Temporal Stream Logic: Synthesis Beyond the Bools

Bernd Finkbeiner[1], Felix Klein[1(✉)], Ruzica Piskac[2], and Mark Santolucito[2]

[1] Saarland University, Saarbrücken, Germany
klein@react.uni-saarland.de
[2] Yale University, New Haven, USA

Abstract. Reactive systems that operate in environments with complex data, such as mobile apps or embedded controllers with many sensors, are difficult to synthesize. Synthesis tools usually fail for such systems because the state space resulting from the discretization of the data is too large. We introduce TSL, a new temporal logic that separates control and data. We provide a CEGAR-based synthesis approach for the construction of implementations that are guaranteed to satisfy a TSL specification for all possible instantiations of the data processing functions. TSL provides an attractive trade-off for synthesis. On the one hand, synthesis from TSL, unlike synthesis from standard temporal logics, is undecidable in general. On the other hand, however, synthesis from TSL is scalable, because it is independent of the complexity of the handled data. Among other benchmarks, we have successfully synthesized a music player Android app and a controller for an autonomous vehicle in the Open Race Car Simulator (TORCS).

1 Introduction

In reactive synthesis, we automatically translate a formal specification, typically given in a temporal logic, into a controller that is guaranteed to satisfy the specification. Over the past two decades there has been much progress on reactive synthesis, both in terms of algorithms, notably with techniques like GR(1)-synthesis [7] and bounded synthesis [20], and in terms of tools, as showcased, for example, in the annual SYNTCOMP competition [25].

In practice however, reactive synthesis has seen limited success. One of the largest published success stories [6] is the synthesis of the AMBA bus protocol. To push synthesis even further, automatically synthesizing a controller for

Supported by the European Research Council (ERC) Grant OSARES (No. 683300), the German Research Foundation (DFG) as part of the Collaborative Research Center Foundations of Perspicuous Software Systems (TRR 248, 389792660), and the National Science Foundation (NSF) Grant CCF-1302327.

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 609–629, 2019.
https://doi.org/10.1007/978-3-030-25540-4_35

an autonomous system has been recognized to be of critical importance [52]. Despite many years of experience with synthesis tools, our own attempts to synthesize such controllers with existing tools have been unsuccessful. The reason is that the tools are unable to handle the data complexity of the controllers. The controller only needs to switch between a small number of behaviors, like steering during a bend, or shifting gears on high rpm. The number of control states in a typical controller (cf. [18]) is thus not much different from the arbiter in the AMBA case study. However, in order to correctly initiate transitions between control states, the driving controller must continuously process data from more than 20 sensors.

If this data is included (even as a rough discretization) in the state space of the controller, then the synthesis problem is much too large to be handled by any available tools. It seems clear then, that a scalable synthesis approach must separate control and data. If we assume that the data processing is handled by some other approach (such as deductive synthesis [38] or manual programming), is it then possible to solve the remaining reactive synthesis problem?

In this paper, we show scalable reactive synthesis is indeed possible. Separating data and control has allowed us to synthesize reactive systems, including an autonomous driving controller and a music player app, that had been impossible to synthesize with previously available tools. However, the separation of data and control implies some fundamental changes to reactive synthesis, which we describe in the rest of the paper. The changes also imply that the reactive synthesis problem is no longer, in general, decidable. We thus trade theoretical decidability for practical scalability, which is, at least with regard to the goal of synthesizing realistic systems, an attractive trade-off.

We introduce Temporal Stream Logic (TSL), a new temporal logic that includes *updates*, such as $[\![\texttt{y} \leftarrowtail \texttt{f x}]\!]$, and predicates over arbitrary function terms. The update $[\![\texttt{y} \leftarrowtail \texttt{f x}]\!]$ indicates that the result of applying function `f` to variable `x` is assigned to `y`. The implementation of predicates and functions is not part of the synthesis problem. Instead, we look for a system that satisfies the TSL specification *for all possible interpretations of the functions and predicates*.

This implicit quantification over all possible interpretations provides a useful abstraction: it allows us to *independently* implement the data processing part. On the other hand, this quantification is also the reason for the undecidability of the synthesis problem. If a predicate is applied to the same term *twice*, it must (independently of the interpretation) return the *same* truth value. The synthesis must then implicitly maintain a (potentially infinite) set of terms to which the predicate has previously been applied. As we show later, this set of terms can be used to encode PCP [45] for a proof of undecidability.

We present a practical synthesis approach for TSL specifications, which is based on bounded synthesis [20] and counterexample-guided abstraction refinement (CEGAR) [9]. We use bounded synthesis to search for an implementation up to a (iteratively growing) bound on the number of states. This approach underapproximates the actual TSL synthesis problem by leaving the interpretation of the predicates to the environment. The underapproximation allows

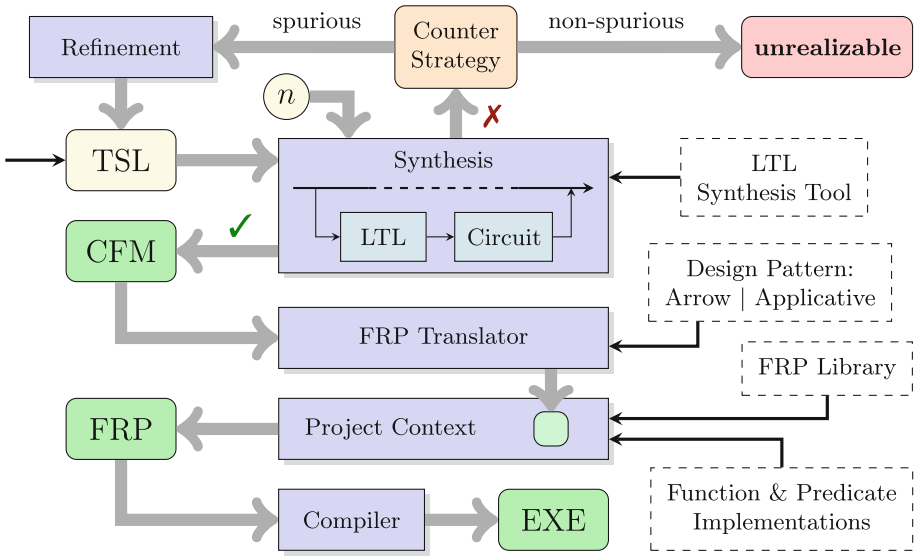


Fig. 1. The TSL synthesis procedure uses a modular design. Each step takes input from the previous step as well as interchangeable modules (dashed boxes).

for inconsistent behaviors: the environment might assign different truth values to the same predicate when evaluated at different points in time, even if the predicate is applied to the same term. However, if we find an implementation in this underapproximation, then the CEGAR loop terminates and we have a correct implementation for the original TSL specification. If we do not find an implementation in the underapproximation, we compute a counter strategy for the environment. Because bounded synthesis reduces the synthesis problem to a safety game, the counter strategy is a reachability strategy that can be represented as a finite tree. We check whether the counter strategy is spurious by searching for a pair of positions in the strategy where some predicate results in different truth values when applied to the same term. If the counter strategy is not spurious, then no implementation exists for the considered bound, and we increase the bound. If the counter strategy is spurious, then we introduce a constraint into the specification that eliminates the incorrect interpretation of the predicate, and continue with the refined specification.

A general overview of this procedure is shown in Fig. 1. The top half of the figure depicts the bounded search for an implementation that realizes a TSL specification using the CEGAR loop to refine the specification. If the specification is realizable, we proceed in the bottom half of the process, where a synthesized implementation is converted to a control flow model (CFM) determining the control of the system. We then specialize the CFM to Functional Reactive Programming (FRP), which is a popular and expressive programming paradigm for building reactive programs using functional programming languages [14].

```
Sys.leaveApp() :
  if (MP.musicPlaying())
    Ctrl.pause()
Sys.resumeApp() :
  pos = MP.trackPos()
  Ctrl.play(Tr,pos)
```

$$\texttt{ALWAYS}\ \big(\texttt{leaveApp Sys} \wedge \texttt{musicPlaying MP} \rightarrow [\![\texttt{Ctrl} \leftarrowtail \texttt{pause}()]\!]\big)$$

$$\texttt{ALWAYS}\ \big(\texttt{resumeApp Sys} \rightarrow [\![\texttt{Ctrl} \leftarrowtail \texttt{play Tr (trackPos MP)}]\!]\big)$$

Fig. 2. Sample code and specification for the music player app.

Our framework supports any FRP library using the *Arrow* or *Applicative* design patterns, which covers most of the existing FRP libraries (e.g. [2,3,10,41]). Finally, the synthesized control flow is embedded into a project context, where it is equipped with function and predicate implementations and then compiled to an executable program.

Our experience with synthesizing systems based on TSL specifications has been extremely positive. The synthesis works for a broad range of benchmarks, ranging from classic reactive synthesis problems (like escalator control), through programming exercises from functional reactive programming, to novel case studies like our music player app and the autonomous driving controller for a vehicle in the Open Race Car Simulator (TORCS).

2 Motivating Example

To demonstrate the utility of our method, we synthesized a music player Android app[1] from a TSL specification. A major challenge in developing Android apps is the temporal behavior of an app through the *Android lifecycle* [46]. The Android lifecycle describes how an app should handle being paused, when moved to the background, coming back into focus, or being terminated. In particular, *resume and restart errors* are commonplace and difficult to detect and correct [46]. Our music player app demonstrates a situation in which a resume and restart error could be unwittingly introduced when programming by hand, but is avoided by providing a specification. We only highlight the key parts of the example here to give an intuition of TSL. The complete specification is presented in [19].

Our music player app utilizes the Android music player library (`MP`), as well as its control interface (`Ctrl`). It pauses any playing music when moved to the background (for instance if a call is received), and continues playing the currently selected track (`Tr`) at the last track position when the app is resumed. In the Android system (`Sys`), the `leaveApp` method is called whenever the app moves to the background, while the `resumeApp` method is called when the app is brought back to the foreground. To avoid confusion between pausing music and pausing the app, we use `leaveApp` and `resumeApp` in place of the Android methods

[1] https://play.google.com/store/apps/details?id=com.mark.myapplication.

```
bool wasPlaying = false

Sys.leaveApp() :
  if (MP.musicPlaying()) :
    wasPlaying = true
    Ctrl.pause()
  else
    wasPlaying = false

Sys.resumeApp() :
  if (wasPlaying)
    pos = MP.trackPos()
    Ctrl.play(Tr,pos)
```

```
ALWAYS ((leaveApp Sys ∧ musicPlaying MP
        → ⟦Ctrl ↼ pause()⟧)
        ∧ (⟦Ctrl ↼ play Tr (trackPos MP)⟧
          AS_SOON_AS resumeApp Sys))
```

Fig. 3. The effect of a minor change in functionality on code versus a specification.

onPause and onResume. A programmer might manually write code for this as shown on the left in Fig. 2.

The behavior of this can be directly described in TSL as shown on the right in Fig. 2. Even eliding a formal introduction of the notation for now, the specification closely matches the textual specification. First, when the user leaves the app and the music is playing, the music pauses. Likewise for the second part, when the user resumes the app, the music starts playing again.

However, assume we want to change the behavior so that the music only plays on resume when the music had been playing before leaving the app in the first place. In the manually written program, this new functionality requires an additional variable `wasPlaying` to keep track of the music state. Managing the state requires multiple changes in the code as shown on the left in Fig. 3. The required code changes include: a conditional in the `resumeApp` method, setting `wasPlaying` appropriately in two places in `leaveApp`, and providing an initial value. Although a small example, it demonstrates how a minor change in functionality may require wide-reaching code changes. In addition, this change introduces a globally scoped variable, which then might accidentally be set or read elsewhere. In contrast, it is a simple matter to change the TSL specification to reflect this new functionality. Here, we only update one part of the specification to say that if the user leaves the app and the music is playing, the music has to play again as soon as the app resumes.

Synthesis allows us to specify a temporal behavior without worrying about the implementation details. In this example, writing the specification in TSL has eliminated the need of an additional state variable, similarly to a higher order `map` eliminating the need for an iteration variable. However, in more complex examples the benefits compound, as TSL provides a modular interface to specify behaviors, offloading the management of multiple interconnected temporal behaviors from the user to the synthesis engine.

3 Preliminaries

We assume time to be discrete and denote it by the set $\mathbb{N}$ of positive integers. A value is an arbitrary object of arbitrary type. $\mathcal{V}$ denotes the set of all values. The Boolean values are denoted by $\mathcal{B} \subseteq \mathcal{V}$. A stream $s\colon \mathbb{N} \to \mathcal{V}$ is a function fixing values at each point in time. An n-ary function $f\colon \mathcal{V}^n \to \mathcal{V}$ determines new values from n given values, where the set of all functions (of arbitrary arity) is given by $\mathcal{F}$. Constants are functions of arity 0. Every constant is a value, i.e., is an element of $\mathcal{F} \cap \mathcal{V}$. An n-ary predicate $p\colon \mathcal{V}^n \to \mathcal{B}$ checks a property over n values. The set of all predicates (of arbitrary arity) is given by $\mathcal{P}$, where $\mathcal{P} \subseteq \mathcal{F}$. We use $B^{[A]}$ to denote the set of all total functions with domain A and image B.

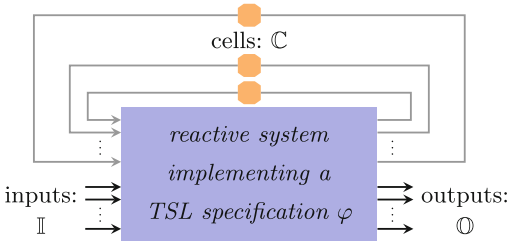
In the classical synthesis setting, inputs and outputs are vectors of Booleans, where the standard abstraction treats inputs and outputs as atomic propositions $\mathcal{I} \cup \mathcal{O}$, while their Boolean combinations form an alphabet $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$. Behavior then is described through infinite sequences $\alpha = \alpha(0)\alpha(1)\alpha(2)\ldots \in \Sigma^\omega$. A *specification* describes a relation between input sequences $\alpha \in (2^{\mathcal{I}})^\omega$ and output sequences $\beta \in (2^{\mathcal{O}})^\omega$. Usually, this relation is not given by explicit sequences, but by a fomula in a temporal logic. The most popular such logic is Linear Temporal Logic (LTL) [43], which uses Boolean connectives to specify behavior at specific points in time, and temporal connectives, to relate sub-specifications over time. The realizability and synthesis problems for LTL are 2ExpTime-complete [44].

An implementation describes a realizing strategy, formalized via infinite trees. A Φ-labeled and Υ-branching tree is a function $\sigma\colon \Upsilon^* \to \Phi$, where Υ denotes the set of branching directions along a tree. Every node of the tree is given by a finite prefix $v \in \Upsilon^*$, which fixes the path to reach a node from the root. Every node is labeled by an element of Φ. For infinite paths $\nu \in \Upsilon^\omega$, the branch $\sigma\langle\nu$ denotes the sequence of labels that appear on ν, i.e., $\forall t \in \mathbb{N}.\ (\sigma\langle\nu)(t) = \sigma(\nu(0)\ldots\nu(t-1))$.

4 Temporal Stream Logic

We present a new logic: Temporal Stream Logic (TSL), which is especially designed for synthesis and allows for the manipulation of infinite streams of arbitrary (even non-enumerative, or higher order) type. It provides a straightforward notation to specify how outputs are computed from inputs, while using an intuitive interface to access time. The main focus of TSL is to describe temporal control flow, while abstracting away concrete implementation details. This not only keeps the logic intuitive and simple, but also allows a user to identify problems in the control flow even without a concrete implementation at hand. In this way, the use of TSL scales up to any required abstraction, such as API calls or complex algorithmic transformations.

Architecture. A TSL formula φ specifies a reactive system that in every time step processes a finite number of inputs $\mathbb{I}$ and produces a finite number of outputs $\mathbb{O}$. Furthermore, it uses cells $\mathbb{C}$ to store a value computed at time t, which can then be reused in the next time step $t+1$. An overview of the architecture of such a system is given in Fig. 4a. In terms of behavior, the environment produces infinite



(a) Architecture

Function Term:

$\tau_F := \mathtt{s_i} \mid \mathtt{f}\ \tau_F^0\ \tau_F^1\ \cdots\ \tau_F^{n-1}$

Predicate Term:

$\tau_P := \mathtt{p}\ \tau_F^0\ \tau_F^1\ \cdots\ \tau_F^{n-1}$

Update:

$[\![\mathtt{s_o} \leftarrowtail \tau_F]\!]$

(b) Term Definitions

Fig. 4. General architecture of reactive systems that are specified in TSL on the left, and the structure of function, predicate and updates on the right.

streams of input data, while the system uses pure (side-effect free) functions to transform the values of these input streams in every time step. After their transformation, the data values are either passed to an output stream or are passed to a cell, which pipes the output value from one time step back to the corresponding input value of the next. The behaviour of the system is captured by its infinite execution over time.

Function Terms, Predicate Terms, and Updates. In TSL we differentiate between two elements: we use purely functional transformations, reflected by functions $f \in \mathcal{F}$ and their compositions, and predicates $p \in \mathcal{P}$, used to control how data flows inside the system. To argue about both elements we use a term based notation, where we distinguish between function terms τ_F and predicate terms τ_P, respectively. Function terms are either constructed from inputs or cells ($\mathtt{s_i} \in \mathbb{I} \cup \mathbb{C}$), or from functions, recursively applied to a set of function terms. Predicate terms are constructed similarly, by applying a predicate to a set of function terms. Finally, an update takes the result of a function computation and passes it either to an output or a cell ($\mathtt{s_o} \in \mathbb{O} \cup \mathbb{C}$). An overview of the syntax of the different term notations is given in Fig. 4b. Note that we use curried argument notation similar to functional programming languages.

We denote sets of function and predicate terms, and updates by $\mathcal{T}_F$, $\mathcal{T}_P$ and $\mathcal{T}_U$, respectively, where $\mathcal{T}_P \subseteq \mathcal{T}_F$. We use $\mathbb{F}$ to denote the set of function literals and $\mathbb{P} \subseteq \mathbb{F}$ to denote the set of predicate literals, where the literals $\mathtt{s_i}$, $\mathtt{s_o}$, $\mathtt{f}$ and $\mathtt{p}$ are symbolic representations of inputs and cells, outputs and cells, functions, and predicates, respectively. Literals are used to construct terms as shown in Fig. 4b. Since we use a symbolic representation, functions and predicates are not tied to a specific implementation. However, we still classify them according to their arity, i.e., the number of function terms they are applied to, as well as by their type: input, output, cell, function or predicate. Furthermore, terms can be compared syntactically using the equivalence relation $\equiv$. To assign a semantic interpretation to functions, we use an assignment function $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$.

Inputs, Outputs, and Computations. We consider momentary inputs $i \in \mathcal{V}^{[\mathbb{I}]}$, which are assignments of inputs $\mathtt{i} \in \mathbb{I}$ to values $v \in \mathcal{V}$. For the sake of readability let $\mathcal{I} = \mathcal{V}^{[\mathbb{I}]}$. Input streams are infinite sequences $\iota \in \mathcal{I}^\omega$ consisting of infinitely many momentary inputs.

Similarly, a momentary output $o \in \mathcal{V}^{[\mathbb{O}]}$ is an assignment of outputs $\mathtt{o} \in \mathbb{O}$ to values $v \in \mathcal{V}$, where we also use $\mathcal{O} = \mathcal{V}^{[\mathbb{O}]}$. Output streams are infinite sequences $\varrho \in \mathcal{O}^\omega$. To capture the behavior of a cell, we introduce the notion of a computation ς. A computation fixes the function terms that are used to compute outputs and cell updates, without fixing semantics of function literals. Intuitively, a computation only determines which function terms are used to compute an output, but abstracts from actually computing it.

The basic element of a computation is a computation step $c \in \mathcal{T}_F^{[\mathbb{O} \cup \mathbb{C}]}$, which is an assignment of outputs and cells $\mathtt{s_o} \in \mathbb{O} \cup \mathbb{C}$ to function terms $\tau_F \in \mathcal{T}_F$. For the sake of readability let $\mathcal{C} = \mathcal{T}_F^{[\mathbb{O} \cup \mathbb{C}]}$. A computation step fixes the control flow behaviour at a single point in time. A computation $\varsigma \in \mathcal{C}^\omega$ is an infinite sequence of computation steps.

As soon as input streams, and function and predicate implementations are known, computations can be turned into output streams. To this end, let $\langle\cdot\rangle \colon \mathbb{F} \to \mathcal{F}$ be some function assignment. Furthermore, assume that there are predefined constants $init_{\mathtt{c}} \in \mathcal{F} \cap \mathcal{V}$ for every cell $\mathtt{c} \in \mathbb{C}$, which provide an initial value for each stream at the initial point in time. To receive an output stream from a computation $\varsigma \in \mathcal{C}^\omega$ under the input stream ι, we use an evaluation function $\eta_{\langle\cdot\rangle} \colon \mathcal{C}^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \to \mathcal{V}$:

$$\eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{s_i}) = \begin{cases} \iota(t)(\mathtt{s_i}) & \text{if } \mathtt{s_i} \in \mathbb{I} \\ init_{\mathtt{s_i}} & \text{if } \mathtt{s_i} \in \mathbb{C} \ \wedge \ t = 0 \\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t-1, \varsigma(t-1)(\mathtt{s_i})) & \text{if } \mathtt{s_i} \in \mathbb{C} \ \wedge \ t > 0 \end{cases}$$

$$\eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{f}\ \tau_0\ \cdots\ \tau_{m-1}) = \langle \mathtt{f} \rangle\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \tau_0)\ \cdots\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \tau_{m-1})$$

Then $\varrho_{\langle\cdot\rangle,\varsigma,\iota} \in \mathcal{O}^\omega$ is defined via $\varrho_{\langle\cdot\rangle,\varsigma,\iota}(t)(\mathtt{o}) = \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{o})$ for all $t \in \mathbb{N}$, $\mathtt{o} \in \mathbb{O}$.

Syntax. Every TSL formula φ is built according to the following grammar:

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

An atomic proposition τ consists either of a predicate term, serving as a Boolean interface to the inputs, or of an update, enforcing a respective flow at the current point in time. Next, we have the Boolean operations via negation and conjunction, that allow us to express arbitrary Boolean combinations of predicate evaluations and updates. Finally, we have the temporal operator next: $\bigcirc\psi$, to specify the behavior at the next point in time and the temporal operator until: $\vartheta \,\mathcal{U}\, \psi$, which enforces a property ϑ to hold until the property ψ holds, where ψ must hold at some point in the future eventually.

Semantics. Formally, this leads to the following semantics. Let $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$, $\iota \in \mathcal{I}^\omega$, and $\varsigma \in \mathcal{C}^\omega$ be given, then the validity of a TSL formula φ with respect to ς and ι is defined inductively over $t \in \mathbb{N}$ via:

$$
\begin{aligned}
\varsigma, \iota, t &\models_{\langle\cdot\rangle} \mathtt{p}\ \tau_0\ \cdots\ \tau_{m-1} &&:\Leftrightarrow\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{p}\ \tau_0\ \cdots\ \tau_{m-1}) \\
\varsigma, \iota, t &\models_{\langle\cdot\rangle} [\![\mathtt{s} \leftarrowtail \tau]\!] &&:\Leftrightarrow\ \varsigma(t)(\mathtt{s}) \equiv \tau \\
\varsigma, \iota, t &\models_{\langle\cdot\rangle} \neg\psi &&:\Leftrightarrow\ \varsigma, \iota, t \not\models_{\langle\cdot\rangle} \psi \\
\varsigma, \iota, t &\models_{\langle\cdot\rangle} \vartheta \wedge \psi &&:\Leftrightarrow\ \varsigma, \iota, t \models_{\langle\cdot\rangle} \vartheta \ \wedge\ \varsigma, \iota, t \models_{\langle\cdot\rangle} \psi \\
\varsigma, \iota, t &\models_{\langle\cdot\rangle} \bigcirc\psi &&:\Leftrightarrow\ \varsigma, \iota, t+1 \models_{\langle\cdot\rangle} \psi \\
\varsigma, \iota, t &\models_{\langle\cdot\rangle} \vartheta\,\mathcal{U}\,\psi &&:\Leftrightarrow\ \exists t'' \geq t.\ \forall t \leq t' < t''.\ \varsigma, \iota, t' \models_{\langle\cdot\rangle} \vartheta \wedge \varsigma, \iota, t'' \models_{\langle\cdot\rangle} \psi
\end{aligned}
$$

Consider that the satisfaction of a predicate depends on the current computation step and the steps of the past, while for updates it only depends on the current computation step. Furthermore, updates are only checked syntactically, while the satisfaction of predicates depends on the given assignment $\langle\cdot\rangle$ and the input stream ι. We say that ς and ι satisfy φ, denoted by $\varsigma, \iota \models_{\langle\cdot\rangle} \varphi$, if $\varsigma, \iota, 0 \models_{\langle\cdot\rangle} \varphi$.

Beside the basic operators, we have the standard derived Boolean operators, as well as the derived temporal operators: *release* $\varphi\,\mathcal{R}\,\psi \equiv \neg((\neg\psi)\,\mathcal{U}(\neg\varphi))$, *finally* $\diamondsuit\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$, *always* $\square\varphi \equiv \mathit{false}\,\mathcal{R}\,\varphi$, the *weak* version of *until* $\varphi\,\mathcal{W}\,\psi \equiv (\varphi\,\mathcal{U}\,\psi) \vee (\square\varphi)$, and *as soon as* $\varphi\,\mathcal{A}\,\psi \equiv \neg\psi\,\mathcal{W}(\psi \wedge \varphi)$.

Realizability. We are interested in the following realizability problem: given a TSL formula φ, is there a strategy $\sigma \in \mathcal{C}^{[\mathcal{I}^+]}$ such that for every input $\iota \in \mathcal{I}^\omega$ and function implementation $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$, the branch $\sigma \wr \iota$ satisfies φ, i.e.,

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

If such a strategy σ exists, we say σ realizes φ. If we additionally ask for a concrete instantiation of σ, we consider the synthesis problem of TSL.

5 TSL Properties

In order to synthesize programs from TSL specifications, we give an overview of the first part of our synthesis process, as shown in Fig. 1. First we show how to approximate the semantics of TSL through a reduction to LTL. However, due to the approximation, finding a realizable strategy immediately may fail. Our solution is a CEGAR loop that improves the approximation. This CEGAR loop is necessary, because the realizability problem of TSL is undecidable in general.

Approximating TSL with LTL. We approximate TSL formulas with weaker LTL formulas. The approximation reinterprets the syntactic elements, $\mathcal{T}_P$ and $\mathcal{T}_U$, as atomic propositions for LTL. This strips away the semantic meaning of the function application and assignment in TSL, which we reconstruct by later adding assumptions lazily to the LTL formula.

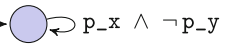
Formally, let $\mathcal{T}_P$ and $\mathcal{T}_U$ be the finite sets of predicate terms and updates, which appear in φ_{TSL}, respectively. For every assigned signal, we partition $\mathcal{T}_U$ into $\biguplus_{\mathtt{s}_\circ \in \mathbb{O} \cup \mathbb{C}} \mathcal{T}_U^{\mathtt{s}_\circ}$. For every $\mathtt{c} \in \mathbb{C}$ let $\mathcal{T}_{U/\mathtt{id}}^{\mathtt{c}} = \mathcal{T}_U^{\mathtt{c}} \cup \{[\![\mathtt{c} \leftarrowtail \mathtt{c}]\!]\}$, for $\mathtt{o} \in \mathbb{O}$ let

$$\square\,([\![\mathtt{y} \leftarrow \mathtt{y}]\!] \vee [\![\mathtt{y} \leftarrow \mathtt{x}]\!]) \wedge \diamondsuit\, \mathtt{p\ x} \rightarrow \diamondsuit\, \mathtt{p\ y}$$

(a) TSL specification

$$\square\, \neg(\mathtt{y\_to\_y} \wedge \mathtt{x\_to\_y}) \wedge \square\,(\mathtt{y\_to\_y} \vee \mathtt{x\_to\_y}) \wedge \diamondsuit\, \mathtt{p\_x} \rightarrow \diamondsuit\, \mathtt{p\_y}$$

(b) initial approximation



(c) spurious counter-strategy

Fig. 5. A TSL specification (a) with input `x` and cell `y` that is realizable. A winning strategy is to save `x` to `y` as soon as `p(x)` is satisfied. However, the initial approximation (b), that is passed to an LTL synthesis solver, is unrealizable, as proven through the counter-strategy (c) returned by the LTL solver.

$\mathcal{T}^{\mathtt{o}}_{U/\mathtt{id}} = \mathcal{T}^{\mathtt{o}}_{U}$, and let $\mathcal{T}_{U/\mathtt{id}} = \bigcup_{\mathtt{s_o} \in \mathbb{O} \cup \mathbb{C}} \mathcal{T}^{\mathtt{s_o}}_{U/\mathtt{id}}$. We construct the LTL formula φ_{LTL} over the input propositions $\mathcal{T}_P$ and output propositions $\mathcal{T}_{U/\mathtt{id}}$ as follows:

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

Intuitively, the first part of the equation partially reconstructs the semantic meaning of updates by ensuring that a signal is not updated with multiple values at a time. The second part extracts the reactive constraints of the TSL formula without the semantic meaning of functions and updates.

Theorem 1 ([19]). *If φ_{LTL} is realizable, then φ_{TSL} is realizable.*

Note that unrealizability of φ_{LTL} does not imply that φ_{TSL} is unrealizable. It may be that we have not added sufficiently many environment assumptions to the approximation in order for the system to produce a realizing strategy.

Example. As an example, we present a simple TSL specification in Fig. 5a. The specification asserts that the environment provides an input `x` for which the predicate `p x` will be satisfied eventually. The system must guarantee that eventually `p y` holds. According to the semantics of TSL the formula is realizable. The system can take the value of `x` when `p x` is true and save it to `y`, thus guaranteeing that `p y` is satisfied eventually. This is in contrast to LTL, which has no semantics for pure functions - taking the evaluation of `p y` as an environmentally controlled value that does not need to obey the consistency of a pure function.

Refining the LTL Approximation. It is possible that the LTL solver returns a counter-strategy for the environment although the original TSL specification is realizable. We call such a counter-strategy *spurious* as it exploits the additional freedom of LTL to violate the purity of predicates as made possible by the underapproximation. Formally, a counter-strategy is an infinite tree $\pi\colon \mathcal{C}^* \rightarrow 2^{\mathcal{T}_P}$, which provides predicate evaluations in response to possible update assignments of function terms $\tau_F \in \mathcal{T}_F$ to outputs $\mathtt{o} \in \mathbb{O}$. W.l.o.g. we can assume that $\mathbb{O}$, $\mathcal{T}_F$ and $\mathcal{T}_P$ are finite, as they can always be restricted to the outputs and terms that appear in the formula. A counter-strategy is spurious, iff there is a branch $\pi \wr \varsigma$ for some computation $\varsigma \in \mathcal{C}^\omega$, for which the strategy chooses an inconsistent evaluation of two equal predicate terms at different points in time, i.e.,

Algorithm 1. Check-Spuriousness

Input: bound b, counter-strategy $\pi\colon \mathcal{C}^* \to 2^{\mathcal{T}_P}$ (finitely represented using m states)

for all $v \in \mathcal{C}^{m \cdot b}$, $\tau_P \in \mathcal{T}_P$, $t, t' \in \{0, 1, \ldots, m \cdot b - 1\}$ **do**
  if $\eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t, \tau_P) \equiv \eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t', \tau_P) \wedge$
  $\tau_P \in \pi(v_0 \ldots v_{t-1}) \wedge \tau_P \notin \pi(v_0 \ldots v_{t'-1})$ **then**
    $w \leftarrow$ `reduce` (v, τ_P, t, t')
    return $\square\big(\bigwedge_{i=0}^{t-1} \bigcirc^i w_i \wedge \bigwedge_{i=0}^{t'-1} \bigcirc^i w_i \rightarrow (\bigcirc^t \tau_P \leftrightarrow \bigcirc^{t'} \tau_P)\big)$
return ‘‘`non-spurious`’’

$$\exists \varsigma \in \mathcal{C}^\omega.\ \exists t, t' \in \mathbb{N}.\ \exists \tau_P \in \mathcal{T}_P.$$
$$\tau_P \in \pi(\varsigma(0)\varsigma(1)\ldots\varsigma(t-1)) \wedge \tau_P \notin \pi(\varsigma(0)\varsigma(1)\ldots\varsigma(t'-1)) \wedge$$
$$\forall \langle\cdot\rangle\colon \mathbb{F} \rightarrow \mathcal{F}.\ \eta_{\langle\cdot\rangle}(\varsigma, \pi\wr\varsigma, t, \tau_P) = \eta_{\langle\cdot\rangle}(\varsigma, \pi\wr\varsigma, t', \tau_P).$$

Note that a non-spurious strategy can be inconsistent along multiple branches. Due to the definition of realizability the environment can choose function and predicate assignments differently against every system strategy accordingly.

By purity of predicates in TSL the environment is forced to always return the same value for predicate evaluations on equal values. However, this semantic property cannot be enforced implicitly in LTL. To resolve this issue we use the returned counter-strategy to identify spurious behavior in order to strengthen the LTL underapproximation with additional environment assumptions. After adding the derived assumptions, we re-execute the LTL synthesizer to check whether the added assumptions are sufficient in order to obtain a winning strategy for the system. If the solver still returns a spurious strategy, we continue the loop in a CEGAR fashion until the set of added assumptions is sufficiently complete. However, if a non-spurious strategy is returned, we have found a proof that the given TSL specification is indeed unrealizable and terminate.

Algorithm 1 shows how a returned counter-strategy π is checked for being spurious. To this end, it is sufficient to check π against system strategies bounded by the given bound b, as we use bounded synthesis [20]. Furthermore, we can assume w.l.o.g. that π is given by a finite state representation, which is always possible due to the finite model guarantees of LTL. Also note that π, as it is returned by the LTL synthesizer, responds to sequences of sets of updates $(2^{\mathcal{T}_{U/\mathtt{id}}})^*$. However, in our case $(2^{\mathcal{T}_{U/\mathtt{id}}})^*$ is an alternative representation of $\mathcal{C}^*$, due to the additional “single update” constraints added during the construction of φ_{LTL}.

The algorithm iterates over all possible responses $v \in \mathcal{C}^{m \cdot b}$ of the system up to depth $m \cdot b$. This is sufficient, since any deeper exploration would result in a state repetition of the cross-product of the finite state representation of π and any system strategy bounded by b. Hence, the same behaviour could also be generated by a sequence smaller than $m \cdot b$. At the same time, the algorithm iterates over predicates $\tau_P \in \mathcal{T}_P$ appearing in φ_{TSL} and times t and t' smaller than $m \cdot b$. For each of these elements, spuriousness is checked by comparing the output of π for the evaluation of τ_P at times t and t', which should only differ if the inputs to the predicates are different as well. This can only happen, if the

passed input terms have been constructed differently over the past. We check it by using the evaluation function η equipped with the identity assignment $\langle\cdot\rangle_{\mathtt{id}}\colon \mathbb{F} \to \mathbb{F}$, with $\langle \mathtt{f} \rangle_{\mathtt{id}} = \mathtt{f}$ for all $\mathtt{f} \in \mathbb{F}$, and the input sequence $\iota_{\mathtt{id}}$, with $\iota_{\mathtt{id}}(t)(\mathtt{i}) = (t, \mathtt{i})$ for all $t \in \mathbb{N}$ and $\mathtt{i} \in \mathbb{I}$, that always generates a fresh input. Syntactic inequality of $\eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t, \tau_P)$ and $\eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t', \tau_P)$ then is a sufficient condition for the existence of an assignment $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$, for which τ_P evaluates differently at times t and t'.

If spurious behaviour of π could be found, then the revealing response $v \in \mathcal{C}^*$ is first simplified using `reduce`, which reduces v again to a sequence of sets of updates $w \in (2^{\mathcal{T}_{U/\mathtt{id}}})^*$ and removes updates that do not affect the behavior of τ_P at the times t and t' to accelerate the termination of the CEGAR loop. Afterwards, the sequence w is turned into a new assumption that prohibits the spurious behavior, generalized to prevent it even at arbitrary points in time.

As an example of this process, reconsider the spurious counter-strategy of Fig. 5c. Already after the first system response $[\![\mathtt{y} \leftarrowtail \mathtt{x}]\!]$, the environment produces an inconsistency by evaluating $\mathtt{p}\ \mathtt{x}$ and $\mathtt{p}\ \mathtt{y}$ differently. This is inconsistent, as the cell $\mathtt{y}$ holds the same value at time $t = 1$ as the input $\mathtt{x}$ at time $t = 0$. Using Algorithm 1 we generate the new assumption $\square([\![\mathtt{y} \leftarrowtail \mathtt{x}]\!] \rightarrow (\mathtt{p}\ \mathtt{x} \leftrightarrow \bigcirc \mathtt{p}\ \mathtt{y}))$. After adding this strengthening the LTL synthesizer returns a realizability result.

Undecidability. Although we can approximate the semantics of TSL with LTL, there are TSL formulas that cannot be expressed as LTL formulas of finite size.

Theorem 2 ([19]). *The realizability problem of TSL is undecidable.*

6 TSL Synthesis

Our synthesis framework provides a modular refinement process to synthesize executables from TSL specifications, as depicted in Fig. 1. The user initially provides a TSL specification over predicate and function terms. At the end of the procedure, the user receives an executable to control a reactive system.

The first step of our method answers the synthesis question of TSL: if the specification is realizable, then a control flow model is returned. To this end, an intermediate translation to LTL is used, utilizing an LTL synthesis solver that produces circuits in the AIGER format. If the specification is realizable, the resulting control flow model is turned into Haskell code, which is implemented as an independent Haskell module. The user has the choice between two different targets: a module built on Arrows, which is compatible with any Arrowized FRP library, or a module built on Applicative, which supports Applicative FRP libraries. Our procedure generates a single Haskell module per TSL specification. This makes naturally decomposing a project according to individual tasks possible. Each module provides a single component, which is parameterized by their initial state and the pure function and predicate transformations. As soon as these are provided as part of the surrounding project context, a final executable can be generated by compiling the Haskell code.

An important feature of our synthesis approach is that implementations for the terms used in the specification are only required after synthesis. This allows

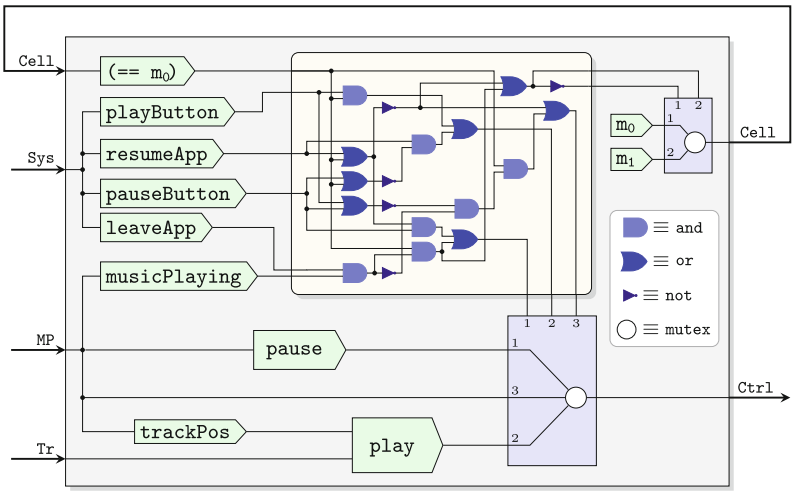


Fig. 6. Example CFM of the music player generated from a TSL specification.

the user to explore several possible specifications before deciding on any term implementations.

Control Flow Model. The first step of our approach is the synthesis of a *Control Flow Model* $\mathcal{M}$ (CFM) from the given TSL specification φ, which provides us with a uniform representation of the control flow structure of our final program.

Formally, a CFM $\mathcal{M}$ is a tuple $\mathcal{M} = (\mathbb{I}, \mathbb{O}, \mathbb{C}, V, \ell, \delta)$, where $\mathbb{I}$ is a finite set of inputs, $\mathbb{O}$ is a finite set of outputs, $\mathbb{C}$ is a finite set of cells, V is a finite set of vertices, $\ell \colon V \to \mathbb{F}$ assigns a vertex a function $\mathtt{f} \in \mathbb{F}$ or a predicate $\mathtt{p} \in \mathbb{P}$, and

$$\delta \colon (\mathbb{O} \cup \mathbb{C} \cup V) \times \mathbb{N} \to (\mathbb{I} \cup \mathbb{C} \cup V \cup \{\bot\})$$

is a dependency relation that relates every output, cell, and vertex of the CFM with $n \in \mathbb{N}$ arguments, which are either inputs, cells, or vertices. Outputs and cells $\mathtt{s} \in \mathbb{O} \cup \mathbb{C}$ always have only a single argument, i.e., $\delta(\mathtt{s}, 0) \not\equiv \bot$ and $\forall m > 0.$ $\delta(\mathtt{s}, m) \equiv \bot$, while for vertices $x \in V$ the number of arguments $n \in \mathbb{N}$ align with the arity of the assigned function or predicate $\ell(x)$, i.e., $\forall m \in \mathbb{N}.\ \delta(x, m) \equiv \bot \leftrightarrow m > n$. A CFM is valid if it does not contain circular dependencies, i.e., on every cycle induced by δ there must lie at least a single cell. We only consider valid CFMs.

An example CFM for our music player of Sect. 2 is depicted in Fig. 6. Inputs $\mathbb{I}$ come from the left and outputs $\mathbb{O}$ leave on the right. The example contains a single cell $\mathtt{c} \in \mathbb{C}$, which holds the stateful memory `Cell`, introduced during synthesis for the module. The green, arrow shaped boxes depict vertices V, which are labeled with functions and predicates names, according to ℓ. For the Boolean decisions that define δ, we use circuit symbols for conjunction, disjunction, and negation. Boolean decisions are piped to a multiplexer gate that selects the respective update streams. This allows each update stream to be passed to an

output stream if and only if the respective Boolean trigger evaluates positively, while our construction ensures mutual exclusion on the Boolean triggers. For code generation, the logic gates are implemented using the corresponding dedicated Boolean functions. After building a control structure, we assign semantics to functions and predicates by providing implementations. To this end, we use Functional Reactive Programming (FRP). Prior work has established Causal Commutative Arrows (CCA) as an FRP language pattern equivalent to a CFM [33,34,53]. CCAs are an abstraction subsumed by other functional reactive programming abstractions, such as Monads, Applicative and Arrows [32,33]. There are many FRP libraries using Monads [11,14,42], Applicative [2,3,23,48], or Arrows [10,39,41,51], and since every Monad is also an Applicative and Applicative/Arrows both are universal design patterns, we can give uniform translations to all of these libraries using translations to just Applicative and Arrows. Both translations are possible due to the flexible notion of a CFM.

In the last step, the synthesized FRP program is compiled into an executable, using the provided function and predicate implementations. This step is not fixed to a single compiler implementation, but in fact can use any FRP compiler (or library) that supports a language abstraction at least as expressive as CCA. For example, instead of creating an Android music player app, we could target an FRP web interface [48] to create an online music player, or an embedded FRP library [23] to instantiate the player on a computationally more restricted device. By using the strong core of CCA, we even can directly implement the player in hardware, which is for example possible with the CλaSH compiler [3]. Note that we still need separate implementations for functions and predicates for each target. However, the specification and synthesized CFM always stay the same.

7 Experimental Results

To evaluate our synthesis procedure we implemented a tool that follows the structure of Fig. 1. It first encodes the given TSL specification in LTL and then refines it until an LTL solver either produces a realizability result or returns a non-spurious counter-strategy. For LTL synthesis we use the bounded synthesis tool BoSy [15]. As soon as we get a realizing strategy it is translated to a corresponding CFM. Then, we generate the FRP program structure. Finally, after providing function implementations the result is compiled into an executable.

To demonstrate the effectiveness of synthesizing TSL, we applied our tool to a collection of benchmarks from different application domains, listed in Table 1. Every benchmark class consists of multiple specifications, addressing different features of TSL. We created all specifications from scratch, where we took care that they either relate to existing textual specifications, or real world scenarios. A short description of each benchmark class is given in [19].

For every benchmark, we report the synthesis time and the size of the synthesized CFM, split into the number of cells ($|\mathbb{C}_{\mathcal{M}}|$) and vertices ($|V_{\mathcal{M}}|$) used. The synthesized CFM may use more cells than the original TSL specification if synthesis requires more memory in order to realize a correct control flow.

Table 1. Number of cells $|\mathbb{C}_{\mathcal{M}}|$ and vertices $|V_{\mathcal{M}}|$ of the resulting CFM $\mathcal{M}$ and synthesis times for a collection of TSL specifications φ. A * indicates that the benchmark additionally has an initial condition as part of the specification.

Benchmark (φ)	$\lvert\varphi\rvert$	$\lvert\mathbb{I}\rvert$	$\lvert\mathbb{O}\rvert$	$\lvert\mathbb{P}\rvert$	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{C}_{\mathcal{M}}\rvert$	$\lvert V_{\mathcal{M}}\rvert$	Synthesis Time (s)
Button								
default	7	1	2	1	3	3	8	0.364
Music App								
simple	91	3	1	4	7	2	25	0.77
system feedback	103	3	1	5	8	2	31	0.572
motivating example	87	3	1	5	8	2	70	1.783
FRPZoo								
scenario_0	54	1	3	2	8	4	36	1.876
scenario_5	50	1	3	2	7	4	32	1.196
scenario_{10}	48	1	3	2	7	4	32	1.161
Escalator								
non-reactive	8	0	1	0	1	2	4	0.370
non-counting	15	2	1	2	4	2	19	0.304
counting	34	2	2	3	7	3	23	0.527
counting*	43	2	2	3	8	4	43	0.621
bidirectional	111	2	2	5	10	3	214	4.555
bidirectional*	124	2	2	5	11	4	287	16.213
smart	45	2	1	2	4	4	159	24.016
Slider								
default	50	1	1	2	4	2	15	0.664
scored	67	1	3	4	8	4	62	3.965
delayed	71	1	3	4	8	5	159	7.194
Haskell-TORCS								
simple	40	5	3	2	16	4	37	0.680
advanced								
gearing	23	4	1	1	3	2	7	0.403
accelerating	15	2	2	2	6	3	11	0.391
steering								
simple	45	2	1	4	6	2	31	0.459
improved	100	2	2	4	10	3	26	1.347
smart	76	3	2	4	8	5	227	3.375

Table 2. Set of programs that use purity to keep one or two counters in range. Synthesis needs multiple refinements of the specification to proof realizability.

Benchmark (φ)	$\lvert\varphi\rvert$	$\lvert\mathbb{I}\rvert$	$\lvert\mathbb{O}\rvert$	$\lvert\mathbb{P}\rvert$	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{C}_{\mathcal{M}}\rvert$	$\lvert V_{\mathcal{M}}\rvert$	Refinements	Synthesis Time (s)
inrange-single	23	2	1	2	4	2	21	3	0.690
inrange-two	51	3	3	4	7	4	440	6	173.132
graphical-single	55	2	3	2	6	4	343	9	1767.948
graphical-two	113	3	5	4	9	-	-	-	¿ 10000

The synthesis was executed on a quad-core Intel Xeon processor (E3-1271 v3, 3.6GHz, 32 GB RAM, PC1600, ECC), running Ubuntu 64bit LTS 16.04.

The experiments of Table 1 show that TSL successfully lifts the applicability of synthesis from the Boolean domain to arbitrary data domains, allowing for new applications that utilize every level of required abstraction. For all benchmarks we always found a realizable system within a reasonable amount of time, where the results often required synthesized cells to realize the control flow behavior.

We also considered a preliminary set of benchmarks that require multiple refinement steps to be synthesizable. An overview of the results is given in Table 2. The benchmarks are inspired by examples of the Reactive Banana FRP library [2]. Here, purity of function and predicate applications must be utilized by the system to ensure that the value of one or two counters never goes out of range. Thereby, the system not only needs purity to verify this condition, but also to take the correct decisions in the resulting implementation to be synthesized.

8 Related Work

Our approach builds on the rich body of work on reactive synthesis, see [17] for a survey. The classic reactive synthesis problem is the construction of a finite-state machine that satisfies a specification in a temporal logic like LTL. Our approach differs from the classic problem in its connection to an actual programming paradigm, namely FRP, and its separation of control and data.

The synthesis of *reactive programs*, rather than finite-state machines, has previously been studied for standard temporal logic [21,35]. Because there is no separation of control and data, these approaches do not directly scale to realistic applications. With regard to FRP, a *Curry-Howard correspondence* between LTL and FRP in a dependently typed language was discovered [28,29] and used to prove properties of FRP programs [8,30]. However, our paper is the first, to the best of our knowledge, to study the synthesis of FRP programs from temporal specifications.

The idea to separate control and data has appeared, on a smaller scale, in the synthesis with *identifiers*, where identifiers, such as the number of a client in a mutual exclusion protocol, are treated symbolically [13]. *Uninterpreted functions* have been used to abstract data-related computational details in the synthesis of synchronization primitives for complex programs [5]. Another connection to other synthesis approaches is our CEGAR loop. Similar *refinement loops* also appear in other synthesis appraches, however with a different purpose, such as the refinement of environment assumptions [1].

So far, there is no immediate connection between our approach and the substantial work on *deductive* and *inductive synthesis*, which is specifically concerned with the data-transformation aspects of programs [16,31,40,47,49,50]. Typically, these approaches are focussed on non-reactive sequential programs. An integration of deductive and inductive techniques into our approach for reactive systems is a very promising direction for future work. Abstraction-based synthesis [4,12,24,37] may potentially provide a link between the approaches.

9 Conclusions

We have introduced Temporal Stream Logic, which allows the user to specify the control flow of a reactive program. The logic cleanly separates control from complex data, forming the foundation for our procedure to synthesize FRP programs. By utilizing the purity of function transformations our logic scales independently of the complexity of the data to be handled. While we have shown that scalability comes at the cost of undecidability, we addressed this issue by using a CEGAR loop, which lazily refines the underapproximation until either a realizing system implementation or an unrealizability proof is found.

Our experiments indicate that TSL synthesis works well in practice and on a wide range of programming applications. TSL also provides the foundations for further extensions. For example, a user may want to fix the semantics for a subset of the functions and predicates. Such refinements can be implemented as part of a much richer *TSL Modulo Theory* framework.

References

1. Alur, R., Moarref, S., Topcu, U.: Counter-strategy guided refinement of GR(1) temporal logic specifications. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 26–33. IEEE (2013). http://ieeexplore.ieee.org/document/6679387/
2. Apfelmus, H.: Reactive-banana. Haskell library (2012). http://www.haskell.org/haskellwiki/Reactive-banana
3. Baaij, C.: Digital circuit in CλaSH: functional specifications and type-directed synthesis. Ph.D. thesis, University of Twente, January 2015. https://doi.org/10.3990/1.9789036538039, eemcs-eprint-23939
4. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: Jagannathan and Sewell [26], pp. 221–234. https://doi.org/10.1145/2535838.2535860, http://doi.acm.org/10.1145/2535838.2535860
5. Bloem, R., Hofferek, G., Könighofer, B., Könighofer, R., Ausserlechner, S., Spork, R.: Synthesis of synchronization using uninterpreted functions. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014, pp. 35–42. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987593
6. Bloem, R., Jacobs, S., Khalimov, A.: Parameterized synthesis case study: AMBA AHB. In: Chatterjee, K., Ehlers, R., Jha, S. (eds.) Proceedings 3rd Workshop on Synthesis, SYNT 2014. EPTCS, Vienna, Austria, 23–24 July 2014, vol. 157, pp. 68–83 (2014). https://doi.org/10.4204/EPTCS.157.9
7. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012). https://doi.org/10.1016/j.jcss.2011.08.007
8. Cave, A., Ferreira, F., Panangaden, P., Pientka, B.: Fair reactive programming. In: Jagannathan and Sewell [26], pp. 361–372. https://doi.org/10.1145/2535838.2535881, http://doi.acm.org/10.1145/2535838.2535881
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643

10. Courtney, A., Nilsson, H., Peterson, J.: The yampa arcade. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, 28 August 2003, pp. 7–18. ACM, (2003). https://doi.org/10.1145/871895.871897, http://doi.acm.org/10.1145/871895.871897
11. Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for Guis. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013, pp. 411–422. ACM (2013). https://dl.acm.org/citation.cfm?doid=2462156.2462161, http://doi.acm.org/10.1145/2462156.2462161
12. Dimitrova, R., Finkbeiner, B.: Counterexample-guided synthesis of observation predicates. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 107–122. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33365-1_9
13. Ehlers, R., Seshia, S.A., Kress-Gazit, H.: Synthesis with identifiers. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 415–433. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_23
14. Elliott, C., Hudak, P.: Functional reactive animation. In: Jones, S.L.P., Tofte, M., Berman, A.M. (eds.) Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP 1997), Amsterdam, The Netherlands, 9–11 June 1997, pp. 263–273. ACM (1997). https://doi.org/10.1145/258948.258973, http://doi.acm.org/10.1145/258948.258973
15. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: an experimentation framework for bounded synthesis. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 325–332. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_17
16. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Grove and Blackburn [22], pp. 229–239. https://doi.org/10.1145/2737924.2737977, http://doi.acm.org/10.1145/2737924.2737977
17. Finkbeiner, B.: Synthesis of reactive systems. In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Dependable Software Systems Engineering. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 45, pp. 72–98. IOS Press (2016)
18. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Vehicle platooning simulations with functional reactive programming. In: Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, 21 April 2017, pp. 43–47. ACM, (2017). https://doi.org/10.1145/3055378.3055385, http://doi.acm.org/10.1145/3055378.3055385
19. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: Synthesis beyond the bools. CoRR abs/1712.00246 (2019). http://arxiv.org/abs/1712.00246
20. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT **15**(5–6), 519–539 (2013). https://doi.org/10.1007/s10009-012-0228-z
21. Gerstacker, C., Klein, F., Finkbeiner, B.: Bounded synthesis of reactive programs. In: Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, 7–10 October 2018, Proceedings, pp. 441–457 (2018). https://doi.org/10.1007/978-3-030-01090-4_26
22. Grove, D., Blackburn, S. (eds.): Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015. ACM (2015). http://dl.acm.org/citation.cfm?id=2737924

23. Helbling, C., Guyer, S.Z.: Juniper: a functional reactive programming language for the arduino. In: Janin and Sperber [27], pp. 8–16. https://doi.org/10.1145/2975980.2975982, http://doi.acm.org/10.1145/2975980.2975982
24. Hsu, K., Majumdar, R., Mallik, K., Schmuck, A.K.: 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), pp. 120–129. ACM (2018)
25. Jacobs, S., et al.: The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants and results. In: SYNT 2017. EPTCS, vol. 260, pp. 116–143 (2017). https://doi.org/10.4204/EPTCS.260.10
26. Jagannathan, S., Sewell, P. (eds.): The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014. ACM (2014). http://dl.acm.org/citation.cfm?id=2535838
27. Janin, D., Sperber, M. (eds.): Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design, FARM@ICFP 2016, Nara, Japan, 24 September 2016. ACM (2016). https://doi.org/10.1145/2975980, http://doi.acm.org/10.1145/2975980
28. Jeffrey, A.: LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In: Claessen, K., Swamy, N. (eds.) Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, 24 January 2012, pp. 49–60. ACM (2012). https://doi.org/10.1145/2103776.2103783, http://doi.acm.org/10.1145/2103776.2103783
29. Jeltsch, W.: Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. Electr. Notes Theor. Comput. Sci. **286**, 229–242 (2012). https://doi.org/10.1016/j.entcs.2012.08.015
30. Krishnaswami, N.R.: Higher-order functional reactive programming without space-time leaks. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA, 25–27 September 2013, pp. 221–232. ACM (2013). https://doi.org/10.1145/2500365.2500588, http://doi.acm.org/10.1145/2500365.2500588
31. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Comfusy: a tool for complete functional synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 430–433. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_38
32. Lindley, S., Wadler, P., Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electr. Notes Theor. Comput. Sci. **229**(5), 97–117 (2011). https://doi.org/10.1016/j.entcs.2011.02.018
33. Liu, H., Cheng, E., Hudak, P.: Causal commutative arrows. J. Funct. Program. **21**(4–5), 467–496 (2011). https://doi.org/10.1017/S0956796811000153
34. Liu, H., Hudak, P.: Plugging a space leak with an arrow. Electr. Notes Theor. Comput. Sci. **193**, 29–45 (2007). https://doi.org/10.1016/j.entcs.2007.10.006
35. Madhusudan, P.: Synthesizing reactive programs. In: Bezem, M. (ed.) Computer Science Logic, 25th International Workshop/20th Annual Conference of the EACSL, CSL 2011, Bergen, Norway, 12–15 September 2011, Proceedings. LIPIcs, vol. 12, pp. 428–442. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011). https://doi.org/10.4230/LIPIcs.CSL.2011.428
36. Mainland, G. (ed.): Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, 22–23 September 2016. ACM (2016). https://doi.org/10.1145/2976002, http://doi.acm.org/10.1145/2976002

37. Mallik, K., Schmuck, A.K., Soudjani, S., Majumdar, R.: Compositional abstraction-based controller synthesis for continuous-time systems. arXiv preprint arXiv:1612.08515 (2016)
38. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980). https://doi.org/10.1145/357084.357090
39. Murphy, T.E.: A livecoding semantics for functional reactive programming. In: Janin and Sperber [27], pp. 48–53. https://doi.org/10.1145/2975980.2975986http://doi.acm.org/10.1145/2975980.2975986
40. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Grove and Blackburn [22], pp. 619–630. https://doi.org/10.1145/2737924.2738007, http://doi.acm.org/10.1145/2737924.2738007
41. Perez, I., Bärenz, M., Nilsson, H.: Functional reactive programming, refactored. In: Mainland [36], pp. 33–44. https://doi.org/10.1145/2976002.2976010, http://doi.acm.org/10.1145/2976002.2976010
42. van der Ploeg, A., Claessen, K.: Practical principled FRP: forget the past, change the future, FRPNow! In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, 1–3 September 2015, pp. 302–314. ACM (2015). https://doi.org/10.1145/2784731.2784752, http://doi.acm.org/10.1145/2784731.2784752
43. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
44. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0035790
45. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Am. Math. Soc. **52**(4), 264–268 (1946). http://projecteuclid.org/euclid.bams/1183507843
46. Shan, Z., Azim, T., Neamtiu, I.: Finding resume and restart errors in android applications. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, 30 October–4 November 2016, pp. 864–880. ACM (2016). https://doi.org/10.1145/2983990.2984011, http://doi.acm.org/10.1145/2983990.2984011
47. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013). https://doi.org/10.1007/s10009-012-0249-7
48. Trinkle, R.: Reflex-frp (2017). https://github.com/reflex-frp/reflex
49. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. STTT **15**(5–6), 413–431 (2013). https://doi.org/10.1007/s10009-012-0232-3
50. Wang, X., Dillig, I., Singh, R.: Synthesis of data completion scripts using finite tree automata. PACMPL 1(OOPSLA), 62:1–62:26 (2017). https://doi.org/10.1145/3133886, http://doi.acm.org/10.1145/3133886
51. Winograd-Cort, D.: Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming. Ph.D. thesis, Yale University, December 2015. http://www.danwc.com/s/dwc-yale-formatted-dissertation.pdf
52. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Synthesis of control protocols for autonomous systems. Unmanned Syst. **1**(01), 21–39 (2013)

53. Yallop, J., Liu, H.: Causal commutative arrows revisited. In: Mainland [36], pp. 21–32. https://doi.org/10.1145/2976002.2976019, http://doi.acm.org/10.1145/2976002.2976019