Greyboxing: towards domain-specific representations for domain-specific languages in electronics design

Richard Lin [1], Rohit Ramesh [2], Prabal Dutta [2], Björn Hartmann [2] and Ankur Mehta [1]

[1] *University of California, Los Angeles, ECE*
[2] *University of California, Berkeley, EECS*

Abstract

While domain-specific languages (DSLs) can be powerful, textual code interfaces can be unfamiliar to domain practitioners compared to mainstream graphical tools. For example, in board-level electronics design, DSLs enable libraries that automate design calculations, but graphical schematics and their visual patterns dominate mainstream practice. An underappreciated yet inexpensive technique to bridge the power-familiarity gap is to exploit existing and familiar graphical tools, but augmented with DSL semantics and interoperability by hijacking graphical notions of annotation and naming. In this short paper, we apply this technique to our electronics DSL by supporting the importing of graphical schematics. Unlike simple format conversions, we map schematic components to DSL library elements and constructs, augmenting a familiar graphical representation with powerful DSL semantics. We believe this technique can generalize to other domains and help blend the power of DSLs with the familiarity of conventional tools.

Keywords: Domain-specific languages. Graphical tooling. PCB design.

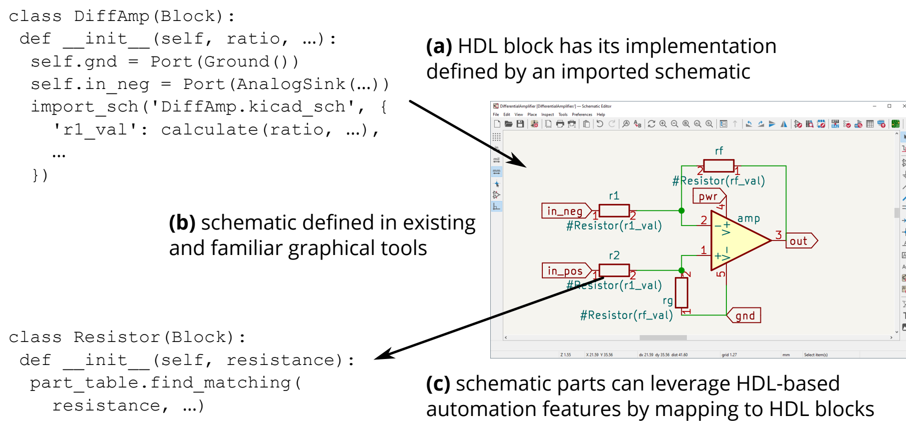

Figure 1. An overview of our system which augments an electronics hardware description language (HDL) with a familiar schematic representation. While HDL code similar to the snippet in (a) and (c) provides powerful programming constructs supporting user-defined design automation, the textual representation can be unfamiliar. This work enables users to define the implementation of an HDL subcircuit with a mainstream schematic editor shown in (b), providing both a more familiar interface and a document that can be shared with other electronics engineers. Beyond simply importing a schematic as a black box with its limited semantics of hardcoded values, the import process provides extended semantics including inline HDL and mapping graphical components to HDL library blocks in (c) which can include powerful and arbitrary design automation functionality like running code to select parts.

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

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

This work is licensed under a Creative Commons Attribution 4.0 International License.

1 Introduction

Domain specific languages (DSLs) can bring significant power to non-software domains through programming techniques that automate repetitive tasks and encapsulate expert knowledge. For example, our Python-embedded hardware description language (HDL) for board-level electronics circuit design [1], which we extend in this work, enables users to describe circuits as code.

With current mainstream circuit design tools, users draw graphical schematics consisting of components and their values. However, the use of concrete values means that these circuits are often

specialized for a particular application and might not be reusable elsewhere. Adding programming capability allows users to encode their design processes instead of merely the end results, enabling libraries of parameterized and generalizable subcircuits that can be shared and re-used.

Though the fundamental HDL concepts and structures were rooted in observations of current design practice [2], a textual interface is still very different and can be unfamiliar compared to schematics. Furthermore, graphical structures in schematics also offer readability benefits, for example with conventions like left-to-right signal flows and visual patterns like common arrangements of components into known functional blocks. Yet, these graphical tools have limited (if any) support for the parametricity needed for re-usable designs, and textual code remains the most common representation for defining more complicated logic such as automatic parts selection from a table. In general, we expect these trade-offs and issues to apply beyond electronics to other domains which could benefit from DSLs, especially domains that may be inherently spatial or visual like mechanical design.

So, there is an overall trade-off between familiar graphical interfaces with their widely understood notations, and powerful but unconventional textual DSLs. While recent work tries to bridge the gap by generating DSL edits from GUI actions [3], this is still part of a text-centric interface. We contribute a work-in-progress system that takes the opposite approach: providing a graphical-first interface through an existing, mature graphical tool, and importing those files into a powerful HDL system in a way that extends the graphical objects with programmatic semantics. We call our approach *greyboxing* because importing externally created objects is similar to blackboxing in digital logic DSLs, but our importing process is not opaque because of the additional semantics. In the rest of this short paper, we review related work on DSL usability in general, describe how our system integrates graphical schematics into an HDL, examine its usage in practice, and close with ideas for extensions.

2 Related Work

While all DSLs must ultimately generate some kind of domain-specific output, some DSLs provide tooling that uses visualization to aid the development process. A common approach that builds off of live programming [4], [5] is a side-by-side system, with a view of the output next to the text editor. Examples include OpenSCAD's [6] 3d viewer and Overleaf's [7] LaTeX PDF render, both of which have relatively fast (seconds) recompilation and provide a tight feedback loop to the user. However, all editing is still done in the text interface, a very different interaction compared to a direct-manipulation, what-you-see-is-what-you-get (WYSIWYG) interface.

More recent work has examined enabling code edits to be generated from the graphical interface. Sketch-n-sketch [8] does this for a DSL for 2d vector graphics, terming this "output-directed programming". SVG-PCB [9] addresses the somewhat similar 2d design problem of circuit board layout and allows users to modify constants in code by dragging points on the visualization. On the circuit design side, we previously developed a mixed textual and graphical IDE [3] for our electronics HDL which generates code snippets such as block instantiations from schematic editor-like interactions on the block diagram visualization.

While these approaches try to combine all the power of a programming language with the intuitiveness of a graphical interface, these systems must typically be built from scratch and may not have the interface polish of mature graphical-only tools. Furthermore, these are still code-first interfaces and provide different nonfunctional degrees-of-freedom, for example requiring the user to manage ordering of code statements, but not supporting spatial placements. However, this metadata can still be important: for example, all participants in our IDE user study [3] discussed shortcomings in the layout of the auto-generated block diagrams compared to manually drawn schematics.

Our technical approach of repurposing an existing graphical editor and leveraging popular and mature tools also has precedent. Ink/Stitch [10] is structured as a plugin to the Inkscape vector graphics editor, applying its interface and graphical primitives to embroidery design without (re)building everything from scratch. However, while it adapts a tool between domains, it is not an end-user programming tool. More broadly, foreign function interfaces (FFIs) in software engineering allow libraries written in one language to be used from another without being rewritten from the ground up.

For PCBs specifically, mainstream design suites like the open-source KiCad[11] as well as many commercial offerings focus on schematics for circuit design. These schematics ultimately define a list of components and connections, which feed into physical board layout and ultimately manufacturing [2]. While much recent work on electronics in the HCI community focuses on the novice experience [12]–[16], some also examines how novices can build PCBs [17]–[19]. More recent PCB design work makes use of subcircuit libraries, including both academic [20]–[23] and commercial work [24], [25] work, and sometimes with electronics modeling for error detection and circuit synthesis. Yet, lack of focus on supporting user-defined libraries limits applicability, something our HDL work [1], [26] addresses by adapting programming constructs and concepts like type systems to support both generalizable libraries and top-level board design in a unified interface. In this work, we extend that HDL with a more familiar graphical interface.

3 System Description

To gain the benefits of graphical interfaces, we add support for importing schematics files produced by an existing and popular graphical schematic tool, KiCad [11], effectively providing a graphical frontend for our HDL. By adding on new semantics to schematics as part of our import flow, we also bolt on some programming capability to the graphical design style.

3.1 Schematic Import

The overall flow is outlined in Figure 1: while there is still an HDL block in (a) to define external references like externally-connectable ports, its implementation is defined by a graphical schematic file. The schematic file in (b) is designed in an unmodified version of KiCad and makes use of its library of graphical circuit symbols.

Because our HDL's block diagram design model of subcircuit blocks, ports on those blocks, and connections between those ports is similar to schematic models of components, pins, and connections, bridging the two is conceptually straightforward at a high level. Schematic components map to HDL block instantiations, and schematic wires map to HDL connections between block ports. Interface points between HDL and schematic are matched by name, for example the `self.gnd` Port in the HDL is connected to the corresponding `gnd` wire label on the schematic.

As schematic files are structured as component objects and wires and not graphical primitives or raster images, the importing process is a straightforward transformation. Furthermore, as the schematic import is structured as a file reference instead of a code generator, there are no additional temporary files to get out of date.

3.2 Bridging Different Models

However, despite similarities in the overarching models, the details differ significantly. Our HDL's blocks are strongly typed with structured data fields, compared with schematic components being merely a graphical object with a free-text value field[1]. For example, a resistor in our HDL has numeric resistance and power fields which allows automatic matching to compatible parts, compared to the diverse formats with which engineers might specify a resistance value on a schematic.

We provide a few different options to bridge the representations. First, HDL blocks can define rules to map components and their values to a block instantiation – for example, the resistor class can parse the value "120kOhm" on a resistor in an imported schematic, or the opamp class can parse the triangle in Figure 1 (b) with no additional parameters. This aims to support schematics as commonly drawn today, especially where static values are sufficient and parameterization is unnecessary. Second, components can be defined with inline HDL in the value field, such as the resistors in Figure 1 (b). This style, while different from conventional schematics, allows parameterization in a graphical environment. Third, blocks can be fully defined in the HDL and correlated to the schematic components by name. This style can be useful for heavily parameterized blocks requiring complex HDL but where graphical definition of connectivity is still desirable.

1 Our HDL also has a richer port structure, for example modeling voltages and currents. But because there is no schematic analogy, there is no schematic integration and this additional modeling must be defined purely in HDL.

To support schematic components mapping to HDL blocks, HDL blocks must define a mapping between HDL port names and schematic pin numbers. This also makes this framework general: user-defined blocks can be used in graphical schematics, without being restricted to a small set of pre-defined blocks.

3.3 Example

The example shown in Figure 1 has a schematic defining a differential amplifier, a signal processing subcircuit that outputs the difference of its input voltages. The imported schematic in (b) would be a more familiar representation to electronics engineers compared to equivalent HDL. However, the HDL in (a) defines an automatic calculation for the resistance, and the resistor HDL in (c) can then find a part number from that resistance specification – automation not possible with graphical schematics.

4 Discussion

Overall, our goal is the best of both worlds – the power of an HDL combined with the familiarity of graphical schematics. Different ways of defining components further allow users to flexibly trade-off between staying within conventional notations but being limited by static component values, to more hybrid notations like inline HDL that incrementally add programming power.

Although we've mainly focused on the designer experience, the schematic documents themselves can also be invaluable for readability and documentation. Because the imported schematics define the block's implementation, they cannot get out of sync unlike comments or supplementary diagrams.

Yet, with blocks capable of having both an HDL and graphical schematic component, users may need to juggle both a text editor and schematic editor as neither alone may be sufficient to fully understand a block. Style guidelines may help reduce confusion by standardizing best practices on how to split responsibilities. However, this seam may still limit tooling, for example naive static checkers would be unable to inspect into inline HDL in schematics.

While we're not sure what a perfect solution might look like, we believe that our approach provides many benefits of both interfaces at a very low engineering cost – just an importer and component mapping definitions. In the bigger picture, this is another point in the conceptual design space of design tools, with graphical schematics and textual HDLs at opposite ends, and this schematic import and our prior mixed graphical-textual IDE somewhere in the middle. Future user studies might help illuminate which design points can be more useful to a wider range of people.

4.1 Future Work

Although the examples here focused on new designs, another application of this approach may be to leverage the vast number of existing schematics and import them. The programming analogy would be foreign function interfaces enabling a new language to reuse libraries designed for other languages. This reuse mitigates a major drawback of newer systems, which may not have the strong community or ecosystem of libraries needed to be productive [27].

The example is also a pretty straightforward mapping from schematic to HDL, with fixed circuit structure and limited parameterization using inline HDL. However, another major advantage of an HDL is the ability to parameterize the circuit structure, for example a block of n LEDs. Though not supported by current schematic editors, informal graphical notations like ellipses can be used to denote repeating circuit blocks. Extensions to the importer might recognize these notations and enable graphical parameterization of the circuit structure.

Finally, DSLs for other domains could also benefit from domain-specific representations and interfaces, and importing can be a potentially inexpensive solution compared to building a custom output directed programming tool. One example might be mechanical design, where a text-driven language like OpenSCAD might provide a way to import 3d parts or even 2d sections defined in a graphical tool like FreeCAD. Bolting on programming semantics might further enable generalizable designs in a graphical tool without pre-existing strong parameterization support.

5 Acknowledgements

This work was supported in part by DARPA grants HR00112110008 and FA8750-20-C-0156 (SD-CPS). The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, nor does this imply any official endorsement. Approved for public release; distribution is unlimited.

References

[1] R. Lin, R. Ramesh, C. Chi, *et al.*, "Polymorphic blocks: Unifying high-level specification and low-level control for circuit board design," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 529–540, ISBN: 9781450375146. DOI: 10.1145/3379337.3415860. [Online]. Available: https://doi.org/10.1145/3379337.3415860.

[2] R. Lin, R. Ramesh, A. Iannopollo, *et al.*, "Beyond schematic capture: Meaningful abstractions for better electronics design tools," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19, Glasgow, Scotland Uk: Association for Computing Machinery, 2019, ISBN: 9781450359702. DOI: 10.1145/3290605.3300513. [Online]. Available: https://doi.org/10.1145/3290605.3300513.

[3] R. Lin, R. Ramesh, N. Jain, *et al.*, "Weaving schematics and code: Interactive visual editing for hardware description languages," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1039–1049, ISBN: 9781450386357. DOI: 10.1145/3472749.3474804. [Online]. Available: https://doi.org/10.1145/3472749.3474804.

[4] J. Edwards, "Example centric programming," *SIGPLAN Not.*, vol. 39, no. 12, pp. 84–91, Dec. 2004, ISSN: 0362-1340. DOI: 10.1145/1052883.1052894. [Online]. Available: https://doi.org/10.1145/1052883.1052894.

[5] S. McDirmid, "Usable live programming," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2013, Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 53–62, ISBN: 9781450324724. DOI: 10.1145/2509578.2509585. [Online]. Available: https://doi.org/10.1145/2509578.2509585.

[6] OpenSCAD. "OpenSCAD." (2022), [Online]. Available: openscad.org (visited on 12/31/2022).

[7] Overleaf. "Overleaf." (2022), [Online]. Available: overleaf.com (visited on 12/31/2022).

[8] B. Hempel, J. Lubin, and R. Chugh, "Sketch-n-sketch: Output-directed programming for svg," in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '19, New Orleans, LA, USA: Association for Computing Machinery, 2019, pp. 281–292, ISBN: 9781450368162. DOI: 10.1145/3332165.3347925. [Online]. Available: https://doi.org/10.1145/3332165.3347925.

[9] L. McElroy, Q. Bolsée, N. Peek, and N. Gershenfeld, "Svg-pcb: A web-based bidirectional electronics board editor," in *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication*, 2022, pp. 1–9.

[10] Ink/Stitch. "Ink/Stitch." (2022), [Online]. Available: inkstitch.org (visited on 12/31/2022).

[11] KiCad. "Kicad eda." (2018), [Online]. Available: http://kicad-pcb.org/ (visited on 09/20/2018).

[12] D. Drew, J. L. Newcomb, W. McGrath, F. Maksimovic, D. Mellis, and B. Hartmann, "The toastboard: Ubiquitous instrumentation and automated checking of breadboarded circuits," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ser. UIST '16, Tokyo, Japan: ACM, 2016, pp. 677–686, ISBN: 978-1-4503-4189-9. DOI: 10.1145/2984511.2984566. [Online]. Available: http://doi.acm.org/10.1145/2984511.2984566.

[13] W. McGrath, D. Drew, J. Warner, *et al.*, "Bifröst: Visualizing and checking behavior of embedded systems across hardware and software," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '17, Québec City, QC, Canada: ACM, 2017, pp. 299–310, ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126658. [Online]. Available: http://doi.acm.org/10.1145/3126594.3126658.

[14] Y. Kim, Y. Choi, H. Lee, G. Lee, and A. Bianchi, "Virtualcomponent: A mixed-reality tool for designing and tuning breadboarded circuits," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–13.

[15] J. Warner, B. Lafreniere, G. Fitzmaurice, and T. Grossman, "Electrotutor: Test-driven physical computing tutorials," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 435–446.

[16] F. Anderson, T. Grossman, and G. Fitzmaurice, "Trigger-action-circuits: Leveraging generative design to enable novices to design and build circuitry," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '17, Québec City, QC, Canada: ACM, 2017, pp. 331–342, ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126637. [Online]. Available: http://doi.acm.org/10.1145/3126594.3126637.

[17] D. A. Mellis, L. Buechley, M. Resnick, and B. Hartmann, "Engaging amateurs in the design, fabrication, and assembly of electronic devices," in *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, ser. DIS '16, Brisbane, QLD, Australia: ACM, 2016, pp. 1270–1281, ISBN: 978-1-4503-4031-1. DOI: 10.1145/2901790.2901833. [Online]. Available: http://doi.acm.org/10.1145/2901790.2901833.

[18] A. Knörig, R. Wettach, and J. Cohen, "Fritzing: A tool for advancing electronic prototyping for designers," in *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, ser. TEI '09, Cambridge, United Kingdom: Association for Computing Machinery, 2009, pp. 351–358, ISBN: 9781605584935. DOI: 10.1145/1517664.1517735. [Online]. Available: https://doi.org/10.1145/1517664.1517735.

[19] J.-Y. Lo, D.-Y. Huang, T.-S. Kuo, *et al.*, "Autofritz: Autocomplete for prototyping virtual breadboard circuits," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19, Glasgow, Scotland Uk: Association for Computing Machinery, 2019, ISBN: 9781450359702. DOI: 10.1145/3290605.3300633. [Online]. Available: https://doi.org/10.1145/3290605.3300633.

[20] R. Ramesh, R. Lin, A. Iannopollo, A. Sangiovanni-Vincentelli, B. Hartmann, and P. Dutta, "Turning coders into makers: The promise of embedded design generation," in *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*, ser. SCF '17, Cambridge, Massachusetts: ACM, 2017, 4:1–4:10, ISBN: 978-1-4503-4999-4. DOI: 10.1145/3083157.3083159. [Online]. Available: http://doi.acm.org/10.1145/3083157.3083159.

[21] D. J. Merrill, J. Garza, and S. Swanson, "Echidna: Mixed-domain computational implementation via decision trees," in *Proceedings of the ACM Symposium on Computational Fabrication*, ser. SCF '19, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2019, ISBN: 9781450367950. DOI: 10.1145/3328939.3329004. [Online]. Available: https://doi.org/10.1145/3328939.3329004.

[22] D. J. Merrill and S. Swanson, "Reducing instructor workload in an introductory robotics course via computational design," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19, Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 592–598, ISBN: 9781450358903. DOI: 10.1145/3287324.3287506. [Online]. Available: https://doi.org/10.1145/3287324.3287506.

[23] J. Garza, D. J. Merrill, and S. Swanson, "Appliancizer: Transforming web pages into electronic devices," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21, Yokohama, Japan: Association for Computing Machinery, 2021, ISBN: 9781450380966. DOI: 10.1145/3411764.3445732. [Online]. Available: https://doi.org/10.1145/3411764.3445732.

[24] Sparkfun. "À la carte." (2020), [Online]. Available: https://alc.sparkfun.com/ (visited on 12/16/2020).

[25] Gumstix. "Geppetto." (2018), [Online]. Available: www.gumstix.com/geppetto/ (visited on 01/02/2020).

[26] R. Lin, R. Ramesh, P. Dutta, B. Hartmann, and A. Mehta, "Computational support for multiplicity in hierarchical electronics design," in *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication*, ser. SCF '22, Seattle, WA, USA: Association for Computing Machinery, 2022, ISBN: 9781450398725. DOI: 10.1145/3559400.3561997. [Online]. Available: https://doi.org/10.1145/3559400.3561997.

[27] L. A. Meyerovich and A. S. Rabkin, "Socio-plt: Principles for programming language adoption," in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2012, Tucson, Arizona, USA: ACM, 2012, pp. 39–54, ISBN: 978-1-4503-1562-3. DOI: 10.1145/2384592.2384597. [Online]. Available: http://doi.acm.org/10.1145/2384592.2384597.