On Vector Graphics Substrates and their Potential

for Postmodern, Phenotropic Notational Freedom

Joel Jakubovic[a] and Camille Gobert[b]

a Charles University, Prague

b Université Paris-Saclay, CNRS, Inria, Laboratoire Interdisciplinaire des Sciences du Numérique

Abstract Pencil-and-paper supports a naïve "notational freedom" where notations can be conjured up on demand. Although programming was born in such a world, it has since been locked in to textual notation. Efforts to support alternative notations have required them to be designed long in advance, or have followed a "Modernist" paradigm demanding investment in a particular drawing, programming, and storage ecosystem. We propose a "Postmodern" alternative: treat SVG diagrams as a *substrate* for notational freedom. By exploiting the *de-facto* status of SVG across editors and browsers, we can largely avoid reinventing various wheels and instead focus on the missing piece: the interpretation and macro-expansion of vector graphics as programs. We sketch a "Vector Vision Engine" (VVE) that extracts spatial properties from SVG diagrams, executes embedded code, and understands user-defined ad-hoc notations. *Static* diagrams, whose processing terminates in arbitrary output, contrast with *"self-raising"* diagrams which bootstrap themselves into interactive apps. We describe some principles and challenges of such an architecture, and close with some open questions for discussion.

Keywords programming, diagrams, postmodernism, notations, substrates, vector graphics, computer vision, vector vision, domain-specific languages, domain-specific notations

The Art, Science, and Engineering of Programming

© Joel Jakubovic and Camille Gobert
This work is licensed under a "CC BY 4.0" license
In *The Art, Science, and Engineering of Programming*; 12 pages.

1 Resurrecting the Old "Notational Freedom"

Although programming is mostly known as a text-centric activity, diagrammatic notations were more common at its inception than we might expect. Arawjo [3] reports that early programs from the 1940s were initially described using rich and diverse notations, such as Zuse's algebra for his *Plankalkül* and the diagrams for ENIAC programs. It was the mass use of *teletype interfaces* that would later commit the enterprise to text. In the following decades, programmers adapted to this notational lock-in; perhaps by drawing on paper or whiteboards [5, 8, 22] or by embedding "ASCII diagrams" in comments and commit messages [12].

This persistent desire for diagrams in programming is reflected in early research on programming systems. Starting from the late 1970s, systems such as ThingLab [7], Boxer [9], Fabrik [13] and Max [25] pioneered the idea of programming by directly manipulating box-shaped objects in a 2D space. Their inability to displace textual dominance motivated hybrid solutions, whereby programmers could write most of the code as text while interacting with specific fragments as domain-specific diagrams. These notably include state machines in Prolog [10] and in C [27], Red-Black trees in Racket [2] and Rocq [24], and electronic [21] and quantum [4] circuitry in Python.

Unfortunately, these systems still fall short of hand-drawn diagrams by providing little in the way of "notational *freedom*". The notations must be designed well in advance and "hard-coded" at considerable cost; the user cannot make up a notation "on the spot" like they can on paper. Projects that go a long way towards realising this capability include software architectures for notational extension, such as Barista [19] and MPS [28], as well as works on interactive programming with hand-drawn graphics, such as Ink & Switch's "programmable ink" experiments [1] and "scoped propagators". [2]

However, such efforts *in turn* tend to follow a *"Modernist"* approach: in order to program with the notations they make possible, one must *commit* to their own version of drawing, programming, and storage functionality; one can't rely on compatibility with other systems. The developers of such a system must also expend considerable resources on creating such functionality. All of this contrasts with more *"Postmodern"* solutions, such as the aforementioned ASCII diagrams, which are compatible with every established programming language and can effectively be read and written by anyone, using any text editor, with the only commitment being that of a text encoding scheme. While not without its downsides, this default "openness" is a compelling advantage.

As potential users and developers of notational freedom, we feel wary of the demands made by Modernist solutions. We're not eager to reimplement features already available in existing software, nor do we look forward to having to convince other users to commit to those implementations. Instead, we follow Kell's call for more Postmodern approaches to programming [16], as well as our respective vision statements from last year's workshop [11, 15], and explore how we might augment a ubiquitous data format—SVG images—into a (programmable) substrate for notational freedom.

[1] https://www.inkandswitch.com/ink/
[2] "Scoped Propagators", Orion Reed, 2024 (https://www.youtube.com/watch?v=zz1tfE6i6qU).

2 A Postmodern Alternative: Vector Graphics Substrates

We reckon that most of what everybody wants from programming notations is covered by vector graphics (VG). Unlike raster bitmaps, we can recognise patterns in VG without the need for advanced and expensive computer vision techniques, just like ChartDetective [23] can reconstruct data tables from vector graphics plots. Specifically, let's seize on the SVG format and exploit its wide editor and browser support. Let's reduce our labour requirements and our demands for user commitment to the precise thing that's *missing* from the existing SVG and programming ecosystems: the interpretation and transformation of SVG pictures. This approach lets users "bring their own client"[3] in at least two respects: the vector graphics editor and the viewing software. By avoiding the Modernist approach, we relieve ourselves of any need to reinvent those wheels.

The concrete technical artefact which we propose to "slot in" to the existing SVG ecosystem is a "Vector Vision Engine" (VVE), which interprets and evolves its input SVG document. Assuming the VVE is written in JavaScript, it would let anyone draw a static image in their preferred vector graphics editor[4]; export it to SVG; and open the SVG file in a web browser to run the VVE on it.

Joel's prototype VVE is a JavaScript library operated from the browser developer tools, acting on the SVG in the DOM.[5] Generally, a VVE should do the following work:

1. Extract discrete relationships (e.g., shape containment, connector endpoint incidence, connector directedness) and cache them in the document as IDs, CSS classes, `data-` attributes, and so on (Figure 1).[6]
2. Identify "native" code embedded as text in the diagram, strip all formatting, and execute it. (Joel's prototype VVE looks for specially-marked boxes containing JavaScript code; see Figures 3 and 4.)
3. Process any "meta-notation" in the diagram which specify the user's ad-hoc notations. (Not yet implemented.)
4. Process further parts of the diagram according to their specified notations. (Currently highly unstable and experimental.)

By combining the *notational* affordances of SVG and the *programmatic* affordances of the JavaScript engine, we end up with a (programmable) *Vector Graphics Substrate*.

[3] https://www.geoffreylitt.com/2021/03/05/bring-your-own-client
[4] All of the diagrams in this paper were created in Mathcha (https://www.mathcha.io/editor).
[5] Joel has a GitHub repository (https://github.com/jdjakub/self-raising-diagrams) and a blog post with more detail about an earlier state of the project [14].
[6] Following TeX's naming conventions [18], this stage could be called the "Vector Visual Cortex"; the rest is perhaps more "cognition" than "vision".

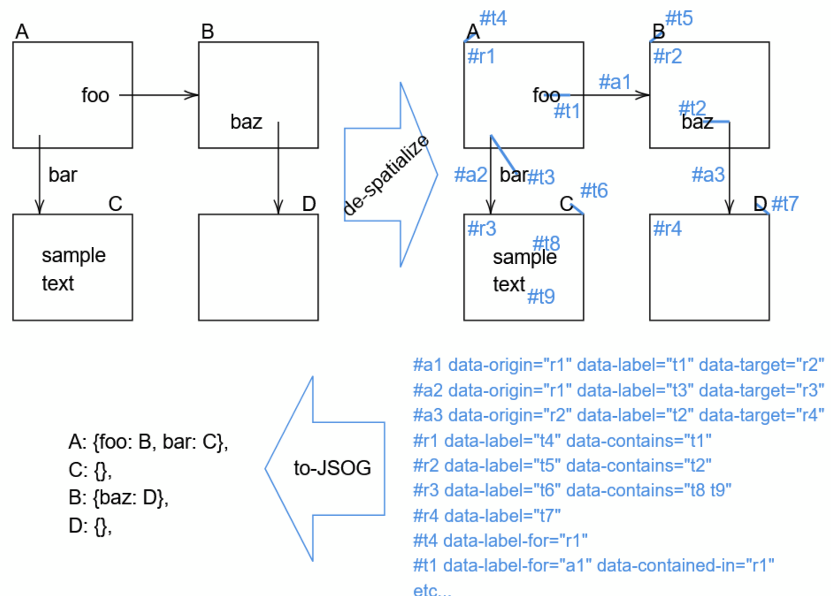


Figure 1 An SVG diagram in the "BoxGraph" notation (top-left) is processed by the VVE. First, apply "Vector Vision" to turn spatial / topological properties into DOM attributes and assign IDs to everything ("de-spatialize"). The output is a modified SVG document (depicted top- and bottom-right). From this point, we can do whatever we want with the extracted "semantic" content of the notation. Here, we create a JavaScript object graph with the same structure as the diagram.

2.1 Static vs. Interactive (Self-Raising) Diagrams

In a *static diagram,* certain notational patterns are "macro-expanded" into further patterns, mutating the document. Any intermediate state is a valid, visible SVG document that can be saved and resumed later. At the final step, patterns are used to produce some non-SVG output, and the engine exits. For example, BoxGraph (Figure 1) outputs JavaScript objects with the structure implied by the diagram; BitsTable (Figure 2) outputs a C struct capturing the bit ranges of the fields in the table. Static diagrams are like "executable documentation": instead of drawing a diagram for human eyes, and manually synchronising the program you have to write, you can just derive the program *from* the documentation.

An *interactive* or *"self-raising"*[7] diagram might macro-expand similarly at first. However, at some point, it is left with JavaScript code for setting up event handlers (see Figure 3 for such a final state). This code is then executed, and the diagram "raises itself" into an interactive web app. It may continue to use VVE functionality on-demand. Instead of the "one-shot" output of a static diagram, a self-raising diagram ends in a continuous interaction with the user.

[7] Joel was thinking of flatpack furniture, origami architecture, and self-extracting archives when he cooked up the name. Improvements welcome!

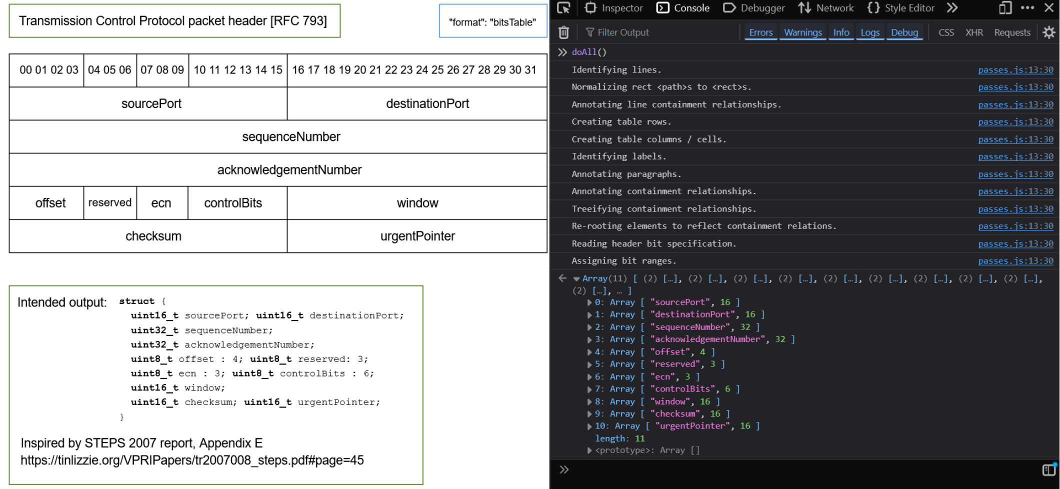


Figure 2 An SVG diagram in the "BitsTable" notation (left) functions as "executable documentation" describing the layout of a TCP packet. It was drawn in a few minutes as a rectangle, horizontal/vertical lines, and text labels (the header row is one long text label). The box below is a "comment", ignored by processing because its border is a very specific "magic" shade of green (#417505). The browser console on the right shows a log of the VVE processing steps and the final output (a small step away from the comment's C struct code).

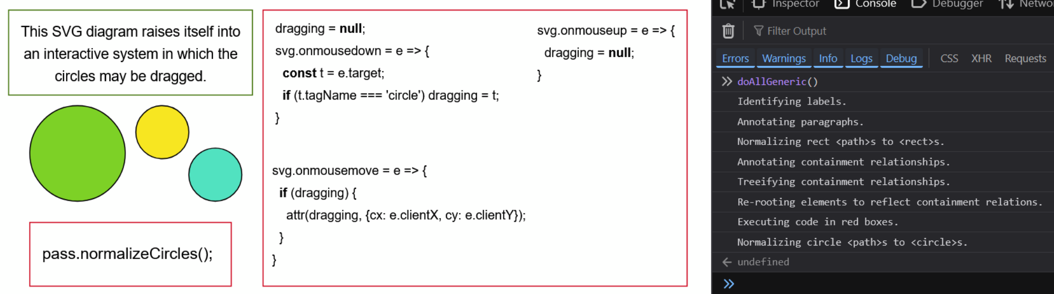
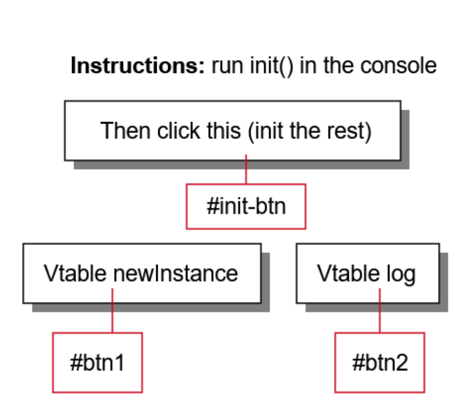


Figure 3 The minimal "Hello World" of self-raising diagrams. Boxes with border colour "Magic Red" (#D0021B) cause their contents to be executed. One such box (left) calls a DOM rewriting operation—the mathcha.io editor exports circles as Bézier <path> elements, but the "circle" concept is semantically meaningful, so we substitute accordingly (normalizeCircles). The other box sets up event handlers that allow the resulting <circle> elements to be dragged. The equivalent JavaScript program would of course contain the same JavaScript as this diagram, but it would additionally need to *describe* the appearance of the circles *in code;* in a self-raising diagram, we at least have the option to *draw* graphical things.




```
  Vtable = send(boxGraph, 'boxNamed:', 'Vtable');
  id_vtable_lookup = send(Vtable, 'methodAt:', 'lookup');

  id_send   = send(send(boxGraph, 'boxNamed:', 'id_send'), 'asJSFunc');
  id_bind   = send(send(boxGraph, 'boxNamed:', 'id_bind'), 'asJSFunc');
  id_vtable = send(send(boxGraph, 'boxNamed:', 'id_vtable'), 'asJSFunc');
}

const btnClick = e => {
  const rect = e.target;
  const para = rect.parentElement.querySelector('g.is-paragraph');
  const str = para.dataset.string;
  const [objName, selector] = str.split(' ');
  id_send(window[objName], selector);
}

byId('btn1').onclick = btnClick;
byId('btn2').onclick = btnClick;

byId('init-btn').onclick = () => {
  parseBoxGraph(); parseAsObjModel(); log('Initialised.');
}
```

Figure 4 "Magic Red" boxes may contain JavaScript code (right) *or* an element ID (left). Element IDs are first applied to their associated elements (identified here by the free endpoint on the "magic red" path). Then, the contents of all JavaScript boxes are executed; with known IDs at authoring time, the author's code can refer to them. Hence, with these two primitives, anything is possible. Here, they are used to make clickable buttons out of rectangles by adding event handlers.

Some caveats: unlike their static counterpart, self-raising diagrams require a specific runtime environment (the browser), risking our viewer-agnosticism. Moreover, since we duplicate editing functionality *inside* the diagram-app, editor-agnosticism is relegated merely to the *initial* bootstrapping diagram—in the extreme case, raising itself into a Modernistic editing environment and undermining the entire approach.

2.2 The Model Is (In) The View

In the Modernist approach, a semantic "model" is maintained separately from its rendered "view", requiring bespoke rendering/editing infrastructure and file formats. In a VG Substrate, the persistent medium (SVG) *is* the view, so we inherit existing editor tooling for free. Now the model is *parsed out from* the view rather than *rendered to* it; we call this The Model Is (In) The View (TMIITV). There is a clear analogy to plain text: semantics are *encoded inside* the presentation medium. The more general idea of approximate pattern recognition has been conceptualised as "phenotropic" programming [20].

The main challenge with this is *layout-affecting mutations*, where inserting or deleting an element may cause cascading changes to positions and sizes of many shapes. We could capitulate to a hidden synchronised model (defeating TMIITV), but we'd first like to try the *incremental* approach to layout, drawing on literature in self-adjusting computation [1, 26].

2.3 No Hidden State

VG Substrates also invite us to strengthen the "externalisability" principle [6]. An example of externalisability is in LopeCode,[8] where users may serialise the current state of a dynamic notebook into a single HTML file; later, this file can be opened and the notebook will resume. For VG Substrates, not only must we be able to externalise the full state of a running interactive diagram (in a form that we can resume later), we also desire that the behaviour of notational elements should depend only on how they *look* in a viewer, not on invisible DOM details. Two *visually* identical diagrams should *behave* identically. However, this is an aspirational goal for realistic usage; we aren't worried about the inevitable pathological cases. For more detail, see Appendix A.

3 Future Directions

We envision a number of future directions to build upon this work:

- Use WebStrates [17] to make notation programming *collaborative* for free (although the distributed execution of self-raising diagrams might cause issues).
- Expose the browser's SVG APIs with user-friendly "handles" (e.g., supply coordinate parameters via a mouse click) to get basic interactive diagram editing for free.
- Distinguish "editing within the rules of a notation" (dragging a box moves the arrows along with it; cannot delete arrowheads) from "editing at a lower level" (VG editor operations that might break the notation).
- Create a postmodern, Flash-like animation editor assembled from a vector graphics editor, a web browser, an advanced VVE, and self-raising notations for animation, reacting to input, etc.

4 Prompts for Discussion

- What sort of thing deserves the name "Vector Graphics Substrate"—should we already call SVG itself a substrate? If not, what else do we need to add?
- SVG permits embedded text strings; does this mean VG substrates strictly *generalise* all textual substrates?
- How important are "robustness under perturbations" and "approximate correctness" for something to qualify as a substrate?
- Is "The Model Is (In) The View" a natural fit for an application like Python Tutor implemented in a VG Substrate?
- What conditions affect whether the modernist vs. postmodern approach is "worth it"? Does the answer differ between the developer and the user perspectives?

[8] https://tomlarkworthy.github.io/lopecode/

- Suppose that advanced AI image perception were available under ideal conditions (instant response, for free, no usage limits, no environmental impact, etc.). Does this make the VG substrates approach *obsolete* for attaining Notational Freedom?[9]
- Under the same assumptions, do the above “substrate” questions have the same answers for raster bitmaps?

A No Hidden State

VG substrates are a good opportunity to enshrine discipline in externalisability [6]. The amount of “divergence” between the runtime state of the self-raising diagram, and the persistable document, must be minimised. A static diagram must contain all the information needed by the VVE to evolve it to its next state: the VVE reads from the document and writes back to it. For a running interactive diagram, one ought to be able to pause it, save the XML, and resume it at a later date by opening it again. Fancy JavaScript code and data structures may be created to optimise or cache operations that *conceptually* take place in the diagram, but the user should not have to care about this. This principle of “all state is in the document” suffices for externalisability. However, this does not quite satisfy us here: there is plenty of room for state that is part of the SVG *document* yet not part of the *diagram*, because it does not correspond to any visual element.

Hence, a stronger principle of “No Hidden State”. The spirit of “Vector Vision” is that the semantics or behaviour of notational elements should only depend on how they *look* in a viewer, not on invisible details of the underlying DOM tree or XML text. Many perturbations of the DOM leave the appearance unchanged; for example, permuting elements whose shapes don’t overlap, or overwriting a circular `<path>` with an identical-looking `<circle>` element (*cf.* `normalizeCircles` in Figure 3). If two diagrams look identical in the viewer, but behave differently because they look different in the DOM inspector, their behaviour depends on “hidden state” and violates the principle. This principle is only meant as an aspirational goal to be pursued reasonably. Although pathological cases exist,[10] the point is that users who use VG tools in a realistic way shouldn’t ever have to fiddle with the XML to make their notation work.

Of course, when we allow the diagrams to be *interactive,* the matter of what counts as “invisible state” gets more complicated. If a self-raising UI can hide or collapse certain elements, it would not make sense to mandate that they are somehow always visible (perhaps with a small graphic meaning “Warning! There is some hidden state here!”). The principle is more important for static diagrams, where show/hide interactions are by definition not possible.

[9] We have responded to a less speculative analogue of this question, concerning the relevance of current AI, in Appendix B.

[10] For example, since it is possible to simulate raster bitmaps in vector graphics by means of tiny squares, a fully general solution would require precisely the computer vision that we chose to avoid—there’s no way around it.

00 01 02 03	04 05 06	07 08 09	10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
anon_field_1				anon_field_2
anon_field_3				
anon_field_4				
anon5	anon6	anon7	anon8	anon9
anon10				anon11

"Please look at the table in this picture and generate a C struct definition for the anonymous field names and their bit offsets / widths, as judged by the vertical lines and the bit indexes in the header row."

Claude Sonnet 4.6:

```
struct anon_packet {
    uint16_t anon_field_1; uint16_t anon_field_2;
    uint32_t anon_field_3;
    uint32_t anon_field_4;
    uint32_t anon5 : 4; uint32_t anon6 : 3; uint32_t anon7 : 3;
    uint32_t anon8 : 6; uint32_t anon9 : 16;
    uint32_t anon10 : 16; uint32_t anon11 : 16;
};
```

Figure 5 The SVG text labels were first edited in the browser Inspector to anonymise their identities, to ensure that the results would be due to perception rather than drawing from trained knowledge about TCP packets. The table itself, with no surrounding context, was then screenshotted and pasted into the Claude Sonnet 4.6 chat window, along with the prompt. After some thinking in English (omitted from the figure), Claude's response ended with a more verbose version of the struct depicted — its C comments were stripped and the whitespace was altered to fit in the figure. Response time was not precisely measured, but felt immediate as is expected of the basic Sonnet tier. Interestingly, Claude referred to it as a "packet" from the beginning and named the struct `anon_packet`.

Another complication is that, given suitable editing operations, it *becomes* possible to tell different DOM structures apart. One shape can be dragged over another, and whether it covers or is covered by the other shape reveals which one comes first in the DOM tree traversal. Whether a rectangle is a `<rect>` element or just four `<line>`s would become apparent when selecting and manipulating different parts of it. We can also just posit a self-raising DOM inspector, or a smaller version, which shows the tag name of the selected element. This suggests it might be better to expose "equivalence class" detection as optional functionality, or to make it explicit in the diagrams via some metadata notation.

B The Role of AI in VG Substrates

In response to workshop feedback, we will give our outlook on the role of AI. To start with, it is beyond doubt that AI image perception substitutes for Vector Vision in *some* use cases of *static* diagrams. As a quick test, Claude Sonnet 4.6 generates correct C struct code for Figure 2's TCP packet from just a screenshot and a paragraph-length prompt (Figure 5). However, while this is very promising and convenient, it does not obsolete Vector Vision code *in general* for static diagrams. Locally-running code may

be read, reasoned about, and tweaked, while even *locally-run* AI models are black boxes. For some applications, the precision and predictability of Vector Vision code will be more appropriate than trusting an AI; put another way, the failure modes of code are less open-ended than the ways that an AI might misinterpret a diagram or a prompt. For *external* AI use, token pricing, service availability, and response latency must also be considered; for local AI use, hardware requirements and compute cost are sure to be greater than the same for "good-enough" Vector Vision code. A natural compromise would be to have AI write the Vector Vision code instead, for some definition of "good enough", to be re-used across many perception tasks.

In the case of *self-raising* diagrams, it is harder to see how AI perception could be beneficially integrated. If a diagram-app is rapidly changing in response to user input (e.g. the layout-affecting mutations in §2.2), it is enough of an engineering challenge to make the relevant Vector Vision code performant enough to keep the user experience smooth. Calling out to even a local AI model on a continuous basis would seem to be at least as hard a challenge. Nevertheless, since a self-raising diagram should have on-demand access to the VVE (§2.1), on-demand access to AI-powered Vector Vision is a natural extension. As with static diagrams, we expect the most sure-fire benefit of AI to be in generating the Vector Vision code which can then be run cheaply and frequently. However, we are certainly interested to explore direct AI perception further down the line in this research.

References

[1] Umut A Acar. "Self-adjusting computation: (an overview)". In: *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. 2009, pages 1–6.

[2] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. "Adding Interactive Visual Syntax to Textual Code". In: *Proceedings of the ACM on Programming Languages*. Volume 4. ACM, 2020, pages 1–28. DOI: 10.1145/3428290.

[3] Ian 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*. CHI '20. ACM, 2020, pages 1–15. DOI: 10.1145/3313831.3376731.

[4] Ian Arawjo, Anthony DeArmas, Michael Roberts, Shrutarshi Basu, and Tapan 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*. UIST '22. ACM, 2022, pages 1–20. DOI: 10.1145/3526113.3545619.

[5] Sebastian Baltes and Stephan Diehl. "Sketches and Diagrams in Practice". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. ACM, 2014, pages 530–541. DOI: 10.1145/2635868.2635891.

[6] Antranig Basman, Luke Church, Clemens N. Klokmose, and Colin B. D. Clark. "Software and How it Lives On: Embedding Live Programs in the World Around Them". In: *PPIG*. 2016. URL: http://www.klokmose.net/clemens/wp-content/uploads/2016/10/ppig-2016.pdf.

[7] Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory*. Technical report SSL-79-3. Xerox PARC, 1979.

[8] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. "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. LaTeX2Solver: a Hierarchical Semantic Parsing of LaTeX Document into Code for an Assistive Optimization Modeling Application, 2007, pages 557–566. DOI: 10.1145/1240624.1240714.

[9] Andrea A. diSessa and Harold Abelson. "Boxer: A Reconstructible Computational Medium". In: *Communications of the ACM* 29.9 (1986), pages 859–868. DOI: 10.1145/6592.6595.

[10] Martin Erwig and Bernd Meyer. "Heterogeneous Visual Languages-Integrating Visual and Textual Programming". In: *1995 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Comput. Soc. Press, 1995, pages 318–325. DOI: 10.1109/VL.1995.520825.

[11] Camille Gobert. "Designing Postmodern Substrate Architectures". In: *Substrates'25 Workshop*. 2025, pages 1–8.

[12] Devamardeep Hayatpur, Brian Hempel, Kathy Chen, William Duan, Philip Guo, and Haijun Xia. "Taking ASCII Drawings Seriously: How Programmers Diagram Code". In: *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. CHI '24. ACM, 2024, pages 1–16. DOI: 10.1145/3613904.3642683.

[13] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. "Fabrik: A Visual Programming Environment". In: *ACM SIGPLAN Notices* 23.11 (1988), pages 176–190. DOI: 10.1145/62084.62100.

[14] Joel Jakubovic. *PAINT'25 Invited Talk Transcript: Notational Freedom via Self-Raising Diagrams*. 2025. URL: https://programmingmadecomplicated.wordpress.com/2025/11/04/paint25-invited-talk-transcript-notational-freedom-via-self-raising-diagrams/.

[15] Joel Jakubovic. "Substrates Vision Statement". In: *Substrates'25 Workshop*. 2025, pages 1–9.

[16] Stephen Kell. "The Operating System: Should There Be One?" In: *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*. PLOS '13. ACM, 2013, pages 1–7. DOI: 10.1145/2525528.2525534.

[17] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. "Webstrates: Shareable Dynamic Media". In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. UIST '15. ACM, 2015, pages 280–290. DOI: 10.1145/2807442.2807446.

[18] Donald E. Knuth. *The TeXbook*. Computers & Typesetting. Addison-Wesley, 1984. ISBN: 978-0-201-13448-3.

[19] Amy J. Ko and Brad A. Myers. "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. ACM, 2006, pages 387–396. DOI: 10.1145/1124772.1124831.

[20] Clayton Lewis. "Phenotropic Programming?" In: *PPIG*. 2018.

[21] Richard Lin, Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Bjoern Hartmann. "Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages". In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. ACM, 2021, pages 1039–1049. DOI: 10.1145/3472749.3474804.

[22] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and Andre van der Hoek. "How Software Designers Interact with Sketches at the Whiteboard". In: *IEEE Transactions on Software Engineering* 41.2 (2015), pages 135–156. DOI: 10.1109/TSE.2014.2362924.

[23] Damien Masson, Sylvain Malacria, Daniel Vogel, Edward Lank, and Géry Casiez. "ChartDetective: Easy and Accurate Interactive Data Extraction from Complex Vector Charts". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, 2023, pages 1–17. DOI: 10.1145/3544548.3581113.

[24] Clément Pit-Claudel. "Untangling Mechanized Proofs". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2020, pages 155–174. DOI: 10/ghs5sn.

[25] Miller Puckette. "Max at Seventeen". In: *Computer Music Journal* 26.4 (2002), pages 31–43. DOI: 10.1162/014892602320991356.

[26] Ganesan Ramalingam and Thomas Reps. "A categorized bibliography on incremental computation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1993, pages 502–510.

[27] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. "Lessons Learned from Developing Mbeddr: A Case Study in Language Engineering with MPS". In: *Software & Systems Modeling* 18.1 (2019), pages 585–630. DOI: 10.1007/s10270-016-0575-4.

[28] Markus Voelter and Sascha Lisson. "Supporting Diverse Notations in MPS' Projectional Editor". In: *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages*. 2014, pages 7–16.