Towards the Usability of Reactive Synthesis: Mapping TSL Misconceptions to Mitigations

Leyi Cui * [†1], **Raven Rothkopf** * [‡1] **and Mark Santolucito** [§1]

[1] *Barnard College, Columbia University, New York, USA*

Abstract

Reactive program synthesis from logical specifications has yet to match the user-friendly approach of example-based programming for spreadsheets, despite its success in specific domains. A main challenge hindering the broader adoption of reactive synthesis is in the complexity of specification engineering in temporal logics. We map out misconceptions and mitigations that arise as users write temporal logic specifications in Temporal Stream Logic. Our goal is to provide a roadmap for future usability work that can elevate temporal specification engineering for synthesis to match the usability support available for software engineering.

1 Introduction

FlashFill [1] is a notable success in program synthesis, allowing users to generate Excel data manipulation programs from simple examples. Yet, programming-by-example is just one approach within program synthesis. For cases where examples are impractical, formal logic-based synthesis offers a viable alternative. This is particularly true for reactive systems operating on infinite input and output streams, where examples can be limiting. These systems are typically instead defined using temporal logic, with reactive synthesis creating a controller that ensures correct interaction between the system and environment, responding aptly to all input scenarios. The synthesis of the AMBA bus protocol from Linear Temporal Logic exemplifies this success [2]. Recent advancements extend reactive synthesis to educational programs [3], FPGA game development [4], musical interfaces [5], and interactive animation creation [6].

Despite its wide array of application domains and substantial research progress, temporal logic specifications remains unfamiliar and challenging to software developers to write. Recent work has explored these challenges [3], [7], yet solutions remain unclear. Unlike software engineering, which has a wealth of debugging techniques, specification engineering for temporal logics lacks similar resources. This work aims to define reactive synthesis challenges in TSL specification writing, suggest mitigation strategies, and highlight areas needing more research.

We focus on Temporal Stream Logic (TSL) [8] - a high-level, temporal logic specification language used in reactive synthesis. It extends Linear Temporal Logic (LTL) [9] with updates and predicates over arbitrary function terms. The synthesis of a TSL specification yields concrete program code corresponding to the reactive system by capturing both reactive properties and data manipulations. TSL has shown promising initial results that could introduce reactive synthesis as a path towards more expressive low/no-code platforms, which currently suffer from limitations in language expressivity [10]. The use of TSL has extended reactive synthesis to new application domains including music [5], video games [11], mobile apps [8], animation [6], and autonomous vehicle controllers [8]. Whether with TSL, LTL, or any other reactive logic [12], [13], there is a critical need for a structured framework for the development of temporal logic specification engineering support tools.

In this work, we present the following key contributions:

1. Based on anecdotal first-hand experience, we outline common misconceptions users have about reactive synthesis when writing specifications in Temporal Stream Logic (TSL).
2. We outline a list of mitigations that assist in the development and debugging of reactive synthesis.
3. We propose a mapping between the misconceptions and mitigations to define the space for future work in the usability of reactive synthesis.

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

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

*Authors contributed equally
†Email: lc3542@barnard.edu
‡Email: rgr2124@barnard.edu
§Email: msantolu@barnard.edu

4. To provide motivation, we explore the real-world repercussions of TSL misconceptions using a benchmark specification from a recently published paper. We then highlight the mitigations that would have prevented those misconceptions.

2 Preliminaries

To make this work self-contained, we formally define some technical terms that are referenced later in our descriptions of the misconceptions and mitigations. Note that understanding the formal definitions of Temporal Stream Logic is not critical to understanding the main contributions of this work. However, it provides extra context for the origin of the misconceptions and the technical challenges of the mitigations. For a full exposition of TSL's formal background, we refer the reader to prior work [8].

2.1 TSL

Temporal Stream Logic (TSL) [8] is a high-level, logical specification language that describes the behavior of a reactive system over discrete time. TSL extends Linear Temporal Logic (LTL) [9] with updates and predicates over arbitrary function terms. With TSL, one can specify a reactive system that reacts to an infinite stream of inputs to produce an infinite stream of outputs. TSL specification can synthesize implementable reactive programs written in JavaScript or Python.

TSL uses the usual LTL operators: next $\bigcirc$ and until $\mathcal{U}$. Additionally, the syntax of TSL contains *predicate terms* τ_P, *function terms* τ_F, and *update terms* τ_U, as defined in the following grammar:

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

$$\begin{aligned} \tau_F &:= \mathtt{s.f}(\tau_F^0, \tau_F^1, \ldots, \tau_F^{n-1}) \\ \tau_P &:= \mathtt{p}(\tau_F^0, \tau_F^1, \ldots, \tau_F^{n-1}) \\ \tau_U &:= [\mathtt{s} \leftarrowtail \tau_F] \end{aligned}$$

TSL also uses the standard derived operators, such as *always* $\square\varphi \equiv \bot\,\mathcal{R}\,\varphi$, *eventually* $\diamondsuit\varphi \equiv true\,\mathcal{U}\,\varphi$, *weak until* $\varphi\,\mathcal{W}\,\psi \equiv (\varphi\,\mathcal{U}\,\psi) \vee (\square\varphi)$, and *release* $\varphi\,\mathcal{R}\,\psi \equiv \neg(\neg\varphi\,\mathcal{U}\,\neg\psi)$.

TSL specifies the behavior of a reactive system by utilizing signals, indicated as $\mathtt{s}$, which contain data values of arbitrary type. Through a TSL specification, it is possible to describe how functions are applied to these signals over time. Signals can either be pure outputs or *cells* that memorize data values so that the outputs of a specific time t are provided as inputs for time $t+1$. These characteristics establish the semantics of TSL, which follow the conventional LTL semantics while integrating predicate evaluations, function evaluations, and update terms. A formal definition of TSL semantics is given in [8].

TSL synthesis can be modelled as a two-player game between the system (choosing moves in the form of outputs $\mathcal{O}$) and the environment (choosing moves in the form of inputs $\mathcal{I}$). A winning strategy of the system in this game will be a finite automaton that can produce output values that satisfy φ in reaction to all possible inputs from the environment. Likewise, a winning strategy of the environment is a finite automaton which provides a sequence of inputs that no system can satisfy.

With TSL, the user may provide constraints on the system and the environment players. Constraints on the environment restrict the possible inputs given to the system and are called assumptions. Assumptions are specified in TSL's `assume` block. Constraints on the system are used to specify the system and its outputs and are called guarantees. Guarantees are specified in TSL's `guarantee` block. These two blocks are desugared into a single TSL formula of the form $assume \Rightarrow guarantee$. Notice that if an assumption is violated, the formula becomes $\bot \Rightarrow guarantee$, and so the guarantees are trivially satisfied. Thus, the environment cannot violate an assumption and still win the game.

2.2 TSL Synthesis

The realizability problem of TSL is stated as: given a TSL formula φ, is there a strategy $\sigma \in \mathcal{I}^+ \to \mathcal{O}$ mapping a finite input steam (since the beginning of time) to an output (at each particular timestep),

such that for any input stream $\iota \in \mathcal{I}^\omega$, and every possible function interpretation (some concrete implementation) $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$, the execution of that strategy over the input $\sigma \wr \iota$ satisfies φ, i.e.,

$$\exists \sigma \in \mathcal{I}^+ \to \mathcal{O}.\ \forall \iota \in \mathcal{I}^\omega.\ \forall \langle\cdot\rangle : \mathbb{F} \to \mathcal{F}.\ \sigma \wr \iota, \iota \vDash_{\langle\rangle} \varphi$$

If such a strategy σ exists, we say that σ realizes φ.

The synthesis problem of TSL asks for a concrete implementation of σ. In TSL synthesis, this model σ can be turned into a Control Flow Model (CFM), an abstract representation of the system that covers all possible behaviors, which can then be represented as program code. A formal definition of the TSL realizability and synthesis is available in [8].

Note here that our synthesis procedures asks for one such strategy σ - depending on the specification, there may be many such strategies that will satisfy φ. It is the task of the synthesis engine to choose one among these solutions (typically, the smallest solution is chosen, especially when using bounded synthesis [14]).

2.3 TSL Counter Strategies

A *counter strategy* for reactive synthesis is a strategy that the environment can use to falsify the system no matter the moves the system makes. This strategy can only be generated from an unrealizable TSL formula. For the reactive synthesis problem, a counter strategy can be formally defined as the strategy $\sigma_{\mathcal{C}}$ that satisfies the TSL formula $\neg\varphi$, where $\mathcal{I}_{\mathcal{C}} = \mathcal{O}$ and $\mathcal{O}_{\mathcal{C}} = \mathcal{I}$ are the inputs and outputs for $\sigma_{\mathcal{C}}$, generated by the system and the environment.

2.4 TSL Example

We demonstrate the advantages of TSL as a specification language for real-world development using the example application of 3D animation. The small specification depicted in Table 1 synthesizes a concrete implementation of a controller for a reactive, animated cube. In this specification, the cube reacts to the left and right arrow key presses, spinning about the y-axis depending on the key pressed. The arrow key values, `pressLeft(e)` and `pressRight(e)`, are delivered as Boolean input streams to the system: `true` while the arrow key is pressed and `false` otherwise. The cube's rotation, `cube.rotation`, is emitted from the system as an integer output stream. Finally, we have one additional signal, an internal signal, or *cell*, `x`. `x` is an integer output value that appears as an input in the next time step.

TSL	**JavaScript**

!(pressR(e) && pressL(e));
}
always guarantee {
pressR(e) -> ([x <- x + 1] W pressL(e));
pressL(e) -> ([x <- x − 1] W pressR(e));
[cube.rotation <- x];
}
	if (pressL(e) && pressR(e)) {
	currentState = 0
	} else if (!pressL(e)) {
	x = x + 1
	cube.rotation = x
	currentState = 0
	} else if (pressL(e) && !pressR(e)) {
	x = x − 1
	cube.rotation = x
	currentState = 1
	}
	} else if (currentState === 1) {
	if (pressL(e) && pressR(e)) {
	currentState = 0
	} else if (!pressL(e) && pressR(e)) {
	x = x + 1
	cube.rotation = x
	currentState = 0
	}
	}

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

We have several requirements for our system to control the cube movements depending on the left and right arrow keys. First, the system can never receive two input signals at the same time.

The mutual exclusion of inputs from the environment, `!(pressLeft(e) && pressRight(e))`, is specified in the `assume` block. Second, once an arrow key is pressed we want the cube to continuously spin in the direction specified by the input until the complementary key is pressed. To satisfy this behavior, we use the weak until operator, $\varphi \mathcal{W} \psi$, which states that either φ is true until ψ is true, or φ is true forever. The line of the plaintext TSL formula, `pressLeft(e) -> ([x <- x + 1] W pressRight(e))`, specifies that when the system receives the input `pressLeft(e)`, the cell `x` will be continuously incremented until the system receives the input `pressRight(e)`. In the same time step, `cube.rotation` is updated with the value of `x` that depends on the input, consequently spinning the cube to either the left or the right. By the semantics of TSL, it is already ensured that assignments to the same cell are mutually exclusive, i.e., `x` can never be incremented and decremented at the same time, further enforced by the mutual exclusion of the system inputs.

When the specification is synthesized using our TSL synthesis tool [15], the result is an automatically generated CFM that satisfies the specified control behavior or the reactive rotation of the 3D cube. This CFM is then used to generate the system's implementation in program code shown in the second column of Table 1.

3 TSL Misconceptions

We summarize key misconceptions and challenges faced by developers in writing TSL. We derived these categories from our personal experience of writing TSL specifications. Identifying what makes TSL hard to write further defines the problem space for future work on improving the learnability and usability of TSL.

3.1 Assume vs. Guarantee

The basic form of a reactive specification is of two blocks: assume and guarantee. Conceptually, the assume block is meant to provide restrictions on the environment and the guarantee is meant to provide specifications for the system to be synthesized. However, reactive synthesis studies [3] have shown that the task of knowing what to specify in each block is error-prone. Users often struggle with with differentiating between variables managed by the environment and variables managed by the system. Knowing which properties to specify in the `assume` block is particularly challenging.

A common mistake in this category is specifying system updates on the right side of an implication in the `assume` block. Such a specification would allow the system to choose a set of updates that violates the assumptions of the specification, and since these blocks are desugared into the form $assume \Rightarrow guarantee$, the predicate evaluates to false, meaning that any consequent will yield true for the entire formula (false implies anything). Take as a concrete example the specification $\square \neg [x \leftarrow\!\!\prec y] \Rightarrow \square(z \wedge \neg z)$. The guarantee is impossible to satisfy on its own, but if the system choose to always update $[x \leftarrow\!\!\prec y]$, the specification is $\bot \Rightarrow \square(z \wedge \neg z)$, which is trivially satisfiable.

3.2 (Non) Reactive Systems

The power of TSL is best utilized when synthesizing reactive, multi-state systems. When looking only at a TSL formula, it can be difficult to determine if a system is indeed reactive. A non-reactive system written in TSL is one without inputs from the environment - the system cannot "react" without a stream of inputs from the environment. While the specification may be realizable, likely, the specification does not capture the user's intentions. Both TSL formulas written for other publications [16] (cf. our Case Study in Sec. 6) and formulas written by members of our lab have unintentionally specified non-reactive systems.

3.3 Temporal Operator Semantics

In order to use TSL to generate reactive systems, developers must first understand the logic they are writing. This process of understanding can be challenging because writing temporal logic requires a shift in mindset from the process of writing code [3]. If a user holds a misconception about the semantics of temporal logic operators, it can quickly lead to unrealizable specifications or realizable specifications that do not have the intended behavior. Furthermore, there are no precautions that

address these misconceptions. The synthesis tool will blindly apply properties and check or generate the requested behavior, whether it is the desired one or not. Therefore, it is critical to help users accurately understand TSL.

We outline three concrete misconceptions about the semantics of TSL's temporal operators, built upon our own experiences and the results of recent work.

Negation refers to a misconception in the understanding of the `!` operator. In our pilot studies, a user confused `! F(x)` with `F(! x)`. The first specification guarantees that `x` will never be true, while the second specification guarantees that `x` will not be true for at least one time-step. The syntactic similarity makes it difficult for new users to understand the significant semantic difference.

WeakU [7] refers to a misconception that confuses the $\mathcal{U}$ operator with its weak variant, $\mathcal{W}$. $\mathcal{U}$ guarantees that its second subterm will eventually hold, while $\mathcal{W}$ does not. As an example, take the TSL specification from Table 1, but with one minor change. In this specification, shown in Figure 1,

```
always assume {
    ! ( pressR(e) && pressL(e) );
}
always guarantee {
    pressR(e) -> ( [x <- x + 1] U pressL(e) );
    pressL(e) -> ( [x <- x - 1] U pressR(e) );
    [cube.rotation <- x];
}
```

Figure 1. Modified specification from Table 1, demonstrating a **WeakU** misconception.

we have replaced $\mathcal{W}$ with $\mathcal{U}$ on lines 5 and 6. This specification is now unrealizable–or has no solution to the synthesis problem–because $\mathcal{U}$ guarantees that once the first subterm is true, the second subterm must eventually be true. Our specification has no assumption on the environment's inputs that fulfill the guarantee of the $\mathcal{U}$ operator. We explore how to debug this class of misconception in Sec. 4.9.

Prior work [7] explores other semantic misconceptions such as **BadStateQuantification**. As with any logic, complex combinations of operators can always lead to confusion, especially for novice users.

3.4 Precedence and Syntactic Misconceptions

Although the semantic complexity of temporal logic is the most significant hurdle to overcome in the development process, TSL's syntactic complexity can also be non-trivial, especially for novices. Users have to deal with the new language paradigm of TSL with its unfamiliar operators and structural form. Syntactic misconceptions and even simple typos can inhibit developers from focusing on the semantic complexity of their specifications.

Prior work has shown that one of the key syntactic misconceptions developers have when working with reactive synthesis is precedence [7]. An operator precedence misconception is defined as a specification that is correct up until missing parentheses. We have found in our pilot studies that other syntactic issues that arise in software engineering also arise in temporal logic specification engineering. As another example, variable shadowing can easily turn input signals into cell signals, as illustrated in our Case Study in Sec. 6.

3.5 Unrealizability

Initial attempts of users to write a temporal logic specification are often unrealizable, meaning no system implements the specification. Debugging unrealizable specifications is challenging because they cannot be executed or simulated (similar to code that does not compile or has a runtime error). A common reason for unrealizability is that assumptions about the environment are incomplete. While unrealizability is not a misconception, many misconceptions lead to unrealizable specifications. Unrealizability is a well-studied phenomenon [17]–[19], and recent work has begun to investigate how unrealizability impacts users in their specification development process [3]. Just as software engineers have tools to debug code that does not fully compile or run (type errors, unit test, break points, etc.), so too is there a need for techniques for fixing unrealizable specifications.

3.6 Underspecification

Even in the case where the user has provided a realizable specification, and every part of that specification correctly matches the users intentions, there is still room for error. In particular, the goal of synthesis is that the user may leave some part of the problem unspecified and the synthesis engine will complete that part of the solution automatically. The goal is that the user specifies all the parts of the problem they care about and leaves other parts unspecified. However, if a user had in mind some constraint on the system and did not include that constraint as part of the specification, one of two things might happen. Either the synthesis engine will find a solution that matches the users intentions (either by luck or some intention inference), or the synthesis engine will find a solution that does not match the users missing part of the specification.

As users develop expertise in reactive synthesis, we have noticed they are often able to correctly guess the behavior of the synthesis engine and strategically omit parts of the specification that can be handled well by the tool. However, novice user do not have this expertise and need ways to inspect the output of the tool to confirm that they did not miss some part of the specification.

4 TSL Mitigations

We outline a list of mitigations that aid in the reactive synthesis development and debugging process. We then explain the motivation behind the mapping from misconceptions to the proposed mitigations.

4.1 Syntax highlighting/warnings

Similar to syntax highlighting in an IDE for software development, we can also use syntax highlighting for TSL. This can use be used to help users recognize particular syntactic constructions in TSL, as well as to alert users to potential semantic issues. As a prototype of such a tool, we built a VSCode extension for TSL. This extension does basic syntax highlighting, but also can alert users to potential semantic errors, for example issues with construction of assumptions, as described in Sec. 3.1 would appear as shown in the screenshot of our tool shown in Fig. 2.

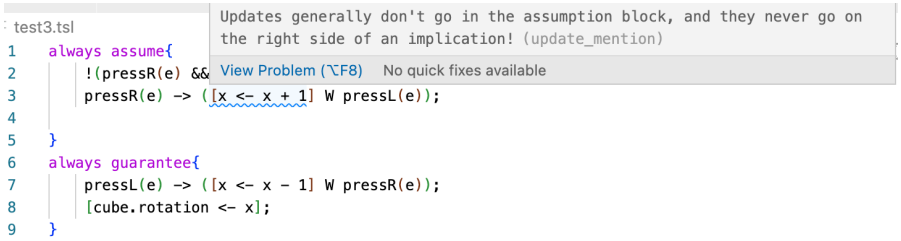

Figure 2. The TSL specification from Table 1 demonstrating a Assumption vs. Guarantee misconception with syntax highlighting and error checking.

4.2 Block based editors

Block-based structure editors, such as Scratch [20], have become widely accepted as tools for learning programming. These tools lighten the cognitive load experienced when programming by guarding against syntactic mistakes. Anecdotally, we noticed that when writing TSL, novice users experience a similar distracting cognitive load of worrying about syntax. When writing temporal logic, users must recall the syntactic and grammatical structure of the new language paradigm of TSL. 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 in their specifications instead of remembering it, shifting the working mental model from recall to recognition.

Recent work has proposed a block-based structure editor for TSL, tslBlocks [6]. In the specification editor, users can drag and drop temporal logic operators, update terms and predicates into locations that are restricted by the grammar of TSL. Fig. 3 demonstrates an example of a specification built in the tslBlocks editor. In Fig. 3, temporal operators, update terms, input/output signals, domain-specific functions, and mathematical operators are all distinguished by color and shape. This mitigation is well

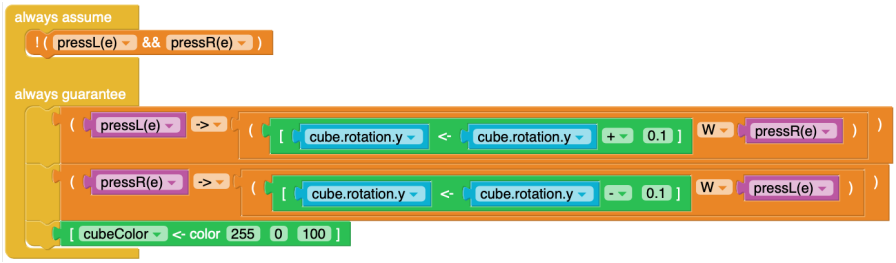

Figure 3. The TSL specification from Table 1 built with the tslBlocks structure editor from **??**.

suited to tackle the syntactic misconceptions from Sec. 3.4, as well as any syntactic mistakes.

4.3 State Machine Visualization

We propose the ability to interactively test a synthesized solution while visualizing the generated Mealy Machine as another mitigation. This mitigation is well suited to tackle the Temporal Operator Semantics (3.3) and Underspecification (3.6) misconceptions. This is particularly helpful to identify specifications that result in fewer states than expected. We built a prototype of a tool that automatically generates an interactive state machine during the synthesis process, as shown in Fig. 4.

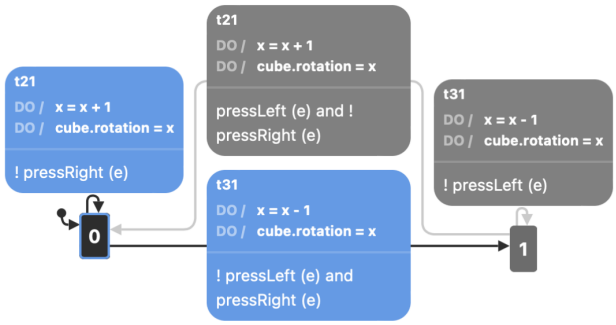

Figure 4. A visualization of the synthesized automaton from the specification in Table 1.

4.4 Reactive System Visualizer

Understanding the inputs, outputs, and cells of a reactive system is a critical part of building a TSL specification. We built a reactive system visualizer to illustrate the behavior of reactive systems that visualizes input, output, and cell values from the TSL specification. While the state machine mitigation described in Sec. 4.3 aims to provide an understanding of the fine-grained behavior of a synthesized system, the reactive system visualization specifically aims to act as a sanity check for ensuring the overall structure of the reactive system matches the user's intention.

4.5 Minimal Assumption Cores

In reactive synthesis, assumptions are constraints put on the environment of the reactive system. Assumptions restrict the possible inputs that the environment can provide to the system. These assumptions are provided by the user, but certain assumptions may not be necessary to realize a specification. Similar to a warning about unused library imports, identifying a minimal assumption core can help prune extraneous assumptions. However, the use case is slightly more subtle - if an assumption is not necessary for realizability there is no guarantee that it will have an impact on the synthesis result. As such, extraneous assumptions may indicate as misconception the user has with another part of the specification.

4.6 Code Pruning

Optimizing the synthesized code from a TSL specification itself is a new mitigation. With readable code, users can map changes in their specifications to changes in the synthesized code and debug their specifications. This mental mapping could allow users better understand the semantics of temporal operators as they observe how changing operations impact the synthesized code.

Pruning invalid transitions generated from violated assumptions is one area of potential optimization of the synthesized code. TSL synthesis generates code to handle violations of assumptions because the synthesized controller must be a system that can handle any stream of input, no matter if that input violates an assumption of the environment or not. This behavior is due to the hardware synthesis roots of reactive synthesis, where the system (a circuit) will continue to operate no matter the given inputs. In the software setting, we now face the question of how to handle invalid transitions.

The generated Mealy machine contains transitions telling the system what to do with inputs that violate the assumptions of the environment (from the `guarantee` block of the TSL specification). These transitions are invalid in the sense that they violate user-specified assumptions. Consequently, this leads to substantial portions of non-executable and perplexing code, particularly in complex systems with numerous states and assumptions. One solution to achieve readable and more optimized synthesized code 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 [6].

This mitigation is well suited to tackle the Temporal Operator Semantics (3.3) misconception.

4.7 Source mapping

To better understand the semantics of a temporal logic formula in relation to the synthesized output, we propose source mapping between the specification and the generated code. Similar to the way source mapping can be used to explore code generation through compiler passes [21], this tool might be used to better understand the synthesis process. For a compiler, source mapping is a one-to-many mapping - a single line of code may generate multiple lines in the compiled target language. However, a key challenge is that, unlike code compilation, in reactive synthesis the mapping from specification to target is many-to-many. That is, not only does a single line of the specification have an impact on many lines of code, a single line of generated code can be attributed to many (non-contiguous) lines of the specification. An effective source mapping algorithm for Temporal Stream Logic synthesis is still an open problem.

4.8 Unrealizable Cores

Another tool used in reactive synthesis to debug unrealizable specifications is the generation of unrealizable cores [22]. Unrealizable cores can be viewed as a fault-localization problem. The core of an unrealizable spec is the minimal set of guarantees–or constraints on the system–that render the synthesis problem unrealizable. After obtaining this minimal set, users have more information about what is causing their specifications to be unrealizable. This mitigation is well suited to tackle the Unrealizability (3.5) misconception.

4.9 Counter Strategies

One of the main challenges of reactive synthesis is dealing with unrealizable specifications or specifications of systems that have no correct controller implementation. After writing a specification, attempting synthesis, and getting that the specification is unrealizable, users may not know where to begin to debug their spec. As a result, substantial tooling has been developed to combat the problem of unrealizability [23], [18], including the generation of counter-strategies specifically for reactive systems [24]. A counter-strategy in program synthesis is a strategy the environment can use to falsify the specification no matter how the system moves. For developers debugging an unrealizable specification, a counter-strategy can provide information about the inputs from the environment that can cause the system to fail and what areas of the specification must be refined.

If a misconception about a temporal operator leads to an unrealizable specification, localizing the fault may lead to a correction in the understanding of temporal operator semantics. This mitigation is

well suited to tackle the Temporal Operator Semantics (3.3) and Unrealizability (3.5) misconceptions. The debugging process outlined above may not have been possible without that use of a counter strategy. How best to present counter strategies to users is still an open question.

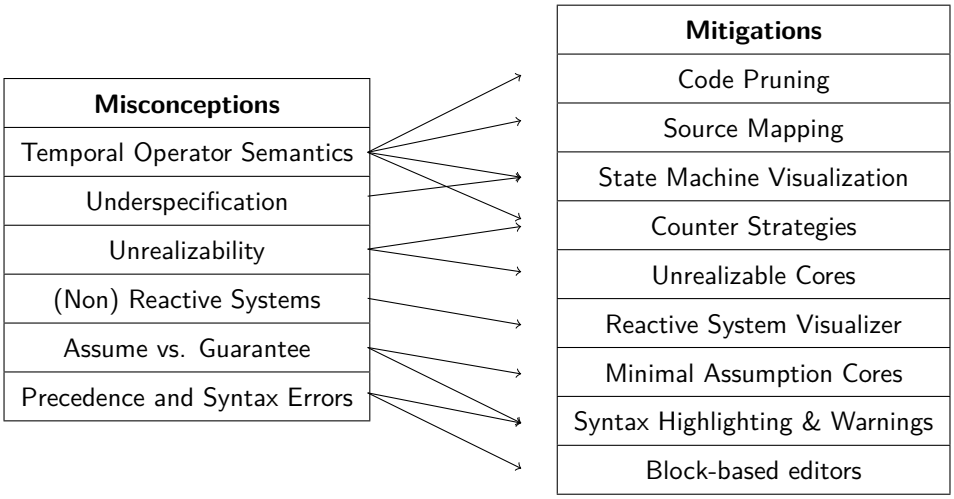

Table 2. This table proposes a mapping between common misconceptions that developers have in their understanding of TSL and current mitigations that can be utilized to combat those misconceptions.

5 Mapping TSL Misconceptions to Mitigations

To summarize, we propose a map, Table 2, between common misconceptions about Temporal Stream Logic and existing mitigations to address those misconceptions. This map can serve as a space for future work in the field of increasing the learnability and usability of reactive synthesis and temporal logic specification languages in practice.

6 Case Study

To further demonstrate the repercussions of TSL misconceptions in reactive synthesis, we provide a practical example taken from a recently published paper [16]. As commented in their benchmark GitHub repository [25], the TSL specification, shown in Fig. 5, aims to specify "a 'bidirectional' (i.e. 'full duplex') escalator capable of moving up or down. Its direction is controlled by the number of people waiting to take the escalator; when there are none, the escalator stops." As we will show below - this natural language specification does not match the TSL specification. Using our framework of misconceptions and mitigations, we identify the misconceptions that cause the mismatch between the natural language specification and the TSL specification, as well as the mitigations that could have helped to correct this specification.

Recall that the `always assume` block specifies constraints on the input the system gets from the environment. Here, the spec restricts people from entering the bottom of the escalator is moving down, and vice versa. Recall also that the `always guarantee` block describes the system's response to input values. Lines 9-12 aim to describe the movement of the escalator from the bottom to the top. Lines 9 and 10 aim to describe that if one person enters the bottom of the escalator, the number of people on the bottom will increase by one and the number of people on the escalator will also increase by one. At the same time, the number of people on the top of the escalator remains the same. Lines 11 and 12 aim to describe that if one person exits the top of the escalator, the number of people on the top will decrease by one, the number of people on the escalator will decrease by one, and the number of people on the bottom of the escalator remains the same. Lines 15-18 follows the same pattern for when the escalator moves up. Lines 21-24 aim to describe the directional components of the escalator. If the number of users on the escalator, the number of users at the bottom, and the number of users on top are all equal to 0, the escalator will stop. Otherwise, if there are more users waiting at the bottom than the top, the escalator will go up, and vice versa.

While this specification is realizable, there are three misconceptions shown in this benchmark

```
//#LIA#
always assume {
  [steps <− bottom()] <−> ![bottom <− add bottom c1()];
  [steps <− up()] <−> ![top <− add top c1()];
}

always guarantee {
  // Bottom movements
  [bottom <− add bottom c1()] && [top <− top] <−> [ users <− add users c1() ];
  [bottom <− bottom] && [top <− sub top c1()] <−> [ users <− sub users c1() ];

  // Top movements
  [bottom <− bottom] && [top <− add top c1()] <−> [ users <− add users c1() ];
  [bottom <− sub bottom c1()] && [top <− top] <−> [ users <− sub users c1() ];

  // Directional components
  eq users c0() && eq bottom c0() && eq top c0() −> [steps <− stop()];
  eq users c0() && gt bottom top −> [steps <− up()];
  eq users c0() && !(gt bottom top) −> [steps <− down];
}
```

Figure 5. The faulty TSL specification for an escalator from [16]

specification which mismatches the natural language specification and the TSL specification.

Misconception: (Non) Reactive Systems. A (Non) Reactive Systems' misconception refers to confusion about the optimal use of TSL (Sec. 3.2). Synthesizing non-reactive systems may not make a specification unrealizable, but poses risks of misalignment with user intentions. For instance, the escalator specification results in a non-reactive system as it doesn't incorporate environmental inputs. This misconception, mainly due to modeling issues, arises because `top` and `bottom` are treated as cells instead of inputs, as well as the fact that there are no temporal operators in the specification.

Misconception: Assume vs. Guarantee. An Assume vs. Guarantee misconception occurs when there's confusion about content allocation between the `always assume` and `always guarantee` blocks in TSL (Sec. 3.1). Here, lines 3 and 4 with system updates specified on the right side of an implication are in fact controlled by the system.

Misconception: Syntactic Problem. In line 24, there is a missing set of parenthesis after `down`, which makes it a input signal rather than a constant function. In this specification, `top` and `bottom` should be inputs, and `up()` and `down()` should be function terms that control the movement of the escalator. Similarly, in line 3 there is a syntactic mistake - `[steps <- bottom()]` should instead be `[steps <- down()]` in order to update the steps with the `down()` command.

7 Related Work

Recent work [7] focused on identifying and categorizing common misconceptions about temporal logic for reactive synthesis. Building upon this, we focus on Temporal Stream Logic, and construct more generalized misconceptions that can be addressed through concrete mitigations in the form of developer support tools. There is an increasing awareness of the need to address the user experience of temporal logic specification engineering [3], [6]. This will potentially have an outsize impact on the viability of teaching temporal logic in classroom settings.

8 Conclusion

Our goal in this work is to give an initial accounting of the various challenges and opportunities facing the synthesis community as we seek to give specification engineering the same first-class treatment as software engineering. We leave for future work a full user study to properly evaluate each of these mitigations and verify the claims made in Table 2. Additionally, we have focused our work on specifications in TSL, in the belief that many of this concepts will generalize to other temporal logics. There are however temporal logics which are more expressive than TSL - for example TSL is a fragment of Temporal Stream Logic Modulo Theories (TSL-MT) [16]. We hypothesize when working with richer temporal logic languages, there may be further misconceptions and mitigations - as one example, the issues of type safety arises in TSL-MT.

References

[1] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

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

[3] D. Ma'ayan and S. Maoz, "Using reactive synthesis: An end-to-end exploratory case study,"

[4] 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)*, 2019, pp. 138–146. DOI: 10.23919/FMCAD.2019.8894261.

[5] 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.

[6] R. Rothkopf, A. L. Cui, H. T. Zeng, A. Sinha, and M. Santolucito, "Towards the usability of reactive synthesis: Building blocks of temporal logic," Plateau Workshop, 2023.

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

[8] 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.

[9] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, ieee, 1977, pp. 46–57.

[10] Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, "Characteristics and challenges of low-code development: The practitioners' perspective," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21, Bari, Italy: Association for Computing Machinery, 2021, ISBN: 9781450386654. DOI: 10.1145/3475716.3475782. [Online]. Available: https://doi.org/10.1145/3475716.3475782.

[11] 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.

[12] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *Proceedings of the 18th international conference on hybrid systems: Computation and control*, 2015, pp. 239–248.

[13] D. J. Fremont and S. A. Seshia, "Reactive control improvisation," in *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, Springer, 2018, pp. 307–326.

[14] C. Gerstacker, F. Klein, and B. Finkbeiner, "Bounded synthesis of reactive programs," in *Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings 16*, Springer, 2018, pp. 441–457.

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

[16] W. Choi, B. Finkbeiner, R. Piskac, and M. Santolucito, "Can reactive synthesis and syntax-guided synthesis be friends?" In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 229–243.

[17] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev, "Diagnostic information for realizability," in *Verification, Model Checking, and Abstract Interpretation: 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008. Proceedings 9*, Springer, 2008, pp. 52–67.

[18] R. Könighofer, G. Hofferek, and R. Bloem, "Debugging formal specifications: A practical approach using model-based diagnosis and counterstrategies," *International journal on software tools for technology transfer*, vol. 15, no. 5-6, pp. 563–583, 2013.

[19] V. Schuppan, "Towards a notion of unsatisfiable and unrealizable cores for ltl," *Science of Computer Programming*, vol. 77, no. 7-8, pp. 908–939, 2012.

[20] 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.

[21] M. Godbolt, *Compiler explorer*, https://godbolt.org/, 2023.

[22] S. Maoz and R. Shalom, "Unrealizable cores for reactive systems specifications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 25–36.

[23] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, IEEE, 2011, pp. 43–50.

[24] R. Alur, S. Moarref, and U. Topcu, "Pattern-based refinement of assume-guarantee specifications in reactive synthesis," in *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, Springer, 2015, pp. 501–516.

[25] B. C. P. L. Lab. "Tslmt: Benchmarks." archived by Jan 28, 2023, Barnard College. (2022), [Online]. Available: https://github.com/Barnard-PL-Labs/temos/blob/art-eval-pldi22/benchmarks/escalator/bidirectional/bidirectional.tslmt.