Towards the Usability of Reactive Synthesis: Building Blocks of Temporal Logic

Raven Rothkopf [*1], **Angel Leyi Cui** [†1], **Hannah Tongxin Zeng** [‡1], **Arya Sinha** [§2] **and Mark Santolucito** [¶1]

[1]*Barnard College, Columbia University, New York, USA*
[2]*Roslyn High School, New York, USA*

Abstract

Temporal logic specifications can be used to synthesize reactive systems by writing high-level descriptions of desired behavior, without the need to manually program a complete system. While synthesis from temporal logics has long been focused on hardware systems, recent work has expanded applications of synthesis to include areas of broader interest, such as mobile apps, visualization, and self-driving cars. These new application areas have the potential to bring new types of users into the synthesis community, but significant usability hurdles remain. In this work, we investigate how Temporal Stream Logic (TSL), a temporal logic specification language, can be made more usable and approachable to programmers of all skill levels. We propose a study design to evaluate the usefulness of an alternative interface for writing TSL to address the syntactic hurdle of temporal logic. We then outline areas for improvement and exploration in TSL and reactive synthesis as a whole.

Keywords: Program Synthesis. Reactive Synthesis. Temporal Stream Logic. Google Blockly. Structure editors. Visual programming.

1 Introduction

Reactive synthesis describes the process of automatically generating a controller from a high-level specification, typically written in temporal logic. This controller is responsible for constantly managing complex interactions between the system and its environment, while ensuring that the system can respond appropriately to all input scenarios. In recent years, reactive synthesis and its applications have been a growing topic of interest. The synthesis of the AMBA bus protocol from a Linear Temporal Logic (LTL) specification [1], [2] is considered one of the largest published success stories in the field. In contrast to LTL, which is well-suited for synthesis of hardware such are the AMBA bus module, Temporal Stream Logic (TSL) [3] is another temporal logic that is more well-suited to synthesis of software. Temporal Stream Logic is a high-level, logical specification language used in reactive synthesis, which extends LTL with updates and predicates over arbitrary function terms, allowing for the separation of "control" and "data". This separation enables a specification to capture both reactive properties and data manipulations. TSL has shown promising initial results that could extend reactive synthesis to new application domains including music [4], video games [5], mobile apps [3], animation (Figure 2), and autonomous vehicle controllers [3]. Despite this wide array of application domains, TSL (and temporal logic specification in general) remains unfamiliar and challenging to even the most experienced programmers. In this work, we explicitly define some of TSL's challenges, propose potential solutions to mitigate them, and identify areas that require further research.

In summary, our contributions are as follows:

1. We introduce a block-based structure editor as a tool to help beginners become comfortable with the syntax of Temporal Stream Logic (TSL).
2. We propose a study design intended to evaluate the usefulness of this tool for those learning TSL.
3. Taking into account areas for improvement, we outline additional directions for research that could make TSL and more broadly, temporal logic languages, more usable for programmers of all experience levels.

*Email: rgr2124@barnard.edu
†Email: lc3542@barnard.edu
‡Email: tz2486@barnard.edu
§Email: asinha23@roslynschools.org
¶Email: msantolu@barnard.edu

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

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

2 Demonstrative Example

Temporal Stream Logic is a language that specifies how a program behaves over time. When synthesized, TSL generates finite automata, specifically Mealy Machines, that produce output depending on their current state and input. These automata can be made into reactive programs when represented in code, responding to constant streams of input and producing constant streams of output.

Table 1. The following example shows a TSL spec and a section of its synthesized JavaScript code

TSL

```
always assume {
  ! ( pressR(e) && pressL(e) );
}
always guarantee {
  pressR(e) → ( [count ← count + 0.1] W pressL(e) );
  pressL(e) → ( [count ← count − 0.1] W pressR(e) );

  [cube.rotation.y ← count];
}
```

JavaScript

```
if ( currentState === 0 ) {
  if ( pressL(e) && pressR(e) ) {
    currentState = 0
  }
  else if ( ! pressL(e) ) {
    count = count + 0.1
    cube.rotation.y = count
    currentState = 0
  }
  else if ( pressL(e) && ! pressR(e) ) {
    count = count − 0.1
    cube.rotation.y = count
    currentState = 1
  }
}
else if ( currentState === 1 ) {
  if ( pressL(e) && pressR(e) ) {
    currentState = 0
  }
  else if ( ! pressL(e) && pressR(e) ) {
    count = count + 0.1
    cube.rotation.y = count
    currentState = 0
  }
}
```

To demonstrate the utility of TSL, we provide a small TSL specification, as shown in Table 1, that synthesizes the controller of an animated cube. Using our synthesis tool [6], the specification generates JavaScript code which realizes that specification and can be used to render an animated cube once integrated with an animation library in JavaScript. In this case, the specification describes a controller that spins the cube to the left or the right based on user input.

The upper-left textbox in Table 1 contains an example of a TSL specification. In TSL, we have the `always assume{}` and `always guarantee{}` blocks. Assumptions about the environment are specified in the `assume` block. The system to be synthesized is specified in the `guarantee` block. In the example, we assume of the environment that, `!(pressR(e) && pressL(e))` - the user will never press the left and right keys at the same time.

The system outlined in the `guarantee` block is comprised of four logical statements. The first, `pressR(e) -> ([count <- count + 1] W pressL(e))` states that pressing the right key implies that a variable called count will be incremented by 1 until the left key is pressed. Thus, when the variable count is used in the second statement, `[cube.rotation.y <- count]`, the cube's rotation around the y-axis will be constantly updated by an incrementing value, spinning the cube to the right. The second logical expression describes the same behavior, but if the left key was pressed, decrementing the variable count and spinning the cube to the left. The fourth logical expression, `[cube.color <- red]`, constantly updates the cube's color value to red, rendering a solid red cube.

3 Mitigating TSL's Syntactic Hurdles

3.1 Background

Having to memorize the syntax of temporal logic complicates the already difficult process of writing such logics. Although temporal logic has a relatively small grammar (compared to say, JavaScript), it is unfamiliar to those who are used to traditional programming languages. Our hypothesis is that the small grammar makes a structure editor uniquely well-suited as an alternative interface for writing temporal logics. Additionally, *a structure editor interface shifts the working mental model from recall*

to recognition [7]. Users manage a heavy cognitive load when writing temporal logic because they must recall the syntactic and grammatical structure of the new language paradigm of TSL as a temporal logic. This reliance on recall inhibits users from focusing on the semantics of their specifications. Displaying all operators and terms in a syntactically correct format enables users to recognize what they need to use instead of remembering it. We propose a block-based structure editor as a tool for TSL to help novices with the specification development process, targeting the syntactic complexity of TSL. As we discuss further in Sec. 6, there is some precedent for using block-based languages for temporal logic [8].

3.2 Google Blockly x TSL

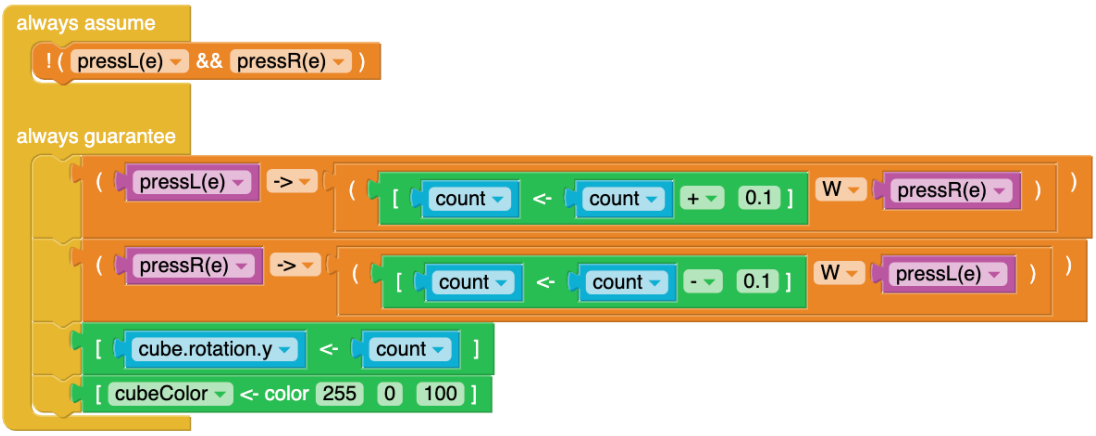

Figure 1. Blockly interface with TSL syntax and grammar, implementing the same specification from Table 1.

We introduce an alternative structured interface for writing TSL specifications using Google Developer's Blockly software [9]. Blockly is an opensource library for creating block-based visual code editors. The blocks have connection points where they can be attached to other blocks, and chained together. Blockly also has a side menu that categorizes each type of block. This menu ensures that each block is easily accessible, and also attaches a significance to where a block is placed in the menu. We are extending Blockly to a specification editor for TSL where users can drag and drop temporal logic operators, update terms and predicates. Fig. 1 demonstrates an example of a specification built with a Blockly editor. In Fig. 1, temporal operators, update terms, input signals, and mathematical operators are all distinguished by color and shape.

We hypothesize that a block-based editor is well-suited for writing TSL in a structured context. Many programmers first introduction to computer science is through the block-based programming language Scratch [10], which evidences it usefulness as an introductory programming environment. Presenting TSL in a similar format lets users familiarize themselves with the mechanics of the interface and spend more time writing specifications rather than learning how to use the tool. Blockly manages the syntactic complexity of TSL in a clean, straightforward way, lightening the cognitive load required to learn and write TSL.

Our preliminary internal evaluations suggest that users benefit from using a structure editor when writing TSL specifications. Therefore, in the next section, we outline a potential user study design to formally evaluate the usefulness of our block-based editor. This study will also highlight and categorize common misconceptions about TSL.

4 Proposed Study Design

In this section, we outline a study design intended to evaluate the usefulness of a block-based editor for reactive synthesis. We first illustrate the environment and context within which our users will write their specifications. Then, we describe our experiment protocol, which is split into four sessions. After describing the protocol, we list the measurements we will take during each session of the study. Finally, we the discuss the hypotheses we expect to validate using the results of the study.

4.1 Environment for Evaluating Blockly

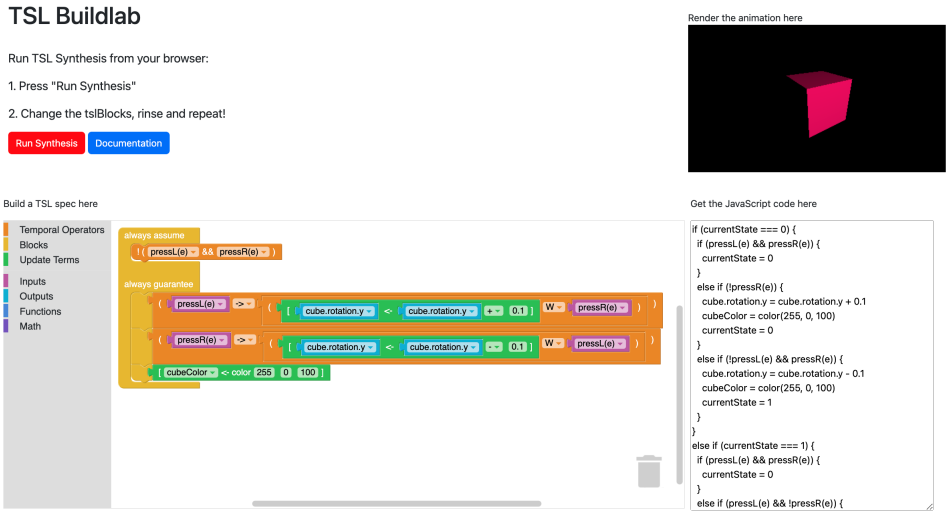

Figure 2. Interface for rendering animations using the specification from Table 1 and Fig. 1

Through iterative design loops, we identified that receiving immediate visual feedback on the correctness of a temporal logic specification helps users better understand the specification they are writing, thus we integrated TSL with JavaScript's Three.js library for rendering animations[1]. In this interface, demonstrated in Fig. 2, users build a specification using blocks, then attempt to synthesize their specification after clicking the "Run Synthesis" button. Once feedback is requested though the button click, users will either see the JavaScript code representation of their specification in the right textbox, or a message describing an error with their specification. An animation using a combination of the Three.js library and the reactive system specified by the user will also be rendered by the interface.

Users can then edit and retest their specification, encouraging an exploratory approach to learning TSL. The interface also supports toggling between building specifications with Blockly and writing them in plaintext. Complete documentation on the functions, primitives, shapes, and other properties that users can utilize to write their specifications is accessible through the top right "Documentation" button.

4.2 Experiment Protocol

The experiment consists of four sessions: a learning session, a task session, an evaluation session, and a creation session. We plan to recruit undergraduate students with a variety of programming experience, from students who have never programmed to students who are graduating computer science majors.

4.2.1 Session 1: Learning

The learning session is an online session that participants are asked to complete at home at their own pace. Participants are instructed to read through a documentation that introduces them to the basic concepts of temporal logic, the syntax of TSL, and several example specifications and their corresponding animations in the Three.js x TSL environment. After they finish the documentation, participants are asked to take a quiz that contains a series of two-alternative choice questions, in which participants must choose the correct specification for a simple animation. Participants need to gain full scores for the quiz, or they are asked to take the quiz again until they get all questions correct. This is to make sure that

1. a common baseline knowledge can be established for all participants before they come into the task session.
2. materials are actually learned and processed into long-term memory, alleviating participants' cognitive load in later sessions. And

1 Prototype interface for building TSL specs to render reactive animations: https://barnard-pl-labs.github.io/tslBlocks

3. the effectiveness of the learning material can be evaluated independently from that of the interface.

The documentation from the learning session remains available to participants in subsequent sessions. This is because we assume that the documentation serves as a guide for TSL semantics rather than syntax. To independently evaluate the effect of the documentation on the experiment, we design the documentation page to prompt participants with a question each time they open the page. The question asks participants to indicate their current intent, with the following (randomized order) multiple choice options:

- I am looking for the meaning of a particular function (semantics)
- I am looking for which function to use for a specific effect (knowledge)
- I am looking for how to write a particular function (syntax)
- I am looking for an example specification or animation (problem solving)
- I am looking for something else (fill in the blank)

4.2.2 Session 2: Tasks

The next day, participants come in-person for the task session where they will write TSL with both the block-based interface as well as the text-based interface. They are randomly assigned to first use either interface 1 (a block-based interface) or interface 2 (a text-based interface) to prevent possible confounding effects of biasing one particular interface due to the sequence that the interfaces are introduced. After watching a brief tutorial on the assigned interface, participants are given two prompts in the form of animation clips that they must replicate on the interface. Each prompt is given one at a time, and the prompt remains available throughout the task. Participants have 10 minutes to complete each one. If time runs out before the participant successfully replicates the prompt, the correct answer is revealed to them. Participants then repeat this process using the other interface, and asked to replicate two different prompts at a similar difficulty level (e.g. rather than spinning the cube left and right, spin it up and down). In total, participants respond to four prompts.

4.2.3 Session 3: Evaluation

Next, in the evaluation session, participants are asked to rate the following subjective measures on a 7-point Likert scale: their prior experience with temporal logics (TSL, LTL, or otherwise), block-based or visual scripting environment, and programming in general; their preferences for the two interfaces; and their confidence in their ability to write TSL at the time of the experiment. These subjective measures can be used to identify individual differences when we analyze the objective measures (such as percentage of correct specifications), and will better reflect an individual's personal experience with the interfaces.

4.2.4 Session 4: Creation

Lastly, in the creation session, participants have 10 minutes to use either of the provided interfaces to create one animation without a prompt. They are free to use the documentation from the learning session to help them in this task. Participants are informed that specifications created in this session will not be judged in terms of correctness or quality. This session is intended to afford a more ecologically valid environment (meaning that it is closer to the actual scenario for writing TSL than experimentally set), and to provide additional data for analysis.

4.3 Measurements

During the task session, per interface, we plan to measure the percentage of correct specifications, correctness of participants' specification, measured as the percentage of replicated rules divided by all rules in the animation prompt, and the time taken to finish each prompt (maximum time is 10 minutes, regardless of whether the specification is functional).

In both the task session and the creation session, we will measure the frequency of feedback requests, which will be measured by the number of times the "Run Synthesis" button is pressed.

The survey in the evaluation session will gather the following information:

- Prior knowledge expertise with coding and with temporal logics.

- Prior tool expertise with block-based or visual programming environment.
- Preference score for the two interfaces.
- Confidence score in participants' ability to write TSL.
- Post-study expertise (measured as correctness in the two-alternative choice questions).

Participants' self-directed specifications in the creation session will not be measured, but they will provide useful information about the practical use cases of TSL for novices.

Overall, the design of the experiment procedure ensures that participants have a basic understanding of temporal logics and the syntax of TSL before starting the tasks. This will allow users to focus on solving the well-defined problems at hand rather than struggling with unfamiliar concepts, and hence actually utilizing the language.

4.4 Identifying TSL Misconceptions

To fully understand what makes TSL difficult to write and build tools that address those difficulties, we must first identify common misconceptions users have in understanding TSL and reactive synthesis. Inspired by work on identifying misconceptions in Linear Temporal Logic (LTL) [11], we will log participants' specifications during the study. Every time a participant attempts to synthesize their specification during the study, we will log:

- The specification
- The synthesized JavaScript code (if synthesis succeeds)
- The error message (if synthesis fails)
- A unique session ID
- The timestamp

With the above information, we will be able to perform both within-individual analysis by tracing one individual's learning trajectory anonymously, and between-individual analysis for finding noticeable patterns in misconceptions among users. We can then cluster those misconceptions into a series of common syntactic and semantic misconceptions. We will include a section in our informed consent process to explain to the users what data will be collected and how it will be used.

4.5 Hypotheses

Our hypotheses are as follows:

1. The block-based interface facilitates a better understanding of TSL semantics and helps participants generate faster and more correct solutions.
2. Most people are novices to TSL, even those who are experienced in writing temporal-logics-based specifications. We expect to find a weak positive relationship between prior expertise, particularly with LTL, and the correctness of participants' responses in the task session. We do not expect prior coding experience to have a significant effect on performance.

5 Open Questions

We identify two key directions of open questions in the pursuit of a synthesis environment that supports new users.

First, we must address the issue of synthesized code quality to make it comprehensible to users. One way to achieve this is to prune the code results so that repetitive states and transitions or those that violate the assumptions of the specification are removed from the output. The algorithm to prune the generated code is still an open research question.

Second, we need to further tailor both the tutorial materials and interface for writing in a synthesis environment. The tutorial materials should convey the fundamental differences between writing programming languages and specification languages. For example, it is crucial to teach the idea of sketching the ideal result of a program and expressing it TSL, without worrying about the implementation details that will satisfy the desired result. Additionally, a further exploration on the habits of novice users and more experienced users when writing TSL is necessary to make a specification interface accessible to users with varying levels of programming and temporal logics experience. One way to make the interface versatile might be to identify a hierarchy of functions ranging from commonly

used to advanced and organize them in a way that allows users to either quickly access the features they need or delve deeper into the interface based on their needs.

6 Related Work

This work addresses the usability of synthesis and the development of synthesis interfaces. Integrating reactive synthesis into the development work-flow has recently been explored through the use of interactive, live data visualizations in programming environments [12] and the generation of program snippets from examples [13]. There have also been efforts to create synthesis interfaces that are usable for both novices and experts [14]. To help introduce programming languages to students, visual programming languages such as Scratch [10], are often used in education. For temporal logics, there is some existing work exploring the need for more usable interfaces [15]. Specifically, the temporal logic framework TLA+ (which is generally focused on verification [16], but also has been used for synthesis [17]) has spurred work into IDE support for temporal logic specifications [18].

While using a structure editor as a tool for synthesis is uninvestigated, combining synthesis with structure editors has seen research [19]. Of particular relevance is the use of Blockly as an interface for the Promela language of the SPIN model checker [8]. Promela is a specification language that integrates both temporal operators as well as program code. In contrast to TSL, Promela is a grammatically larger language, and we hypothesize that this will make TSL an even better fit for Blockly. Additionally, Promela is used for model checking instead of synthesis - in model checking both the specification and model must be provided by the user whereas in synthesis only the specification must be provided. We hypothesize that the synthesis setting will thereby reduce the labor overhead of using the block-based language (clicking and dragging in the visual interface compared to keyboard navigation). Techniques that address the usability and limitations of program synthesis have also been examined in graphical domains [20].

An evalutation of Promela [8] listed two advantages of using visual programming language tools for programming education: 1) Students' difficulty in understanding a particular syntax of the modeling language for a model checker is reduced. And 2) The education cost (time and human resources) to correct syntax errors in the models written by students is reduced. Particularly, they envisioned a visual programming language for model checkers for educational purposes, rather than professional usage.

7 Conclusions

In this work, we focus on the learnability and usability of synthesis. We introduced a block-based structure editor to help users write Temporal Stream Logic (TSL) specifications. We proposed a framework for evaluating the effectiveness of a structure editor in the context of reactive synthesis. Although the main difficulty with writing TSL is semantic complexity, users must first overcome their syntactic misconceptions to focus on the semantics of their specifications. Overcoming this hurdle is a non-trivial task and is critical to the learning process. Unfamiliar syntax can often deter novices from continuing to work with temporal logic. We illustrated the usefulness of our work and introduced several future directions based on case studies and related work.

References

[1] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 179–190.

[2] R. Bloem, S. Jacobs, and A. Khalimov, "Parameterized synthesis case study: Amba ahb (extended version)," *arXiv preprint arXiv:1406.7608*, 2014.

[3] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, "Temporal stream logic: Synthesis beyond the bools," in *International Conference on Computer Aided Verification*, Springer, 2019.

[4] W. Choi, M. Vazirani, and M. Santolucito, "Program synthesis for musicians: A usability testbed for temporal logic specifications," in *Asian Symposium on Programming Languages and Systems*, Springer, 2021, pp. 47–61.

[5] G. Geier, P. Heim, F. Klein, and B. Finkbeiner, "Syntroids: Synthesizing a game for fpgas using temporal logic specifications," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, 2019, pp. 138–146.

[6] B. C. P. L. Lab. (2021). Tsltools: Library and tools for the tsl specification format. accessed on 8-18-2022, Barnard College, [Online]. Available: https://barnard-pl-labs.github.io/tsltools/.

[7] J. Nielsen, *Ten usability heuristics*, 2005.

[8] S. Yamashita, M. Tsunoda, and T. Yokogawa, "Visual programming language for model checkers based on google blockly," in *International Conference on Product-Focused Software Process Improvement*, Springer, 2017, pp. 597–601.

[9] N. Fraser, "Google blockly-a visual programming editor," *URL: http://code.google.com/p/blockly. Accessed Aug 2022*, 2014.

[10] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, *et al.*, "Scratch: Programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[11] B. Greenman, S. Saarinen, T. Nelson, and S. Krishnamurthi, "Little tricky logic: Misconceptions in the understanding of ltl," *arXiv preprint arXiv:2211.01677*, 2022.

[12] C. Omar, D. Moon, A. Blinn, I. Voysey, N. Collins, and R. Chugh, "Filling typed holes with live guis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 511–525.

[13] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova, "Small-step live programming by example," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 614–626.

[14] A. M. McNutt and R. Chugh, "Integrated visualization editing via parameterized declarative templates," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–14.

[15] N. De Francesco, A. Santone, and G. Vaglini, "A user-friendly interface to specify temporal properties of concurrent systems," *Information Sciences*, vol. 177, no. 1, pp. 299–311, 2007.

[16] C. Newcombe, "Why amazon chose tla+," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings 4*, Springer, 2014, pp. 25–39.

[17] G. Moreira, C. Vasconcellos, and J. Kniess, "Fully-tested code generation from tla+ specifications," in *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing*, 2022, pp. 19–28.

[18] R. Cuinat, C. Teodorov, and J. Champeau, "Specedit: Projectional editing for tla+ specifications," in *2020 IEEE Workshop on Formal Requirements (FORMREQ)*, 2020, pp. 1–7. DOI: 10.1109/FORMREQ51202.2020.00008.

[19] B. Hempel and R. Chugh, "Tiny structure editors for low, low prices!(generating guis from tostring functions)," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2020, pp. 1–5.

[20] J. Lubin, N. Collins, C. Omar, and R. Chugh, "Program sketching with live bidirectional evaluation," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020.