# Can Reactive Synthesis and Syntax-Guided Synthesis Be Friends?

Wonhyuk Choi
Columbia University
New York, USA
wonhyuk.choi@columbia.edu

Bernd Finkbeiner
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
finkbeiner@cispa.de

Ruzica Piskac
Yale University
New Haven, USA
ruzica.piskac@yale.edu

Mark Santolucito
Barnard College, Columbia University
New York, USA
msantolu@barnard.edu

## Abstract

While reactive synthesis and syntax-guided synthesis (SyGuS) have seen enormous progress in recent years, combining the two approaches has remained a challenge. In this work, we present the synthesis of reactive programs from Temporal Stream Logic modulo theories (TSL-MT), a framework that unites the two approaches to synthesize a single program. In our approach, reactive synthesis and SyGuS *collaborate* in the synthesis process, and generate executable code that implements both reactive and data-level properties.

We present a tool, `temos`, that combines state-of-the-art methods in reactive synthesis and SyGuS to synthesize programs from TSL-MT specifications. We demonstrate the applicability of our approach over a set of benchmarks, and present a deep case study on synthesizing a music keyboard synthesizer.

***Keywords:*** Reactive Synthesis, Syntax-Guided Synthesis, Program Synthesis

## 1 Introduction

*Reactive synthesis*, in the tradition of Church's Problem [13], and deductive synthesis, in its modern form of *syntax-guided synthesis (SyGuS)* [1], have both seen a tremendous revival of interest in recent years. The synthesis of the industrial AMBA AHB bus protocol [7] as well as many other device drivers [38] are considered milestone successes in using the reactive synthesis for industrial software. In parallel, the integration of the Flash Fill feature [20] into Microsoft Excel has also brought to light various SyGuS-based methods.

While both directions of synthesis research have the same basic goal of automatically generating correct programs, their methods and application areas are complementary. In *reactive synthesis*, we are interested in finding a reactive system, such as a hardware circuit, that implements a given temporal specification. If such system exists, we say that it *realizes* the specification. Reactive synthesis focuses on the potentially infinite interaction between the system and its environment, and constructs intricate control strategies that ensure that the system reacts appropriately to any possible move by the environment. Typically, reactive synthesis approaches rely on automata transformations to construct finite-state control strategies that can be represented as Mealy or Moore machines [36]. While reactive synthesis is focused on generating reactive systems, the goal of *SyGuS*-style synthesis techniques is to find a data-transforming function between input and output data. A SyGuS solver takes as input a specification, written in some logical formalism, and a set of input/output examples. The goal is to find a function describing a relation between input/output values. SyGuS methods consider a syntactically restricted search space: the synthesized functions must be expressible as terms of a given grammar. This grammar is also an input parameter of a SyGuS problem. A SyGuS solver finds a solution and uses an SMT solver to verify that the found term correctly realizes the specification.

Ideal application areas for reactive synthesis are *control-dominated* programming problems, such as reactive protocols and circuits [10]; ideal applications of SyGuS are *data-dominated* programming problems, such as spreadsheet functions and other data manipulations. Nevertheless, modern applications do not fall exclusively into either category. Smartphone apps, for example, are reactive in the sense that there is a continuous interplay between the actions of the user (such as clicking on buttons) and actions by the app (such as requesting some information over the network). On the other hand, data processing, such as the organization of the music tracks in a music player app, is equally important for the correct functioning of the app. As a result, such applications are difficult to synthesize with either approach: reactive synthesis reasons about the reactive behavior but not about the data, SyGuS reasons about the data, but not about the reactive behavior.

In this paper, we present a program synthesis technique that combines reactive synthesis and SyGuS, allowing us to automatically generate non-trivial reactive programs, such as

*PL'18, January 01–03, 2018, New York, NY, USA*
2018.

a Linux kernel scheduler. Our method leverages the strengths of both approaches, and is well-suited to construct programs where control and data both play an important role. As the specification language we use an input formalism that is powerful enough to express the specification that includes simultaneously both temporal properties as well as data-transformations. The main idea behind our synthesis technique is to automatically derive additional temporal constraints from the data-transformation based properties of the specification. To do this, we first identify data transformation tasks in the specification, and then, using SyGuS, we generate functions that implement these data transformations. Next, using reactive synthesis we integrate these functions into a control structure that satisfies the reactive requirements.

As the interface between SyGuS and reactive synthesis, we use TSL [19], a variant of temporal logic that is sufficiently powerful to express the requirements on both data transformations and reactive behavior. Temporal Stream Logic (TSL) [19] extends linear-time temporal logic (LTL) with first-order variables. TSL also support first-order functions and predicates. Of particular interest is the *update* predicate $[x \leftarrowtail v]$ that assigns a new value $v$ to some variable $x$, where $v$ can be an arbitrary function term. The introduction of predicate and function terms in a temporal logic makes it possible to capture both reactive properties and data-level manipulations within a common specification. Existing synthesis procedures for TSL [18, 19] are reactive in its nature and as such they do not employ any particular theory reasoning but they treat functions and predicates as *uninterpreted* symbols. Although these synthesis procedures can synthesize a music player app and its basic functionality [19], its specification is highly complex and contains many workarounds as not being able to apply semantics of some predicates and functions is a serious obstacle.

We illustrate the problems that TSL synthesis is facing using the following simple:

$$\square(x = 0 \rightarrow \diamondsuit x = 2)$$
$$\square([x \leftarrowtail x + 1] \vee [x \leftarrowtail x - 1])$$

The system described by this specification must eventually set the counter value to $x = 2$ after the counter reaches $x = 0$. To do this, the system can apply one of two updates in every time step: either incrementing or decrementing the $x$ counter. Conceptually, this specification can be satisfied with a simple system, denoted by $\mathcal{S}$, that always chooses to increment the counter.

However, the above specification is not realizable in TSL logic, because TSL synthesis techniques treat the symbols for addition, subtraction, and equality as uninterpreted. Yet, it is clear to us that the system $\mathcal{S}$ implements the given specification. To synthesize such a system, we need to combine an understanding of temporal properties with an understanding of Linear Integer Arithmetic.

We use this example to outline the basic steps of our synthesis procedure. We first extract predicate and function symbols to form the grammar for the SyGuS problem. By running a SyGuS solver on the problem extracted from the specification, we obtain a sequential program $[x \leftarrowtail x + 1]; [x \leftarrowtail x + 1]$, which increases the value of $x$ from 0 to 2. Next, we analyze this program to add the temporal understanding that this program would take two time steps to execute given the constraints of our system, and augment the original specification with an additional assumption:

$$\psi = \square(x = 0 \wedge [x \leftarrowtail x+1] \wedge \bigcirc[x \leftarrowtail x+1] \rightarrow \bigcirc\bigcirc x = 2),$$

which states that whenever $x$ is 0, its value can be changed to 2 by incrementing $x$ twice. Formula $\psi$ is valid in TSL and it is safe to add it as an *assumption* to the original specification. The resulting formula $\psi \rightarrow \varphi$ is passed to the reactive synthesis algorithm. While in the TSL, "+" is only an uninterpreted function symbol, the assumption formula $\psi$ provides additional constraints that "+" has to satisfy. Those constraints are derived from the semantics of "+". Note that formula $\varphi$ is unrealizable in TSL, but $\psi \rightarrow \varphi$ is realizable.

In this paper we define a general synthesis procedure for TSL *modulo theories* (TSL-MT) [17], which combines reactive synthesis and SyGuS. Existing synthesis methods for TSL [19] cannot handle specifications in TSL-MT because TSL assumes functions and predicates terms are uninterpreted. This means TSL requires the specification to be satisfied for all possible interpretations of the symbols, rather than only by the interpretations admitted by the specific theories. In general, we must therefore *weaken* the TSL formula with additional assumptions that are valid in the theories under consideration but do not hold in general for uninterpreted symbols.

Our synthesis procedure takes as input a TSL-MT formula and outputs an executable program code. The synthesis procedure supports any theory extension that is also supported by the underlying SyGuS solver.

We have empirically tested our approach by implementing a tool, `temos`, and evaluated it on a set of benchmarks for reactive systems. We also present a case study where we build an online tool that allows users to synthesize and then run `JavaScript` code from a TSL-MT specification.

In summary, our contributions are as follows:

1. We introduce the synthesis problem of TSL modulo theories (TSL-MT) as a way of synthesizing reactive systems over first-order theories.
2. We present a method for the extraction of data transformation tasks from TSL-MT specifications, and for the translation of programs that implement such tasks into valid TSL-MT assumptions.
3. We demonstrate the practical applications of the approach by synthesizing executable code for a Linux process scheduler as well as a music keyboard synthesizer.

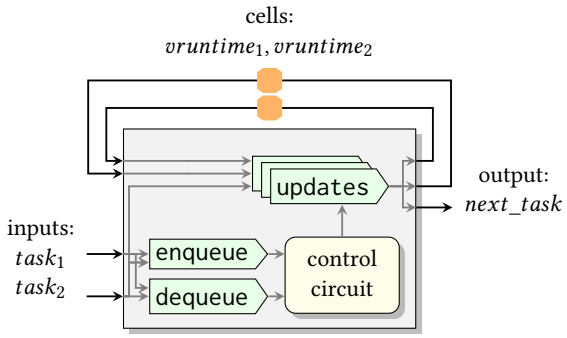



**Figure 1.** The process scheduler uses predicate evaluations to control which function will be applied to generate the updated values in each time step.

## 2 Motivating Example

One of the most complicated tasks that we have synthesized is the Linux *Completely Fair Scheduler* (CFS) [31, 40]. We use it as an illustrative example to show how TSL-MT can be used to specify behavior with both temporal guarantees and data-transformation guarantees. In Linux, CFS is the default scheduler for most user processes, with its policy `SCHED_NORMAL`. Whenever CFS needs to schedule a new task, it does so by choosing the task with the lowest virtual runtime (vruntime), a weighted value of how long each task has used the CPU. Diagrammatically, this can be modeled as a reactive system as shown in Fig. 1.

For better readability, the scheduler depicted in Fig. 1 is a simplified version with only two processes. The scheduler takes as input two signals, $task_1$ and $task_2$, and outputs the signal $next\_task$ which is the next task to be scheduled. A signal in TSL plays the same role as a variable in the first-order logic: signals are arguments of function and predicate expressions. In this particular case, each signal is actually an integer variable corresponding to the process ID generated by the kernel. Internal signals, so called "cells", are internal variables. They can be updated, and their updated values are used in further computations. This way cells are both: input and output signals.

A simplified specification for CFS with two processes could be written as shown in Fig. 2.

Here, enqueue and dequeue are environmental predicates, and weight is a positive integer that the user (the environment) can modify. The weak until operator, $\varphi \mathcal{W} \psi$, states that either $\varphi$ holds until $\psi$ happens, or $\varphi$ holds always. In our particular example, the specification states that if a task is dequeued, it cannot be scheduled until it is again enqueued. The first four formulas define the usual properties of a scheduler that enqueues and dequeues tasks, with liveness guarantees for enqueued tasks. More interestingly, the last four formulas define the fairness of the scheduler. They specify that scheduling a task updates its vruntime with respect to its

$$\square([next\_task \leftarrowtail task_1] \vee [next\_task \leftarrowtail task_2] \vee [next\_task \leftarrowtail idle])$$
$$\square(\text{enqueue } task_1 \rightarrow (\diamondsuit[next\_task \leftarrowtail task_1]) \vee \diamondsuit \text{dequeue } task_1)$$
$$\square(\text{enqueue } task_2 \rightarrow (\diamondsuit[next\_task \leftarrowtail task_2]) \vee \diamondsuit \text{dequeue } task_2)$$
$$\square(\text{dequeue } task_1 \rightarrow (\neg[next\_task \leftarrowtail task_1]) \mathcal{W} \text{enqueue } task_1)$$
$$\square(\text{dequeue } task_2 \rightarrow (\neg[next\_task \leftarrowtail task_2]) \mathcal{W} \text{enqueue } task_2)$$
$$\square([next\_task \leftarrowtail task_1] \leftrightarrow [vruntime_1 \leftarrowtail vruntime_1 + weight_1])$$
$$\square([next\_task \leftarrowtail task_2] \leftrightarrow [vruntime_2 \leftarrowtail vruntime_2 + weight_2])$$
$$\square(vruntime_1 < vruntime_2 \rightarrow \neg[next\_task \leftarrowtail task_2])$$
$$\square(vruntime_2 < vruntime_1 \rightarrow \neg[next\_task \leftarrowtail task_1])$$

**Figure 2.** The TSL-MT specification for a simplified Linux CFS.

weight (assumed to be positive), and that the system should always choose the task with the lowest vruntime to run next.

Even though the specification is just a simplified version of a real scheduler, it contains components of both reactive control and data manipulation: the system must *react* to the environment by applying *data transformations* to virtual runtimes. Standard synthesis methods based on temporal logic cannot synthesize this system, because it requires knowledge of how data transformations alter its state. For instance, the synthesis algorithm must be aware that it is possible to satisfy the liveness guarantee $\diamondsuit[next\_task \leftarrowtail task_1]$ by ensuring that $vruntime_1 < vruntime_2$ does not hold forever: incrementing $vruntime_1$ at each timestep will eventually make it greater than $vruntime_2$. In other words, the synthesis must infer the following data properties:

$$\square(vruntime_1 < vruntime_2 \wedge ([vruntime_1 \leftarrowtail vruntime_1 + weight_1] \wedge [vruntime_2 \leftarrowtail vruntime_2] \mathcal{W} vruntime2 < vruntime_1) \rightarrow \diamondsuit vruntime2 < vruntime_1)$$

$$\square(vruntime_2 < vruntime_1 \wedge ([vruntime_2 \leftarrowtail vruntime_2 + weight_2] \wedge [vruntime_1 \leftarrowtail vruntime_1] \mathcal{W} vruntime1 < vruntime_2) \rightarrow \diamondsuit vruntime1 < vruntime_2)$$

In the following sections, we present a method of generating these reactive data transformation specifications by

dividing up the synthesis task into two parts: a *reactive synthesis* component, and a *syntax-guided synthesis* component. We then allow the synthesis tasks to collaborate, and ultimately produce a program.

# 3 Preliminaries

## 3.1 Temporal Stream Logic

*Temporal Stream Logic (TSL)* [19] extends linear-time temporal logic (LTL) [35] with uninterpreted predicates and functions. TSL is a specialized logic for reactive synthesis, specifically for the synthesis of functional reactive programs [16]. With TSL, one can specify a reactive system that reacts to an infinite stream of inputs to produce an infinite stream of outputs. Based on such a specification, TSL synthesis produces a reactive program in a high-level language such as `Haskell`.

TSL is based on the usual LTL operators next $\bigcirc$ and until $\mathcal{U}$. Additionally, the syntax of TSL contains *predicate terms* $\tau_P$, *function terms* $\tau_F$, and *update terms* $\tau_U$, as defined in the following grammar:

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

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

We also use the standard derived operators, such as *release* $\varphi\,\mathcal{R}\,\psi \equiv \neg(\neg\varphi\,\mathcal{U}\,\neg\psi)$, *always* $\Box\varphi \equiv \bot\,\mathcal{R}\,\varphi$, *eventually* $\Diamond\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$, and *weak until* $\varphi\,\mathcal{W}\,\psi \equiv (\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$. All TSL formulas can be transformed into a *negation normal form*, where negations only appear in front of predicate and update terms, and all operators are in the set $\{\wedge, \vee, \bigcirc, \mathcal{U}, \mathcal{R}\}$.

TSL describes the behavior of a reactive system in terms of signals, denoted with $\mathsf{s}$, which carry data values of arbitrary type. A TSL specification describes how the functions may be applied to these signals over time. Signals can be pure outputs or *cells* that memorize data values such that the outputs of time $t$ is available as a input for time $t+1$. These properties define the semantics of TSL, which follow the usual LTL semantics with the addition of predicate evaluations, function evaluations, and update terms. A formal definition of the TSL semantics is available in [19].

The realizability problem of TSL is stated as: given a TSL formula $\varphi$, is there a strategy $\sigma \in \mathcal{I}^+ \to \mathcal{O}$ mapping a finite input steam to an output, 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 $\varphi$, i.e.,

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

If such a strategy $\sigma$ exists, $\sigma$ realizes $\varphi$. The synthesis problem of TSL asks for a concrete instantiation of $\sigma$. In TSL synthesis, this model $\sigma$ can be abstracted as a Control Flow Model (CFM), which can then be implemented as program code. A formal definition of the TSL realizability and synthesis is available in [19].

## 3.2 First-Order Theories

A First-Order Theory $\mathcal{T}$ is a class of models over First-Order Logic (FOL) with the some signature $\Sigma$ [5]. More precisely, we can define a first-order theory as a tuple $(\Sigma_F, \Sigma_P, \mathcal{A})$ where:

- $\Sigma_F$ a set of function symbols, $\Sigma_P$ a set of predicate symbols. The set $\Sigma = \Sigma_F \cup \Sigma_P$ is known as a *signature*.
- $\mathcal{A}$ a set of closed FOL formulae over $\Sigma_F$, $\Sigma_P$, and $\mathcal{V}$ a set of variables. Then $\mathcal{A}$ is called the axioms of $\mathcal{T}$.

Solving for constraint satisfiability of these first-order theory formulas is known as the *Satisfiability Modulo Theories* (SMT) [5] problem, which has seen large progress in the previous two decades.

## 3.3 TSL Modulo Theories (TSL-MT)

TSL-MT [17] generalizes TSL with first-order theories. Given a First-Order Theory $\mathcal{T} = (\Sigma_F, \Sigma_P, \mathcal{A})$, a TSL-MT formula $\varphi_\mathcal{T}$ is constructed as follows:

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

$$\begin{aligned} \mathsf{f} &\in \Sigma_F \\ \tau_F &:= \mathsf{s} \;\mid\; \mathsf{f}(\tau_F^0, \tau_F^1, \ldots, \tau_F^{n-1}) \\ \mathsf{p} &\in \Sigma_p \\ \tau_P &:= \mathsf{p}(\tau_F^0, \tau_F^1, \ldots, \tau_F^{n-1}) \\ \tau_U &:= [\mathsf{s} \leftarrowtail \tau_F] \end{aligned}$$

TSL-MT specifications can refer to any arbitrary first-order theory, such as the theory of Linear Integer Arithmetic (LIA) or the Theory of Arrays. TSL is the special case of TSL-MT where all symbols are from the Theory of Uninterpreted Functions.

## 3.4 Syntax-Guided Synthesis

Syntax-Guided Synthesis (SyGuS) [1] is a framework for program synthesis that applies both semantic and syntactic restrictions on the possible space of solutions. A SyGuS problem is defined in two parts: a second-order formula $\exists f.\, \forall x.\, \varphi[f, x]$ over some first-order theory $\mathcal{T}$, and a context-free grammar $\mathcal{R}$ that defines the possible syntactic forms of $f$. A solution $f$ for the SyGuS problem then is a lambda term $\lambda x.e$ of the same type as $f$ such that $\forall x.\, \varphi[\lambda x.\, e, x]$ is valid in $\mathcal{T}$ and where $e$ can be generated by the grammar $\mathcal{R}$.

# 4 Synthesis from TSL-MT Specifications

The synthesis problem of TSL-MT is to generate a model that satisfies a TSL-MT formula. This is a generalization of the synthesis problem of TSL. In TSL synthesis, since all functions and predicates are uninterpreted, we search for a

strategy $\sigma$ that satisfies the specification $\varphi$ over all possible function implementations $\langle\cdot\rangle$ (equation (1)). However, in TSL-MT, we search for a strategy $\sigma$ that satisfies the specification $\varphi_{\mathcal{T}}$ with function and predicate implementations as defined by the user in the selected first order theory $\mathcal{T}$ (equation (2)):

$$\exists \sigma \in \mathcal{I}^+ \to \mathcal{O}.\ \forall \iota \in \mathcal{I}^\omega.\ \forall \langle\cdot\rangle : \mathbb{F} \to \mathcal{F}.\ \sigma \wr \iota, \iota \models_{\langle\cdot\rangle} \varphi \quad (1)$$

$$\exists \sigma \in \mathcal{I}^+ \to \mathcal{O}.\ \forall \iota \in \mathcal{I}^\omega.\ \sigma \wr \iota, \iota \models_{\langle\cdot\rangle} \varphi_{\mathcal{T}} \quad (2)$$

The synthesis problem of TSL is then a sub-problem of the synthesis problem of TSL-MT: TSL is equivalent to TSL-MT with the Theory of Uninterpreted Functions being the selected first order theory $\mathcal{T}$. A consequence of this relation is that since TSL synthesis is undecidable [19], the general problem of TSL-MT synthesis is also undecidable and hence incomplete.

In order to synthesize a program from a TSL-MT specification, we first decompose the formula into its corresponding TSL formula, the set of its predicate literals, and the set of data transformation obligations. On the predicate literals, we perform a consistency checking step to produce assumptions that the environment does not produce invalid inputs. With our data transformation obligations, we perform our main task, using SyGuS to implement data transformation programs. We combine these programs with the obligations and encode them as assumptions in TSL. The resulting (weakened) TSL formula is fed into reactive synthesis. If the TSL formula is realizable, we synthesize a program; otherwise, we refine by returning to SyGuS for further assumptions.

Figure 3 shows an overview of the procedure. In the following, we describe each part of the process in detail.

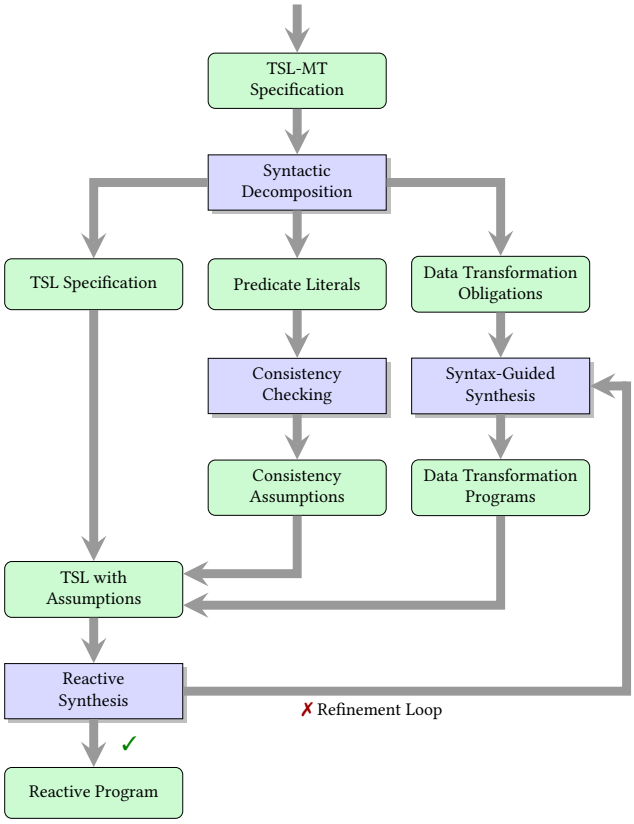



**Figure 3.** Overview of the Synthesis Procedure

### 4.1 Syntactic Decomposition

We begin by decomposing the TSL-MT specification into a TSL specification, predicate literals, and data transformation obligations. We obtain the predicate literals by parsing the TSL-MT formula, and produce the TSL specification by removing the semantics of function and predicate terms from TSL-MT.

To generate the necessary data transformations, we frame the problem as enumerating data transformation *obligations*, analogous to the *proof obligations* of a Hoare Triple [22]. From some predicate evaluation state, the *pre-condition*, we define a future predicate evaluation we want to reach, the *post-condition*. Then, unlike a Hoare Triple, which has a predefined *command*, we can frame our obligation as a program synthesis problem searching for the data transformation program $\mathcal{S}$ that satisfies the pre- and post-conditions:

| Pre-condition | Program | Post-condition |
|---|---|---|
| $p_{pre}(s_1 \cdots s_m)$ | $\mathcal{S}$ | $(\bigcirc^n \mid \Diamond)\ p_{post}(s_1 \cdots s_m)$ |

To generate the obligations, we choose the pre-condition and post-conditions from the predicate literals in the specification. However, since the specification states that the post-condition must hold in the future, we need to find the temporal operator associated with the post-condition. In order to obtain the set of temporal operators assigned to a literal, we first transform the original specification to negation normal form (NNF).

Then, for each predicate literal in the NNF, we group it into its *temporal atom*, the formula connected to the literal by only logical connectives. We traverse up the AST towards the root in order to find the temporal operators associated with this temporal atom. If the traversal encounters a $\bigcirc$ operator, we simply increment the count of $\bigcirc$ operators we have seen thus far and add a postcondition of the form $\bigcirc^{\text{count}}$ p into the list of our post-conditions, where p is the predicate literal.

If we traverse upwards and find a $\mathcal{U}$ operator associated with the temporal atom, we can divide it two cases, whether the atom is on the left or the right of the operator. If it is on the right-hand side, by the semantics of the $\mathcal{U}$ operator, we know that a satisfying model must be able to produce $\Diamond$ p, where p. Similarly, if the temporal atom is on the left hand side of the $\mathcal{U}$ operator, then the model will need p $\to \bigcirc$ p as long as the right-hand side of $\mathcal{U}$ operator is not true. At this point, since $\Diamond\Diamond$ p $\equiv \Diamond\bigcirc$ p $\equiv \Diamond$ p, we can exit the loop without reaching the root of the AST since further traversals will not add complexity to the temporal clauses associated with the predicate literal.

After we obtain the pre- and post-conditions, we generate a powerset to obtain all possible boolean combinations. Since temporal operators are associated with each postcondition, the *maximum time* (i.e. a $\Diamond$, or if none exists, the maximum number of $\bigcirc$ operators) of a postcondition is assigned as its temporal operator. With these pre- and post-conditions, decomposition can generate all necessary data transformation obligations.

A formalized version of the procedure is available in Algorithm 1.

### 4.2 Consistency Checking

In a reactive system, the environment may produce any arbitrary data value within the domain as inputs. In a TSL-MT specification, users write *predicate terms* on these inputs to specify how the system should react to the environment.

However, not all predicate terms are satisfiable. Given a background theory $\mathcal{T}$, certain predicates or their conjunctions may be unsatisfiable; the environment may be unable to produce any values (models) that satisfy the predicate term. For instance, consider the following TSL-MT formula:

$$\square(x < y \rightarrow [mutex \leftarrowtail x]) \wedge$$
$$\square(y < x \rightarrow [mutex \leftarrowtail y])$$

Intuitively, this specification states that *mutex* is updated with $\texttt{min}(x, y)$. Even though the specification is realizable in TSL-MT, reducing it to TSL without the semantics of $<$ renders it unrealizable. Since reactive synthesis has no knowledge of arithmetic, it assumes the environment can produce values of $x$ and $y$ that simultaneously satisfy $x < y$ and $y < x$. As the system cannot update *mutex* with two values at the same timestep, synthesis cannot produce a model.

However, in a standard interpretation of $<$ (e.g., Real Arithmetic), $(x < y) \wedge (y < x)$ is unsatisfiable. Therefore, the environment can never produce values of $x$ and $y$ that satisfy the predicate terms simultaneously, and we need to add the *consistency assumption* $\square\neg(x < y \wedge y < x)$ to TSL. With this assumption, reactive synthesis is able to produce a model.

In general, we can generate these consistency assumptions by enumerating the powerset of all predicate literals in the specification. Then, we check the satisfiability of these predicates with an SMT solver, and add assumptions of the form $\square\neg p$ for all unsatisfiable $p$. In the worst case, this produces $O(2^n)$ satisfiability queries. While exponential, in practice this process can be done quickly, as modern SMT solvers are capable of solving queries with millions of constraints.

### 4.3 Syntax-Guided Synthesis

After we identify the data transformation obligations from the TSL-MT specification, we can apply syntax-guided synthesis to generate functions that implement the transformations. Depending on the temporal operator associated with the post-condition, we have two different kinds of grammars for SyGuS: one for *sequential programs* with fixed number of executions, and one for *looping programs* with arbitrary long length. We discuss the context-free grammar for each kind of program, and explain the encoding to a TSL formula.

**Algorithm 1:** Syntactic Decomposition

**Input:** TSL-MT Specification
**Output:** Data Transformation Obligations

```
1  predLiterals = all predicate literals in the TSL-MT formula;
2  preconditions = predLiterals;
3  postconditions = {};
4  dtoList = {};
   /* Traverse the AST backwards to collect temporal operators */
5  foreach p in predLiterals do
6      numNext = 0;
7      temporalAtom = GET-TEMPORAL-ATOM(p);
8      while TRUE do
9          parent = PARENT-AST-NODE(temporalAtom);
           /* Reached root of AST */
10         if parent == ∅ then
11             break;
           /* Reached 𝒰 */
12         else if parent.temporalOperator == 𝒰 then
13             if parent.Right == temporalAtom then
14                 postconditions.append(◇p);
15             else
16                 postconditions.append(○p);
17             end
18             break;
           /* Reached ○ */
19         else
20             ++numNext;
21             postconditions.append(○^numNext p);
22             temporalAtom = parent;
23         end
24     end
25 end
   /* Form Data Transformation Obligations with the pre- and post-conditions */
26 foreach p_pre in POWERSET(preconditions) do
27     foreach p_post in POWERSET(postconditions) do
28         dtoList.append(p_pre, p_post);
29     end
30 end
31 return dtoList;
```

#### 4.3.1 Sequential Programs.

We first examine synthesis of *sequential programs* for data transformation obligations with explicitly defined number of timesteps. The obligation

contains two parts: the pre-condition $p_{pre}$, and the post-condition $\bigcirc^n p_{post}$. The predicate terms $p_{pre}$ and $p_{post}$ define the semantic constraints for our SyGuS problem. For the syntactic constraint, we consider the case of building the grammar for a function $f_i$ that transforms some signal term $s_i$ in the obligation. In order to construct the grammar, we find the set of functions and signals $\mathcal{F}$ that can update the value of $s_i$. We can then construct our context-free grammar for the function $f_i(s_1 \cdots s_m)$ as follows:

$$\mathcal{S} ::= \mathcal{F}\ \mathcal{S} \mid s_i$$

We can build a SyGuS query from the semantic and syntactic constraints, and send it to a SyGuS solver. The solver will produce a function that satisfies the constraints. However, it still remains to convert this into a meaningful form for reactive synthesis. Therefore, we translate the resulting function into a TSL assumption by "unrolling" each node of the AST. We traverse the AST from the leaves up to the root, transforming each node of the AST into an update term. Since TSL can apply functions over multiple timesteps, we prefix a number of temporal next operators $\bigcirc$ to the update terms that corresponds to its distance from the leaves. This procedure results in a chain of update terms that we call $upd$:

$$[s_1 \leftarrow f_{10}(s_1 \cdots s_m)] \wedge \cdots [s_m \leftarrow f_{m0}(s_1 \cdots s_m)] \wedge$$
$$\bigcirc([s_1 \leftarrow f_{11}(s_1 \cdots s_m)] \wedge \cdots [s_m \leftarrow f_{m1}(s_1 \cdots s_m)]) \wedge \cdots$$
$$\bigcirc^{n-1}([s_1 \leftarrow f_{1n}(s_1 \cdots s_m)] \wedge \cdots [s_m \leftarrow f_{mn}(s_1 \cdots s_m)])$$

Here, $f_{ij}$ is the update term for the $i$th signal at time $j$.

With our TSL encoding of the SyGuS function, we then combine it with the pre- and post-conditions of the data transformation obligation to produce a TSL assumption

$$\square(p_{pre} \wedge upd \rightarrow \bigcirc^n p_{post})$$

We formalize the procedure in Algorithm 2, and prove its correctness in Theorem 4.1.

**Algorithm 2:** Sequential Programs to TSL

**Input:** Function AST, $p_{pre}$, $p_{post}$
**Output:** TSL Assumption

```
assumption = "□ p_pre";
timeSteps = 0;
astNode = leaf;
while astNode != root do
    assumption += " ∧ ○^timeSteps astNode";
    astNode = astNode.parent;
    ++timeSteps;
end
assumption += " → ○^timeSteps p_post";
return assumption
```

**Theorem 4.1** (Soundness of SyGuS-TSL Translation for Sequential Programs)**.** *Let $\varphi_{\text{TSL-MT}}$ be a TSL-MT specification and $\varphi_{TSL}$ the TSL specification with assumptions generated from the SyGuS-TSL translation procedure via Algorithm 2. Then, for all models $M_{TSL} \models \varphi_{TSL}$, there exists $M_{\text{TSL-MT}}$ such that $M_{\text{TSL-MT}} \models \varphi_{\text{TSL-MT}}$.*

*Proof.* Consider the generated assumption:

$$\square(p_{pre} \wedge upd \rightarrow \bigcirc^{n+1} p_{post})$$

where $p_{pre}$ is some predicate literal $p_1(s_1 \ldots s_m)$ and $p_{post}$ also some predicate literal $p_2(s_1 \ldots s_m)$. Then using the computation function from [19], we can define the predicate evaluation of $p_{post}$ at time $t + n + 1$ as follows:

$$\eta(\varsigma, \iota, t + n + 1, p_2(s_1 \cdots s_n)) =$$
$$\langle p_2 \rangle \eta(\varsigma, \iota, t + n + 1, s_1) \cdots \eta(\varsigma, \iota, t + n + 1, s_n)$$

Without loss of generality, let us choose an arbitrary signal $s_i$, and consider its evaluation at time $t + n + 1$:

$$\eta(\varsigma, \iota, t + n + 1, s_i) =$$
$$\eta(\varsigma, \iota, t + n, \varsigma(t + n)(s_i)) =$$
$$\eta(\varsigma, \iota, t + n - 1, \varsigma(t + n)(\varsigma(t + n - 1)(s_i))) =$$
$$\cdots$$
$$\eta(\varsigma, \iota, t, \varsigma(t + n)(\varsigma(t + n - 1)(\cdots \varsigma(t)(s_i))))$$

Therefore, the evaluation of the term $s_i$ at time $t + n + 1$ can be defined as the chain of computations $\varsigma(t) \cdots \varsigma(t + n)$ on the term $s_1$ at time $t$.

Now, we consider that $upd$ was the chain of update terms, $u_{10} \wedge \cdots u_{m0} \wedge \bigcirc(u_{11} \wedge \cdots u_{m1}) \cdots \bigcirc^n(u_{1n} \wedge \cdots u_{mn})$, generated by the step-by-step AST deconstruction from the function derived by the following program synthesis query:

$$\exists f_1 \cdots f_m. \forall s_1 \cdots s_m \models p_{pre}.$$
$$f_1(s_1 \cdots s_m) \cdots f_m(s_1 \cdots s_m) \models p_{post}$$

Since each step of the computation on signal $s_i$ at any time $j$ corresponds to the update term $u_{ij}$, we obtain that the value of signal $s_i$ at time $t + n + 1$ is equivalent to the value $f_i(s_1 \cdots s_m)$ for the function $f_i$ synthesized by our SyGuS query. This holds for all signals $s_i$, and since we have $f_1(s_1 \cdots s_m) \cdots f_m(s_1 \cdots s_m) \models p_{post}$, this implies that at time $t + n + 1$, $s_1 \cdots s_m \models p_{post}$. This is exactly the semantics of $\bigcirc^{n+1} p_{post}$, and this concludes the proof. □

Note that applying this process naïvely is insufficient to satisfy the data transformation obligation. Apart from the semantic and syntactic constraints, we have a third constraint, the *temporal constraint*, that the post-condition must hold after exactly $n$ timesteps. However, since we translate each node of the AST into an update term, we can easily enforce this constraint by restricting the height of the AST produced by SyGuS. When we send our query to SyGuS, we stop its search depth at depth $n$, and only consider the solutions that have height $n$. This means that our search for sequential programs is always bound by two factors: its syntax and its AST height, making our procedure computationally inexpensive.

We now present two examples to demonstrate the process.

**Example 4.2.** Consider the following specification TSL Modulo Linear Integer Arithmetic (LIA):

$$\square([x \leftarrow x+1] \vee [x \leftarrow x-1]) \wedge$$
$$\square(x = 0 \rightarrow \bigcirc\bigcirc x = 0)$$

From the specification, we can construct a data transformation obligation that takes $x = 0$ as a pre-condition and $\bigcirc\bigcirc x = 0$ as a post-condition. This produces the following program synthesis problem:

$$\exists f.\ f(0) = 0$$

This search is bound in two ways. First, the possible syntax of $f$ is restricted by the update terms that appear in the specification. This constraint produces the following context-free grammar for $f$:

$$\begin{aligned} \mathcal{S} ::= \mathcal{S} + 1 \\ \mid \mathcal{S} - 1 \\ \mid x \end{aligned}$$

We also bound SyGuS with the restriction that the height of the AST must be exactly 2, since our post-condition must occur after exactly two timesteps. We can then run the query with a SyGuS solver, which can produce two possible results:

1. $f(x) = ((x+1)-1)$
2. $f(x) = ((x-1)+1)$

While either of these results are valid, let us assume that $f(x) = ((x+1)-1)$ was returned from the solver. From this SyGuS result, we can encode the function into TSL and combine it with the obligation to produce a TSL assumption

$$\square x = 0 \wedge [x \leftarrow x+1] \wedge \bigcirc[x \leftarrow x-1] \rightarrow \bigcirc\bigcirc x = 0$$

With this environment assumption, the original formula becomes realizable in TSL synthesis.

**Example 4.3.** We consider the synthesis problem of plain TSL formulas. In [19], the authors present a method of synthesizing TSL formulas by underapproximating the specification with Linear Temporal Logic (LTL). This is a sound, but incomplete procedure as the synthesis problem of TSL is undecidable while LTL synthesis is decidable. Therefore, the TSL synthesis procedure adds refinements whenever the approximation insufficiently captures the semantics of update terms in TSL; however, this refinement loop required significant effort to analyze counter-strategies and was never implemented automatically and all refinements were done manually.

However, since plain TSL specifications are just TSL modulo the Theory of Uninterpreted Functions, we can use our SyGuS approach to easily automate the refinement loop. We consider the modified example from [19]:

$$\square([y \leftarrow y] \vee [y \leftarrow x]) \wedge$$
$$\square(p\ x \rightarrow \bigcirc p\ y)$$

A possible model for this formula is to take the value of $x$ whenever $p\ x$ is true and save it $y$. However, if we produce an underapproximation of the TSL specification to LTL

$$\square\ \neg(\texttt{y\_to\_y} \wedge \texttt{x\_to\_y}) \wedge$$
$$\square\ (\texttt{y\_to\_y} \vee \texttt{x\_to\_y}) \wedge$$
$$\square\ (\texttt{p\_x} \rightarrow \bigcirc\ \texttt{p\_y})$$

the formula loses the semantic meaning of update terms and allows the spurious counter-strategy $\square(\texttt{p\_x} \wedge \neg\texttt{p\_y})$. This counter-strategy is spurious because a model can update the value of y by using the update term $[y \leftarrow x]$, and circumvent the possibility that $\texttt{p\_x} \wedge \neg\texttt{p\_y}$ is always true. TSL synthesis produces this counter-strategy because the LTL underapproximation does not know how to complete the data transformation obligation of $p\ x$ to $\bigcirc p\ y$. If we view the problem in our TSL-MT framework, then we can create a SyGuS query with the semantic constraint

$$\exists f.\ \forall x.\ x \models\ p(x).\ f(y) \models p(f(y))$$

which captures our data transformation obligation task that whenever $p(x)$ is true, $p(y)$ should be true in the next time step. For our function $f(y)$, we have the grammar

$$\mathcal{S} ::= x \mid y$$

Sending this to a SyGuS solver will return $f(y) = x$, which we can then translate into a TSL assumption:

$$\square(p\ x \wedge [y \leftarrow x] \rightarrow \bigcirc p\ y)$$

With this assumption, the LTL underapproximation now sufficiently captures the semantics of update terms for realizability.

**4.3.2 Programs With Loops.** It is also possible for a data transformation obligation to have a post-condition with a *reachability* property, with a *eventually* operator $\diamondsuit$ assigned to it. For these obligations, it is still possible that sequential programs can generate solutions, i.e. the example from the introduction, but the grammar may not always be sufficiently expressive in certain cases. Therefore, we introduce the idea of loops, or equivalently, recursion to our grammar. Let $\mathcal{F}_i$ be the update terms available in the specification for signal $s_i$. Then for each function $f_i$ that updates $s_i$, we can construct a context-free grammar for a recursive function $f_i(s_1 \cdots s_m)$:

$$\begin{aligned} \mathcal{S} ::= \texttt{IF}\ \ p_{post}\ \ \texttt{THEN}\ \ s_i \\ \texttt{ELSE}\ \ \mathcal{S} \\ \mid \mathcal{F}_i\ \mathcal{S} \end{aligned}$$

Intuitively, this is the grammar from Section 4.3.1 extended with recursion. Accordingly, translating a synthesized recursive function to TSL requires encoding of the loop property. We achieve this by using the temporal operators *weak until* $\mathcal{W}$ and *eventually* $\diamondsuit$:

$$\square(p_{pre} \wedge (upd\,\mathcal{W}\,p_{post}) \rightarrow \diamondsuit p_{post})$$

Here, *upd* is the "body" of the loop, and $p_{pre}$ and $p_{post}$ are the predicate literals of the pre- and post-condition. Intuitively, this formula describes the property that applying *upd* an unknown number of times will eventually lead to $p_{post}$, which corresponds to the semantics of our looping function.

In Algorithm 3, we formalize the procedure for encoding looping programs to TSL. In Theorem 4.4 we prove the soundness of the translation procedure for transforming the result of the SyGuS solver into a TSL assumption.

**Algorithm 3:** Looping Programs to TSL

**Input:** Function AST, precond, postcond
**Output:** TSL Assumption

assumption = "$\square p_{pre}$";
timeSteps = 0;
astNode = leaf;
**while** *astNode != root* **do**
  assumption += "$\wedge \bigcirc^{\text{timeSteps}}$";
  **if** *astNode.isLoop* **then**
    assumption += astNode.loopBody $\mathcal{W}$ astNode.condition;
  **else**
    assumption += astNode;
  **end**
  astNode = astNode.parent;
  ++timeSteps;
**end**
assumption += "$\rightarrow \diamondsuit p_{post}$";
**return** *assumption*

**Theorem 4.4** (Soundness of SyGuS-TSL Translation for Recursive Programs)**.** *Let $\varphi_{\text{TSL-MT}}$ be a TSL-MT specification and $\varphi_{TSL}$ the TSL specification with assumptions generated from the SyGuS-TSL translation procedure via Algorithm 3. Then, for all models $M_{TSL} \models \varphi_{TSL}$, there exists $M_{\text{TSL-MT}}$ such that $M_{\text{TSL-MT}} \models \varphi_{\text{TSL-MT}}$.*

*Proof.* Consider the TSL assumption that we obtain from the translation procedure:

$$\square(p_{pre} \wedge (upd\, \mathcal{W}\, p_{post}) \rightarrow \diamondsuit p_{post})$$

Intuitively, this statement can be understood as "when we are at predicate evaluation $p_{pre}$, we can apply update terms *upd* some finite amount of times to reach predicate evaluation $p_{post}$". Since $\varphi\, \mathcal{W}\, \psi = (\varphi\, \mathcal{U}\, \psi) \vee (\square \varphi)$, we can have two cases:

1. $p_{pre} \wedge (upd\, \mathcal{U}\, p_{post}) \rightarrow \diamondsuit p_{post}$
2. $p_{pre} \wedge \square upd \rightarrow \diamondsuit p_{post}$

Since $\diamondsuit \varphi = \top\, \mathcal{U}\, \varphi$, the first case is trivial, and we are only interested in the second case.

Now, consider our program synthesis query:

$$\exists f_1 \cdots f_m. \forall s_1 \cdots s_m \models p_{pre}.$$
$$f_1(s_1 \cdots s_m) \cdots f_m(s_1 \cdots s_m) \models p_{post}$$

For all $i$, this results in the following function:

$$f(s_1 \cdots s_n) = \texttt{IF}\ \ p_{post}\ \ \texttt{THEN}\ \ s_i$$
$$\texttt{ELSE}\ \ f_1(upd_1(s_1 \cdots s_m) \cdots upd_m(s_1 \cdots s_m))$$

As this function was a solution to our SyGuS query, this is correct by construction, with respect to the SyGuS solver. In particular, it shows that 1) the recursive function terminates and 2) that when it terminates, the resulting value from the functions satisfy $p_{post}$. These combined show that $\square(p_{pre} \wedge upd\, \mathcal{W}\, p_{post} \rightarrow \diamondsuit p_{post})$ is a valid assumption, concluding the proof. □

Note that while we require synthesis of recursive functions, most SyGuS solvers do not yet support this feature. In our experimental evaluation in Section 5, we built a wrapper around CVC4 [6] to synthesize recursive functions (cf. Section 5.1).

**Example 4.5.** Consider the following specification:

$$\square[x \leftarrow x + 1] \vee [x \leftarrow x - 1] \wedge$$
$$\square 0 < x \rightarrow \diamondsuit x = 0$$

Sequential programs alone cannot solve the obligation with pre-condition $x < 0$ and post-condition $\diamondsuit x = 0$, as SyGuS will produce a different each AST for each value of $x$. Therefore, we introduce a grammar with recursion:

$$\begin{aligned} \mathcal{S} ::= &\ \texttt{IF}\ \ x = 0\ \ \texttt{THEN}\ \ x \\ &\ \texttt{ELSE}\ \ \mathcal{S} \\ \mid &\ \mathcal{F} \\ \mathcal{F} ::= &\ \mathcal{F} + 1 \\ \mid &\ \mathcal{F} - 1 \\ \mid &\ x \end{aligned}$$

A SyGuS solver can then produce the solution

$$f(x) = \texttt{IF}\ \ x = 0\ \ \texttt{THEN}\ \ 0\ \ \texttt{ELSE}\ \ f(x + 1)$$

Now, we can encode the function into TSL and combine it with the obligation to produce the TSL assumption

$$\square(x < 0 \wedge ([x \leftarrow x + 1]\, \mathcal{W}\, x = 0) \rightarrow \diamondsuit x = 0)$$

### 4.4 Reactive Synthesis

After we obtain assumptions from consistency checking and syntax-guided synthesis, we combine them with the TSL specification to run reactive synthesis. In many cases, syntax-guided synthesis will provide reactive synthesis with sufficient knowledge to synthesize our goal, a program that reasons both about control and data.

However, in certain cases, it is still possible for TSL synthesis to return *unrealizable* even though the original TSL-MT formula is realizable. This happens because when we find

the chain of update terms $upd$ satisfying the obligation, restrictions from the reactive control may not allow it to occur when the pre-condition $p_{pre}$ is true. This causes the assumption of the form $p_{pre} \wedge upd \rightarrow \Diamond p_{post}$ or $p_{pre} \wedge upd \rightarrow \bigcirc^n p_{post}$ to be "unhelpful", since the system has no way to "execute" these data transformations.

To resolve this issue, we present a refinement loop that generates different programs for the data transformation obligation. We first find which assumption is "unhelpful" by translating the assumption into a guarantee of the form $\square(p_{pre} \rightarrow upd)$. If adding the guarantee makes the specification unsatisfiable, this implies that the system can never update values with $upd$ whenever $p_{pre}$ is true. Therefore, we need to find a different chain of update terms $upd$ for the given obligation. We re-execute SyGuS synthesis for the same pre-condition and post-condition pair and force it produce a different function from our previous result. This result can then also be encoded into a TSL assumption, with which we can rerun synthesis.

The refinement loop is formally presented in Algorithm 4.

**Algorithm 4:** Reactive Synthesis and Refinement Loop

```
Input: formula, the TSL specification with
       assumptions
while TRUE do
    isRealizable, model =
     REACTIVE-SYNTHESIS(formula);
    if isRealizable then
        return model;
    end
    foreach assumption in ASSUMPTIONS(formula) do
        p_pre, upd, p_post = assumption;
        guarantee = □(p_pre → upd);
        isSAT = CHECK-SAT(guarantee + formula);
        /* Found the "unhelpful" assumption */
        if ! isSAT then
            newAssumption = SyGuS(p_pre, p_post,
              notValidSolution=upd);
            formula = formula + newAssumption;
            break;
        end
    end
end
```

**Example 4.6.** Consider the following TSL-MT specification in Linear Integer Arithmetic:

$$\square[x \leftarrow x+1] \rightarrow \bigcirc[x \leftarrow x] \wedge$$
$$\square x = 0 \rightarrow \Diamond x = 2$$

With $x = 0$ as the pre-condition and $\Diamond x = 2$ as the post-condition, our process will produce the following assumption:

$$\square(x = 0 \wedge [x \leftarrow x+1] \wedge \bigcirc[x \leftarrow x+1] \rightarrow \bigcirc^2 x = 2)$$

However, due to the specification $[x \leftarrow x+1] \rightarrow \bigcirc[x \leftarrow x]$, the system cannot "follow" the data transformation program to go from $x = 0$ to $x = 2$. We can check this by considering the satisfiability of the specification

$$\square x = 0 \rightarrow [x \leftarrow x+1] \wedge \bigcirc[x \leftarrow x+1] \wedge$$
$$\square[x \leftarrow x+1] \rightarrow \bigcirc[x \leftarrow x] \wedge$$
$$\square x = 0 \rightarrow \Diamond x = 2$$

which produces UNSAT. This implies that we need another function for $\exists f. \forall x.\ x = 2.\ f(x) = 2$. We rerun synthesis for the data transformation obligation, ignore the result $f(x) = ((x+1)+1)$, and generate the assumption

$$\square x = 0 \wedge [x \leftarrow x+1] \wedge$$
$$\bigcirc[x \leftarrow x] \wedge \bigcirc^2[x \leftarrow x+1] \rightarrow \bigcirc^3 x = 2$$

Adding this assumption and running TSL synthesis returns realizable, completing the refinement.

### 4.5 Limitations

While our approach to TSL-MT synthesis is sound (cf. Theorems 4.1 and 4.4), it suffers from two sources of undecidability: the undecidability of TSL synthesis [19] and the undecidability of syntax-guided synthesis [8]. This renders any approach to TSL-MT synthesis undecidable and hence incomplete. Moreover, while there is recent work in proving unrealizability for SyGuS [24], currently most exisiting SyGuS solvers do not halt on unrealizable inputs, meaning that our approach of combining reactive synthesis and SyGuS solvers will generally fail to return unrealizability results. There are some decidable fragments of TSL-MT (e.g. a empty first-order theory without any functions or predicates would reduce TSL-MT to Linear Temporal Logic) that could have complete synthesis procedures; further research is needed to explore these fragments.

A third source of undecidability stems from the grammars used in our encoding of TSL-MT to SyGuS. As an example, consider an example where the model's behavior depends on uncontrollable inputs. We modify Example 4.5 by adding an additional constraint that updating $x$ must depend on $y$, an environment variable that is true infinitely often:

$$\square(!y \rightarrow [x \leftarrow x]) \wedge \square\Diamond y$$

In this case, the correct function to be synthesized would be

$$f(x) = \texttt{IF } x = 0 \texttt{ THEN } 0 \texttt{ ELSE (IF } y \texttt{ THEN } [x \leftarrow x] \texttt{ ELSE } x)$$

However, this function is not synthesizable according our SyGuS grammar as defined in Section 4.3.2. This is due to a gap in our grammar where we do not allow nested conditionals.

Future work is necessary to explore relative completeness – assuming we have a complete oracle for TSL and a complete oracle for SyGuS synthesis, can we construct a complete synthesis procedure for TSL-MT?

# 5 Evaluation

## 5.1 Implementation

We implemented a tool, `temos` for synthesizing TSL-MT specifications. The tool is written in `Rust`, on top of CVC4 [6] as our SyGuS backend and `tsltools` [18] and `Strix` [34] for TSL synthesis.

We use CVC4 for the SyGuS synthesis subproblems, as it achieves state-of-the-art performance for most SyGuS benchmarks [2]. However, as most SyGuS solvers do not yet support synthesis of recursive functions, we added a wrapper around CVC4 to synthesize looping functions. Whenever we have a universally quantified pre-condition, we instantiate a few models that satisfy the condition. We can then set up independent data transformation obligations with these models as pre-conditions, and synthesize them as sequential programs. This produces different functions and AST's for each pre-condition value. For each AST, we try to find repeated fragments in the tree and compare each to find the "loop body" of the recursive function. Since we already know the base case condition, this is sufficient to synthesize a recursive function.

We use `tsltools` to transform TSL specifications into the standard `tlsf` [27] format. We then use `Strix`, a state-of-the-art tool for LTL synthesis [28], to synthesize an AIGER [26] circuit. We then use `tsltools` to convert the AIGER circuit into a Control Flow Model (CFM) [19] and generate program code. `tsltools` supports many different targets for code generation, including high-level languages like `Haskell` and `JavaScript` and hardware description languages like C$\lambda$aSH, `yosys`, or `nextpnr` [4, 39].

Thanks to our procedure's modularity, we can leverage state-of-the-art tools in both program synthesis and reactive synthesis. As we use the standardized SyGuS and TLSF formats, future advancements (both in theory and engineering) can be directly integrated into our tool.

## 5.2 Experimental Results

We evaluated our tool on four classes of benchmarks[1] from adapted from [12, 19] and present the results in Table 1.

We note that despite the large number of assumptions that we may generate, the synthesis time does not grow exponentially. Generally, the reactive synthesis time dominates the time taken for synthesis. This suggests while our TSL-MT synthesis procedure may add assumptions that are unneeded for realizability, it is possible that it is faster than a *lazy* approach. A lazy approach runs reactive synthesis multiple times and only adds the necessary assumptions for synthesis to complete. While a lazy approach will generate only the necessary environment assumptions synthesis, it needs to run reactive synthesis multiple times. Since our results show that a single reactive synthesis procedure takes longer than many Syntax-guided Synthesis problems, it is likely that a lazy approach will take significantly longer than our eager procedure.

To evaluate the overhead incurred by our approach, we also compared our results to that of an oracle, presented in Figure 4. We have chosen to compare our results against oracle as the synthesis problem of TSL-MT is a new problem; a comparison with other procedures is not possible because there are no alternative tools synthesizing TSL-MT formulas. Instead, we compare our evaluation results against an oracle that represents the best possible scenario.

We assume that the oracle is capable of constructing the TSL with assumptions *minimum realizability core* [14] from a TSL-MT formula. We construct this oracle formula by taking the TSL with assumptions formula we generate from our procedure, and then running it through the minimum realizability core feature of `tsltools`. This removes all unnecessary assumptions from the TSL with assumptions formula, which reduces the complexity of the reactive synthesis procedure. Therefore, synthesis from an oracle represents a theoretical best possible runtime by 1) incurring no overhead in assumption generation (generated via SyGuS in our procedure) and 2) not adding superfluous environment assumptions that increase the synthesis run time. Overall, we found that while our algorithm takes at worst more than twice the time of the oracle, it is not prohibitively expensive. This is thanks to `Strix` lazily building reachable states on demand, avoiding the overhead that may incur from superfluous environment assumptions.

## 5.3 Case Study: Music Synthesizer

For a case study, we synthesized a music synthesizer in `JavaScript` as a real-world example of TSL-MT synthesis. The synthesized tool applies music effects, such as vibrato (LFO) and FM synthesis, depending on which notes the user plays. The synthesized system changes the frequencies randomly, but abides by liveness guarantees that these effects must occur infinitely often. As an example of a concrete TSL-MT specification, we show the formula for the vibrato functionality of the synthesizer in Fig. 5. We present a demo playing the popular Jazz tune *Autmun Leaves*, available at https://vimeo.com/647965386. The demo takes in a TSL-MT specification and transforms it into `JavaScript` code that immediately runs on top of the standard audio libraries `WebAudio` and `WebMIDI`. Afterwards, the synthesized system responds to the environment – the keyboard player – and applies music effects accordingly.

[1] Full listing available at https://github.com/Barnard-PL-Labs/temos/tree/art-eval-pldi22/benchmarks

| Benchmark ($\varphi$) | $\lvert \varphi \rvert$ | $\lvert \mathbb{P} \rvert$ | $\lvert \mathbb{F} \rvert$ | $\lvert \psi \rvert$ | $\psi$ Generation (s) | TSL Synthesis (s) | Sum (s) | Synthesized LoC |
|---|---|---|---|---|---|---|---|---|
| **Music Synthesizer** | | | | | | | | |
| Vibrato | 10 | 2 | 2 | 21 | 0.431 | 0.914 | 1.345 | 206 |
| Modulation | 33 | 4 | 4 | 41 | 2.012 | 3.983 | 5.995 | 1352 |
| Intertwined | 58 | 4 | 4 | 41 | 2.157 | 3.178 | 5.335 | 1366 |
| Multi-effect | 27 | 6 | 6 | 45 | 3.145 | 81.470 | 84.615 | 1463 |
| **Pong** | | | | | | | | |
| Single-Player | 27 | 1 | 1 | 5 | 0.043 | 0.571 | 0.614 | 169 |
| Two-Player | 49 | 2 | 2 | 12 | 0.181 | 0.625 | 0.806 | 195 |
| Bouncing | 27 | 3 | 2 | 25 | 0.418 | 0.808 | 1.226 | 169 |
| Automatic | 27 | 5 | 2 | 54 | 0.541 | 0.988 | 1.529 | 214 |
| **Escalator** | | | | | | | | |
| Simple | 29 | 1 | 2 | 2 | 0.011 | 0.434 | 0.445 | 166 |
| Counting | 57 | 2 | 2 | 8 | 0.100 | 0.592 | 0.692 | 241 |
| Bidirectional | 57 | 5 | 11 | 9 | 0.340 | 2.291 | 1.121 | 279 |
| Smart | 65 | 8 | 2 | 34 | 3.034 | 0.935 | 3.969 | 179 |
| **CPU Scheduler** | | | | | | | | |
| Round Robin | 21 | 2 | 4 | 16 | 0.149 | 0.740 | 0.889 | 252 |
| Load Balancer | 39 | 3 | 4 | 12 | 0.531 | 2.128 | 1.345 | 208 |
| Preemptive | 54 | 4 | 4 | 12 | 0.548 | 0.765 | 1.313 | 356 |
| CFS | 81 | 8 | 5 | 12 | 0.533 | 2.443 | 2.976 | 2825 |

**Table 1.** Experimental Results. $\varphi$ refers to the original specification, and $\mathbb{P}$ and $\mathbb{F}$ refer to the number of unique predicate and update terms in the specification. $\psi$ refers to the assumptions generated by the algorithm. LoC refers to lines of code.

### 5.4 Case Study: Linux Kernel Scheduler

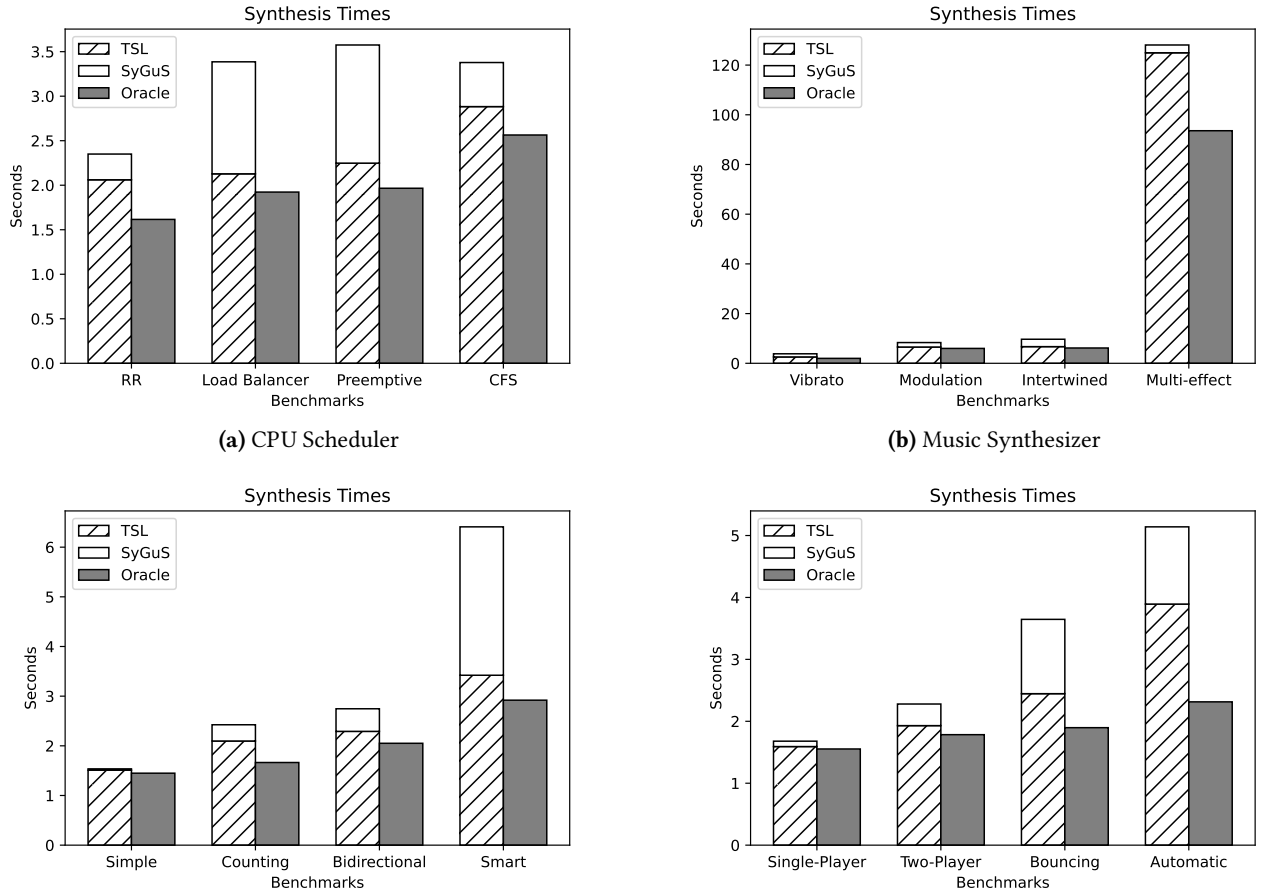
We used our procedure described in Section 4 and the CFS specification from Section 2 to synthesize a scheduler for the Linux `v4.15` kernel. We added our synthesized code as an additional scheduler `sched_class` as defined in the kernel source `/kernel/sched/sched.h`. The code listing is available online at https://pastebin.com/TMZAmFWM.

The reactive predicates enqueue and dequeue of the specification coincide with the semantics of the `enqueue_task` and `dequeue_task` of the `sched_class` struct. Since the synthesized code needs to update the virtual runtimes of each task periodically, we link the result this to the `task_tick` function of the kernel, which is a timer interrupt that happens periodically to update the runtime statistics of the each task. Note that with this case study we are tackling a different type of scheduling problem than is encountered with device drivers [9] - here we are generating program code that manages high level data abstractions in the kernel such as a task.

## 6 Related Work

The key issue we address with this work is synthesizing programs that have both control-dominated and data-dominated components. While our approach is to combine reactive synthesis and SyGuS, there are other approaches to tackling the synthesis of systems with both a control and a data component that focus on using exclusively either reactive synthesis or SyGuS.

Temporal Stream Logic (TSL) [19] was presented as a logic to separate the concerns of control and data, but only offered support for synthesis on the reactive side of the problem. Synthesizing programs (rather than models) with LTL has been explored, but without TSL-MT's interface between control and data, this work was limited to synthesizing programs over Boolean values [32]. Some systems handle synthesis problems with both a reactive and data component, such as Abstraction Based Controller Synthesis [23], but again this approach but the full burden of synthesis on the reactive synthesis engine by framing the entirety of the problem as a reactive system. Other temporal logics combine data and reactivity, such as Signal Temporal Logic (STL) [33] or Metric Temporal Logic [29]. However synthesis procedures for logics such as STL [37] must fully rely on reactive synthesis techniques, as the logic does not have the same clean separation of control and data that TSL provides through functions and update terms. From the SyGuS perspective, prior work has encoded the fully control-dominated problem of reactive motion planning in SyGuS, but lacks a data-transformation focused component [11]. In contrast, our approach leverages TSL-MT to allow reactive synthesis and SyGuS to play to their respective strengths of control and data.




**(a)** CPU Scheduler

**(b)** Music Synthesizer

**(c)** Elevator

**(d)** Pong

**Figure 4.** Comparison of synthesis times with an oracle. Crossed lines signify Reactive Synthesis, and dotted bars signify the assumption generation with Syntax-Guided Synthesis. The oracle synthesis time is shown to the right.

Our approach to synthesizing TSL-MT specifications is to eagerly add assumptions to approximate the TSL-MT specification into a TSL specification. This mirrors the eager approach to solving SMT queries [5] that translate SMT problems into classic satisfiability (SAT) problems, i.e. bit-blasting [21] or the UCLID SMT Solver [30]. While a lazy approach – such as one using a Counter-example Guided Abstraction Refinement [15] may be faster – may be more efficient, an eager approach allows us to easily leverage off-the-shelf SMT solvers in an extensible manner.

In our implementation, we timeout unbounded SyGuS queries because our backends do not have support for unrealizability. Recently, there has been work translating SyGuS problems as reachability analysis or Constrainted Horn Clauses (CHC) [24, 25], which are capable of showing unrealizability for SyGuS queries. This work could be used to improve the running times of our tool.

In our motivating example and case study, we synthesize a Linux kernel process scheduler. Applying synthesis to generate low-level systems code, particularly for scheduling, has seen a variety of research. This includes work such as synthesizing schedulers for synchronization [9], synchronization of GPU kernels [3], and regression-free synthesis for concurrency [10].

## 7 Conclusions

We have introduced a method for combining syntax-guided synthesis and reactive synthesis to synthesize a program with both reactive control and data transformations. We present Temporal Stream Logic Modulo Theories as a framework to combine the two different synthesis frameworks, and provide a tool, `temos`, to implement the procedure. We demonstrate the practicality of our approach by an experimental evaluation and two case studies where we synthesize a `JavaScript` music keyboard synthesizer, as well as a Linux process scheduler.

```
//#RA#
always guarantee {
  G F [lfo <- True()];
  G F [lfo <- False()];

  lte lfoFreq c10() -> [lfo <- False()]
    U gt  lfoFreq c10();
  gt  lfoFreq c10() -> [lfo <- True()]
    U lte lfoFreq c10();
  [lfo <- False()] ->
    [lfoFreq <- add lfoFreq c1()];
  [lfo <- True()] ->
    [lfoFreq <- sub lfoFreq c1()];
}
```

**Figure 5.** The TSL-MT benchmark specification for Music Synthesizer: Vibrato. The LFO should be off until the LFO frequency is greater than the constant 10 (`c10()`), and on until it is less than 10. The frequency should increase by 1 when turned off, and decrease by 1 when turned on. The `#RA#` annotation indicates that this specification uses the Real Arithmetic theory extension of TSL-MT.

## References


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

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

[3] Sourav Anand and Nadia Polikarpova. 2018. Automatic Synchronization for GPU Kernels. In *Formal Methods in Computer Aided Design, FMCAD*. IEEE. https://doi.org/10.23919/FMCAD.2018.8602999

[4] C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis.* Ph.D. Dissertation. University of Twente. https://doi.org/10.3990/1.9789036538039 eemcs-eprint-23939.

[5] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer.

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV (Lecture Notes in Computer Science, Vol. 6806)*. Springer. https://doi.org/10.1007/978-3-642-22110-1_14

[7] Roderick Bloem, Swen Jacobs, and Ayrat Khalimov. 2014. Parameterized synthesis case study: AMBA AHB (extended version). *arXiv preprint arXiv:1406.7608* (2014).

[8] Benjamin Caulfield, Markus N Rabe, Sanjit A Seshia, and Stavros Tripakis. 2015. What's Decidable about Syntax-Guided Synthesis? *arXiv preprint arXiv:1510.08393* (2015).

[9] Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. 2015. From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis. In *Computer Aided Verification - 27th International Conference, CAV (Lecture Notes in Computer Science, Vol. 9207)*. Springer. https://doi.org/10.1007/978-3-319-21668-3_11

[10] Pavol Černỳ, Thomas A Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2014. Regression-free synthesis for concurrency. In *International conference on computer aided verification*. Springer.

[11] Sarah Chasins and Julie L. Newcomb. 2016. Using SyGuS to Synthesize Reactive Motion Plans. In Proceedings Fifth Workshop on *Synthesis (Electronic Proceedings in Theoretical Computer Science, Vol. 229)*. Open Publishing Association. https://doi.org/10.4204/EPTCS.229.3

[12] Wonhyuk Choi, Michel Vazirani, and Mark Santolucito. 2021. Program Synthesis for Musicians: A Usability Testbed for Temporal Logic Specifications. In *Asian Symposium on Programming Languages and Systems*. Springer.

[13] Alonzo Church. 1957. Applications of recursive arithmetic to the problem of circuit synthesis *(Summaries of the Summer Institute of Symbolic Logic at Cornell University, Vol. 1)*.

[14] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Andrei Tchaltsev. 2008. Diagnostic information for realizability. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer.

[15] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer.

[16] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*.

[17] Bernd Finkbeiner, Philippe Heim, and Noemi Passing. 2021. Temporal Stream Logic modulo Theories. arXiv:2104.14988 [cs.LO] https://arxiv.org/abs/2104.14988.

[18] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Synthesizing functional reactive programs. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*.

[19] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Temporal stream logic: Synthesis beyond the bools. In *International Conference on Computer Aided Verification*. Springer.

[20] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *POPL*. https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/

[21] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. 2014. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *International Conference on Computer Aided Verification*. Springer.

[22] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969).

[23] Kyle Hsu, Rupak Majumdar, Kaushik Mallik, and Anne-Kathrin Schmuck. 2018. Multi-layered abstraction-based controller synthesis for continuous-time systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*.

[24] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving unrealizability for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer.

[25] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[26] Swen Jacobs. 2014. Extended AIGER format for synthesis. *arXiv preprint arXiv:1405.5793* (2014).

[27] Swen Jacobs, Felix Klein, and Sebastian Schirmer. 2016. A high-level LTL synthesis format: TLSF v1. 1. *arXiv preprint arXiv:1604.02284* (2016).

[28] Swen Jacobs, Guillermo Perez, Roderick Bloem, Armin Biere, et al. 2020. The 7th Reactive Synthesis Competition. http://www.syntcomp.org/wp-content/uploads/2020/07/SYNTCOMP2020-SYNT.pdf. Accessed: 2021-05-18.

[29] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-time systems* 2, 4 (1990).

[30] Shuvendu K Lahiri and Sanjit A Seshia. 2004. The UCLID decision procedure. In *International Conference on Computer Aided Verification*. Springer.
[31] Robert Love. 2010. *Linux kernel development*. Pearson Education.
[32] Parthasarathy Madhusudan. 2011. Synthesizing reactive programs. In *Computer Science Logic (CSL'11)-25th International Workshop/20th Annual Conference of the EACSL*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[33] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer.
[34] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. 2018. Strix: Explicit Reactive Synthesis Strikes Back!. In *Computer Aided Verification - 30th International Conference, CAV (Lecture Notes in Computer Science, Vol. 10981)*. Springer. https://doi.org/10.1007/978-3-319-96145-3_31
[35] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*. IEEE.
[36] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of an Asynchronous Reactive Module. In *Automata, Languages and Programming, 16th International Colloquium (Lecture Notes in Computer Science, Vol. 372)*. Springer. https://doi.org/10.1007/BFb0035790
[37] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M. Murray, and Sanjit A. Seshia. 2015. Reactive Synthesis from Signal Temporal Logic Specifications. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control* (Seattle, Washington) *(HSCC '15)*. Association for Computing Machinery, New York, NY, USA.
[38] Leonid Ryzhyk and Adam Walker. 2016. Developing a Practical Reactive Synthesis Tool: Experience and Lessons Learned. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV (EPTCS, Vol. 229)*. https://doi.org/10.4204/EPTCS.229.8
[39] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. 2019. Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE.
[40] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2014. *Operating system concepts essentials*.