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

In *The Art, Science, and Engineering of Programming*; 10 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, 20] 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 [23] 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 [25], Red-Black trees in Racket [2] and Rocq [22], and electronic [19] 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. Works by Bret Victor,[1] Ink&Switch,[2] and JetBrains[3] go a long way towards realising this capability.

However, such efforts 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 established programming languages and tooling. They 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 (e.g. drawing rectangles), 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* an substrate for notational freedom.

[1] "Drawing Dynamic Visualisations", 2013 (https://vimeo.com/66085662).
[2] https://www.inkandswitch.com/inkbase and https://www.inkandswitch.com/crosscut
[3] https://www.jetbrains.com/mps/

2 A Postmodern Alternative: Vector Graphics Substrates

We reckon that 90% of what everybody wants from programming notations is covered by Vector Graphics (VG), as opposed to raster bitmaps.[4] Unlike the latter, we can recognise and "parse" patterns in VG without the need for advanced and expensive Computer Vision / AI (see, e.g. ChartDetective [21]). Specifically, let's seize on the SVG format and exploit its *de-facto* nature; it has 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. The user can then "bring their own client"[5] in at least two respects: the VG 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.

The workflow would be as follows:

1. Draw a static diagram in your preferred VG editor (all of the diagrams in this paper were created in Mathcha[6]).
2. Export it to SVG.
3. Run the VVE on it.

Joel's prototype VVE is a JS library operated from the browser developer tools, acting on the SVG in the DOM. The 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).[7]
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 JS 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 browser JS engine, we end up with a (programmable) *Vector Graphics Substrate*.

[4] Apart from the Piet esolang, we can't think of any programming notation that requires more than VG — and might we even vectorise a bitmap as lots of small coloured squares?

[5] https://www.geoffreylitt.com/2021/03/05/bring-your-own-client

[6] https://www.mathcha.io/editor

[7] Following TEX's naming conventions, this stage could be called the "Vector Visual Cortex"; the rest is perhaps more "cognition" than "vision".

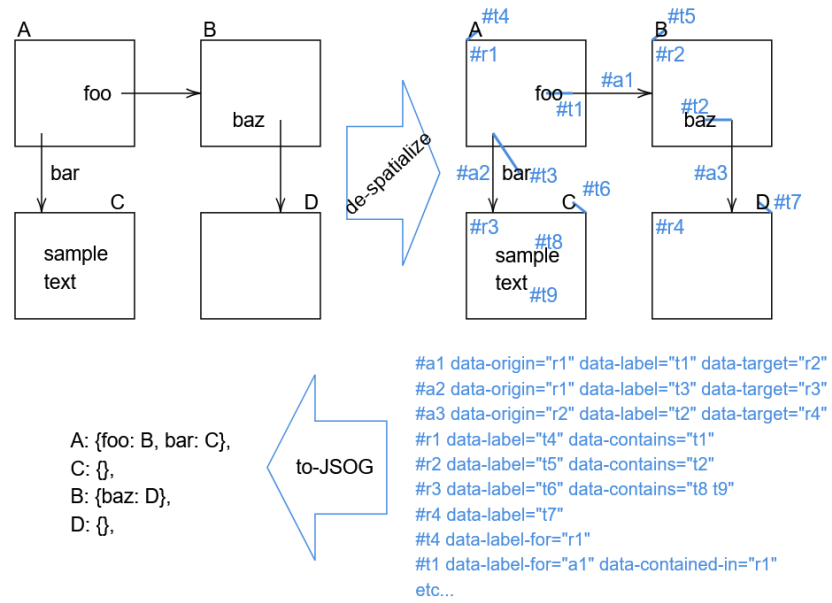

Figure 1 An SVG diagram in the "BoxGraph" notation (top-left) is processed by the VVE. First, apply "Vector Vision" to turn spatial / topological properties into DOM attributes and assign IDs to everything ("de-spatialize"). The output is a modified SVG document (depicted top- and bottom-right). From this point, we can do whatever we want with the extracted "semantic" content of the notation. Here, we create a JS 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 JS 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"*[8] *diagram* might macro-expand similarly at first. However, at some point, it is left with JS 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.

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

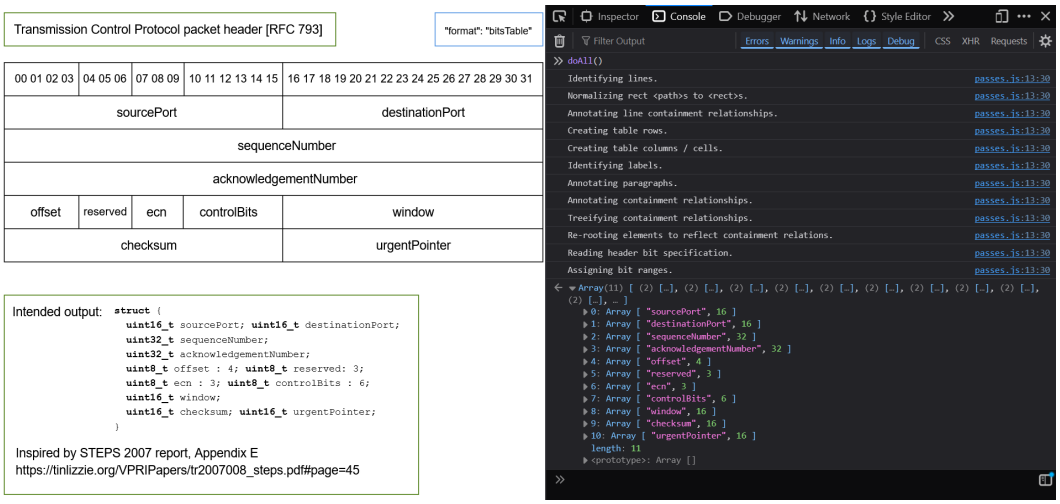

Figure 2 An SVG diagram in the "BitsTable" notation (left) functions as "executable documentation" describing the layout of a TCP packet. It was drawn in a few minutes as a rectangle, horizontal/vertical lines, and text labels (the header row is one long text label). The box below is a "comment", ignored by processing because its border is a very specific "magic" shade of green. 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).

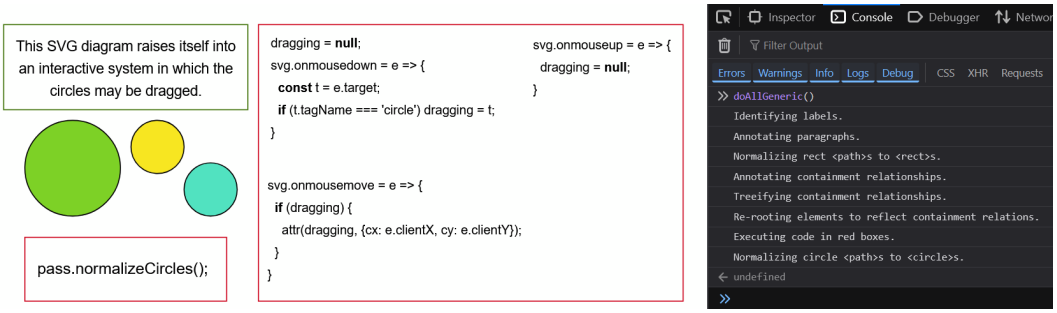

Figure 3 The minimal "Hello World" of self-raising diagrams. Boxes with border colour "Magic Red" (#D0021B) cause their contents to be executed. One such box (left) calls a DOM rewriting operation — the mathcha.io editor exports circles as Bézier <path> elements, but the "circle" concept is semantically meaningful, so we substitute accordingly (normalizeCircles). The other box sets up event handlers that allow the resulting <circle> elements to be dragged. The equivalent JS program would of course contain the same JS 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.

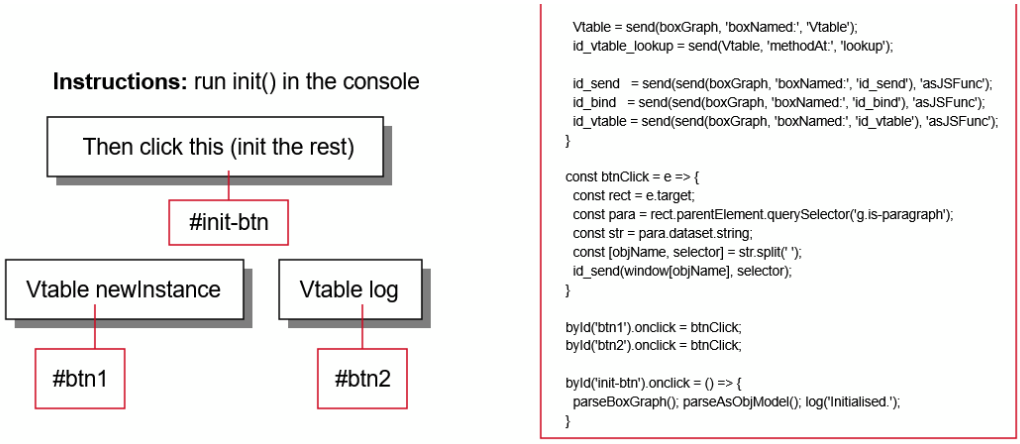

Figure 4 "Magic Red" boxes may contain JS code (right) *or* an element ID (left). Element IDs are first applied to their associated elements (identified here by the free endpoint on the "magic red" path). Then, the contents of all JS boxes are executed; with known IDs at authoring time, the author's 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: self-raising diagrams require a specific runtime environment (in our case, the browser), risking our viewer-agnosticism. Moreover, to the extent that 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 [18].

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, 24].

2.3 No Hidden State

VG Substrates also invite us to strengthen the "externalisability" principle [6]. Not only must we be able to externalise the full state of a running interactive diagram;

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. (This is an aspirational goal for realistic usage; we aren't worried about the inevitable pathological cases.)

3 Future Directions

- Run it on WebStrates [17] and get *collaborative* notation programming for free(?)
- Expose the browser's SVG APIs with user-friendly "handles" (e.g. supply coordinate parameters via a mouse click) and get basic interactive diagram editing for free.
- We must distinguish "editing within the rules of a notation" (dragging a box moves the arrows along with it; cannot delete arrowheads) vs. "editing at a lower level" (VG editor operations that might break the notation).
- A postmodern, emergent Flash Studio assembled from a VG editor, the browser, an advanced VVE, and self-raising notations for animation, reacting to input, etc.

4 Prompts for Discussion

1. 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?
2. SVG permits embedded text strings; does this mean VG substrates strictly *generalise* all textual substrates?
3. How important are "robustness under perturbations" and "approximate correctness" for something to qualify as a substrate?
4. Is "The Model Is (In) The View" a natural fit for an application like Python Tutor implemented in a VG Substrate?
5. What conditions affect whether the modernist vs. postmodern approach is "worth it"? Does the answer differ between the developer and the user perspectives?
6. 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?
7. Under the same assumptions, do the above "substrate" questions have the same answers for raster bitmaps?

5 Resources

Joel has a GitHub repo[9] and a blog post with more detail about an earlier state of the project [14].

[9] https://github.com/jdjakub/self-raising-diagrams

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. ISBN: 978-1-4503-6708-0. 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. ISBN: 978-1-4503-9320-1. 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. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635891.

[6] Antranig Basman, L. Church, C. 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. ISBN: 978-1-59593-593-9. 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. ISBN: 978-0-8186-7045-9. 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. ISBN: 979-8-4007-0330-0. 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. ISBN: 978-1-4503-2460-1. 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. Charlotte, NC, USA: Association for Computing Machinery, 2015, pages 280–290. ISBN: 9781450337793. DOI: 10.1145/2807442.2807446. URL: https://doi.org/10.1145/2807442.2807446.

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

[19] 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. ISBN: 978-1-4503-8635-7. DOI: 10.1145/3472749.3474804.

[20] 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.

[21] 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. ISBN: 978-1-4503-9421-5. DOI: 10.1145/3544548.3581113.

[22] Clément Pit-Claudel. "Untangling Mechanized Proofs". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2020, pages 155–174. ISBN: 978-1-4503-8176-5. DOI: 10/ghs5sn.

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

[24] 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.

[25] 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.