-formal programming* [5], [7].

We discuss emerging challenges with semi-formal programming (section 4). First, how the gulfs of evaluation and execution [8] between user and program are both helped by the use of semi-formal notations, but also made more challenging since researchers must design more malleable, semi-formal

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

interfaces. Second, we explain the challenge of integrating different semi-formal representations into shared computing environments.

Next, we propose axes for the design space of semi-formal representations. These axes fall into two categories: syntactic and semantic. The two syntactic axes — unstructured vs. structured and spatial vs. linguistic — refer to the external form of the representation — the way it looks. For example, natural language is often unstructured and linguistic, whereas a binary tree is structured and spatial. The three semantic axes — concrete vs. abstract, vague vs. detailed, and figurative vs. literal — denote how content is expressed in that representation. A unit test is concrete, representing a single example, whereas a whole program abstracts behavior over all possible inputs. A napkin sketch of a UI is more vague than a design document. A poem describing a mechanical process is more figurative than a step-by-step manual. We demonstrate the descriptive power of these axes by analyzing semi-formal programs created as inputs to tldraw's "Make Real" prototype [9] (subsection 5.3).

Finally, we propose some solutions to the gulf and integration problems (section 6). The gulf problems may be addressed with *proactive disambiguation*, where a foundation model guides a user to disambiguate or formalize their specification. This process could further be systematized by developing semi-formal toolkits to reuse common patterns in semi-formal programming. Integration may be tackled using code as a shared substrate or perhaps by creating semi-formal bidirectional lenses between different representations when it is not possible to generate code.

2 Related Work

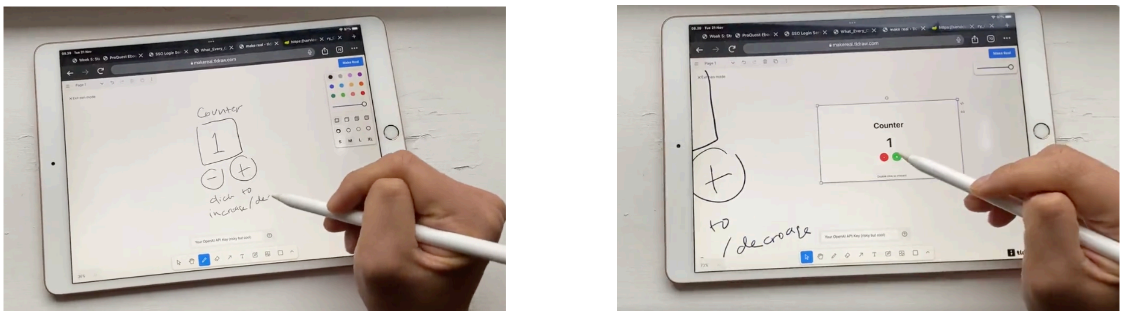
Notational and Semi-formal Programming. These terms refer to emerging paradigms proposed by the authors [5], [7]. If we take a "program" to mean any collection of instructions to control the operation of a machine, then foundation models vastly expand the kinds of "programs" we can write. Instead of being limited to code (in either text or visual form) or a fixed set of representations created by interface researchers and designers, a user can employ many different kinds of *semi-formal* representations that may be more familiar to them than code. Indeed any visual form (and non-visual form, but we won't consider them here) can be used as an input medium.

Programming Environments. Computation is an important tool in the scientific discovery workflow [10]. Scientists report using software, and writing programs [10]. Jupyter, a computational notebook, is a popular environment for scientists to write programs [11]. Tools exist to support source control [12], code execution control [13], and collaboration [14] in computational notebooks, however the design of these tools has been centered around a data scientist user group. To extend the design space and yield innovation, we must broaden work to include new user groups, such as scientists, and to be open to a departure from traditional programming interfaces to exploring tablets, phones, smartphones, and other tools and devices [15].

AI-Assisted Programming. Users program with the assistance of AI tools, such as Github Copilot and ChatGPT. Github Copilot works as an intelligent autocomplete that sits in the development environment, and ChatGPT can generate code and propose alternatives to code based on natural language prompts, to help with programming tasks such as debugging [16]. However, checking the correctness of the code produced is difficult, *"writing code is not the bottleneck in need of optimization. Conceiving the solution is. Any time 'saved' through Copilot and it's ilk is immediately nullified by having to check it's correctness..."*, says a commenter on HackerNews [16], [17]. Sarkar and colleagues posited, "While LLM-assisted programming is currently geared towards expert programmers, arguably the greatest beneficiaries of their abilities will be non-expert end-user programmers." [16]. Thus, our work aims to answer this call by envisioning the possible futures of AI-assisted programming for non-expert programmers.

3 Experience Report: Translation Work Challenges of End-User Programming

To explore existing challenges with end-user programming and how foundation models may help address them, we present an experience report based on a project of one of the authors involving re-implementing machine learning models. Though the author is a proficient programmer, they do not usually write machine learning code, use Jupyter notebooks, or make charts and UIs in Python.



(a) Sketching UI and notating expected interaction (b) Verifying the output by trying it out

Figure 1. An online user illustrates the tldraw Make Real workflow on a pen-based tablet, sketching a web UI for a number counter and getting the UI back in less than ten seconds [9], [18]. Make Real is a small extension to the tldraw canvas editor that uses GPT-4V with a custom prompt to generate UI code that is placed back in the canvas. This is an early example of the kind of workflows that foundation models are beginning to enable.

As a result, they are able to serve as a proxy for an end-user scientist.

The project required comparing different extensions of the neural radiance fields (NeRF) model [19] inside a Jupyter notebook, creating a shared interface for exploring them, and communicating the differences using statistical charts.[1] The project has three main steps:

1. Convert a PDF description of each NeRF model into code.
2. Create a custom GUI for navigating the output of a NeRF.
3. Compare results across models using charts.

Each of these steps is complicated, requiring high cognitive load and *translation work* between different representations [1]: translating an academic paper to code, an informal UI specification to a UI in Jupyter, and existing screenshots of model comparison charts to charts using data in the notebook.

First consider translation between academic papers and code. If source code for a given paper is not available online, one must work slowly and methodically through a paper to translate each equation to code. This requires understanding the details of each symbol and filling in implicit pieces that are not described in the text. While researchers have made progress towards improving paper readability [20], the process remains tedious and error-prone.

Moreover, *verifying* that a translation is correct can be challenging, since a scientist is not merely translating between two representations, but between the languages of two *cultures*: academic research and programming [1]. In disciplines other than computer science, this gap is much larger since end-users may not be adept programmers, and they must think about their domain-specific concepts while also programming.

Next consider the challenges of UI creation. After converting academic descriptions of NeRF models to code, the author wants to expose some variables first via simple direct manipulation through sliders, buttons, and menus, and then with a more complex 3D inspector. Such interfaces are typically prototyped with pen and paper before they are programmed (Figure 1a). But to translate this intent to code, a user must employ a library like ipywidgets or use a JavaScript framework to make a fully custom visualization. This translation work is difficult enough for programming experts who must switch from writing code in their domain to UI code, but it is even more challenging for end-users outside the field. Unlike folks doing machine learning from a computer science or software engineering background, end-users often report a lack formal education in software development [10]. Such a user would need to be aware of the library or have the time, resources, and desire to look for and find it. Once they found such a library, they would need to learn it, potentially in a new language in the case of JavaScript. This is a significant amount of effort for UI design, especially when many interfaces are compositions of well-known pieces like sliders, buttons, and menus that control program variables.

1 NeRF is a deep learning method that addresses the problem of constructing 3D representation of a scene from 2D images; the original paper has since spurred many follow-up papers proposing extensions and refinements.

As with paper-to-code translation, UI specification requires translation between cultural artifacts, like UI sketches or natural language, and code.

Authoring charts is similar to authoring UIs; however, unlike with simple UIs for modifying variables, users often forage for examples they want to retarget with their own data when working with charts [21], [22]. As with UI authoring, a user must learn a domain-specific API and translate their intent into the right API calls. While tools exist that address parts of this problem [23], they are limited to a fixed range of input representations like the DOM. As a result they cannot be used for other kinds of inputs like sketches, screenshots, or natural language.

Multi-modal program synthesis using foundation models can help with all three steps. For step one, a user can prompt a foundation model to directly convert an academic paper to commented, runnable code. For step two, a foundation model can convert a user sketch to code for a custom widget for their use case (Figure 1b). For the third step, the user can communicate their intent using natural language or a screenshot, and the model will generate code to import and call a specific visualization API. In each of these cases, foundation models are able to do most if not all the translation work for the user, bridging the cultural gap between domain-specific representations and code.

4 Emerging Challenges for Semi-Formal Programming

While foundation models help us solve the problem of translating various semi-formal representations (PDF, diagram, natural language, etc.) to code, the paradigm of semi-formal programming presents new challenges. Based on our experiences using synthesis tools and foundation models for writing code, we identify two salient challenges: *verification* and *integration* of foundation models' translations.

4.1 Gulfs of Evaluation and Execution

First, how do users *verify* that synthesized outputs are any good? Expert end-users who are familiar with their own artifacts, but not with code, face the task of verifying code outputs. As suggested by Arawjo et al., a system that automatically translates code back to a representation similar to the input may help bridge this cultural translation task [5]. For example, as is already the case in ChatGPT, if a user provides a screenshot of a bar chart to be converted to code, the system can run the generated code so the user may compare their input specification to the visual output and not just the code itself. However, more complex programs, such as those generated from whole papers, are likely to be difficult to verify even if they are translated back to documentation or TeX. In these cases, proxy representations, such as generated examples or benchmarks, may be more useful than representations of the entire output.

This problem is an instance of the gulfs of evaluation and execution between a user and a semi-formal representation [8]. In traditional settings, gulfs are considered between just one interface, the one created by the system designers. However, many of the benefits of semi-formal programming arise from their malleability. The right interface for the job is often task-specific and cannot always be anticipated by designers. Therefore, it seems necessary to consider the *design space* of semi-formal representations rather than specific instances. Users must be able to evaluate the representations output by the model, and they must be able to direct the model to produce refined or updated representations.

We identify two notions of correctness that are salient in semi-formal programmming: *technical correctness* and *stylistic correctness*. Technical correctness refers to the standard notion of correctness with respect to a specification. A user may verify the output is correct via spot-checks, debugging, or formal verification. *Stylistic correctness* refers the match between the user's stylistic expectations and the output. Even if the code is technically correct, if it is not meaningful to the user, then it is not stylistically correct. For example, if the output follows a functional programming style, but the user has only ever seen imperative code, then the output is not stylistically correct, as the user struggles to make sense of it. Similarly, if the variable names are seemingly nonsensical confusing the user what they refer to, then the program is stylistically incorrect. Different communities have different cultural conventions about the output form, and it is often not specified up front, but only as a rebuttal to stylistically incorrect output. To resolve issues with both kinds of correctness, the user must iteratively re-prompt the model (e.g., "there is a bug," or "make sure you output a Python function named foo") or revise their drawing (as seen in notational programming [5]).

4.2 Integration

Second, how does the user *integrate* different synthesized outputs together into a cohesive program? When working in a notebook environment, different pieces of code, such as a NeRF model and a UI to manipulate it, are connected using program variables. However, in a semi-formal programming context a user is not composing multiple pieces of code, but multiple semi-formal representations like an academic paper and a UI wireframe. Connecting or specifying shared variables across the paper and the wireframe instead of their corresponding generated code is challenging. While variables and data structures provide a lingua franca for code across domains, there is no clear equivalent for more general semi-formal representations.

The fields of PL and HCI are well-positioned to offer solutions to these challenges including approaches to verifying correctness and designs for semi-formal programming environments that facilitate integration across representations.

5 The Design Space of Semi-Formal Representations

The semi-formal programming challenges outlined in section 4 concern semi-formal interfaces. The gulfs lie between the user and semi-formal representations, and integration challenges lie between different semi-formal representations. To begin to address these challenges, we see a need for an account of the *design space* of semi-formal representations. Such a design space would allow us to *generate*, *describe*, and *evaluate* [24] semi-formal representations. These aspects would aid in addressing the gulfs of evaluation and execution for semi-formal representations as well as give a unified way to discuss them so that they might be integrated together more easily in shared contexts. Drawing on existing literature we propose some initial axes for such a design space. For several of the axes we identify useful related work that explores the axis in depth. We then use these axes to describe different semi-formal representations created by users of tldraw's "Make Real" prototype.

5.1 Syntactic Axes

Syntactic axes refer to the external form of a semi-structured representation.

Unstructured vs. Structured. Semi-formal representations differ in how formally specified their schema are. For example, code is highly structured since it must conform to the grammar of the language it's written in. Natural language is typically mostly unstructured, but is frequently augmented with lists, asides, headings, and other kinds of structured information. Shipman and Marshall explore the tradeoffs between unstructured and structured representations in "Formality Considered Harmful" [25]. While unstructured representations may be less precise, they place less burden on users to learn the details of a formal language or schema. Unstructured representations also meet users where they are in their own thinking and design processes. Ideas or tasks rarely start fully formalized or structured, but instead typically take shape gradually. Environments that support unstructured or semi-structured representations may be able to support users at these earlier stages.

Spatial vs. Linguistic. Information may be encoded *spatially*, taking advantage of humans' innate visual processing abilities to process multidimension information, or *linguistically* using linear arrangements of information where spatial positioning is less meaningful. For example, diagrams make heavy use of spatial encoding, and convey much of their meaning through spatial relationships between elements [26], [27]. On the other hand, while text does use spatial relationships to convey meaning (reordering a sentence would change its meaning, for example), much of text's meaning is conveyed linguistically. Text can often be reflowed to different line widths without changing its meaning. Larkin and Simon explore tradeoffs between sentential (linguistic) and diagrammitc (spatial) representations in "Why a Diagram is (Sometimes) Worth Ten Thousand Words" [28]. For example, while sentential representations are useful for presenting linearly ordered information (such as events unfolding in time), diagrammatic representations afford non-linear traversal since an element can be related to many elements at once (such as by being clustered in a group). Many semi-formal representations

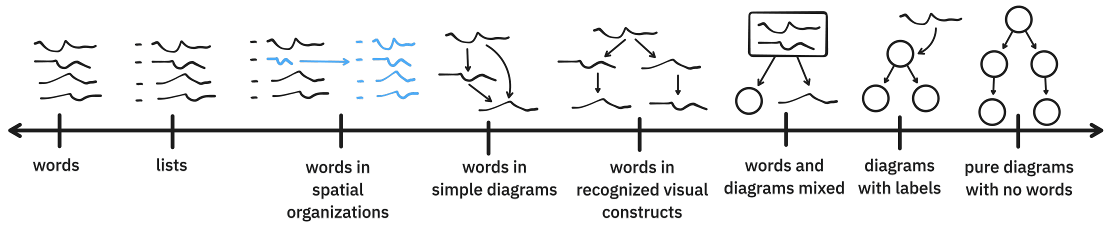

words
lists
words in spatial organizations
words in simple diagrams
words in recognized visual constructs
words and diagrams mixed
diagrams with labels
pure diagrams with no words

Figure 2. The words (linguistic) to diagrams (spatial) spectrum from Walny et al. [29].

comprise both spatial and linguistic elements, for example as part of an analysis of whiteboard usage, Walny et al. produce a words-to-diagrams spectrum, which we re-present in Figure 2 [29].

5.2 Semantic Axes

Semantic axes refer to the content expressed in a semi-formal representation.

Concrete vs. Abstract. A semi-formal representation can represent a single example or an entire collection. For example, UI *designers* often use concrete mockups of app screens with example content to express their intent and low-fidelity mockups to abstract away decisions like color. Meanwhile UI *developers* typically author abstract components to describe an application. While code tends toward the abstract, unit tests and examples are more concrete. More generally, different communities may expect different levels of concreteness. Concrete representations are especially useful early in a task when abstractions are not yet known or understood. But they can also be useful later in the process when abstractions are not important. For example when programming a one-off task, a concrete description may suffice since there is only one relevant example.

Vague vs. Detailed. Different semi-formal representations can render information at different levels of fidelity. For example, a natural language specification for a bar chart could be simply "Create a bar chart." A more detailed specification might be "Create a bar chart to visually represent the monthly sales data for a clothing store over the last year. The chart should have twelve bars, each representing a month from January to December. The height of each bar should correspond to the total sales in that month." Note that while vague representations are often more abstract, this is not always the case. Consider the difference between a UI mockup that elides the choice of color versus a UI component that accepts color as an input. While both the mockup and the component abstract over color, the mockup is vague, because it does not specify anything about color, whereas the component is detailed, because its behavior is precisely described for any possible color.

Figurative vs Literal. An element of a semi-formal representation may convey meaning figuratively, through cultural convention and metaphor, or via more direct representation such as using color in a UI mockup to directly express the desired color of a button. Lakoff and Johnson explore the use of metaphor to convey meaning in depth in "Metaphors We Live By" [30]. Notably, metaphor is a useful tool for addressing translation work, because it can allow a user to express intent or interpret output using relationships to concepts in their domain language rather than directly translating ideas to a target language such as code.

Within the space of semi-formal representation, traditional programming occupies just a small slice. Code is typically a structured, linguistic representation that represents information in an abstract, detailed, and figurative way. By sketching a larger design space we make room for other kinds of specification of program behavior including ones that may better fit existing representations of end-users.

5.3 Applying the Design Space to tldraw's "Make Real" Prototype

Tldraw is an open-source React library and canvas drawing editor that supports sketching, diagramming, and annotation in the browser.[2] Following the release of OpenAI's GPT4-V, Figma Engineer Sawyer Hood made a prototype "Make Real" feature to convert sketches of web UIs into live web-

2 https://github.com/tldraw/tldraw

sites (Figure 1). This prototype was soon forked and improved by tldraw [9]. To use the feature, a user sketches a semi-formal representation of their desired UI, selects the sketch, and then presses the "Make Real" button in the top-right corner. Upon launch in November 2023, the feature went viral on 𝕏, with many users trying it out and exploring the range of semi-formal representations that could be "made real." This marks one of the first well-known examples of integrating a foundation model into a graphical user interface outside of chat. Using our design space, we can describe the semi-formal representations used to prompt "Make Real." Figure 3 provides example analyses.

6 Towards Semi-Formal Programming Environments

6.1 Addressing the Gulfs

Proactive Disambiguation. When a semi-formal program is ambiguous, the gulf of execution can be large since the user may not know how to disambiguate a specification or even that it needs to be disambiguated at all. As demonstrated with Google's Gemini, it is possible to prompt a foundation model to identify such ambiguities and then generate a semi-formal representation (such as a clarifying question or UI) in response to help the user disambiguate their intent [34]. This approach leverages the ability for a foundation model to pick a suitable UI for the given disambiguation task.

Semi-formal Toolkits. The simplest form of proactive disambiguation uses a foundation model to create a UI for disambiguation. However, end-users may expect responses that use specific semi-formal representations, or a designer of an end-user environment may want to ensure certain tasks always use a specific representation. In cases like these where repeatability and control are desired it may be useful to have *semi-formal toolkits*, which would give users more control over the creation of semi-formal representations. Such a toolkit could allow a user to specify a representation or compose existing representations with each other as well as control when such representations are used by a foundation model. Our design space parameterization could prove useful for *generating* designs within these toolkits.

6.2 Addressing Integration

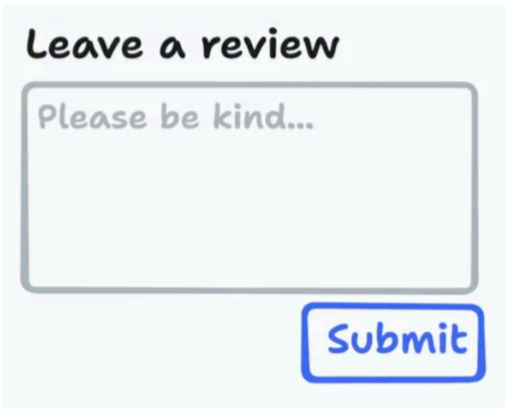
Integrating different semi-formal representations with each other presents another challenge. We propose two approaches.

Code As Substrate. In cases such as the experience report (section 3), it may be possible to use semi-formal representations only for code generation. In such cases, a shared code environment such as a Jupyter Notebook can allow variable definitions and uses to span semi-formal representations. This approach was explored by Arawjo et al. [5].

Semi-formal Bidirectional Lenses. However, a code representation of a semi-formal program may require a very formalized specification, since imprecise or ambiguous portions of a spec may not be representable in a traditional program [7]. For example, in less formal environments like canvas editors, an end-user may operate mostly with partially formalized data like a collection of mockups. In these cases, it may be useful to create *bidirectional* lenses between semi-formal representations such that two representations of the same concept, such as the state machine diagram and wireframe mockup in Figure 3c, can be related without relying on generated code as a shared representation. One could imagine updates to the wireframe updating the state machine and vice versa. This approach is similar in spirit to that of Glue [35], a visualization environment that lets end-users "glue" together related datasets using formal bidirectional lenses. However, because semi-formal lenses would not operate on code, they would likely need to be semi-formally specified themselves.

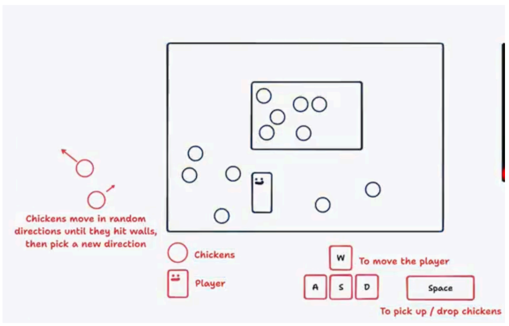
7 The Semi-Formal Programming Era

The invention of foundation models marks the beginning of a new era for end-user programming: semi-formal programming. Techniques from programming languages — like compilation, synthesis, verification, and analysis — have the potential to help communities we have previously been unable to reach because of cultural barriers to programming. But we have to meet them half way. We must relax our formal guarantees, trading them for the ability to reason about programs that are not fully formalized and embracing informal ways of knowing.



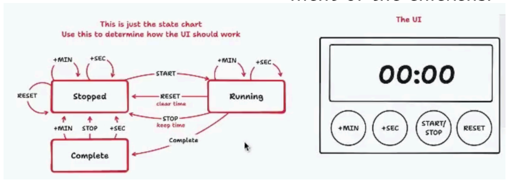
(a) A semi-formal program for a review box [31].

- unstructured vs. **structured**
  This is highly structured, conforming to common conventions of UI design.
- **spatial** vs. **linguistic**
  This incorporates both spatial and linguistic elements, most closely resembling "words and diagrams mixed" in Figure 2.
- **concrete** vs. abstract
  This represents a single state of the application rather than many states.
- figurative vs. **literal**
  This depiction closely maps to the intended output.
- vague vs. **detailed**
  Though it provides a detailed description of the form of the UI, and its function is implied by cultural context, the behavior of the "Submit" button is not specified.


Chickens move in random directions until they hit walls, then pick a new direction
Chickens
Player
W
To move the player
A
S
D
Space
To pick up / drop chickens

(b) A semi-formal program describing a small game where a player tries to put chickens in a pen [32].

- **unstructured** vs. structured
  This representation comprises pieces that are positioned haphazardly around the edge of a main UI region.
- **spatial** vs. linguistic
  This most closely falls into the "diagrams with labels" category in Figure 2.
- concrete vs. **abstract**
  This representation abstracts over many states by specifying behaviors and roles.
- figurative vs. **literal**
  Though the UI itself is figurative, using a circle to represent a chicken and a rectangle with a smiley face for a player, the specification uses literal mappings, such as direct representations of the WASD keys and the space bar.
- vague vs. **detailed**
  Behavior is specified precisely, especially the movement of the chickens.



(c) A state machine specification and a wireframe of a simple counter app. The analysis below covers the state machine only, since the wireframe is similar to that of Figure 3a. Together they comprise a single semi-formal program [33].

- unstructured vs. **structured**
  The state machine is a highly structured representation, conforming to a well-known schema.
- **spatial** vs. linguistic
  This representation is highly spatial. It is closest to "pure diagram with no words" rather than "words in recognized visual constructs" since the words act as short labels rather than longer pieces of prose.
- concrete vs. **abstract**
  This specification abstracts over all possible states of the UI.
- **figurative** vs. **literal**
  The states "Running," "Stopped," and "Complete" are figurative depictions rather than literal descriptions of the interface. On the other hand, most of the transitions map literally to the buttons in the UI.
- vague vs. **detailed**
  The specification attempts to cover every case, and is nearly a full description of the UI's behavior.

Figure 3. A collection of semi-formal representations for UIs created for tldraw's "Make Real" demo by 𝕏 users. We attempt to categorize each example according to our design space axes.

References

[1] I. Arawjo, "To write code: The cultural fabrication of programming notation and practice," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–15.

[2] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani, "Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists," in *Proceedings of the 2020 CHI conference on human factors in computing systems*, 2020, pp. 1–12.

[3] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[4] C. Wang, Y. Feng, R. Bodik, I. Dillig, A. Cheung, and A. J. Ko, "Falx: Synthesis-powered visualization authoring," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–15.

[5] I. Arawjo, A. DeArmas, M. Roberts, S. Basu, and T. Parikh, "Notational programming for notebook environments: A case study with quantum circuits," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, 2022, pp. 1–20.

[6] R. Bommasani, D. A. Hudson, E. Adeli, *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[7] J. Pollock, A. Satyanarayan, and D. Jackson, "Language model agents enable semi-formal programming," *The Ninth Workshop on Live Programming (SPLASH LIVE 2023)*, 2023.

[8] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct manipulation interfaces," *Human–computer interaction*, vol. 1, no. 4, pp. 311–338, 1985.

[9] S. Ruiz, *Make real, the story so far — tldraw.substack.com*, https://tldraw.substack.com/p/make-real-the-story-so-far, [Accessed 06-12-2023], 2023.

[10] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" In *Proceedings of the International Workshop on Software Engineering for Computational Science and Engineering*, Piscataway, NJ, USA: IEEE, 2009, pp. 1–8.

[11] J. Perkel, "Why jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, pp. 145–146, 2018. DOI: 10.1038/d41586-018-07196-1.

[12] nbdime, *Nbdime*, https://github.com/jupyter/nbdime.

[13] papermill, *Papermill*, https://github.com/nteract/papermill.

[14] A. Y. Wang, A. Mittal, C. Brooks, and S. Oney, "How data scientists use computational notebooks for real-time collaboration," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. Computer-Supported Cooperative Work and Social Computing, pp. 1–30, 2019. DOI: 10.1145/3359141.

[15] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo, "The design space of computational notebooks: An analysis of 60 systems in academia and industry," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, Piscataway, NJ, USA: IEEE, 2020, pp. 1–11. DOI: 10.1109/VL/HCC50065.2020.9127201.

[16] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213*, 2022.

[17] username: omginternets, *Github copilot is generally available (github.blog)*, https://news.ycombinator.com/item?id=31825742.

[18] D. IA. "Siguen apareciendo ejemplos de uso espectaculares de la función make it real de @tldraw, la inteligencia artificial que genera html y css a partir de dibujos." Tweet. Accessed on 2023-12-03. (2023), [Online]. Available: https://x.com/diarioiaweb/status/1727995094513418514.

[19] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[20] A. Head, K. Lo, D. Kang, *et al.*, "Augmenting scientific papers with just-in-time, position-sensitive definitions of terms and symbols," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–18.

[21] M. Bostock, *10 years of open-source visualization*, https://observablehq.com/@mbostock/10-years-of-open-source-visualization, 2021.

[22] M. Bostock, *Eyeo 2013 - mike bostock*, https://vimeo.com/69448223, 2013.

[23] P. Ziegler and L. Caraco, *Reviz: A lightweight engine for reverse engineering data visualizations from the dom*, https://github.com/parkerziegler/reviz/blob/main/paper/reviz.pdf, 2021.

[24] M. Beaudouin-Lafon, “Designing interaction, not interfaces,” in *Proceedings of the working conference on Advanced visual interfaces*, 2004, pp. 15–22.

[25] F. M. Shipman and C. C. Marshall, “Formality considered harmful: Experiences, emerging themes, and directions on the use of formal representations in interactive systems,” *Computer Supported Cooperative Work (CSCW)*, vol. 8, pp. 333–352, 1999.

[26] B. Tversky, “Spatial schemas in depictions,” in *Spatial schemas and abstract thought*, vol. 79, 2001, p. 111.

[27] J. Wagemans, J. H. Elder, M. Kubovy, *et al.*, “A century of Gestalt psychology in visual perception: I. Perceptual grouping and figure-ground organization,” *Psychol Bull*, vol. 138, no. 6, pp. 1172–1217, Nov. 2012.

[28] J. H. Larkin and H. A. Simon, “Why a diagram is (sometimes) worth ten thousand words,” *Cognitive science*, vol. 11, no. 1, pp. 65–100, 1987.

[29] J. Walny, S. Carpendale, N. H. Riche, G. Venolia, and P. Fawcett, “Visual thinking in action: Visualizations as used on whiteboards,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2508–2517, 2011.

[30] G. Lakoff and M. Johnson, *Metaphors we live by*. University of Chicago press, 2008.

[31] tldraw. “(can’t believe this works).” Tweet. Accessed on 2023-12-03. (2023), [Online]. Available: https://twitter.com/tldraw/status/1724410017963503894.

[32] tldraw. “This thing can do games too?? http://makereal.tldraw.com.” Tweet. Accessed on 2023-12-03. (2023), [Online]. Available: https://twitter.com/tldraw/status/1724926175301013506.

[33] tldraw. “State charts + wireframes + annotations.” Tweet. Accessed on 2023-12-03. (2023), [Online]. Available: https://twitter.com/tldraw/status/1725083976392437894.

[34] P. Nandy, *Gemini: Reasoning about user intent to generate bespoke experiences — youtube.com*, https://www.youtube.com/watch?v=v5tRc_5-8G4, [Accessed 06-12-2023], 2023.

[35] A. A. Goodman, “Principles of high-dimensional data visualization in astronomy,” *Astronomische Nachrichten*, vol. 333, no. 5-6, pp. 505–514, 2012.