Searching for Incidental Specifications

Jeremy Ferguson[1*], Kevin Ye[2*], Jacob Yim[3*] and Justin Lubin[4]

[1] *jmfergie@berkeley.edu; University of California, Berkeley, USA*
[2] *yekeviny@berkeley.edu; University of California, Berkeley, USA*
[3] *jacobyim@berkeley.edu; University of California, Berkeley, USA*
[4] *justinlubin@berkeley.edu; University of California, Berkeley, USA*
[*] *Authors contributed equally.*

Abstract

Program synthesizers—tools that automatically generate code based on a user-provided specification—aim to lighten the burden of programming by ideally swapping the cost of manually performing a task or writing code for a task with the cost of writing a specification for that task. A concern of fundamental importance to the usability of these synthesizers is therefore to ensure that the specification cost is less than the manual cost. In this short paper, we discuss methods from human-computer interaction, techniques from software engineering and data mining, and theories from cognitive science that could directly help answer the question of what tasks are amenable to having specifications derived from users' existing behaviors and processes rather than having users go out of their way to provide something new to the synthesizer—thereby ensuring the cost of specification is appropriately lower. We hope that this discussion is useful for synthesis designers and sparks interest in further investigation into how we can derive specifications from users' existing behaviors and processes.

Keywords: Program synthesis. Specifications. User-centered. Methodology.

1 Introduction

Across many domains, experts are increasingly faced with the need for automation, from large-scale document processing in law, to analyzing DNA sequences in biology, to describing highly concurrent systems in traditional software engineering, to emerging complex fields like proof engineering in program verification. Typically, solving these tasks will require some form of manual effort (e.g., manually coding a solution, or simply manually analyzing data by hand), which we can imagine has some temporal or cognitive cost associated with it; we might loosely associate some cost $\textsc{ManualCost}(t)$ to each task t in a domain.

The promise of *program synthesis* is for the computer to automatically write code given some form of specification that describes the desired solution. When it works, it (very roughly) swaps the cost of manually achieving an outcome for writing the *specification* for the outcome; for a task t, we can call this new cost $\textsc{SpecificationCost}(t)$.

Barring any additional benefits a program synthesizer may provide (e.g. exposing the user to a new way of thinking), a fundamental assumption behind program synthesizers is therefore that for the tasks t in a given domain, we have

$$\textsc{SpecificationCost}(t) \leq \textsc{ManualCost}(t).$$

The question then is: When is this inequality achieved?[1]

One way to ensure that the manual cost of a task is an upper bound to the specification cost is to derive the specification itself from the user's typical manual process. Then, any effort that the user puts into crafting a specification is work that would have needed to be done *anyway* manually. (It also means that the user does not need to learn any new interfaces or interaction models.) Jayagopal, Lubin, and Chasins [1] dubbed this kind of specification an *incidental specification* and provide insights into its impact on the learnability of program synthesizers by novice programmers, but they do not provide any guidance on how to actually *find* tasks that readily admit these kinds of specification for synthesis designers.

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

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

1 In this paper, we focus on how to achieve this specific inequality, but we emphasize that this inequality is often necessary but certainly not sufficient for when synthesis could be useful. We refer to the following blog post by Michael Greenberg for a discussion of generalizing this inequality into a relatively more sufficient condition for when synthesis could be useful: http://www.weaselhat.com/2022/05/24/when-is-program-synthesis-worthwhile/.

In this short paper, we contribute a discussion of a few avenues that we find particularly appropriate in the **search for incidental specifications**: methods from human-computer interaction, techniques from software engineering and data mining, and theories from cognitive science. We hope that this discussion is useful for synthesis designers and sparks interest in brainstorming and collecting a plurality of methods for finding useful synthesis specifications that can be derived from users' existing behaviors and processes.

2 Avenues of Searching for Incidental Specifications

2.1 Human-Computer Interaction Methods

Incidental specification is effective for building program synthesizers because programmers' normal behavior naturally encapsulates the specification. Thus, when searching for opportunities for program synthesis via incidental specification, it is critical to first understand programmers' behavior *without* the use of program synthesis.

Research in the field of human-computer interaction (HCI) offers some insight into how we may study programmers' behavior. In this section, we examine five HCI-inspired techniques for conducting studies of programmer behavior, and weigh their advantages and disadvantages through the lens of designing a program synthesizer with incidental specification.

2.1.1 Fixed-Task In-Lab Think-Aloud Studies

One method of examining programmer behavior involves assigning study participants a programming task and asking them to think out loud while writing code in a controlled lab environment. Thinking aloud enables synthesizer designers to learn about participants' thought processes while coding, which may help match specifications to the ways programmers already reason about their code. A fixed-task in-lab study allows researchers to analyze participants' responses to controlled tasks and enables comparison between different participants' approaches to the same problems.

At the same time, this type of study takes place in a different, more regulated environment than participants' normal working conditions, and results may be biased by the types of tasks chosen for the study. As such, fixed-task in-lab studies may grant investigators control over the testing environment at the cost of ecological validity. This increased control may be helpful for developing program synthesizers for specific tasks. However, for more open-ended studies of program synthesis applications, these studies may not generalize well to real-world settings.

The use of fixed-task in-lab think-aloud studies of programmers is well-documented; for example, Brandt et al. applied in-lab think-aloud methodology to investigate programmers' use of web resources [2], while Vaithilingnam, Zhang, and Glassman evaluated the usability of large language model-based program synthesizers [3].

2.1.2 Contextual Inquiry

A similar technique for directly observing participants is *contextual inquiry*, wherein researchers act as apprentices to the participants and observe their day-to-day work [4]. Unlike an in-lab study, contextual inquiry takes place in the participant's everyday work environment, and the work performed is not assigned by the investigator.

Contextual inquiry offers advantages and disadvantages opposite to those of a fixed-task in-lab study: results are placed in the context of participants' normal work environment, but the study is less directed than one in a controlled lab. Thus, contextual inquiry may be most useful for broadly identifying program synthesis applications without a specific task in mind.

Like in-lab studies, contextual inquiry is also fairly common in studies of programmer behavior; see, for example, Ko's study of expert programmers in an event-based environment [5].

2.1.3 Design by Immersion

A third approach involves having the researchers themselves participate in the work of the domain of interest. This technique, called *design by immersion*, originates in visualization research but may also be applicable to designing programming tools [6].

Design by immersion gives researchers a firsthand perspective on programmers' experiences but can require many months at a time to complete. Program synthesizer designers using immersion must fully familiarize themselves with the skills of their users. For some researchers already familiar with their target domain, this may not require much additional learning; in these cases design by immersion may lead to valuable insights without too much overhead.

To the authors' knowledge, design by immersion is not common in programming languages research or studies of programmer behavior as a whole; however, we believe that similar techniques may have potential to yield valuable insights.

2.1.4 Discourse Analysis

A fourth HCI method involves drawing conclusions from existing discourse about programming tasks. Discourse analysis can take many different forms: for example, a researcher might read online forums or watch coding livestreams to understand how users speak or think about tasks. By understanding the way these tasks are spoken about and described, researchers may be able to identify opportunities for program synthesis with incidental specification models that leverage users' existing tendencies.

Unlike think-aloud studies and contextual inquiry, discourse analysis often requires no resources for finding participants and running a study and many sources—like conversations on online forums—are free to access. However, finding the right resources to analyze (and performing the analysis) can require significant effort on the part of the researcher. Without the context afforded by controlled studies or contextual inquiries, conclusions drawn from discourse analysis may also be harder to generalize, requiring the reader to "fill in the blanks" to provide the context.

Examples of discourse analyses of programmer behavior exist in the literature; see, for example, Zhang et al.'s study of developers of deep learning applications [7].

2.1.5 Natural Programming

A final HCI technique that may be useful is the "natural programming" design process, which emphasizes understanding users' language and thinking processes while solving problems rather than how they use existing tools [8]. Incorporating natural programming could involve observing users' non-coding approaches to solving a problem, including their language and conceptual ideas.

Because tools that reflect the way users naturally think about their task may be easier to operate, this may be attractive for designing more accessible program synthesizers. One major concern for our purposes, however, is that a natural programming approach investigates how users might ideally solve a task, not how they would currently solve it using the tools available to them. In our case, the specific implementation patterns that users already perform are critical as they are what comprise the incidental specifications we are searching for. As a result, natural programming may be best used when combined with one of the previously mentioned techniques, like contextual inquiry or other in-lab studies.

2.2 Software Engineering and Data Mining Techniques

We now discuss how techniques from the fields of software engineering and data mining could be used to find incidental specifications.

2.2.1 Commit Mining

One possible method of determining incidental specifications is to analyze commit histories from a version control website like GitHub. This has the benefit of needing no new input from the programmer, since this would just use the regular sequence of commits made by them. Each commit could be categorized as an addition, meaning new code or functionality was created; a removal; or a refactor, meaning changes that do not change semantics of the program; or some combination. Refactors in particular can be seen as improvements or refinements of a program, which gives one form of incidental specification: the initial implementation can be thought of as a more direct or easier-to-produce artifact for the programmer, whereas the refactored code is more difficult to achieve but somehow more desirable. If we can identify refactors as well as patterns contained in them, we may therefore be able to *view the "before" code as an incidental specification for the "after" code*, as described by Lubin and Chasins in the context of functional programmers' code authoring [9].

However, a major downside of this approach is that extracting this information from commits is at best very difficult, and often impossible. For example, commits are often too coarse-grained, too fine-grained, or too nonlinear to readily correspond to individual refactors, and determining exactly when a change is a refactor requires deep reasoning about the semantics of a program. These drawbacks motivate our next section on data mining program histories obtained via editor telemetry.

2.2.2 Data Mining Program Histories

Although program commit mining as described in the previous section has the advantage that each commit is easy to parse using off-the-shelf tooling, researchers do not get to see the fine-grained structure of the process of how the programmer composes the program. In this section, we provide some background on relevant techniques in data mining to overcome this challenge, well as a brief discussion of an early-stage project we have worked on based on some of these techniques.

Specifically, our project explores using pattern mining to extract information about a program's history over time (collected via editor telemetry) to try to understand how the user constructed the program. We note that this is related to but distinct from code specification mining (as in Daikon [10]) or code pattern extraction (as in GrouMiner [11]), as we are interested in patterns in programmers' *construction processes*, not final programs.

Any such system that aims to understand user behavior via collecting and analyzing edit histories must face at least the three following challenges:

1. *How to parse incomplete programs*, as most of the recorded program states will be syntactically incomplete [12], [13];
2. *How to extract events from sequences of parses*; and
3. *How to determine which events are recurring and meaningful*, which is a sign of a possible incidental specification.

In the remainder of this section, we provide an overview of how we tackled these three areas in our beginning-stage prototype system as an example, which we hope will inspire others to think about how to data mine program histories.

An Example

1. *Parsing Incomplete Programs.* We handled parsing incomplete programs by using an Earley parser with Aho's error correcting grammar [14]. In this formalism, errors in the grammar are fixed with additions, removals, and substitutions. Then, the parse tree is reconstructed using an algorithm from Graham and Harrison [15, Chapter 3] with a modification to account for the costs from the error correcting grammar. Finally, after translating the parse trees of these incomplete programs to abstract expression trees (ASTs) with holes, we have a suitable representation of the history of a computer program. In particular, the history is encoded as a list of ASTs, and each edit can be extracted by diffing neighboring ASTs.

2. *Event Extraction.* To determine the events that occurred over time, we extracted features from the diffs of neighboring ASTs with holes. The features that we analyzed only depended on single diffs, but there are more nuanced features which look at multiple diffs (for example, whether neighboring edits are of the same type); these more complicated features might allow us to analyze deeper patterns in program writing. Another possibility we considered is synthesizing features in a feedback-driven way, automatically optimizing feature extraction via inductive synthesis or reinforcement learning based on what the system ranks as "interesting" (as described next).

3. *Pattern Mining Events.* We used *sequential pattern mining* to determine which sequences of extracted events were recurring and meaningful. Broadly speaking, the goal of sequential pattern mining is to discover interesting and meaningful patterns and rules on data with "time or sequential ordering information" [16]. In general, starting with a sequence database consisting of multiple sets

of sequential data (such as extracted events), sequential pattern mining algorithms find patterns which correspond to subsequences of events. The relevant metric for our use-case is the *support* of a pattern, which is a measure of how often it occurs. Generally the minimum support *minsup* is a parameter in a pattern mining algorithm to determine what subsequences occur frequently enough to be considered meaningful.

Two variants of pattern mining stood out as particularly suitable for our purposes. The first is *episode mining*, which is a pattern mining variant that looks within a single sequence of events instead of multiple sequences. The aim is to find rules of the form $X \rightarrow Y$ called *episode rules*, which represent X appearing before Y within a set time window. Another related variant is *periodic pattern mining*, which is similar to episode mining in that it looks only at a single sequence of events, but it considers periodically repeating patterns [16]. These seemed to be the best choices for our problem, since our program histories are encoded as a sequence of edits.

For our implementation, we used SPMF, an off-the-shelf open source data mining library which consists of many pattern mining algorithms [17]; in particular, we used its implementation of the POERM (Partially Ordered Episode Rule Miner) [18] sequential pattern mining algorithm.

2.3 Cognitive Science Theories

In this last section, we describe how two prominent cognitive science theories can help narrow in on promising incidental specifications. We hope that others will build upon these connections with cognitive science and uncover how other theories can aid in the search for incidental specifications.

2.3.1 Cognitive Load

Cognitive load theory describes how people keep information in their heads and access it while performing a task. Working memory is the section of memory that contains pieces of information actively being used and has been shown to only have capacity for approximately seven items at a time [19]. The rest of the information must be cycled in and out of long-term memory, and can be stored and retrieved more efficiently using organizational structures called schemas. For instance, when programmers are writing code, they are keeping many pieces of information in their mind at a time, such as various syntactic rules, design patterns, and domain-specific concepts that they are implementing [20], [21]. The cognitive load depends on several factors related to working memory, primarily the maximum amount of information that must be held in working memory at any given time [22]. Within the theory of cognitive load, there also exists the concept of intrinsic and extraneous load. An intrinsic cognitive load refers to the inherent mental difficulty of the task at hand, whereas an extraneous cognitive load is one that relates to added difficulty in how the task is being solved, or how a learner is being taught the information.

With this framework in mind, we can consider program synthesis tasks by an approximation of the cognitive load required to perform them and look for potential incidental specifications by finding tasks that admit a less user-preferable solution with approximately low cognitive load and a more user-preferable solution with approximately high cognitive load.[2] Then, when users choose the less preferable (but easier) solution, synthesis could pick up on the incidental specification and return the preferable solution at, ideally, no additional cognitive cost. Although measuring cognitive load directly is challenging, looking for synthesis tasks in this light can serve as a rough guide for finding incidental specifications.

An Example Concretely, one program synthesis task that we have ourselves explored is the task of NumPy synthesis. NumPy synthesis in this context refers to the problem of translating code written in native Python and translating it into vectorized NumPy code. This task can be very challenging task for novices (and even experts), but the runtime speedup can be significant.

There may be a high cognitive load associated with this task, as there is a need to keep many

2 As in Section 2.2.1, the less preferable (but easier) solution can serve as an incidental specification for the more preferable (but harder) solution.

different NumPy functions in one's working memory to try and figure out which ones will be useful for solving the problem at hand. The kinds of programs we looked at were fairly simple programs, operating over large arrays. In this case, the intrinsic cognitive load might be relatively small, due to the simplicity of the operations being done on the data. However, the need to translate it into NumPy might create a much higher extraneous cognitive load.

In this case, the incidental specification is simply the original Python program, which can then be translated into NumPy using a synthesis technique such as verified lifting [23], [24]. The synthesis engine is provided specifications for the NumPy functions, separate from anything the individual programmer provides. Because the program contains all the information needed to do the translation, there is no need for the programmer to do anything else.

2.3.2 Problem Decomposition

Another widely-studied area of cognitive science that can be applied to this study of incidental specifications is the area of problem decomposition. The way that people break down and solve problems is of great importance when searching for problems that are well-suited for applying program synthesis. One important distinction that has been made when studying problem decomposition in the context of programming are the two strategies of hierarchical and opportunistic programming [25]. Hierarchical programming refers to a top-down, structured method of programming whereby components are implementing in descending levels of abstraction in a fashion that is designed ahead of time. Opportunistic programming, on the other hand, refers to a style of moving between different levels of abstraction and changing the approach dynamically as the design evolves.

With this theoretical framework in mind, we can search for incidental specifications by analyzing prospective tasks and specifications and how they support the existing problem decomposition strategies that are frequently used for that task. When a programming task lends itself to hierarchical programming, there may often be some design that the programmer is following to move through the levels of abstraction in a top-down fashion. This design process can provide additional information about the desired program that could aid in constructing a specification. If the programmer is following such a pattern anyway (including creating artifacts like interface files or function stubs as well as top-down program sketches), this would make a specification that used this design an incidental specification.

On the other hand, if a task naturally lends itself to opportunistic programming, then the synthesizer will have less information from which to derive an incidental specification. If the programmer is jumping between different levels of abstraction—for example, by writing functions one at a time—the specification must therefore be derived from only the information within that function. While it may be possible to derive an incidental specification from this information, it may also be more difficult to do so; nevertheless, an understanding of the mode of problem decomposition that users typically undergo when solving problems in a domain may aid synthesis designers in leveraging users' incidental behaviors.

3 Conclusion

We conclude by stressing that these avenues are by no means exhaustive and only begin to scratch the surface of how we can leverage users' existing behaviors and processes in program synthesis. So, we hope that by asking the following question, we can make deeper interdisciplinary collaborations and uncover exciting new user-centered insights for program synthesis:

With your unique background, how would you search for incidental specifications?

Acknowledgements

We would like to thank our PLATEAU mentor, Tianyi Zhang, as well as the members of PLAIT Lab at UC Berkeley for their excellent feedback on this paper. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1752814 and is supported in part by NSF Grant FW-HTF 2129008 as well as by gifts from Google and the Chan Zuckerberg Biohub.

References

[1] D. Jayagopal, J. Lubin, and S. E. Chasins, "Exploring the learnability of program synthesizers by novice programmers," *The 35th Annual ACM Symposium on User Interface Software and Technology*, 2022. DOI: 10.1145/3526113.3545659.

[2] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09, Boston, MA, USA: Association for Computing Machinery, 2009, pp. 1589–1598, ISBN: 9781605582467. DOI: 10.1145/1518701.1518944.

[3] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22, New Orleans, LA, USA: Association for Computing Machinery, 2022, ISBN: 9781450391566. DOI: 10.1145/3491101.3519665.

[4] H. Beyer and K. Holtzblatt, *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann Publishers Inc., 1997, ISBN: 9780080503042.

[5] A. J. Ko, "A contextual inquiry of expert programmers in an event-based programming environment," in *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '03, Ft. Lauderdale, Florida, USA: Association for Computing Machinery, 2003, pp. 1036–1037, ISBN: 1581136374. DOI: 10.1145/765891.766135.

[6] K. W. Hall, A. J. Bradley, U. Hinrichs, *et al.*, "Design by immersion: A transdisciplinary approach to problem-driven visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 109–118, 2020. DOI: 10.1109/TVCG.2019.2934790.

[7] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2019, pp. 104–115. DOI: 10.1109/ISSRE.2019.00020.

[8] B. A. Myers, J. F. Pane, and A. J. Ko, "Natural programming languages and environments," *Commun. ACM*, vol. 47, no. 9, pp. 47–52, Sep. 2004, ISSN: 0001-0782. DOI: 10.1145/1015864.1015888.

[9] J. Lubin and S. E. Chasins, "How statically-typed functional programmers write code," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485532.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99, Los Angeles, California, USA: Association for Computing Machinery, 1999, pp. 213–224, ISBN: 1581130740. DOI: 10.1145/302405.302467.

[11] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09, Amsterdam, The Netherlands: Association for Computing Machinery, 2009, pp. 383–392, ISBN: 9781605580012. DOI: 10.1145/1595696.1595767.

[12] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer, "Hazelnut: A bidirectionally typed structure editor calculus," *SIGPLAN Not.*, vol. 52, no. 1, pp. 86–99, Jan. 2017, ISSN: 0362-1340. DOI: 10.1145/3093333.3009900.

[13] Y. S. Yoon and B. A. Myers, "A longitudinal study of programmers' backtracking," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014, pp. 101–108. DOI: 10.1109/VLHCC.2014.6883030.

[14] A. V. Aho and T. G. Peterson, "A minimum distance error-correcting parser for context-free languages," *SIAM J. Comput.*, vol. 1, pp. 305–312, 1972.

[15] S. L. Graham and M. A. Harrison, "Parsing of general context-free languages," in ser. Advances in Computers, M. Rubinoff and M. C. Yovits, Eds., vol. 14, Elsevier, 1976, pp. 77–185. DOI: 10.1016/S0065-2458(08)60451-9.

[16] P. Fournier-Viger, C.-W. Lin, U. Rage, Y. S. Koh, and R. Thomas, "A survey of sequential pattern mining," *Data Science and Pattern Recognition*, vol. 1, pp. 54–77, Feb. 2017.

[17] P. Fournier-Viger, J. C.-W. Lin, A. Gomariz, *et al.*, "The SPMF Open-Source Data Mining Library Version 2," in *Machine Learning and Knowledge Discovery in Databases*, B. Berendt, B. Bringmann, É. Fromont, *et al.*, Eds., Springer International Publishing, 2016, pp. 36–40, ISBN: 978-3-319-46131-1.

[18] P. Fournier-Viger, Y. Chen, F. Nouioua, and J. C.-W. Lin, “Mining partially-ordered episode rules in an event sequence,” in *Intelligent Information and Database Systems*, N. T. Nguyen, S. Chittayasothorn, D. Niyato, and B. Trawiński, Eds., Springer International Publishing, 2021, pp. 3–15, ISBN: 978-3-030-73280-6.

[19] G. A. Miller, “The magical number seven, plus or minus two: Some limits on our capacity for processing information,” *Psychological Review*, vol. 63, no. 2, pp. 81–97, 1956. DOI: 10.1037/h0043158.

[20] B. Shneiderman, “Exploratory experiments in programmer behavior,” *International Journal of Computer & Information Sciences*, vol. 5, no. 2, pp. 123–143, Jun. 1976, ISSN: 1573-7640. DOI: 10.1007/BF00975629.

[21] D. Françoise and F. Bott, *Software design: Cognitive aspects*. Springer, 2002.

[22] J. Sweller, “Cognitive load during problem solving: Effects on learning,” *Cognitive Science*, vol. 12, no. 2, pp. 257–285, 1988. DOI: 10.1207/s15516709cog1202_4.

[23] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama, “Verified lifting of stencil computations,” *SIGPLAN Not.*, vol. 51, no. 6, pp. 711–726, Jun. 2016, ISSN: 0362-1340. DOI: 10.1145/2980983.2908117.

[24] Metalift Contributors, *Metalift: A program synthesis framework for verified lifting applications*, https://metalift.pages.dev, Accessed: 2022-03-31, 2022.

[25] R. Guindon, “Designing the design process: Exploiting opportunistic thoughts,” *Human-Computer Interaction*, vol. 5, no. 2, pp. 305–344, 1990. DOI: 10.1207/s15327051hci0502&3_6.