Human-in-the-Loop Program Synthesis for Live Coding

Mark Santolucito
msantolu@barnard.edu
Barnard College
New York City, NY, USA

Abstract

Live Coding is a creative coding practice, where the act of programming itself constitutes a performance. The code written during a Live Coding performance often generates media, for example a continuous stream of music or video. One of the challenges of Live Coding is in finding a balance in the language design, such that the language is both expressive enough for the artist, as well as simple enough to be programmed in real-time. In order to reduce the overhead of manually coding every part of a Live Coding performance, we propose a tool for Live Coding that leverages program synthesis to simplify the process. Program synthesis retains the "show your code" ethos of Live Coding performances, while also lowering the barrier to entry to the performance practice.

CCS Concepts: • **Software and its engineering**; • **Applied computing** → **Sound and music computing**;

Keywords: Live Coding, program synthesis, computer music

ACM Reference Format:
Mark Santolucito. 2021. Human-in-the-Loop Program Synthesis for Live Coding. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design (FARM '21), August 27, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3471872.3472972

1 Introduction

The promise of fully automated software construction is intuitively alluring. As such, program synthesis has seen an explosion of academic progress in recent years. The term "program synthesis" refers to automatically generating code to satisfy some specification. Like functional programming, a specification describes *what* the code should do, without going into details about *how* it should be done.

The specification for program synthesis can take many forms. As one example, in the paradigm of *programming by example* [7] (PBE), a user provides a set of pairs of input-output examples that illustrate the desired behavior of the code. From these examples, a PBE engine generates code that generalizes from the examples to create a program that handles the unspecified examples as well.

The starting point for our project is the insight that Live Coding, from the world of computer music, is a setting that revolves around specification refinement and hence is well-suited for human-in-the-loop synthesis. Live Coding is a performative practice of coding still in its nascent stages of definition [28]. Generally, a Live Coding performance consists of code that continually generates some media (often audio [19, 21] or video [17]). The performer changes the code throughout the performance, thereby shaping the media that is being produced. Live Coding falls roughly into the category of Live Programming [29, 30] - as the code changes, the output (the audio) is updated in real time. A key component to Live Coding is allowing viewers to watch the evolution of the code itself.

Live Coding is, by its nature, an iterative process where specifications are continuously refined. During the evolution of code throughout a Live Coding performance, there is no single correct state of the code (i.e. specification) that the performer must achieve. The process of writing code for a Live Coding performance is exploratory. As a result, the implicit specification of the intended code is constantly shifting. Due to this constantly shifting specification, program synthesis and live coding make an attractive pair.

By augmenting Live Coding environments with synthesis, we gain a more fluid interface to the creative ideas a Live Coding artist may want to express. On the other hand, program synthesis-enabled Live Coding environments gives us a testbed for exploring the design space of program synthesis interfaces. This may yield insight into how a synthesis-driven feedback loop for software development might be most effectively designed.

In this work, we present an initial prototype tool for a program synthesis-enabled Live Coding environment. We implement our tool using Javascript and WebAudio on the frontend, allowing the tool to run in the browser and achieve

FARM '21, August 27, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8613-5/21/08…$15.00
https://doi.org/10.1145/3471872.3472972

a high level of accessibility. The backend synthesis engine is implemented using Syntax Guided Synthesis, or SyGuS [1], with CVC4 [22]. SyGuS allows us to synthesis small functional programs, which we then embed into Javascript templates in the Live Coding environment. The key contributions of this work are:

- Outline a synthesis algorithm specialized to a step-sequencer Live Coding environment.
- Present an implementation of our synthesis-enabled live coding environment that integrates program synthesis, program repair, and live programming. This tool is open source and available to test online[1]
- Discuss the future challenges making challenges in a program synthesis-enabled Live Coding environment.

2 Background and Related Work

Our goal with a program synthesis-enabled Live Coding environment is to gain a better understanding of both Live Coding and program synthesis. In particular for program synthesis, one major issue is that there has been slow adoption in real world settings. One important aspect to adoption of program synthesis is in capturing the natural interactions a programmer has with code. Prior work found that even if users are able to complete a task more efficiently with program synthesis, they still find the process of coding to be more helpful than the fully automated synthesis approach [23]. To address this, the need for program synthesis that maintains the interactive aspect of coding as a process (sometimes called human-in-the-loop synthesis), stands as a key hurdle in increasing the impact of program synthesis on developers.

There was an early understanding that program synthesis suffered from usability issues in its design, particularly with the black box approach [18]. The importance of human-in-the-loop processes from a usability perspective has also been well established in the field of computer music. There is ample evidence that fully automated tools are not a complete solution for constructing digital artifacts, especially in creative domains [31]. Live Coding arose partially as an answer to fully automated tools for generative music [4], providing a way for users to interact and shape generative processes for music in real-time by changing the code of these processes.

Recognizing that, just as creating music, programming is an inherently creative activity, we chose to explore Live Coding as our context for human-in-the-loop program synthesis. In fact, the intersection of iterative refinement of code and program synthesis has been explored through Live Programming, a close relative of Live Coding.

[1]Code and a live demo are available at https://github.com/Barnard-PL-Labs/SequencerLiveCoding.

2.1 Live Coding and Live Programming

Live Programming [29] allows users to edit the code of their program and immediately see the effect of those changes on the output. The key motivation is to tighten the edit-compile-run cycle of traditional program development, thereby closing the gap between users making a change in code and seeing the impact of that change. In the context of program synthesis, this has been called the user-synthesizer gap [9].

Live Coding [4] is distinct from Live Programming, but closely related in many ways. In both cases, there is a focus on live re-evaluation of code. Live programming is largely framed as a style of IDE and a software development environment, whereas Live Coding specifically focuses on the performative practice of the evolution of code. In this sense, Live Coding is a particular way of utilizing a Live Programming environment.

There has been great progress at the intersection of Live Programming and program synthesis [10, 16, 32]. The need for an interactive synthesis process was identified early [12]. Recently, the Sketch-n-Sketch tool that looks at output directed programming [16]. In this line of work, the authors explore how direct manipulation of the intended output can be turned back into program transformations automatically. The authors mention the possible extension of program synthesis (as opposed to hand-coded transformations) as a future direction. Similar work [26] has looked at Live Programming and simple program transformations to repair code. In our prior work, we have built a preliminary demo to explore general purpose program synthesis in a Live Programming environment [24]. There has been further work on establishing a formal foundation for Live Programming and program synthesis [10]. Furthermore, the interface issues with interactive program synthesis are also an area of active development [32]. In this work, we focus on how Live Coding can act as a platform to explore how general purpose synthesis tools might be integrated into Live Programming environments.

2.2 Program Synthesis

The Syntax Guided Synthesis (SyGuS) format language [1] was introduced in an effort to standardize the specification format of program synthesis, including PBE synthesis problems. The SyGuS language specifies synthesis problems through two components - a set of constraints (e.g. input-output examples), and a grammar (a set of functions). The goal of a SyGuS synthesis problem is to construct a program from functions within the given grammar that satisfies the given constraints. With this standardized synthesis format and an ever expanding set of benchmarks, there is now a yearly competition of synthesis tools [2], which pushes the frontier of scalable synthesis further. Although CVC4's support for SyGuS is typically the best of the SyGuS competition, even with this state-of-the-art tool, many synthesis problems still take on the order of seconds to complete [2]. The need

for fast synthesis times is concretely motivated by Live Coding, as the performance (and code editing) is happening in real-time. Some work has explored program synthesis in the context of music [25], but has not yet tackled Live Coding specifically.

In summary, prior work has provided an initial exploration of how program synthesis works in an interactive environment. The prior work on synthesis and Live Programming naturally leads us to explore Live Coding as an application domain of human-in-the-loop synthesis.

3 System Overview

Our synthesis-enabled Live Coding environment specifically focuses on Live Coding a *step-sequencer*, a hardware device used by professional musicians to create looping musical patterns (or, *sequences* of musical *steps*). A step-sequencer provides a simple interface to the performer - a set of buttons that control when/which notes should be played. Rather than work with a physical device, we used a software model of a basic step-sequencer [6]. In a programmatic sense, a step-sequencer is a two-dimensional array, where each subarray is an instrument, and each element of the subarrays (the state of each button) are musical beats.

Following a classic live programming interface model, we have two panes for user interaction - a code editor and a direct manipulation editor (the top and bottom of Fig. 1 respectively). The Code Editor is a Live Programming interface that allows the user to manually change the code. Whenever an update to the code is detected, the code is rerun to generate a new state of the output, and the output displayed in the direct manipulation editor is also updated. The direct manipulation editor allows the user to directly change the state of the output of the code, which is in effect a change to the specification of the code. Any change in the direct manipulation editor triggers program synthesis, and the Code Editor is updated to reflect the new, synthesized code that matches the changes in the Code Editor.

One of our key design goals is that the code on the code editor and the state of the output in the direct manipulation editor are always in alignment. To be more specific about the concept of keeping code and output state in alignment, we define two properties that we should maintain on our system. First, at every point in time, the output in the direct manipulation editor should reflect the output of the code. Second, the output is only updated by user interactions, either to the output itself or to the code.

4 Human-in-the-Loop Synthesis

Our proposed Live Coding environment uses a synthesis driven program-repair model to constantly update the code to match the provided pattern. On the interface level, this is a programming-by-demonstration interaction - the user demonstrates the intended beat with the GUI (direct manipulation editor), and the repair engine generates code to match the data in the GUI. Our approach is to transform this Live Coding, beat manipulation problem into a programming-by-example (PBE) problem. As programming-by-example problem, we use CVC4 [22], an SMT solver with support for SyGuS, to run SyGuS queries. We map the grammatical elements of SyGuS to generic JavaScript code, and put this result inside Javascript templates before displaying the code to the user.

4.1 Stage 1: Initial Synthesis

More specifically, to obtain a PBE problem, we start by treating each pattern/instrument (e.g. Tom, Kick, Snare, etc) as a separate problem. Each beat itself is an array of length 16 with values representing the note to be played (0 for no beat, 1 for a quiet beat, 2 for a loud beat) at the time step corresponding to the index of the array. As is typical in step sequencers, time is quantized into 16 steps. To transform this array into a PBE problem, we view the array as a function mapping time to notes.

As an example, the pattern shown in Fig. 2 would generate the PBE constraints as shown in Fig. 3. Notice that we truncate the constraints at the position of the last non-zero beat. We do this so that synthesis has some opportunity to generalize the given pattern. Transforming from SyGuS language output to JavaScript, the language of our Live Coding environment, we then obtain the following term: `1 - (i % 2)`. To apply this term over the array of beats, we leverage Javascript's support for higher order functions (in this case, a `map` specifically), using a code snippet template as shown in Fig. 5.

This is a desirable solution as it not only satisfies the given examples, but it also reasonably generalizes across the rest of the beat. The solution hypothesizes that the pattern the user would like to complete is `[1,0,1,0,1...]` for the full 16 beats. However, this is of course not the pattern the user has provided yet. If we return exactly the code as shown in Fig. 5, the code and the direct manipulation editor pattern are no longer in alignment. One of the key design goals of this system is that the code is always in a state that generates the state displayed in the direct manipulation editor. However, the synthesized code violates this as it is forcing a change on the state of the system that was not directly induced by the user. To remedy this situation, we overwrite the second half of the array which is all zeros, using code shown in Fig. 6. This allows us to keep our general solution, while still ensuring the code does not overstep in its predictions.

4.2 Stage 2: Live Programming

In the second stage of our human-in-the-loop model of synthesis, the user may iterate on the synthesized code. For example, if the user likes the synthesized code shown in Fig. 6,

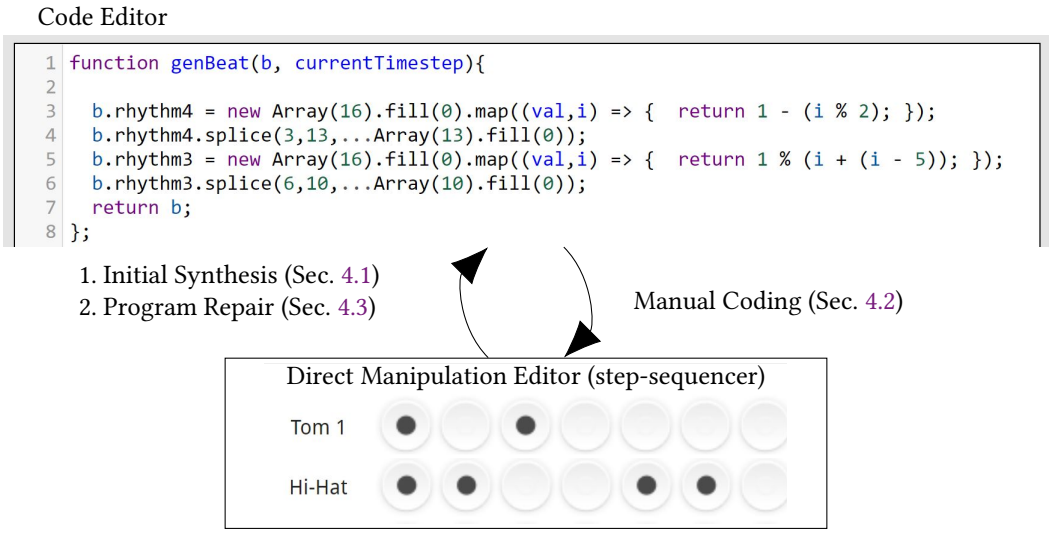

Figure 1. An overview of a prototype of the interface and workflow of a human-in-the-loop synthesis tool for live coding a step-sequencer.

Figure 2. A pattern entered into the step-sequencer. As an array: `[1,0,1,0,1,0,0...]`.

```
(constraint (== f(0) 1))
(constraint (== f(1) 0))
(constraint (== f(2) 1))
(constraint (== f(3) 0))
(constraint (== f(4) 1))
```

Figure 3. Constraints for the pattern shown in Fig. 2

```
(define-fun f ((i Int)) Int (- 1 (mod i 2)))
```

Figure 4. The possible synthesis solution to the SyGuS input-output example constraints shown in Fig. 3

```
b.rhythm1 = new Array(16).fill(0).map((val,i) =>
{ return 1 - (i % 2); });}
```

Figure 5. The final JavaScript code resulting from synthesis

```
b.rhythm1 = new Array(16).fill(0).map((val,i) =>
    { return 1 - (i % 2); });
b.rhythm1.splice(5,11,...Array(11).fill(0));
```

Figure 6. Synthesized code in a template

the user may comment out the second line so that the pattern in the direct manipulation editor adopts the generalized code solution. This is a key component to human-in-the-loop synthesis - that synthesis is not forced to over-generalize. Rather than imposing the generalization of the synthesized program on the state the user is directly manipulating, we synthesize code that contains the generalization, but has not activated it. The user can inspect the code, and make edits as appropriate. We are synthesizing a program with the *expectation* that it is not the final program the user needs. We expect that the user will iterate on this code.

4.3 Stage 3: Program Repair

In the last stage of the program synthesis cycle, the user may change the specification through the direct manipulation editor interface. This implicitly indicates that we need to run synthesize with the updated specification. At this stage, we have two possible scenarios. First, we have the scenario where the code has been unchanged by the user since the last synthesis request. In the second situation, the code has been modified by the user, and we must find a way to merge the user provided code and the synthesized code.

In the first case, merging the new code with the old code is relatively straightforward. As long as we have restricted the target DSL to be small enough, we can directly replace the key components. In the case of the `.map` templates we have presented, we need only to replace the anonymous function in the map, and bounds on the `.splice` operation. However, when we are using less restrictive templates, we may need a

more sophisticated approach. We believe that this approach will be specific to the DSL and templates.

The situation is more complex when the user has made arbitrary edits to the provided code. In this case, the template may not be directly present in the code anymore. The naive solution is to simply overwrite the user's code with the synthesized result. Any strategy will require an interface that clearly communicates to the user how the code is being repaired. It is possible that the user would prefer to have some ability to select which repair strategy should be used at any given time.

4.4 Handling Interaction

In contrast to the traditional approach to synthesis, where a specification is provided and code is returned, we must now consider a larger spectrum on interactions. First, we consider how to handle a failed synthesis attempt. Whereas a traditional synthesis engine can simply tell the user the tool was not able to synthesize a solution, this does not work in our setting. Recall that our system design requires that the code always reflects the status of the step-sequencer (or generally, the direct manipulation editor). If a change has been made in the content editor, we must find some valid change to reflect this in the code. When the synthesis engine is not able to find a solution within the prescribed timeout, we can fall back on code transformations. If the user has clicked one step in the step-sequencer (e.g. turned on step 7) and we cannot find a new program with synthesis, we can keep the old synthesis result and add a direct indexing operation to adjust the code according (e.g. b.rhythm1[7]=1).

Additionally, in human-in-the-loop synthesis, we have no control over when the user will update a specification or update code. Thus, we must have a strategy to address the resulting race conditions. As one example, consider the situations where a synthesis request is still running, and the direct manipulation editor is updated by the user. The change to the direct manipulation editor represents a change in the specification, and a new synthesis request will be generated. If the previous synthesis request has still not completed, we now have two synthesis requests for different specifications running at the same time. To resolve this race condition, we tag each synthesis request with the state of the direct manipulation editor when it begins. If we receive a synthesis result with a tag that does not match the current state of the direct manipulation editor, we ignore it and wait for the new synthesis query to finish. Formalizing these strategies to cover the full set of issues that arise in this setting is still an open question.

The speed of synthesis is also a major concern - especially in a Live Coding environment where reactivity is important to the performance. If we were to run each pattern as a separate synthesis query, the total synthesis time, using a 500ms timeout over six beat patterns, would be 3 seconds when run serially. It may be possible to overcome this to some extent by running each synthesis request in parallel, but there is a clear need for faster synthesis tools on a lower level.

An additional open question is how we can integrate synthesis tools for more general code generation (i.e. beyond the templates demonstrated in Fig. 6). Currently, our proposed approach to synthesis is limited to Linear Integer Arithmetic (LIA) problems and only replaces subexpressions in `.map` statements. However, the scope of SyGuS problems that CVC4 can solve is much larger than this, and can handle much more complex synthesis (e.g. bitvectors, strings, UIF, datatypes). We leave to future work the exploration of strategies to more fully integrate the power of SyGuS to JavaScript specific synthesis.

5 Discussion

Automated code synthesis is an area of research with a long history (cf. the Church synthesis problem [5]). However, due to the problem's undecidability and high computational complexity for decidable fragments, for almost 50 years the research in program synthesis was mainly focused on addressing theoretical questions and the size of synthesized programs was relatively small. However, the state of affairs has drastically changed in the last decade. By leveraging advances in automated reasoning and formal methods, there has been a renewed interest in software synthesis. Research in program synthesis has recently focused on developing efficient algorithms and tools, and has found use in the industrial software application, FlashFill [13, 14].

However, beyond FlashFill, which embeds PBE program synthesis into spreadsheets, the adoption of program synthesis into developer workflows in production has been limited. We believe that this limited adoption is in part due to the fact that existing techniques are designed to be black-boxes for users. Synthesis techniques are designed in such a way that the user provides a specification (such as input-output examples in PBE) and the tool returns complete program code [1]. However, the generated code may not match the user's intention. The generated code will not match the user's intention when the specification is not specific enough and the program synthesis engine incorrectly generalized from the user's specification. This issue is fundamentally unavoidable in program synthesis - for example in PBE, input-output examples are always an underspecification (in that the input-output examples only partially describe the behavior of the intended program).

When using program synthesis to synthesize code, the user has two options to fix synthesized code that does not match the user's intentions. First, the user may update the specification and re-run program synthesis to generate a new program from scratch. Second, the user may manually edit the code. When manually editing code, the environment should automatically update the specification to match the

code. In a PBE synthesis environment, this takes the form of updating the input-output examples by rerunning the new code. The automatic update of the specification is called Live Programming [24, 30]. However, once the code has been edited manually, the program synthesis approach can no longer be directly used as it would generate new code and not take into account the user code edits. The inability to switch back and forth between synthesis and code editing is a problem because the refinement of specification and code are complementary activities. That is, the process of editing code is a critical step that helps users refine the specification of their desired system. To allow program synthesis to take user code edits into account, we turn to the area of "program repair" [27]. Some work has begun to explore the intersection of live programming and synthesis [11]. Program repair takes existing code, and automatically modifies (similarly to how program synthesis automatically generates) it to fit a new specification.

In order to make program synthesis a viable software development method, we must gain a better understanding of how program synthesis, Live Coding, and program repair can be integrated together into a single program synthesis toolchain. By combining these three modes of programming together, we can begin to explore human-in-the-loop program synthesis. One short-term benefit of human-in-the-loop program synthesis is to broaden the scope of coding tasks that program synthesis techniques can handle. Program synthesis techniques are currently limited to synthesizing snippets of code [2, 3], although recent applications have extended to more media-rich domains, such as generation of visualizations [8, 15, 20]. By taking an iterative approach to synthesis [33], program synthesis engines should be able to more effectively build upon previous synthesis results, rather than synthesizing code from scratch each time. The longer term goal of human-in-the-loop synthesis is to achieve a better integration of program synthesis tools with the development process than has currently been explored.

6 Conclusion

The largest challenge we currently face in exploring human-in-the-loop program synthesis is a lack of a suitable application domain for prototyping. The ideal setting is one where we can ask users to interact with a human-in-the-loop program synthesis algorithm and have them naturally explore the interaction between the program synthesis engine and the user. Live Coding exactly matches this setting and as such is a motivating application domain to explore not just for artistic purposes, but also for scientific inquiry. We have presented a prototype tool that demonstrates one way of how Live Coding and program synthesis can be combined. Our key next steps are to further explore the design space of program synthesis-enable Live Coding through user studies and artistic practice.

Acknowledgments

This work was partially completed while working on the grant supported by the National Science Foundation under Grant No. CCF-2105208.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[2] Rajeev Alur, Dana Fisman, Saswat Padhi, Andrew Reynolds, Rishabh Singh, and Abhishek Udupa. 2019. The 6th Competition on Syntax-Guided Synthesis. https://sygus.org/comp/2019/results-slides.pdf. Accessed: 2019-11-20.

[3] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017).

[4] Andrew R Brown and Andrew Sorensen. 2009. Interacting with generative music through live coding. *Contemporary Music Review* 28, 1 (2009), 17–29.

[5] Alonzo Church. 1963. Application of Recursive Arithmetic to the Problem of Circuit Synthesis. *Journal of Symbolic Logic* 28, 4 (1963), 289–290. https://doi.org/10.2307/2271310

[6] cwilso. 2021. Shiny Happy MIDI Drum Machine. https://github.com/cwilso/MIDIDrums

[7] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA.

[8] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2020. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. arXiv:2006.08381 [cs.AI]

[9] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 614–626.

[10] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 614âĂŞ626. https://doi.org/10.1145/3379337.3415869

[11] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. *Small-Step Live Programming by Example*. Association for Computing Machinery, New York, NY, USA, 614âĂŞ626. https://doi.org/10.1145/3379337.3415869

[12] Joel Galenson, Philip Reames, Rastislav Bodik, and Koushik Sen. 2013. A Hint in the Right Direction: Interactive Synthesis with Partial Dynamic Specifications. (2013).

[13] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*. 317–330.

[14] William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *PLDI*. 317–328.

[15] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch. *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (Oct 2019). https://doi.org/10.1145/3332165.3347925

[16] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 281–292.

[17] Olivia Jack. 2021. Hydra. https://hydra.ojack.xyz/

[18] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (2009), 65–67.

[19] Alex McLean. 2014. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. 63–70.

[20] Andrew M McNutt and Ravi Chugh. 2021. Integrated Visualization Editing via Parameterized Declarative Templates. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (May 2021). https://doi.org/10.1145/3411764.3445356

[21] Miguel Cerdeira Negrão. 2018. NNdef: Livecoding Digital Musical Instruments in SuperCollider Using Functional Reactive Programming. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design* (St. Louis, MO, USA) *(FARM 2018)*. Association for Computing Machinery, New York, NY, USA, 8 pages.

[22] Andrew Reynolds and Cesare Tinelli. 2017. SyGuS Techniques in the Core of an SMT Solver. *arXiv preprint arXiv:1711.10641* (2017).

[23] Mark Santolucito, Drew Goldman, Allyson Weseley, and Ruzica Piskac. 2018. Programming by Example: Efficient, but Not "Helpful". In *PLATEAU at SPLASH*. Also presented at SYNT 2018.

[24] Mark Santolucito, William T. Hallahan, and Ruzica Piskac. 2019. Live Programming By Example. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*.

[25] Mark Santolucito, Kate Rogers, Aedan Lombardo, and Ruzica Piskac. 2018. Programming-by-example for Audio: Synthesizing Digital Signal Processing Programs. In *Functional Art and Music (FARM) at ICFP*.

[26] Christopher Schuster and Cormac Flanagan. 2016. Live programming by example: using direct manipulation for live program synthesis. In *LIVE Workshop*.

[27] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. 2005. Finding and fixing faults. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 35–49.

[28] TOPLAP. 2020. TOPLAP draft manifesto. https://toplap.org/wiki/ManifestoDraft

[29] Bret Victor. 2000. Inventing on Principle. Available at https://vimeo.com/36579366.

[30] Bret Victor. 2012. Learnable Programming : designing a programming system for understanding programs. (2012). Available at http://worrydream.com/LearnableProgramming/.

[31] Ge Wang. 2019. Humans in the Loop: The Design of Interactive AI Systems. https://hai.stanford.edu/blog/humans-loop-design-interactive-ai-systems

[32] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627âĂŞ648. https://doi.org/10.1145/3379337.3415900

[33] Yan Zhang, Béatrice Bérard, Lom Messan Hillah, Fabrice Kordon, and Yann Thierry-Mieg. 2014. Controllability for Discrete Event Systems Modelled in VeriJ. *Int. J. Crit. Comput.-Based Syst.* 5, 3/4 (Sept. 2014), 218âĂŞ240. https://doi.org/10.1504/IJCCBS.2014.064668