Program Synthesis for Musicians: A Usability Testbed for Temporal Logic Specifications

Wonhyuk Choi[1], Michel Vazirani[1], and Mark Santolucito[2]

[1] Columbia University, New York, NY 10027, USA
[2] Barnard College, Columbia University, New York, NY 10027, USA
{wonhyuk.choi, mvv2114}@columbia.edu, msantolu@barnard.edu

Abstract. In recent years, program synthesis research has made significant progress in creating user-friendly tools for Programming by example (PBE) and Programming by demonstration (PBD) environments. However, program synthesis from logical specifications, such as reactive synthesis, still faces large challenges in widespread adoption. In order to bring reactive synthesis to a wider audience, more research is necessary to explore different interface options. We present The SynthSynthesizer, a music-based tool for designing and testing specification interfaces. The tool enables researchers to prototype different interfaces for reactive synthesis and run user studies on them. The tool is accessible to both researchers and users by running on a browser on top of a `docker`-containerized synthesis toolchain. We show sample implementations with the tool by creating dropdown interfaces, and by running a user study with 21 users.

Keywords: Reactive Synthesis · Program Synthesis · Computer Music

1 Introduction

Over the last two decades, program synthesis has seen much progress [15] and researchers have made significant headway into making program synthesis accessible to a wider audience [13]. Specifically, research in Programming by example (PBE) and Programming by demonstration (PBD) has led to a wide array of user-friendly tools [8, 22, 23, 31], including Wrangler [21], StriSynth [14], Sketch-n-Sketch [16].

However, building user-friendly tools for program synthesis from logical specifications remains a challenge. In particular, for reactive synthesis [4], despite development in both theory [3, 33] and tooling [19], the complexity of writing specifications has limited the adoption of reactive synthesis to a highly technical audience. In order to bring synthesis to non-technical users, more research is necessary to understand effective means of creating logical specifications.

In this paper, we present The SynthSynthesizer, a tool that enables researchers to try out different interfaces and logic fragments for reactive synthesis. Researchers can define interfaces by simply implementing a single `JavaScript` function, after which a non-technical audience can interact with the tool. In

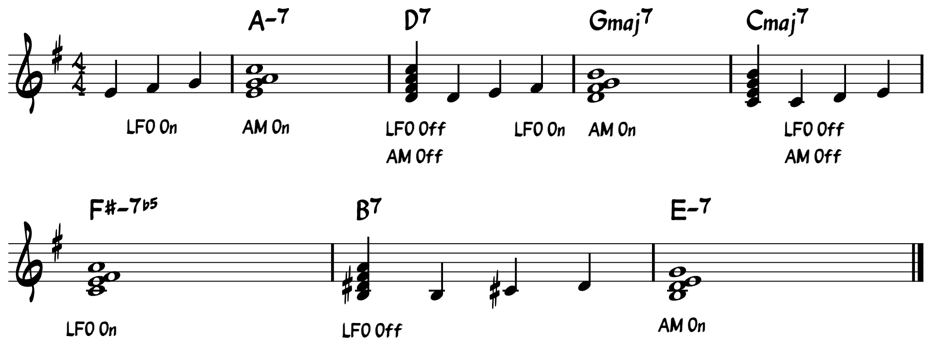


Fig. 1: Autumn Leaves Lead Sheet indicating the changes in signal topology

order to appeal to a larger base of users, The SynthSynthesizer uses computer music as an reactive environment that is also interactive and creative.

The SynthSynthesizer runs on a browser, making it easily to deploy user studies. The tool is also easy to install and modify for researchers; the synthesis toolchain is provided in `docker` container so that even researchers without a deep knowledge of reactive synthesis can explore the specification interface space.

Using our tool, we explored dropdowns as a way of specifying reactive control by implementing three different interfaces. We ran a user study on these interfaces by presenting them to 21 participants with a mix of music and programming backgrounds. From the study, we found that users experienced a tradeoff between ease of use and expressivity, and enjoyed the no-code nature of synthesis. These experiences motivate further exploration of specification interface design, which our tool aims to facilitate.

In summary, our contributions are as follows:

1. We present The SynthSynthesizer, a music-based tool that enables rapid prototyping and user studies for studying different reactive synthesis interfaces.
2. We explored dropdowns as an interface for specifying reactive control, and implemented the example interfaces using our tool.
3. We ran a user study with 21 users, and found a tradeoff between expressivity and ease-of-use as well as a possible appeal of synthesis to a wider user-base.

2 Motivating Example

As an illustrative example, consider a user that would like a reactive system to manipulate audio signals as phrases of a music piece are played. Specifically, AM synthesis should be toggled whenever the note G4 is played, and LFO vibrato should be toggled whenever the note E4 is played, as shown in Figure 1.

To build such a reactive system, a user could write a program that specifies when and how the signals should change. However, writing this program is generally not an easy task. It not only requires the user to be a competent programmer, but also requires them to be comfortable using specific API's such as `Web Audio`. Moreover, even if a user can write such a program, the solution is verbose, requiring nearly 100 lines of code to satisfy two logical conditions.

In order to concisely encapsulate the time-varying nature of the signal topology, the user might turn to reactive synthesis. In this case, the user would need to choose a temporal logic and write a specification that determines when AM synthesis and LFO vibrato are toggled. Such a specification, using Temporal Stream Logic (TSL) [11], can be written as follow:

$$\square(\texttt{play G4} \leftrightarrow [\texttt{AM} \leftarrowtail \texttt{toggle AM}]) \wedge$$
$$\square(\texttt{play E4} \leftrightarrow [\texttt{LFO} \leftarrowtail \texttt{toggle LFO}])$$

Though reactive synthesis brings the user closer to building their instrument, this solution generally involves too much prerequisite knowledge. Users must understand the notion of formal guarantees, time steps, and other particularities of temporal logic, making the approach unrealistic for a broad population.

To overcome the above challenges, we need a framework where non-technical users can easily specify temporal properties. Here, we present The SynthSynthesizer as a tool for exploring the space of such frameworks, where researchers can prototype different interfaces and run user studies.

3 Preliminaries

Temporal Stream Logic (TSL) is a logic designed around the synthesis of reactive programs [11]. TSL is built upon the same temporal logic operators (i.e. next $\bigcirc$, until $\mathcal{U}$) found in logics such as Linear Temporal Logic. In addition, TSL introduces *predicate terms* τ_P, *function terms* τ_F, and *update terms* to describe reactive systems that manipulate data. In TSL, the conceptualization of a reactive system revolves around signals $\texttt{s}$ which carry data values of arbitrary complexity; A TSL specification describes how functions should be applied to these signals over time. Signals may be pure outputs, or *cells*, as a one-timestep delayed input. These terms are defined as shown in the grammar of TSL below:

$$\varphi \;:=\; \tau \in \mathcal{T}_P \cup \mathcal{T}_U \;\mid\; \neg\varphi \;\mid\; \varphi \wedge \varphi \;\mid\; \bigcirc\varphi \;\mid\; \varphi\,\mathcal{U}\,\varphi$$

$$\tau_F := \texttt{s} \;\mid\; \texttt{f}(\tau_F^0, \tau_F^1, \ldots, \tau_F^{n-1})$$
$$\tau_P := \texttt{p}(\tau_F^0, \tau_F^1, \ldots, \tau_F^{n-1})$$
$$\tau_U := [s \leftarrowtail \tau_F]$$

4 The SynthSynthesizer

In this section, we introduce The SynthSynthesizer, a testbed tool for running user studies on different logical fragments and interfaces for program synthesis.

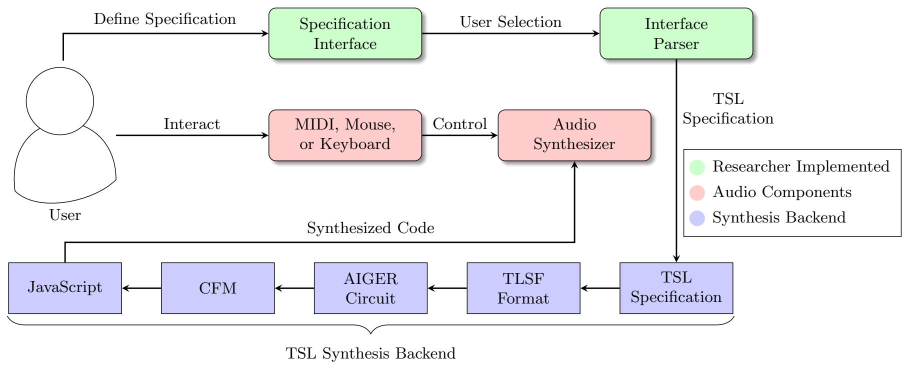


Fig. 2: Overview of The SynthSynthesizer

The framework allows researchers to create interfaces by defining them through `HTML` and implementing a single function in `JavaScript` to parse the interface. The tool also allows researchers to experiment with different fragments of logics, and explore the tradeoffs between expressivity and usability.

The overview of the process is shown in Figure 2. First, a user submits their specification through an interface. This gets parsed into a logic formula, which is then synthesized into `JavaScript` code. The resulting code is embedded back into the tool, controlling the audio synthesizer that the user plays with either their mouse, QWERTY keyboard, or USB MIDI controller. The researcher is free to use any temporal logic that can synthesize to `JavaScript` (such as LTL), but we include our TSL synthesis backend for completeness and usability.

We implemented the audio components of The SynthSynthesizer using Web Audio [28] and Web MIDI [36], both standard Web APIs maintained by the W3C. The frontend uses framework-less `JavaScript`, and the backend runs on `Node.js`. The server backend is responsible for synthesizing TSL specifications to `JavaScript`, with `Strix` [26] as its synthesis backend and `tsltools` [10] to convert between formats such as `TLSF` [20]and `AIGER` [18].

We designed The SynthSynthesizer so that researchers can easily access the tool. Most notably, installation is hassle-free: we provide a `docker` container with all the dependencies pre-installed. In particular, this makes the tool accessible to researchers outside the formal methods community; researchers do not a deep understanding of the synthesis procedure to use our tool. Additionally, since the tool runs on a web browser, running user studies is as simple as just sharing a link. A live demo of the tool is available at https://tslsynthesissynthesizer.com.

5 Evaluation

As an example of how The SynthSynthesizer can be used to explore interfaces for synthesis, we implemented three separate interfaces and presented them to users for a user study. Each interface utilizes a different fragment of TSL.

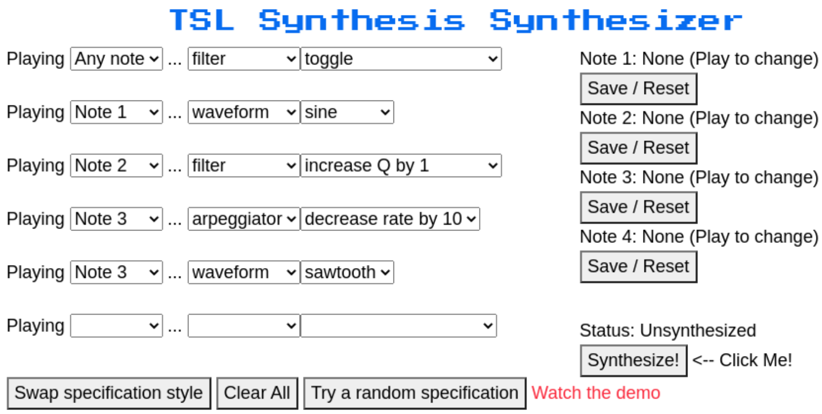


Fig. 3: Interface of TSL$_\alpha$

5.1 Interface Implementations

We explored dropdowns as an interface for specifying reactive control, as dropdowns are a ubiquitous design element. We created two dropdown interfaces with different frontends, and included a third written interface as a control case. All three interfaces use TSL to synthesize user specifications, but with varying parts of the grammar and subsequently varying levels of expressivity.

We now present each implementation separately.

TSL$_\alpha$. In our first implementation, we use a fragment of TSL that we call TSL$_\alpha$. Let $\tau_U \in \mathcal{T}_U$ update terms, $\tau_p \in \mathcal{T}_P$ predicate terms. Then, every formula φ in TSL$_\alpha$ is built according to the following grammar:

$$\varphi := \Box \tau_u \mid \Box \tau_p \leftrightarrow \tau_u \mid \varphi \wedge \varphi$$

The syntax of TSL$_\alpha$ allows users to specify predicates that reconfigure the signal flow topology of the underlying synthesizer. In particular, the TSL specification in the motivating example can be captured by TSL$_\alpha$.

The grammar is concise, allowing us to build a compact interface as in Figure 3. With this interface, users can define specifications by selecting from a set of predefined options. TSL$_\alpha$ specifications also synthesize quickly; 1,000 random synthesis queries took, on average, only 1.76 seconds (cf. Appendix A.2).

TSL$_\beta$. Our second implementation still features a dropdown interface, but with a more expressive grammar. Its syntax is constructed as follows:

$$\psi = \tau_u \mid \tau_p \leftrightarrow \tau_u \mid \tau_u \rightarrow \tau_u$$
$$\varphi = \Box \psi \mid \Box \tau_u \rightarrow \psi \; \mathcal{W} \; \neg \tau_u \mid \varphi \wedge \varphi$$

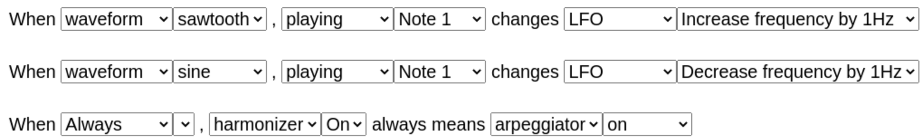


Fig. 4: Partial Interface of TSL_β

Expanding upon TSL_α, TSL_β adds terms of $\tau_u \rightarrow \tau_u$ and the weak until operator $\mathcal{W}$ for more complex specifications. For instance, a specification

$$\begin{aligned} &\square[\texttt{waveform} \leftarrowtail \texttt{sine}()] \leftrightarrow (\texttt{play C4} \leftrightarrow [\texttt{amFreq} \leftarrow \texttt{double amFreq}]) \\ \mathcal{W}\ &\neg[\texttt{waveform} \leftarrowtail \texttt{sine}()] \wedge \neg[\texttt{waveform} \leftarrowtail \texttt{waveform}] \end{aligned}$$

states that playing C4 doubles AM frequency only when the waveform is sine.

To suit the additional complexity in TSL_β, we arranged the dropdowns as natural language sentences for user readability. The interface is shown in Figure 4. This fragment of TSL also synthesizes quickly, with a mean of 10.09 seconds for 1,000 random specification synthesis queries (cf. Appendix A.2).

TSL_μ. TSL_μ subsumes TSL_α and TSL_β by offering the full syntax of TSL, but with the restriction that predicates cannot be applied to cells (cf. Appendix A.1). We can easily implement the tool using a written interface. Here, users type TSL_μ formulas directly into a textbox, accessing the syntax TSL_μ without any restrictions. In a user study, this interface would serve as the control case. Since the UI is a simple textbox, we omit a figure of TSL_μ.

5.2 User Study

We presented the TSL_α, TSL_β, and TSL_μ instantiations of The SynthSynthesizer to 21 users for a usability study. The participants were recruited through online forums focused on programming and computer music, such as reddit or discord. Users first watched a video tutorial[3]and answered preliminary questions on a scale of 1 (not at all experienced) - 7 (very experienced), to rate their own experience in music ($mean = 4.0, SD = 2.2$), audio signal processing ($mean = 2.6, SD = 2.1$), and programming ($mean = 4.5, SD = 2.0$). The users then manipulated the tool to define specifications, synthesize them, and interact with the resulting reactive system. Afterwards, users responded to a variety of questions, such as rating each interface on its 'Ease of Use' and 'Flexibility', or answering if they had a favorite interface and why. The full list of questions is included in Appendix A.3. Note that we did not time users for any of their activities, since our user study was focused on creativity and music production instead of concrete task completion.

From the user study, we found that participants found TSL_α and TSL_β equally understandable (Q2) and intuitive (Q3), while also being expressive and

[3] https://tslsynthesissynthesizer.com/tutorial.html

flexible (Q4). However, while users rated TSL_μ to be expressive and flexible, participants rated its usability to be lower than TSL_α and TSL_μ across all questions. Since we organized our study by showing TSL_α, TSL_β, and TSL_μ in the same sequential order, we intentionally created a bias for users to have a more solid understanding of TSL and temporal logic by the time they reached the TSL_μ interface. However, as users still expressed difficulty in using TSL_μ, this strengthens our claim that we need a more user-friendly interface than text-based interfaces to expose reactive synthesis to a wider audience. From this, we do not conclude that dropdowns are necessarily the right choice of interface - instead we remark that this is complex design space that requires further investigation.

A total of 18 participants responded to an optional qualitative question asking which interface was their favorite. Three chose TSL_α, nine chose TSL_β, and six chose TSL_μ. The preference for the more complex interfaces shows how users are intrigued by the expressivity and possibilities of TSL. Although larger fragments of TSL make interfaces harder to use, users are willing to accept a more complex logic if the interface for the specifications is sufficiently constrained. The balance struck by TSL_β was also reflected in the user explanations. One user responded *"TSL_β: offers the most flexibility while still being incredibly intuitive."* and another user responded *"TSL_β had the best tradeoff in intuitiveness/ease of use and freedom/flexibility"*. Two other users mentioned they preferred to avoid writing code, responding *"TSL_β! It felt like it had a lot more layers that you could add on, without the complexity of writing your own code to make it work."* and *"TSL_β. It has lots of flexibility and no need to write code."*.

A video of users interacting with the tool is available at tslsynthesissynthesizer.com/demo.html. Visualizations of the user study results are available in Appendix A.4.

6 Related Work

The SynthSynthesizer is a tool for exploring logic and interface design for program synthesis with temporal logics. In recent years, there has been an increased interest in usability design of language tools [5], including program synthesis tools [7, 32]. Frameworks to bring program synthesis to broader audiences have also been explored in the context of games [25], graphics [16], and data science [35], but synthesis tools for non-technical users have not yet included reactive synthesis specifically. The tool Flax [34, 35] specifically looks at nontraditional interfaces to synthesis by using visualization as a mode of specification.

Some existing tools have explored the usability design space of temporal logics for more technical users. TERMITE [29, 30] was designed to bring reactive synthesis to software developers. Another critical design problem in the usability of reactive synthesis is the task of providing explanations for reactive synthesis results [1]. Additionally, the UPPAAL tool provides an application-specific engineered interface for TCTL (timed computation tree logic) specifications; however, UPPAAL is more focused on verification than synthesis [2].

While interfaces for interactive music generation with reactive synthesis is a new research problem, computer-assisted composition has a long history [6]. In terms of usability, recent results have found that users preferred to have more control over automated music generation rather than having a monolithic end-to-end model [17]. Similarly, user studies on a music generation tool for video editing [12] found participants objecting to too much automation, as it made them feel as if they had not created music. These insights can directly contribute to interface research of reactive synthesis, since synthesized automata may be counter-intuitive to users.

We have built our tool around TSL [11], but our interface could be used to explore specification interface for other temporal logic. Of particular interest would be adding support for TSL-Modulo Theories [9, 24], which would allow for more fine-grained manipulations of music parameters.

7 Conclusions

We have introduced The SynthSynthesizer, a music-based user study tool that allows rapid prototyping of different fragments of logic and interfaces. We hope our tool can be used to start research into designing interfaces for different logics, and make synthesis more accessible to a broader audience.

References

1. Baumeister, T., Finkbeiner, B., Torfah, H.: Explainable reactive synthesis. In: Automated Technology for Verification and Analysis (2020)
2. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0 (2006)
3. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive (1) designs. Journal of Computer and System Sciences (2012)
4. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Journal of Symbolic Logic (1963)
5. Coblenz, M., Davis, A., Hofmann, M., Huang, V., Jin, S., Krieger, M., Liang, K., Wei, B., Yong, M.S., Aldrich, J.: User-centered programming language design: A course-based case study (2020)
6. Cope, D.: An expert system for computer-assisted composition. Computer Music Journal **11**(4), 30–46 (1987)
7. Crichton, W.: Human-centric program synthesis. CoRR **abs/1909.12281** (2019)
8. Ferdowsifard, K., Ordookhanians, A., Peleg, H., Lerner, S., Polikarpova, N.: Small-step live programming by example. In: Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology. pp. 614–626 (2020)
9. Finkbeiner, B., Heim, P., Passing, N.: Temporal stream logic modulo theories. CoRR **abs/2104.14988** (2021)
10. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Synthesizing functional reactive programs. In: International Symposium on Haskell (2019)
11. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: Synthesis beyond the bools. In: Computer Aided Verification (2019)

12. Frid, E., Gomes, C., Jin, Z.: Music creation by example. In: CHI '20. ACM (2020). https://doi.org/10.1145/3313831.3376514
13. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices (2011)
14. Gulwani, S., Mayer, M., Niksic, F., Piskac, R.: Strisynth: synthesis for live programming. In: International Conference on Software Engineering (2015)
15. Gulwani, S., Polozov, O., Singh, R., et al.: Program synthesis. Foundations and Trends® in Programming Languages (2017)
16. Hempel, B., Lubin, J., Chugh, R.: Sketch-n-sketch: Output-directed programming for svg. In: User Interface Software and Technology (2019)
17. Huang, C.A., Koops, H.V., Newton-Rex, E., Dinculescu, M., Cai, C.J.: AI song contest: Human-ai co-creation in songwriting. CoRR **abs/2010.05388** (2020)
18. Jacobs, S.: Extended aiger format for synthesis. arXiv:1405.5793 (2014)
19. Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J., Sankur, O., Tentrup, L.: The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In: SYNT@CAV (2017)
20. Jacobs, S., Klein, F., Schirmer, S.: A high-level ltl synthesis format: Tlsf v1. 1. Synthesis Workshop at CAV (2016)
21. Kandel, S., Paepcke, A., Hellerstein, J., Heer, J.: Wrangler: Interactive visual specification of data transformation scripts. In: CHI (2011)
22. Lerner, S.: Projection boxes: On-the-fly reconfigurable visualization for live programming. In: CHI (2020)
23. Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. ICFP (2020)
24. Maderbacher, B., Bloem, R.: Reactive synthesis modulo theories using abstraction refinement. arXiv preprint arXiv:2108.00090 (2021)
25. Mayer, M., Kuncak, V.: Game programming by demonstration. In: New Ideas, New Paradigms, and Reflections on Programming Software
26. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Computer Aided Verification (2018)
27. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: International Colloquium on Automata, Languages, and Programming (1989)
28. Rogers, C.: Web audio api specification. World Wide Web Consortium (2021)
29. Ryzhyk, L., Walker, A.: Developing a practical reactive synthesis tool: experience and lessons learned. Workshop on Synthesis at CAV (2016)
30. Ryzhyk, L., Walker, A., Keys, J., Legg, A., Raghunath, A., Stumm, M., Vij, M.: User-guided device driver synthesis. In: OSDI (2014)
31. Santolucito, M.: 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 (2021)
32. Santolucito, M., Goldman, D., Weseley, A., Piskac, R.: Programming by example: Efficient, but not" helpful". In: PLATEAU@SPLASH (2018)
33. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: International Symposium on Automated Technology for Verification and Analysis (2007)
34. Wang, C., Feng, Y., Bodik, R., Cheung, A., Dillig, I.: Visualization by example (POPL) (2019)
35. Wang, C., Feng, Y., Bodik, R., Dillig, I., Cheung, A., Ko, A.J.: Falx: Synthesis-powered visualization authoring. In: CHI Conference on Human Factors in Computing Systems (2021)
36. Wilson, C., Kalliokoski, J.: Web midi api. W3C, Working Draft (2021)

A Appendix

A.1 TSL$_\mu$ and its Decidability

For our tool, we use the TSL fragment TSL$_\mu$ that has no predicate term application on cell values. While our tool has many internal cell values – such as modulation frequencies or waveforms – predicate terms are only applied to fresh user inputs (i.e. which notes they pressed, the velocity of key press, etc.). This allows us to use the fragment TSL$_\mu$, which is decidable, unlike the full syntax of TSL.

Here, we formalize the definition of TSL$_\mu$ and prove the decidability of its synthesis problem.

Definition 1 (TSL$_\mu$). *Let function terms τ_F and update terms τ_U be defined as in Section 3. Let predicate terms τ_P be defined as follows:*

$$\tau_P := \mathfrak{p}(s_{i_0}, s_{i_1}, \cdots s_{i_j})$$

where s_{i_j} refers to input signals, and $\mathfrak{p}$ any predicate. Then, a TSL$_\mu$ formula is defined by the following syntax:

$$\varphi := \tau \in \mathcal{T}_P \cup \mathcal{T}_U \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\psi$$

Intuitively, this is a fragment of TSL where predicate terms are evaluated only on input signals, and not cells. In particular, synthesizing this fragment of TSL is decidable.

We now show that synthesis of this fragment of TSL is decidable by showing that every TSL$_\mu$ formula can be reduced to an LTL formula.

Theorem 1 (TSL$_\mu$-LTL equivalence). *Every TSL$_\mu$ formula can be transformed to an equivalent LTL formula in polynomial time.*

Proof. In TSL synthesis, the environment player chooses the predicate terms τ_P and the system player chooses the update terms τ_U. In TSL$_\mu$, the environment inputs τ_P's are always fresh at each timestep, and their values do not depend on previous outputs τ_U of the system player.

Now, we can use the translation procedure from TSL to LTL presented in [11]:

$$\varphi_{LTL} = \square\Big(\bigwedge_{\mathtt{s_o}\in\mathbb{O}\cup\mathbb{C}} \bigvee_{\tau\in\mathcal{T}_{U/\mathtt{id}}^{\mathtt{s_o}}} \big(\tau \wedge \bigwedge_{\tau'\in\mathcal{T}_{U/\mathtt{id}}^{\mathtt{s_o}}\setminus\{\tau\}} \neg\,\tau'\big)\Big)$$
$$\wedge\ \textsc{SyntacticConversion}\big(\varphi_{TSL}\big)$$

Finkbeiner et. al show the soundness of this procedure, that the realizability of φ_{LTL} implies the realizability of φ_{TSL}. In the full syntax of TSL, this procedure may still produce φ_{LTL} that returns unrealizable even though φ_{TSL} is realizable since the procedure removes the semantic meanings of update terms. However, in TSL$_\mu$, the environment inputs do not depend on the previous system outputs,

Interface type	Realizable	Unrealizable	Timeout	Median (s)	Average (s)
TSL_α	446	554	0	1.72	1.76
TSL_β	911	1	71	51.50	10.09

Table 1: Synthesis times for different grammars

and no semantic interpretation of update terms is necessary; it follows that an unrealizable φ_{LTL} always implies an unrealizable φ_{TSL} formula.

Furthermore, this procedure is bounded in polynomial time with respect to the formula size. The first part of the equation partially reconstructs the semantic meaning of updates by ensuring that a signal is not update with multiple values at a time. This is bounded in the size of update terms, $\binom{n}{2} \in \mathcal{O}(n^2)$. The second part of the equation simply transforms predicate terms to environment inputs and update terms to system outputs, and is in done in linear time, so the entire procedure is bounded in polynomial time.

Finally, we state the decidability as a corollary.

Corollary 1 (Decidability of $\mathbf{TSL}_\mu$ synthesis). *The synthesis problem of TSL_μ is decidable.*

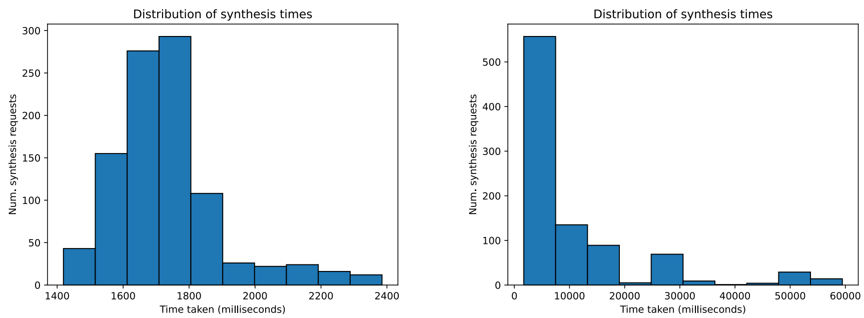
Proof. The syntheis problem of LTL is `2EXP-COMPLETE` [27]. Therefore, it follows from Theorem 1 that the synthesis problem of TSL_μ is also `2EXP-COMPLETE`, and decidable.

A.2 Experimental results

In order for users to interact with an interface, it is necessary that it synthesizes in a reasonable amount of time. Therefore, we decided to measure synthesis times of our TSL fragments by randomly generating 1,000 specifications using The SynthSynthesizer's random specification generator. The runtimes of random specifications is particularly relevant to our tool, as the interfaces for TSL_α and TSL_β included a "generate random specification" button, allowing users to explore the specification design space without needing to have a goal in mind. The random specification generator chooses an option randomly from each dropdown menu in the UI, effectively doing a random search through the combinatorial space of all possible specifications in TSL_α and TSL_β. We did not run a experimental result on the TSL_μ syntax as we did not include random generation of specifications for TSL_μ.

Synthesis was executed on a quad-core Intel Xeon processor (2.30 Ghz, 16Gb RAM) running Ubuntu 64bit LTS 18.04. Timeout was defined as any synthesis request that took over 10 seconds. Average and median time exclude these timed out synthesis requests. The results are shown in Table 1.

Overall, we found that TSL_α specifications synthesized much faster than TSL_β specifications, without any timeouts. This was an expected result, given the relative simplicity of TSL_α's grammar compared to that of TSL_β. However,



(a) Synthesis time distribution of TSL_α (b) Synthesis time distribution of TSL_β

Fig. 5: Synthesis times of 1000 random specifications

we were surprised to find that only one TSL_β specification was unrealizable. After a careful investigation, we discovered that the additional complexity in the grammar more tightly constrained each specification. Since each specification made weaker requirements, the grammar had less probability to create mutually exclusive specifications.

We visualize the distribution of the synthesis times in Figure 5. TSL_α synthesis times follow a quasi-Gaussian distribution, but even the longest-taking query completes in under 2.4 seconds. On the other hand, the distribution of TSL_β specifications skew right; the number of specifications decreases with increasing synthesis time. The majority of specifications synthesize quickly, with 68.5% specifications taking less than 10 seconds to synthesize. From our experimental results, we see a clear tradeoff between expressivity and synthesis times. TSL_α has a limited grammar, but on average synthesis takes less than two seconds to complete. On the other hand, TSL_β uses a larger fragment of TSL and provides more expressivity to the user, but at the cost of timeout; 7.1% of specifications timed out, and on average took almost 10 times as longer to synthesize than TSL_α.

A.3 User Study Questions

In this section, we present the full set of questions for the comprehensive user study in Tables 2, 3, 4. Note that Q5 is repeated in the table because the question is phrased slightly different for TSL_μ. The question is meant to ask about the intuitiveness of the structure of the specification interface. For TSL_α and TSL_β, the specification interface is structured around dropdown menus. For TSL_μ, the specification interface is structured around a text box.

A.4 User Study Results Visualizations

In this section, we present visualizations of the user study results. Figure 6a shows the user responses for each question for each separate interface. Figures

Question Number	Question
Q1	Intuitiveness
Q2	Understandability
Q3	Ease of Use
Q4	Flexibility and Expressivity

Table 2: Please rate the TSL_[x] interface for creating and synthesizing specifications from 1 to 7 (7 is highest) on the following

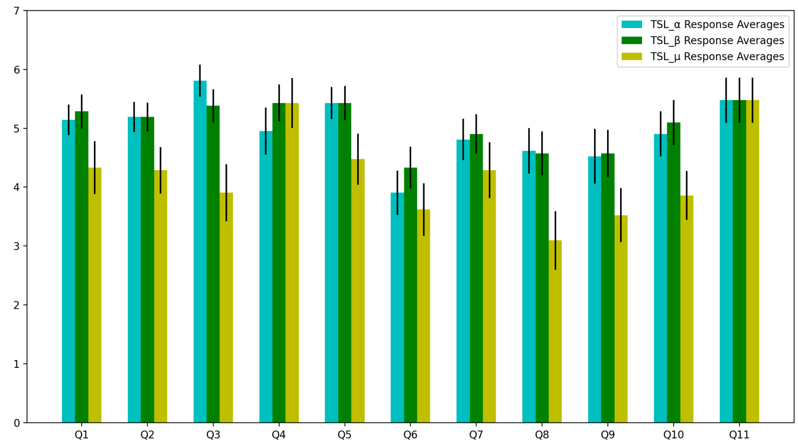
Question Number	Question
Q5(α, β)	The dropdown menus in TSL_[x] are an intuitive interface for specifying control flow
Q5(μ)	The text box in TSL$_\mu$ is an intuitive interface for specifying control flow
Q6	TSL_[x] can help me create music that I previously wanted to create
Q7	TSL_[x] can give me new ideas for music that I hadn't thought of
Q8	I can teach others how to use TSL_[x]
Q9	I would use TSL_[x] again to make music
Q10	I understand what specifications in TSL_[x] mean
Q11	After clicking "Synthesize!", the program did what I expected it to
QD.α	I understood sequential structure of the dropdown menus
QD.β	I understood the natural language descriptions between the dropdown menus
QS.μ	I understood the syntax of TSL_[x]

Table 3: On a scale from 1 to 7, how much do you agree with the following statements about TSL_[x]

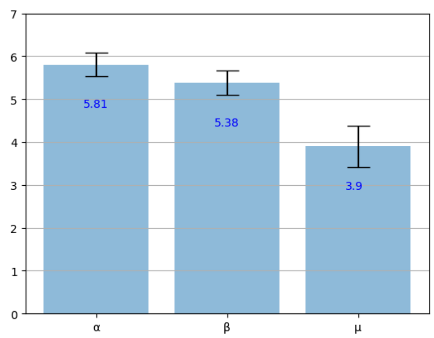
Question Number	Question
QG.1	What are your general thoughts on TSL_[x]?
QG.2	Which of the three specification interfaces was your favorite? Why?
QG.3	Would you like to share anything else?

Table 4: Paragraph Responses

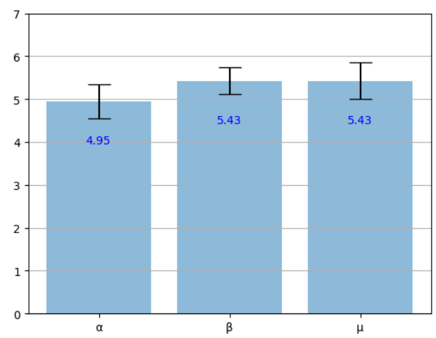
6b and 6c demonstrate the tradeoff between flexibility and ease-of-use of TSL_α, TSL_β, and TSL_μ.



(a) User Study Average Ratings with error bars



(b) TSL Interfaces Average Rating for Ease of Use with error bars



(c) TSL Interfaces Average Rating for Flexibility with error bars

Fig. 6: User Study Average Ratings