How Domain Experts Create Conceptual Diagrams and Implications for Tool Design

Dor Ma'ayan[*1,2] **Wode Ni**[*2] **Katherine Ye**[2] **Chinmay Kulkarni**[2] **Joshua Sunshine**[2]

[1]Technion - Israel Institute of Technology
Haifa, Israel
dorma10@campus.technion.ac.il

[2]Carnegie Mellon University
Pittsburgh, PA
woden, kqy, chinmayk, sunshine@cs.cmu.edu

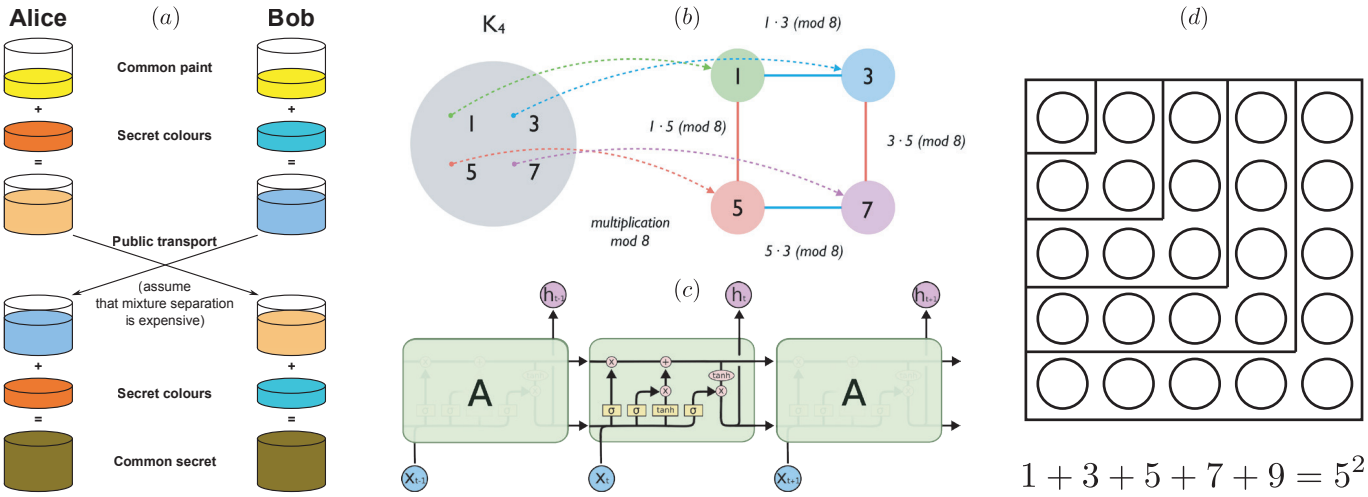


Figure 1. Diagrams explain concepts visually in many domains, *e.g.*: (a) Diffie-Hellman key exchange with colors representing prime multiplication [82]. (b) Linking two views of the Klein 4-group [84]. (c) Unrolling a recurrent LSTM network [58]. (d) Natural numbers as 2D areas in a visual proof [28].

ABSTRACT

Conceptual diagrams are used extensively to understand abstract relationships, explain complex ideas, and solve difficult problems. To illustrate concepts effectively, experts find appropriate visual representations and translate concepts into concrete shapes. This translation step is not supported explicitly by current diagramming tools. This paper investigates how domain experts create conceptual diagrams via semi-structured interviews with 18 participants from diverse backgrounds. Our participants create, adapt, and reuse visual representations using both sketches and digital tools. However, they had trouble using current diagramming tools to transition from sketches and reuse components from earlier diagrams. Our participants also expressed frustration with the slow feedback cycles and barriers to automation of their tools. Based on these results, we suggest four opportunities of diagramming tools—exploration support, representation salience, live engagement, and vocabulary correspondence—that together enable a **natural diagramming** experience. Finally, we discuss possibilities to leverage recent research advances to develop natural diagramming tools.

Author Keywords

Conceptual Diagramming; Diagram Authoring; Information Visualization

INTRODUCTION

Visual representations of knowledge allow us to understand and disseminate information more effectively than text alone [49]. This paper focuses on conceptual diagrams, which communicate conceptual, procedural, and metacognitive knowledge [41] in visual form.

Conceptual diagrams (sometimes also called *explanatory diagrams* or *concept visualization* [1]), provide a graphical overview of conceptual models—the relationship between concrete and abstract entities [29]. By giving abstract concepts visual representations, these diagrams help explain concepts

[*]Authors contributed equally and names are in alphabetical order.

CHI '20, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376253

to oneself and communicate them with others. Explaining concepts using visuals is profoundly important for dissemination of scientific knowledge and for learning. For instance, the use of diagrams in a scientific publication positively correlates with higher scientific impact [45]. Further, creating diagrams improves learning, both when diagrams are created for others [13], and when they are created for self-explanation [4].

While conceptual diagramming is clearly an important form of knowledge work, unfortunately, tools for creating conceptual diagrams are still limited. Current tools for diagramming of conceptual, procedural, and metacognitive knowledge [41] stand in tension between: a) General-purpose drawing tools such as Illustrator and Figma that offer simple pen-and-canvas or box-and-arrow metaphors, but are *viscous* [24]—users must constantly commit to exact positions, sizes, and styling of shapes. b) Dedicated diagramming tools such as Lucidchart and Gliffy that allow rapid changes, but rely heavily on templates, limiting diagrammers to a fixed set of visual representations.

This paper argues that this relatively limited support for diagramming in tools is in part because the process of diagramming is poorly understood. For instance, often diagrammers start with informal media such as paper or whiteboards, and edit diagrams digitally before they are presented, but how do diagrammers manage the evolution of diagrams? How do diagrammers utilize the strengths and cope with the limitations of their tools? Which tools are chosen for what purposes? Such a detailed understanding of the process can help design interactive tools to support diagramming.

This paper contributes a description of the process of creating conceptual diagrams, the difficulties people face while diagramming, and opportunities for tool design. These findings are based on interviews with 18 domain experts from a wide variety of disciplines such as math, computer science, architecture, and education. Our interviews reveal that diagrammers have diverse interactions with visual representations in both physical sketches and digital tools, including finding, creating, storing, and reusing representations. When diagrammers transition from sketches to digital tools, their tool selections are influenced by their *sense of control* over object placement and diagram layout. Participants were concerned with two kinds of control: local object placement, and global diagram layout. Current tools, both those that use programming languages (PL) and those that use direct manipulation (DM) as their interactive metaphor trade-off one kind of control to support the other more effectively. Consequently, we found that diagrammers invented their own set of *ad hoc* and personal reuse patterns to iterate, simplify, and automate the diagramming workflow.

One implication of our results is the opportunity to design tools informed by the processes of diagramming, and practices that domain experts already use, making digital diagramming more intuitive and efficient. We identify four key opportunities for **natural** [56] diagramming tools that allow diagrammers to express their ideas visually the same way they think about them:

- *Exploration support*: supporting exploratory behaviors such as undo and backtracking during both abstract-level, breath-first exploration of the design space and low-level refinements of visual details.
- *Representation salience*: allowing explicit creation and management of visual representations, *i.e.* the *mappings* from domain constructs to shapes instead of geometric primitives themselves.
- *Live engagement*: providing diagrammers with the sense of agency by designing for liveness and directness of the diagramming experience.
- *Vocabulary correspondence*: enabling diagrammers to interact with their diagrams using vocabularies that is conventional in their domain.

For each of the opportunities, we survey existing techniques from relevant areas to provide tool designers with technical insights on how it might be implemented.

BACKGROUND & RELATED WORK

This section provides background on three areas of related work that informed our research: research on the theory behind conceptual diagrams, existing tools for visualization, and empirical research on diagramming-related activities. Related work that is directly relevant to the implications of our work is described in the Implications section.

Conceptual diagrams and their benefits

Conceptual diagrams differ from data visualizations, which are visual representations of concrete and factual, rather than conceptual, information. Data visualization techniques enable people to understand how quantities relate to each other and gain valuable factual knowledge about the world. Ervin [19] distinguishes between *pictorial* and *propositional* graphics: instead of directly depicting data, diagrams (propositional graphics in Ervin's terms) constitute knowledge and embody media-independent abstractions for inference-making [44]. In addition to knowledge representation, conceptual diagrams are also a medium for creativity and exploration, since they do not require early commitments to design decisions and focus on the *form* of possible solutions [14].

Diagrams have been shown to have cognitive benefits to reasoning and problem solving [44, 40, 48]. Compared to textual representations, diagrams facilitate fast recognition and direct inference by making the most relevant information explicit and easily findable [44]. As an external representation of abstract structures of tasks, diagrams can work together with one's mental representation and are an indispensable part for accomplishing distributed cognitive tasks [86]. Hegarty and Kozhevnikov [30] distinguish between *pictorial* and *schematic* visual representations and show that schematic representations of relative spatial relationships significantly outperform pictorial ones that encode visual appearances.

In addition to their values as an external, static representation of knowledge, diagrams are also beneficial when people learn *with*, instead of *from* them [75]. In educational contexts, explicit training of drawing, including the creation of new visual representations and adoption of new ones, significantly improve students' ability to work with multiple representations and improve learning, reasoning, and communication skills [2].

Moreover, creating diagrams as visual explanations also improves learning, since they can act as a check for completeness and a medium for inference [5]. In general, people do not need formal training in visual design to create and interpret effective diagrams and learners at all levels can benefit tremendously from creating diagrams [66].

Bill Thurston famously wrote "people have very powerful facilities for taking in information visually... On the other hand, they do not have a very good built-in facility for *inverse vision*, that is, turning an internal spatial understanding back into a two-dimensional image [74]." In our study, we investigate how domain experts transform high-level concepts to diagrams.

Existing designs of diagramming tools

Although many diagramming tools support both text-based and graphical interfaces, we categorize current diagramming tools by their dominant mode of interaction: programming-language based (PL) tools and direct manipulation (DM) tools.

We use PL tools to refer to text-based diagramming tools, including imperative or declarative programming languages, libraries, frameworks, and embedded domain-specific languages. General-purpose tools such as Processing [61], Asymtote [7], PGF/TikZ, and Paper.js [1] provide program constructs that model graphical primitives and operations akin to those in Scalable Vector Graphics (SVG) [83]. Many of their shared disadvantages are well summarized in TikZ's manual [73]: "steep learning curve, no WYSIWYG, small changes require a long recompilation time, and the code does not really "show" how things will look like." Domain-specific tools allow diagram specifications that are higher-level and specialized to the problem domain to smoothen the learning curve. They are developed either from scratch (*e.g.* GraphViz and the DOT language for graph visualization [17]) or on top of general-purpose tools (*e.g.* TikZ's extensions, `tkz-euclide` for Euclidean geometry). However, many of them still inherit the other disadvantages from above.

DM tools represent interactive diagramming tools that support WYSIWYG interfaces and direct interaction with shapes. Akin to PL tools, general-purpose DM tools such as Adobe Illustrator, Inkscape, and Figma also have similar sets of primitives, but often provide a large number of widgets or drawing tools (*e.g.* Illustrator CC has nearly 100 built-in tools [2]). To overcome the disadvantage of their highly manual interaction model, both Illustrator and Inkscape provide language bindings or command-line tools for automation, but they still suffer from the above problems of PL tools. Popular domain-specific diagramming tools such as draw.io and Gliffy are template editors that provide predefined, mostly box-and-arrow style shapes, limiting users to a narrow set of diagrams. Research prototypes such as Sketchpad [70] and ThingLab [6] automate diagram layout using constraint solving, but many edit actions like selection and shape construction remain manual. Other prototypes like Apparatus [3] and Bret Victor's dynamic visualization tool [9] incorporate some limited programmatic operations (*e.g.* macro recording, variable declaration, and computed properties) via direct interactions.

As discussed by Satyanarayan *et al.* in [65], data visualization tools have transformed over the past decade. The major advances are characterized by three "waves": (1) improvement of individual charts' quality, (2) theories and tools that enable mass-production of visualizations, and (3) the convergence of tools [15]. Whereas the benefits of conceptual diagrams are clear and theoretical foundations exist, most of the diagramming tools are still not easily scalable and there are large gaps in existing technologies, notably between PL and DM tools. In other words, the 2nd wave of conceptual diagramming is still not here. In this paper, we aim to gain a deep understanding of people's diagramming process to drive the design of tools that fill these gaps.

Empirical studies on diagramming-related activities

Although conceptual diagrams are widely studied as a powerful visual representation in multiple domains, there has not been a significant amount of prior work that focuses on the *authoring* of conceptual diagrams, especially with digital tools.

However, prior work in related activities such as note-taking and whiteboarding suggests some insights for both understanding these activities and opportunities for tool design. Studies on sketches in STEM [80] and software engineering [11] suggest a need for automating the process of sketching and preserving transient sketches such as whiteboard drawings with appropriate tools. In similar activities such as annotating documents, personal annotations undergo dramatic changes such as significant substantiation and clarification when they are shared on public platforms [47]. Digitization of the analog pen-and-paper interface attempts to make the transformation process smoother. While digital ink tools imitate the pen-and-paper experience and provide more versatility and power, there still exist gaps between the manual and digital experience of sketching due to conflicting affordances of analog pen and digital ink [64].

Given the lack on the prior work on this topic, this paper directly investigates the process of creating conceptual diagrams using digital tools.

METHOD

Participants and recruitment

We conducted interviews with 18 participants (13 male, 5 female). Participants were recruited through posts on social media, and our research group website. Of these, four participants were university faculty, 10 were PhD students or postdocs, one was a professional masters student, one was a K-12 math instructor, one is an independent software developer, and one is an enterprise software engineer. Prospective participants filled out a survey which allowed us to screen participants for our interviews. We selected the interviewees based on the following criteria: the interviewee (1) creates conceptual diagrams on a frequent basis and (2) uses digital diagramming tools to create these diagrams.

[1] http://paperjs.org/

[2] https://helpx.adobe.com/illustrator/user-guide.html

[3] http://aprt.us/

Domain	Participant
Abstract algebra	P1, P17
Category theory	P4
Discrete mathematics	P14
Computer graphics	P16, P6, P10, P3
Algorithms	P12
Topology	P7
Human-computer interaction	P13, P18
Programming language theory	P11
Software engineering	P9, P2
User interface design	P5
Architecture	P8
K-12 education	P15

Table 1. Interview participants' primary domains

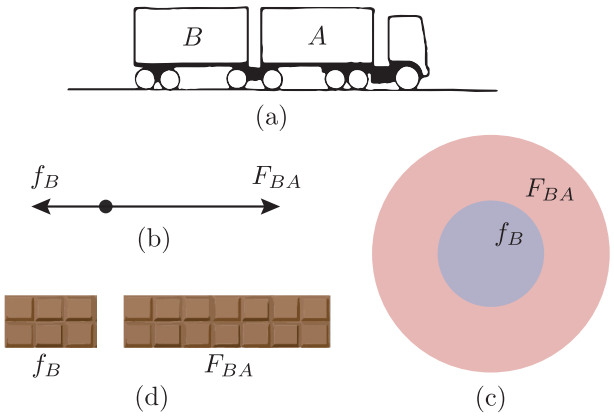


Figure 2. A good visual representation (b) of forces on a truck (a) is easily understandable, whereas (c) loses essential information and (d) is non-standard and harder to understand.

Because we had more potential interviewees than we originally envisioned (64 in all), we used a saturation method [3] to determine the number of participants. We conducted several batches of interviews (with 2-3 interviews per batch consisting of diverse participants), and did a preliminary analysis of the transcripts from each batch. When the analysis stopped revealing new insights, we stopped interviewing more participants.

Semi-structured interviews

Interviews lasted between 30 and 80 minutes and were semi-structured. Interviews were conducted either in person (7 participants) or online using Skype (11 participants.) We encouraged participants to bring any digital and hand drawn diagrams that they had previously created that they could share with us. We also encouraged them to have a pen and paper (or a whiteboard) available to draw during the interview.

Four of the authors are involved in the development of PENROSE [57, 84], a new diagramming tool. Our initial interview questions were developed to inform PENROSE's design. The focus of the interview eventually broadened to participants' past experience diagramming (using the critical incident technique [21]), tool preferences, and reuse practices. The full interview protocol is included as supplementary material. Example questions from our script are: (*e.g.* "*What is the last diagram you made?*" and "*What is the diagram you are most proud of?*"), accompanied by appropriate follow-up questions and requests for participants to share diagrams under discussion. Table 1 includes all the participants categorized by the primary focus of their work.

Analysis

Interviews were video recorded and transcribed using either human or machine transcription. The first two authors then manually validated and corrected any transcription errors.

We employed thematic analysis methods [8] to analyze interview transcripts. The first two authors began by conducting an open coding session and discussed initial insights for every batch of interviews. Then, following all interviews, the authors discussed the codes and created a coding guide with operationalized definitions of codes. Using the agreed coding guide, one of the authors did a second phase of coding. While conducting the second coding phase, the author also summarized the transcripts using sticky notes containing highlights of the interview sessions. All authors then reviewed both the codebook with the sticky notes to further refine the set of codes.

Finally, the authors analyzed the codes by clustering lower-level codes during multiple interactive discussion sessions. Through the higher-level clusters, a few themes with high numbers of codes emerged, such as *Reuse* and *Representation*. We present these themes and the resulting insights next.

RESULTS

In this section, we present the results from our analysis of the interview data. The section is organized in terms of the high-level themes that emerged from our analysis.

Representation finding

When illustrating a concept visually, a crucial step is to decide how every abstract object will be represented graphically. For example, Larkin and Simon, in their classic paper "Why a Diagram is (Sometimes) Worth Ten Thousand Words," chose to represent forces with arrows in the diagram shown in Figure 2b [44]. This step, which we call *representation finding*, is crucial to diagram effectiveness. If Larkin and Simon had represented forces using concentric circles with different radii instead of arrows (Figure 2c), the directionality of the forces would be lost. If they had represented forces with chocolate bars with different lengths (Figure 2d), the diagram would have been inconsistent with other physics diagrams and the extraneous detail would have distracted from the core purpose.

This process of *representation finding* usually preceded the creation of any formal or informal diagrams. Participants engaged in two representation finding activities: (1) seeking and finding existing representations from prior work and (2) creating novel representations.

Diagrammers seek existing representations from prior work

In many domains, there are well-established visual representations for abstract concepts and objects. Therefore, diagram-

mers tended to look at existing diagrams for representations when starting to create their representations:

> "*Sometimes I look for inspiration in other papers just to know what kinds of standard people are using. Sometimes there are some conventions that people actually use in my field like how to represent a camera for instance. So you kind of have to stick with these conventions.*" (P3)

Diagrammers generate new representations to tell new stories

Other domains lack standardized representations and diagrammers creatively *generate* their own representations:

> "*The whole purpose of those diagrams [in my book] is to make something that has never been seen before visually obvious... Why didn't anybody draw that picture before? I have been taking something that almost seems completely confusing or unimportant and having a picture that makes you know what's going on... is truly satisfying.*" (P1)

When creating diagrams for explanatory purposes, diagrammers also carefully craft visual representations to ensure that the diagram is intuitive and clear for their target audience. For instance, P8 developed new representations to reduce visual complexity:

> "*When a diagram has too many working elements, it becomes too hard for your brain to process it. If you can boil it down to two main things interacting, that will make the diagram much more intuitive to someone. It's very much about choosing the right colors, lines... putting the emphasis in the right place.*" (P8)

Diagrammers use sketches to discover appropriate representations

Sketching plays an important role in generating new visual representations or choosing among existing ones. For instance, P8, P12, P5, P9, and P13 reported iterative processes of refining their visual representations as they sketch. For instance, P9 described the evolution of diagram sketches and the changes of visual representations of the design of a complex camera-supporter-projector system, also shown in Figure 3:

> "*At this stage, I don't even know how these machines would be connected, so there's lines, but at this later stage I was actually thinking about 'Oh, how are we going to represent these things and compute with them in practice?' So I did arrows. There's also certainly increased complexity in the beginning thing, I'm just looking at the situation of a single camera projectors supporters system. Then here on the next page I'm starting to look at different configurations of multiple cameras and projectors.*" (P9)

Choosing the right tools

When participants eventually chose to move to a digital medium, their choice of tool was systematic, if not conscious. Specifically, we found participants' preferred either programming-language based (PL) tools or direct-manipulation based (DM) ones. Below, we analyze the reasons for their preferences.

People choose DM tools for faster feedback and global control

DM tools were often described as "easier" and thus have lower barriers to entry when compared with PL tools. One particularly common reason for choosing DM tools was the need to place shapes in relations with other shapes, which is difficult to do without immediate visual feedback.

> "*So I like [a DM tool] because it gives me this very fine control over how things are aligned and when they're straight up and down.*" (P2)

Because of the synchronized visual preview, DM tools provide better support for *global* control over diagram layout, i.e. the relationships among graphical primitives. Diagrammers used DM tools similar to how they used pen and paper, to offload their working memory [44], but with the additional benefits provided by interactions supported by the tools:

> "*I'm trying to draw things down on the papers [or DM tools] because my head is getting crowded and I need to be able to keep track of everything on [digital] paper and be able to interact with it the same way I would in my head.*" (P7)

People choose programming languages for better abstraction and local control

Comparing to the easy *global* control provided by DM tools, PL tools make *local* control easier: they let users control *local* placements of shapes by specifying exact pixel coordinates:

> "*I want this [a shape] to be exactly a hundred pixels... because there's definitely times where I want to get this right to this point and it's hard to do that with the mouse.*" (P15)

Global layout of diagrams can be specified more precisely using PL tools, but requires more advanced programming skills and, as discussed, more time commitment:

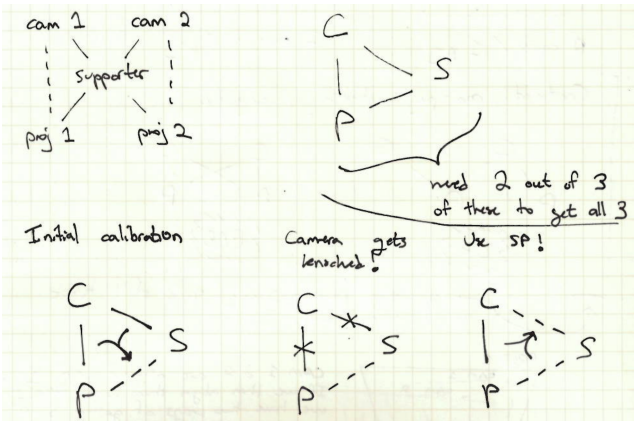
> "*If it's something where the relationships among the things you want to specify in a precise way, then it's a lot easier, if you know how to program, to introduce a programming language where you can specify exactly the relationships you want, how you want them to change, and so forth.*" (P1)

Programming languages provide affordances to create abstractions and automate the diagramming process:

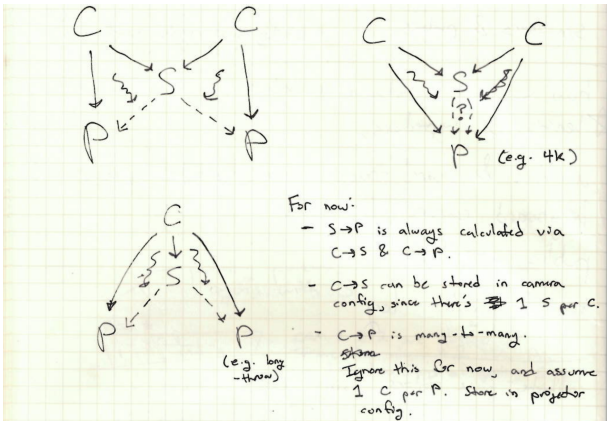
> "*Once you have made a visualization [using PL tools], if you want to tweak things about it, you can. Just put what you do into a script, add some parameters, and you could repeatedly get the same visualization with variations... It will generate the thing automatically, you don't have to create a whole picture by hand again.*" (P1)

The complexity of languages, however, incurs a higher upfront cost and steep learning curve, making more diagrammers without programming background reluctant to commit to them. Another downside of PL tools is that they often require compile-and-run cycles and hence delayed feedback:

> "*There's a long learning curve on [a PL tool] and then it's slow. It's a lot of typing and a lot of iterative–'I type something and I see what it looks like'. So there's a lot of delays in modifying [the diagram].*" (P15)



(a) An initial sketch represents connectivity as line segments.



(b) A later sketch represent connectivity as arrows.

Figure 3. P9 made sketches to explain projector and camera calibration. The complexity of these sketches increased and visual representations evolve over time.

> "*I'm willing to put in the effort, but it's like 20% of the time is that, and like 80% of the time is fighting with LaTeX.*" (P11)

Advanced diagrammers use PL tools to automate their diagramming workflows

In some cases, diagrammers find the need to create families of similar diagrams for the use of, for instance, writing a textbook or a problem set. A small number of our interviewees automated their diagramming workflow extensively by leveraging the abstraction affordances of PL tools. One automation pattern is to parameterize complex diagrams and generate multiple instances with variations to explore design alternatives and populate diagram collections:

> "*If I invest the time upfront to just write it, parameterized by, and then I do the diagram in terms of n and k. And then later I realize that, 'Aw, n = 15 and k = 4 is just a mess!' Okay, I go to the top of the file and I set n = 12 and k = 3 and I re-render it and it looks like this, and I go, 'That's what I want.' If I'm not sure, okay, let's try n = 10, try that. You can just make a new diagram in 15 seconds instead of four hours, but they also demand more time and skills up front. If I'm just going to do one diagram, it's not worth it.*" (P1)

Another pattern is creating *ad hoc*, embedded domain-specific languages that allow specification of diagrams at a higher level:

> "*I have learned a style that is highly idiomatic and not something that I could teach someone else... you look at the sort of syntactic objects that you're going to work with in a certain proof theory, and you define the macros at the top level.*" (P11)

As commented by P1 and P11 above, the automation requires a high upfront cost and advanced skill sets, and only advanced diagrammers invested in the skills and time commitment to do so.

Reusing elements from earlier diagrams

Diagrammers backtrack frequently and informally track prior versions

Diagrammers were iterative even in the formal diagramming stage, which involves frequent backtracking behaviors. Therefore, keeping track of version history becomes an essential task for diagrammers.

In the case of DM tools, however, versioning can be challenging in many existing tools, due to the lack of textual storage formats. As a result, *ad hoc* solutions are again created to compensate for this limitation such as keeping multiple versions of the diagram on the same canvas, as illustrated in Figure 4 and described by P5:

> "*I ... duplicate each [art board], change something about that, and take it out again. That's really helpful not only to present the overall trajectory of the process, but then you can go back and reference a previous state without having to look through the undo history and destroy all of your redos. I [like to] branch out fractally with different areas that are relevant to me.*" (P5)

In theory, standard version control systems such as `git` [76] make it easy to track versions with PL tools. However, even with their textual file formats, versioning can still be challenging for PL tools because textual representations of diagrams can be too low-level to be human readable. As a result, P6 tracks prior versions without using standard version control systems:

> "*I rely either on Dropbox to store different images or I have my own custom-made back-up system that keeps hard copies of things... I have a separate script that every day pulls all of my folders and keeps copies of them if there is any change.*" (P6)

Diagrammers organize reuse libraries by representation

The most common form of library we saw diagrammers maintain was a "cheat sheet". Cheat sheets are configuration files that contain low-level parameters such as line-weight settings

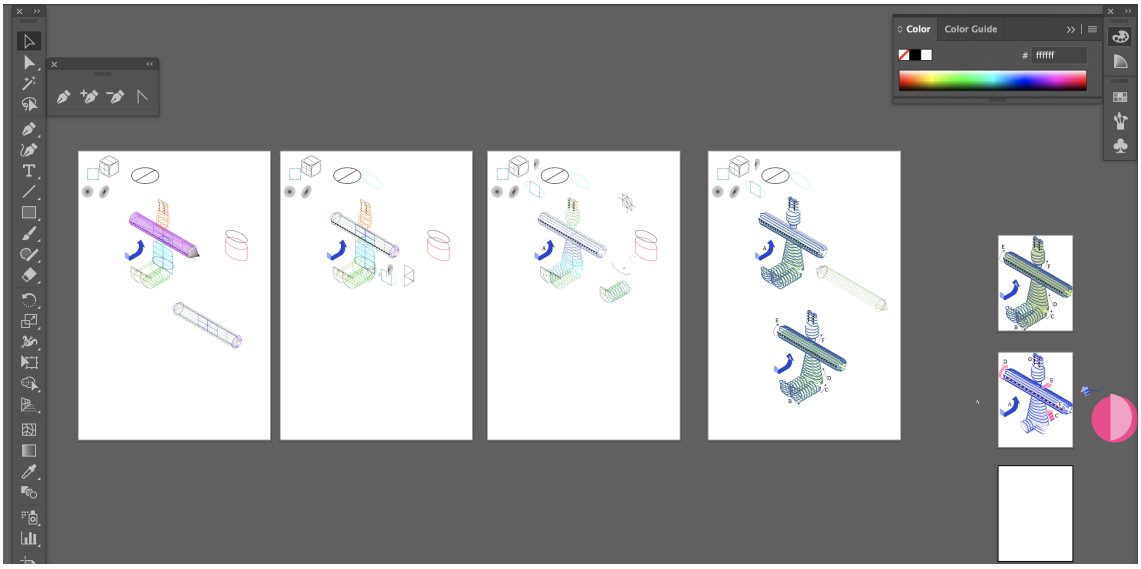
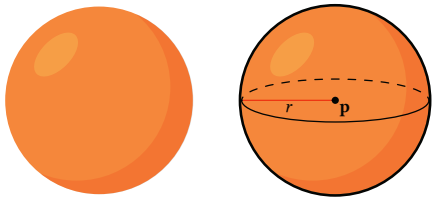


Figure 4. P13 manually tracks versions of a diagram in Illustrator using multiple canvases.




```
=== COLORS ===
* Orange
Base: #f7883c
Dark: #000 @ 15% (overlay)
Darker: #000 @ 30% (overlay)
Light: #fff @ 20% (overlay)
Lighter: #fff @ 40% (overlay)
=== STROKES ===
Main (silhouette only): #000 2pt thickness (solid)
Secondary: #000 1pt thickness (solid)
Tertiary: #000 @ 50% (overlay) 1pt thickness
Behind: #000 @ 1pt thickness (dashed: 6pt dash, 7pt gap,
      round cap, align to corners)
=== FONT ===
Base: Linux Libertine (add with TikZ directly in TeX)
Size: 11pt
```

Figure 5. P3 uses a cheat sheet to track frequently used style attributes.

and hexadecimal strings of colors (P3, P6, P13). Diagrammers used cheat sheets to reduce stylistic inconsistencies and to simplify the repetitive, manual tasks in the diagramming process with cheat sheets. For instance, one participant said:

> "*Usually I have this little* `txt` *file where I basically remember the color so I have the color codes for primary and secondary [objects]... I saved the [line] width as well for primary secondary [objects], and that's kind of like my cheat sheet that I reuse.*" (P3)

This is one area where tool support was mostly lacking. For instance, participants often took notes manually (sometimes these notes were handwritten.)

More advanced diagrammers maintained collections of existing diagrams or diagram components, organized by representation. For instance, one participant keeps a document of previous visualizations, as shown in Figure 6:

> "*I keep a document, that is almost all the TikZ diagrams I've ever had to because I find that they helped me think about how to represent diagrams for new situations.*" (P7)

Another participant collects commonly used diagram components in a personal library:

> "*So over the time I've settled on specific representations for the camera and the light source, I keep copies of them. I have my own small library of things.*" (P6)

IMPLICATIONS: NATURAL DIAGRAMMING

The results discussed above suggest unique strengths and weaknesses of existing PL and DM diagramming tools. This section offers some possibilities to combine the strengths of PL and DM tools and create more intuitive and efficient next-generation tools. Just as previous work advocated for creating programming tools "for people to express their ideas in the same way they think about them" [56] as an opportunity for *natural programming*, we advocate for creating diagramming tools for **natural diagramming**. Natural diagramming presents four distinct opportunities to leverage the strengths while alleviating the weaknesses of existing tools: exploration support, representation salience, live engagement, and vocabulary correspondence. We end the description of each of these natural diagramming opportunities by highlighting existing techniques that may be further developed to achieve it.

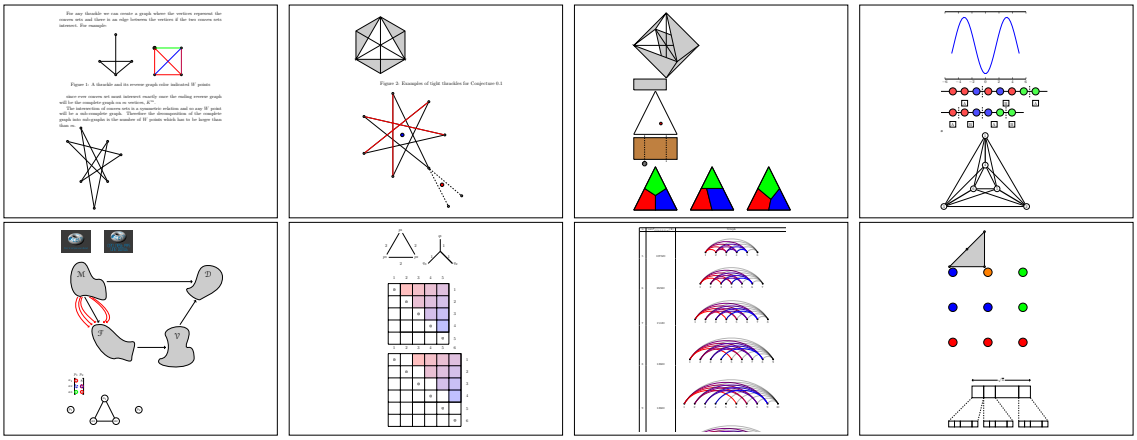


Figure 6. P7 organizes all previously made TikZ diagrams in a single document.

Exploration support

Exploration pervades the diagramming process for making conceptual diagrams, from choosing representation to determining stylistic details. We characterize two types of exploratory activities by adopting terms from Goel [23]: (a) *Lateral transformations* involve ideation and exploration at the high-level to broaden the design space, *e.g.* finding the appropriate visual representation for forces in Figure 2. (b) *Vertical transformations* are more detailed refinements to a pre-determined visual representation, such as deciding the arrowhead style in Figure 2b.

Sketching is an important, if not necessary, part of the design process [10]. Our participants produced physical sketches to explore design alternatives before transitioning to digital tools, which are perceived as a medium of higher commitment. They performed this type of exploration by looking at prior work and multiple alternatives *laterally*. Early informal sketches naturally affords lateral transformations [23].

Diagrammers also attempt to leverage precision, automation, and abstraction afforded by digital tools. After the visual representation stabilizes, they move to digital tools and refine this determinate representation, i.e. *vertically* refine their design. Unfortunately, existing tools do not provide sufficient flexibility to support even small vertical changes. Whereas DM tools require significant manual efforts to propagate local changes, PL tools require a high upfront cost to create abstractions that reduce future repetition. As a result, our participants described various *ad hoc* workarounds to perform common activities for exploration such as backtracking, versioning, and reuse in both PL and DM tools.

In essence, a gap exists between *drafting* and *crafting*, a design dilemma that has plagued other sketching tools (such as digital pens) [64]. Solving this dilemma requires tools that continuously support diagrammers to explore lateral and vertical changes, allowing more fluidity to make, track, and revert changes.

One solution to the problem of exploration and change management can be seen in tools for exploratory programming. *Exploratory Programming* (EP) characterizes a practice of experimenting and prototyping adopted by programmers across a wide range of domains [67, 39]. Data scientists reuse and iterate on small snippets of scripts to analyze data exploratively. VARIOLITE [38] support local versioning with "variant boxes" around regions of code. Software engineers often backtrack by manually deleting or re-typing code when developing software [85]. *Selective undo* techniques allow complex backtracking in code editing [54], which was also shown to be effective for painting applications [55].

Some systems also show opportunities for automatic vertical refinement. For instance, to support the transition from freehand sketches to final UI implementation, SILK recognizes hand-drawn shapes and translates them to real UI components [43]. DREAMSKETCH generates 3D models from sketches [37]. Sketches for conceptual diagrams, however, can often be intentionally ambiguous and unstructured, making recognition, beautification, and generation of reusable diagram components challenging [71].

Finally, in addition to better support for history management, another approach may be to reduce unnecessary changes through improved abstractions. *Program synthesis* can be used to generate reusable functions from examples of lower-level interactions with a potentially non-programmatic interface [27]. SKETCH-N-SKETCH is a vector drawing tool with synchronized code and graphical views of the same drawing [31]. It synthesizes reusable functions from direct manipulation of objects in the graphical view and thereby enables users to avoid repetition. Ellis *et al.* [16] synthesize imperative programs from hand-drawn sketches. Currently, synthesizing high-level abstractions and surfacing them in a non-obstructive, meaningful manner are still open problems for future conceptual diagramming tools.

Representation salience

Constructing and interpreting representations are crucial skills for learning new concepts and developing domain expertise [2]. Many of our participants track prior representations, curate reuse libraries, search for ones created by others, and inventively create new ones in their *representation finding* phase. This suggests an opportunity for natural diagramming tools

that support **representation salience**, treating visual representations as first-class entities and providing operations to easily interact with them.

Bret Victor uses the analogy of climbing "up" (*abstraction*) and "down" (*concretization*) the *ladder of abstraction* to characterize the process of understanding complex systems using visual representations [78]. Abstraction and parametrization of diagrams allows fast generation of families of similar diagrams, which can be used in multiple places or serve as a set of design alternatives. In existing tools without easy access to abstraction constructs, generating design alternatives can be a time-consuming manual process. Our participants use existing abstraction constructs such as macros and functions to encode representations, but they tend to be highly personalized and brittle solutions. As a result, these custom abstractions are rarely scalable or composable, as one frequent TikZ user said *"macros are terrible, [I make macros that are] 20 or 30 braces deep... it's just hard to write and edit."* (P11) In addition, these poor abstractions still require significant time investment and force diagrammers to concretize concepts manually instead. Consequently, diagrammers' representational encodings are scattered in manually curated personal libraries, online examples, and cheatsheets of lower-level elements.

For visual representation to be *salient*, both the underlying structures and mappings to visual elements need to be encoded in the diagramming system explicitly. Further, these encodings must be specified with *manageable, scalable, and composable abstraction constructs* that allow diagrammers to move "up" and "down" the ladder of abstraction easily.

The management of visual representations at different levels of abstraction can be seen in many fields. For instance, Gross [25] tackles the problem of the fixed low-level representations of computer-aided design (CAD) tool by supporting gradual transition from sketches to more structured diagrams and suggesting concrete representations given early conceptual sketches. Data visualization tools such as Dashiki [52] and Draco [53] can manage of multiple representations of the underlying data.

The lack of representation salience often manifest in highly *viscous* diagramming tools that operate on low-level primitives and lose deeper *semantics* of graphical components. To solve this problem in data visualization, the *grammar of graphics* [81] formalizes a rich set of operations to transform data into visual components. For mathematical diagrams, PENROSE [57] includes two domain-specific languages that decouple visual representations from declaration of abstract objects and encode visual representations by pattern-matching on the objects and declaring visual elements. Apart from these domain-specific solutions above, however, conceptual diagramming tools still lack a general and accessible approach to specify problem domains and the visual representations thereof.

Live Engagement

Hutchins *et al*. introduce *direct engagement*, "the *qualitative* feeling that one is directly engaged with *control* of the objects," as an important criterion for effective interfaces [35]. Our results show that direct engagement for diagramming can be perceived differently depending on the kinds of interfaces and the sense of *control* they afford, *i.e*. the sense of agency over essential operations in conceptual diagramming [46]. DM tool design affords continuous representations of objects and immediate visibility of incremental changes [68]. As a result, they naturally afford a sense of *global* control over the global rearrangement of diagram layout. Yet, *local* control over precise specification of visual properties and creation of high-level abstractions is still challenging in direct-manipulation tools. On the other hand, while these same operations are directly supported in PL tools, our participants reported frustration with their high latency and long compile-and-run cycles. Our results suggest immediate visual feedback (or *liveness* [72]) is also essential for abstract operations. In other words, the sense of control and direct engagement is diffused among DM and PL tools.

Traditional direct manipulation interfaces can be augmented by novel interaction and programming techniques such as programmatic brushes [36], *programming by example* [26] and *programming by manipulation* [32]. *Bidirectional programming* [12] proposes to combine direct manipulation and textual programming by surfacing both direct manipulation and text-based interfaces and synchronizes and synchronizing changes in both directions: (i) from the program text to the output (liveness) and (ii) from the output to the program text (direct engagement). These techniques, although currently limited to narrower domains, provide promising directions towards bridging the gap between PL and DM tools.

There have been also significant advances in programming languages to support liveness. *Live programming* techniques can provide responsive and continuous feedback on program changes. The methodology has been increasingly adopted in computer science education, web development, and traditional programming environments [50, 33, 62]. Tanimoto [72] proposes four levels of liveness with the top level featuring "stream-driven updates" and "informative, significant, responsive and live" visual representations of programs' dynamic behaviors. Techniques such as incremental compilation and type inference [51] and typed holes [59] allow fast compilation and facilitate liveness of programming environments. However, many live programming systems only offer one-way updates from the program to its output visual representation, inhibiting the opportunities for direct interactions with the visuals.

Vocabulary correspondence

Conceptual diagrams are made of "abstract and topological" [20] shapes that are mapped from domain objects in the *content model* [63]. In Figure 2b, the concept of two counteracting forces (domain objects) is mapped to two arrows, but the exact styling and lengths of the arrows do not change the meaning of the diagram. As a result, users' vocabularies for conceptual diagramming are often abstract, topological, and domain-specific. Therefore, there is an opportunity for natural diagramming tools that support **vocabulary correspondence** by having a grammar or an interface that directly maps to users' vocabulary for diagramming.

As shown in our results, diagrammers define new abstractions to both automate and *naturalize* their diagramming process. More advanced diagrammers create abstractions to quickly generate new diagram instances and fit their mental models, but, even for advanced diagrammers, abstractions are brittle and often not share-able, due to the lack of tool support. In addition, many participants described diagrams in terms of relative, high-level relationships such as "smaller"/"bigger" and "overlaping"/"non-overlapping". But the tools they use tend to operate on absolute units and do not provide support for specifying such relationships. One participant shared a vision of an ideal tool:

> "*The best tool... would have fairly high level primitives. I might say 'Okay, I want it to be symmetric in this way. I want this thing always to be attached to that.' I want to be able to define my own higher level primitives.*" (P2)

In other words, large semantic and articulatory distances [35] still exist between interaction metaphors and diagrammers' vocabulary, creating an opportunity improve the closeness of mapping [24] while maintaining users' control and the expressiveness of diagramming tools.

One opportunity to do so is allow users to introduce their own vocabulary to the tool, for instance, through domain-specific languages (DSL). DSLs provide focused expressive power within specific problem domains at the cost of generality [77]. For instance, domain-specific diagramming systems such as GraphViz allow succinct, high-level specification of diagrams and leverage domain knowledge to solve for diagram layouts algorithmically [17]. So far, DSLs have been most useful areas such as graph visualization, but they may prove to be useful elsewhere too. To allow end-users to introduce their own DSLs, *language workbenches* may be a viable implementation route. These workbenches allow efficient definition, reuse, and composition of DSLs [18], but much work remains, as existing language workbenches such as MPS [79] are still complex to learn for end-users like many of our participants.

Another way to model abstract and topological relationships is using high-level constraints, an idea that has existed since the invention of SketchPad [70]. Constraint-based systems are extensively used in Computer-Aided Design (CAD) tools [69]. CAD users commonly use constraints in *parametric drawing*, exploring different configurations of complex shapes. Automatic formatting of documents, digital drawings, and web pages are often modeled as constraints and can be optimized by solvers [34, 60]. To further simplify the process of constraint specification, some systems allow visual interactions with constraints [32, 22] while others intelligently infer constraints by examples [42]. By offloading the burden of low-level specification to constraint solvers, diagrammers often lose *control* of diagram elements, which poses usability challenges to future diagramming tools.

CONCLUSION AND FUTURE WORK

Conceptual diagrams are essential for understanding concepts, communicating ideas, and improving instructions effectively in many fields. This paper provides the first empirical study of how domain experts create conceptual diagrams.

Our results demonstrate *representation finding* as a vital step in the diagramming process and the role that sketches play in this step. However, due to limitations of current tools, notably the trade-offs between direct manipulation tools and programming languages, reusing representations is still challenging. As a result, diagrammers creatively circumvent these limitations by employing a set of *ad hoc* techniques to reuse diagram components and to scale up diagram production.

Based on our results, we introduce the concept of **natural diagramming** and four opportunities for natural diagramming support: exploration support, representation salience, live engagement, and vocabulary correspondence. For each of them, we discussed how recent advances from various research communities can help improve existing tools and design future tools.

Future work can leverage the substantial amount of conceptual diagrams that exist in the wild and perform large-scale analysis to gain a more quantitative understanding of existing diagrams. Similarly, it is possible to leverage the large number of current diagramming tools, some of which support a subset of natural diagramming. In particular, it may be possible to isolate and quantify the benefits of each opportunity of natural diagramming introduced in this paper by analysing existing tools. A deeper, more systematic analysis of the relationships and trade-offs among the four natural diagramming opportunities may further inform tool designers to make design decisions more critically. Further, this study does not focus on collaboration support for conceptual diagramming. Future work may explore how to support, for instance, *vocabulary correspondence* when multiple diagrammers from diverse backgrounds collaborate on a single conceptual diagram.

Natural diagramming embodies our vision for future diagramming tools—tools that seamlessly and naturally translate diagrammers' high-level ideas to beautiful and illustrative diagrams. This paper advances this goal by articulating a concrete vision for systems designers to create more effective diagramming tools.

ACKNOWLEDGMENTS

The questions raised in this study were inspired by many discussions with the Penrose team, especially with Keenan Crane and Jonathan Aldrich. We thank our participants for their helpful insight and for being willing to share their visualization work. We greatly appreciate Robert Ochshorn and Reduct for transcribing our interviews. We also thank Brad Myers, Michael Hilton, Mary Beth Kery, Adam Perer, Sarah Chasins, and Tatiana Vlahovic for their helpful feedback on our work. This material is based on work supported by the National Science Foundation under awards #1560137, #1852260, #1910264, and by the U.S. Department of Defense. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

REFERENCES

[1] 2019. Gallary of Concept Visualization. `http://conceptviz.github.io/`. (2019). Accessed: 2019-09-18.

[2] Shaaron Ainsworth, Vaughan Prain, and Russell Tytler. 2011. Drawing to learn in science. *Science* 333, 6046 (2011), 1096–1097.

[3] H Russell Bernard and Harvey Russell Bernard. 2013. *Social research methods: Qualitative and quantitative approaches*. Sage.

[4] Eliza Bobek and Barbara Tversky. 2016a. Creating visual explanations improves learning. *Cognitive Research: Principles and Implications* 1, 1 (2016), 27.

[5] Eliza Bobek and Barbara Tversky. 2016b. Creating visual explanations improves learning. *Cognitive Research: Principles and Implications* 1, 1 (07 Dec 2016), 27. DOI: http://dx.doi.org/10.1186/s41235-016-0031-6

[6] Alan Borning. 1981. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. Program. Lang. Syst.* 3, 4 (Oct. 1981), 353–387. DOI: http://dx.doi.org/10.1145/357146.357147

[7] John C. Bowman and Andy Hammerlindl. 2008. Asymptote: A Vector Graphics Language. *TUGboat: The Communications of the TEX Users Group* 29, 2 (2008), 288–294.

[8] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[9] Bret Victor. 2013. Drawing Dynamic Visualizations. (2013). https://vimeo.com/66085662 Stanford HCI Seminar.

[10] Bill Buxton. 2010. *Sketching user experiences: getting the design right and the right design*. Morgan kaufmann.

[11] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. 2007. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 557–566. DOI: http://dx.doi.org/10.1145/1240624.1240714

[12] Ravi Chugh. 2016. Prodirect manipulation: bidirectional programming for the masses. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 781–784.

[13] Ruth C Clark and Richard E Mayer. 2016. *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning*. John Wiley & Sons.

[14] Fehmi Dogan and Nancy J Nersessian. 2002. Conceptual diagrams: representing ideas in design. In *International Conference on Theory and Application of Diagrams*. Springer, 353–355.

[15] Elijah Meeks. 2018. Third Wave Data Visualization. (2018). https://youtu.be/itChfcTx7ao Tapestry Conference keynote.

[16] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 6059–6068. http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf

[17] John Ellson, Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. 2004. Graphviz and dynagraph–static and dynamic graph drawing tools. In *Graph drawing software*. Springer, 127–148.

[18] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Vol. 8225. Springer International Publishing, Cham, 197–217. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_11

[19] Stephen M Ervin. 1990a. Designing with diagrams: a role for computing in design education and exploration. *The Electronic Design Studio, The MIT Press, Cambridge, Massachusetts* (1990), 107–122.

[20] Stephen M. Ervin. 1990b. Designing with Diagrams: A Role for Computing in Design Education and Exploration. In *The Electronic Design Studio, The MIT Press, Cambridge, Massachusetts*. The MIT Press, Cambridge, Massachusetts, 107–122.

[21] John C Flanagan. 1954. The critical incident technique. *Psychological bulletin* 51, 4 (1954), 327.

[22] Michael Gleicher and Andrew Witkin. 1994. Drawing with Constraints. *Vis. Comput.* 11, 1 (Jan. 1994), 39–51. DOI:http://dx.doi.org/10.1007/BF01900698

[23] Vinod Goel. 1995. *Sketches of thought*. MIt Press.

[24] T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing* 7, 2 (June 1996), 131–174. DOI: http://dx.doi.org/10.1006/jvlc.1996.0009

[25] Mark D Gross. 1996. The electronic cocktail napkin–a computational environment for working with design diagrams. *Design studies* 17, 1 (1996), 53–69.

[26] Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

[27] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (July 2017), 1–119. DOI:http://dx.doi.org/10.1561/2500000010

[28] Pat Hanrahan. 2009. Systems of thought. *EuroVis 2009 keynote address* (2009), 10–12.

[29] Robert L Harris. 2000. *Information graphics: A comprehensive illustrated reference*. Oxford University Press, Chapter Conceptual diagram, 100.

[30] Mary Hegarty and Maria Kozhevnikov. 1999. Types of Visual–Spatial Representations and Mathematical Problem Solving. *Journal of Educational Psychology* 91, 4 (1999), 684–689. DOI: http://dx.doi.org/10.1037/0022-0663.91.4.684

[31] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 379–390. DOI: http://dx.doi.org/10.1145/2984511.2984575

[32] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by Manipulation for Layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 231–241. DOI: http://dx.doi.org/10.1145/2642918.2647378

[33] Christopher D Hundhausen and Jonathan L Brown. 2007. What You See Is What You Code: A âĂIJliveâĂİ algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing* 18, 1 (2007), 22–47.

[34] Nathan Hurst, Wilmot Li, and Kim Marriott. 2009. Review of automatic document formatting. In *Proceedings of the 9th ACM symposium on Document engineering*. ACM, 99–108.

[35] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human-computer interaction* 1, 4 (1985), 311–338.

[36] Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. 2018. Extending manual drawing practices with artist-centric programming tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 590.

[37] Rubaiat Habib Kazi, Tovi Grossman, Hyunmin Cheong, Ali Hashemi, and George W Fitzmaurice. 2017. DreamSketch: Early Stage 3D Design Explorations with Sketching and Generative Design.. In *UIST*. 401–414.

[38] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1265–1276. DOI:http://dx.doi.org/10.1145/3025453.3025626

[39] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.

[40] Kenneth R. Koedinger. 1992. Emergent Properties and Structural Constraints: Advantages of Diagrammatic Representations for Reasoning and Learning. In *Proc. AAAI Spring Symposium on Reasoning with Diagrammatic Representations*. 154–169.

[41] David R Krathwohl and Lorin W Anderson. 2009. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. Longman.

[42] David Kurlander and Steven Feiner. 1993. Inferring Constraints from Multiple Snapshots. *ACM Trans. Graph.* 12, 4 (Oct. 1993), 277–304. DOI: http://dx.doi.org/10.1145/159730.159731

[43] James A Landay and Brad A Myers. 1994. *Interactive sketching for the early stages of user interface design*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

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

[45] P. Lee, J. D. West, and B. Howe. 2018. Viziometrics: Analyzing Visual Information in the Scientific Literature. *IEEE Transactions on Big Data* 4, 1 (March 2018), 117–129. DOI: http://dx.doi.org/10.1109/TBDATA.2017.2689038

[46] Hannah Limerick, David Coyle, and James W. Moore. 2014. The Experience of Agency in Human-Computer Interactions: A Review. *Frontiers in Human Neuroscience* 8 (2014). DOI: http://dx.doi.org/10.3389/fnhum.2014.00643

[47] Catherine C Marshall and AJ Brush. 2004. Exploring the relationship between personal and public annotations. In *Proceedings of the 4th ACM/IEEE-CS joint conference on Digital libraries*. ACM, 349–357.

[48] Richard E. Mayer. 2002. Multimedia Learning. In *Psychology of Learning and Motivation*. Vol. 41. Academic Press, 85–139. DOI: http://dx.doi.org/10.1016/S0079-7421(02)80005-6

[49] Richard E Mayer. 2003. The promise of multimedia learning: using the same instructional design methods across different media. *Learning and instruction* 13, 2 (2003), 125–139.

[50] Sean McDirmid. 2007a. Living It Up with a Live Programming Language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 623–638. DOI: http://dx.doi.org/10.1145/1297027.1297073

[51] Sean McDirmid. 2007b. Living It Up with a Live Programming Language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 623–638. DOI: http://dx.doi.org/10.1145/1297027.1297073

[52] M. McKeon. 2009. Harnessing the Information Ecosystem with Wiki-Based Visualization Dashboards. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (Nov. 2009), 1081–1088. DOI: http://dx.doi.org/10.1109/TVCG.2009.148

[53] Dominik Moritz, Chenglong Wang, Gregory Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2019). http://idl.cs.washington.edu/papers/draco

[54] Brad A. Myers, Ashley Lai, Tam Minh Le, YoungSeok Yoon, Andrew Faulring, and Joel Brandt. 2015a. Selective Undo Support for Painting Applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 4227–4236. DOI: http://dx.doi.org/10.1145/2702123.2702543

[55] Brad A. Myers, Ashley Lai, Tam Minh Le, YoungSeok Yoon, Andrew Faulring, and Joel Brandt. 2015b. Selective Undo Support for Painting Applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 4227–4236. DOI: http://dx.doi.org/10.1145/2702123.2702543

[56] Brad A Myers, John F Pane, and Amy Ko. 2004. Natural programming languages and environments. *Commun. ACM* 47, 9 (2004), 47–52.

[57] Wode Ni, Katherine Ye, Joshua Sunshine, Jonathan Aldrich, and Keenan Crane. 2017. Substance and Style: domain-specific languages for mathematical diagrams. In *Domain-Specific Language Design and Implementation (DSLDI'17)*.

[58] Christopher Olah. 2015. Understanding LSTM Networks. (2015). http://colah.github.io/posts/2015-08-Understanding-LSTMs/ Accessed: 2019-09-18.

[59] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 14.

[60] S. Oney, B. A. Myers, and J. Brandt. 2013. Euclase: A Live Development Environment with Constraints and FSMs. In *2013 1st International Workshop on Live Programming (LIVE)*. 15–18. DOI: http://dx.doi.org/10.1109/LIVE.2013.6617342

[61] Casey Reas and Ben Fry. 2006. Processing: Programming for the Media Arts. *AI & SOCIETY* 20, 4 (Sept. 2006), 526–538. DOI: http://dx.doi.org/10.1007/s00146-006-0050-9

[62] Steven P. Reiss, Qi Xin, and Jeff Huang. 2018. SEEDE: Simultaneous Execution and Editing in a Development Environment. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 270–281. DOI: http://dx.doi.org/10.1145/3238147.3238182

[63] Clive Richards. 2002. The Fundamental Design Variables of Diagramming. In *Diagrammatic Representation and Reasoning*, Michael Anderson, Bernd Meyer, and Patrick Olivier (Eds.). Springer London, London, 85–102. DOI: http://dx.doi.org/10.1007/978-1-4471-0109-3_5

[64] Yann Riche, Nathalie Henry Riche, Ken Hinckley, Sheri Panabaker, Sarah Fuelling, and Sarah Williams. 2017. As We May Ink?: Learning from Everyday Analog Pen Use to Improve Digital Ink Experiences. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 3241–3253. DOI: http://dx.doi.org/10.1145/3025453.3025716

[65] Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John Stasko, John R. Thompson, Matthew Brehmer, and Zhicheng Liu. 2020. Critical Reflections on Visualization Authoring Systems. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2020). http://idl.cs.washington.edu/papers/reflections-vis-authoring

[66] Daniel L Schwartz, Jessica M Tsang, and Kristen P Blair. 2016. *The ABCs of how we learn: 26 scientifically proven approaches, how they work, and when to use them*. WW Norton & Company, Chapter V is for Visualization.

[67] Beau Sheil. 1983. Environments for exploratory programming. *Datamation* 29, 7 (1983), 131–144.

[68] B. Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (1983), 57–69. DOI: http://dx.doi.org/10.1109/MC.1983.1654471

[69] Johannes Strömberg. 2006. Integrating Constraints with a Drawing CAD Application. *Stockholm University* (2006).

[70] Ivan E Sutherland. 1964. Sketchpad a man-machine graphical communication system. *Simulation* 2, 5 (1964), R–3.

[71] Masaki Suwa and Barbara Tversky. 1997. What do architects and students perceive in their design sketches? A protocol analysis. *Design studies* 18, 4 (1997), 385–403.

[72] Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. *J. Vis. Lang. Comput.* 1, 2 (June 1990), 127–139. DOI: http://dx.doi.org/10.1016/S1045-926X(05)80012-6

[73] Till Tantau. *The TikZ and PGF Packages*. http://sourceforge.net/projects/pgf/

[74] William P. Thurston. 1998. On Proof and Progress in Mathematics. *New directions in the philosophy of mathematics* (1998), 337–355.

[75] Christine D. Tippett. 2016. What Recent Research on Diagrams Suggests about Learning with Rather than Learning from Visual Representations in Science. *International Journal of Science Education* 38, 5 (March 2016), 725–746. DOI: http://dx.doi.org/10.1080/09500693.2016.1158435

[76] Linus Torvalds and Junio Hamano. 2010. Git: Fast version control system. *URL http://git-scm. com* (2010).

[77] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.* 35, 6 (June 2000), 26–36. DOI:http://dx.doi.org/10.1145/352029.352035

[78] Bret Victor. 2011. Up and Down the Ladder of Abstraction: A systematic approach to interactive visualization. (2011). http://worrydream.com/LadderOfAbstraction/ [Accessed: 2019-09-17].

[79] M. Voelter and V. Pech. 2012. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. 1449–1450. DOI: http://dx.doi.org/10.1109/ICSE.2012.6227070

[80] J. Walny, S. Carpendale, N. Henry Riche, G. Venolia, and P. Fawcett. 2011. Visual Thinking In Action: Visualizations As Used On Whiteboards. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec 2011), 2508–2517. DOI: http://dx.doi.org/10.1109/TVCG.2011.251

[81] Hadley Wickham. 2010. A layered grammar of graphics. *Journal of Computational and Graphical Statistics* 19, 1 (2010), 3–28.

[82] Wikipedia contributors. 2019. Diffie-Hellman key exchange — Wikipedia, The Free Encyclopedia. (2019). https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange [Online; accessed 05-April-2019].

[83] Eric Willigers, Chris Lilley, Dirk Schulze, Bogdan Brinza, David Storey, and Amelia Bellamy-Royds. 2018. *Scalable Vector Graphics (SVG) 2*. Candidate recommendation. W3C. https://www.w3.org/TR/2018/CR-SVG2-20181004/.

[84] Katherine Ye, Keenan Crane, Jonathan Aldrich, and Joshua Sunshine. Designing extensible, domain-specific languages for mathematical diagrams. In *Off the Beaten Track (OBT'17)*.

[85] Young Seok Yoon and Brad A Myers. 2014. A longitudinal study of programmers' backtracking. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 101–108.

[86] Jiaje Zhang and Donald A Norman. 1994. Representations in distributed cognitive tasks. *Cognitive science* 18, 1 (1994), 87–122.