Synthesizing Functional Reactive Programs

Bernd Finkbeiner
Saarland University
Germany

Felix Klein
Saarland University
Germany

Ruzica Piskac
Yale University
CT, USA

Mark Santolucito
Yale University
CT, USA

Abstract

Functional Reactive Programming (FRP) is a paradigm that has simplified the construction of reactive programs. There are many libraries that implement incarnations of FRP, using abstractions such as Applicative, Monads, and Arrows. However, finding a good control flow, that correctly manages state and switches behaviors at the right times, still poses a major challenge to developers.

An attractive alternative is specifying the behavior instead of programming it, as made possible by the recently developed logic: Temporal Stream Logic (TSL). However, it has not been explored so far how Control Flow Models (CFMs), resulting from TSL synthesis, are turned into executable code that is compatible with libraries building on FRP. We bridge this gap, by showing that CFMs are a suitable formalism to be turned into Applicative, Monadic, and Arrowized FRP.

We demonstrate the effectiveness of our translations on a real-world kitchen timer application, which we translate to a desktop application using the Arrowized FRP library `Yampa`, a web application using the Monadic `Threepenny-GUI` library, and to hardware using the Applicative hardware description language `ClaSH`.

CCS Concepts • **Software and its engineering**;

Keywords Reactive Synthesis, TSL, FRP

ACM Reference Format:
Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Synthesizing Functional Reactive Programs. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3331545.3342601

Haskell '19, August 22–23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342601

1 Introduction

Reactive programs implement a broad class of computer systems whose defining element is the continued interaction between the system and its environment. Their importance can be seen through the wide range of applications, such as embedded devices [Helbling and Guyer 2016], games [Perez 2017], robotics [Jing et al. 2016], hardware circuits [Khalimov et al. 2014], GUIs [Czaplicki and Chong 2013], and interactive multimedia [Santolucito et al. 2015].

Functional Reactive Programming (FRP) [Courtney et al. 2003; Elliott and Hudak 1997] is a paradigm for writing programs for reactive systems. The fundamental idea of FRP is to extend the classic building blocks of functional programming with the abstraction of a *signal* to describe values varying over time. In contrast to sequential programs being executed step by step, FRP programs lead to stream processing networks that manage state and switch between behaviors dependent on the user input. FRP programs can be exceptionally efficient. For example, a network controller recently implemented as an FRP program on a multicore processor outperformed all its contemporary competing implementations [Voellmy et al. 2013].

Building a reactive program is a complex process, of which the most difficult part is finding a good and correct high-level design [Piterman et al. 2006]. Furthermore, even once this design has been fixed, implementing the system still remains a highly error-prone process [Shan et al. 2016]. While FRP helps with the latter problem by embedding the concept of time into the type system, it still leaves the challenge of switching between behaviors and managing state efficiently open. The use of temporal logic has been explored to test properties of FRP programs [Perez and Nilsson 2017], however testing still leaves space for possible errors.

A solution for solving the design challenge has been proposed with Temporal Stream Logic [Finkbeiner et al. 2019], a specification logic to specify the temporal control flow behavior of a program. The logic enforces a clean separation between control and data transformations, which also can be leveraged in FRP [Elliott and Hudak 1997]. Temporal Stream Logic (TSL) is used in combination with a reactive synthesis engine to automatically create an abstract model of temporal control called a Control Flow Model (CFM) satisfying the

```
yampaButton :: SF (Event MouseClick) Picture
yampaButton = proc click → do
  rec
    count    <- init 0 -< newCount
    newCount <- arr f1 -< (click, count)
    pic      <- arr f2 -< count
  returnA -< pic

f1 :: (Event MouseClick, Int) → Int
f1 (click, count)
  | isEvent click = count + 1
  | otherwise     = count

f2 :: Int → Picture
f2 = text . show
```

Figure 1. A button written with the FRP library Yampa.

given specification. TSL combines Boolean and temporal operations with *predicates* $\mathsf{p}\ \mathsf{s}_\mathsf{i}$, evaluated on inputs s_i, and *updates* denoted by $[\![\mathsf{s}_\mathsf{o} \leftarrowtail \mathsf{f}\ \mathsf{s}_\mathsf{i}]\!]$, which map pure functions f to inputs s_i and pipe the result to an output s_o. An example for a TSL specification is given by

$$\square\Big((\mathsf{event\ click} \leftrightarrow [\![\mathsf{count} \leftarrowtail \mathsf{increment\ count}]\!]) \\ \wedge [\![\mathsf{screen} \leftarrowtail \mathsf{display\ count}]\!]\Big)$$

which states that the counter must always be incremented if and only if there is a click event, and that the value of the counter is displayed on a screen.

An implementation that satisfies the specified behavior, built using the FRP library Yampa [Courtney et al. 2003], is shown in Fig. 1. The program implements a button in a GUI which shows a counter value that increments with every click. The Yampa FRP library uses an abstraction called arrows [Hughes 2000], where the arrows define the structure and connection between functions [Liu and Hudak 2007]. As mentioned before, they can be used to cleanly separate data transformations into pure functions, creating a visually clear separation between the control flow and the data level. In the example program of Fig. 1, this separation is clearly visible. The `yampaButton` is the "arrow" part of the code, which defines a control flow. The functions `f1` and `f2` are the "functional" part, describing pure data transformations.

In the TSL specification, function applications, like `click`, `increment` or `display`, are not tied to a particular implementation. Instead, the synthesis engine ensures that the specification is satisfied for all possible implementations that may be bound to these placeholders, similar to an unknown polymorphic function that is used as an argument in a functional program. Thus, the implementation of Fig. 1 indeed satisfies the given specification by implementing `event` with `isEvent` of the `yampa` library, `increment` with `(+1)`, and `display` with `text` of the `gloss` library and `show`.

The immediate advantage of synthesis over manual programming is that, if the synthesis succeeds, then there is a guarantee that the constructed program satisfies the specification. Sometimes, the synthesis does not succeed, and this also leads to interesting results. An example is given by the FRPZoo [Gélineau 2016] study, which consists of implementations for the same program for 16 different FRP libraries. The program consists of two buttons that can be clicked on by the user: a `counter` button, which keeps track of the number of clicks, and a `toggle` button, which turns the counter button on and off. To our surprise, after translating the written-English specification from the FRPZoo website into a formal TSL specification, the synthesis procedure was not able to synthesize a satisfying program. By inspecting the output of the synthesis tool, we noticed that the specification is actually unrealizable. The problem is that the specification requires the counter to be incremented whenever the `counter` button is clicked. At the same time, the counter must be reset to zero whenever the `toggle` button is clicked. This creates a conflict when both buttons are clicked together. To obtain a solution, we had to add the assumption that both buttons are never pressed simultaneously.

While the work of Finkbeiner et al. 2019 discusses the synthesis process for creating the CFM in detail, it does not elaborate on how a CFM is actually turned into FRP code, which is necessary to finally generate an executable. In this work we explore this process, and show how Causal Commutative Arrows (CCA) form a foundational abstraction for FRP in the context of program synthesis. From this connection between CCA and a CFM, we build a system to generate library-independent FRP across a range of FRP abstractions.

There is no single style of FRP which is generally accepted as canonical. Instead, FRP is realized through a number of libraries, which are based on fundamentally different abstractions, such as Monadic FRP [Apfelmus 2013; Ploeg and Claessen 2015; Trinkle 2017], Arrowized FRP [Courtney et al. 2003; Winograd-Cort 2015], and Applicative FRP [Apfelmus 2012; Baaij 2015].

We show that our system is flexible enough to handle all of these abstractions, by demonstrating translations from a CFM to `Threepenny-GUI` [Apfelmus 2013], `Yampa` [Courtney et al. 2003], and `ClaSH` [Baaij 2015]. The CFM used to generate the code is synthesized from a TSL specification that describes the behavior of a kitchen timer application. It was obtained from a real-world kitchen timer, as depicted in Fig. 2, that is retailed by the German company *Dirk Rosssmann GmbH*.

We do not envision FRP synthesis as a replacement for FRP libraries, but rather as a complement. Through synthesis and code generation, users automatically construct FRP programs in these libraries that provide critical interfaces to input/output domains. Furthermore, we show that TSL synthesis generates code as expressive as CCA. While this power is sufficient for many applications, the FRP libraries still provide an interface to more powerful language abstraction features, in case the user chooses to use them.

Figure 2. Real-world kitchen timer that is retailed by the company *Dirk Rosssmann GmbH*.

In summary, the paper makes the following contributions:

1. We describe the process of automatically generating library-independent FRP control code from TSL specifications.
2. We examine the relation between CFMs and CCA, and compare the differences between various FRP abstractions during the translation process.
3. We demonstrate our translations on a kitchen timer application, built as a desktop application using the Arrowized FRP library `Yampa`, as a web application using the Monadic library `Threepenny-GUI`, and to hardware using the Applicative hardware description language `ClaSH`.
4. We provide an open-source tool for the synthesis of FRP programs from TSL specifications[1]

2 Preliminaries

We assume time to be discrete and denote it by the set $\mathcal{T}ime$ of positive integers. A value is an arbitrary object of arbitrary type, where we use $\mathcal{V}$ to denote the set of all values. We consider the Boolean values $\mathcal{B} \subseteq \mathcal{V}$ as a special subset, which are either `true` $\in \mathcal{B}$ or `false` $\in \mathcal{B}$.

A signal $s : \mathcal{T}ime \to \mathcal{V}$ is a function fixing a value at each point in time. The set of all signals is denoted by $\mathcal{S}$, usually partitioned into input signals $\mathcal{I}$ and output signals $\mathcal{O}$.

An n-ary function $f : \mathcal{V}^n \to \mathcal{V}$ determines a new value from n given values. We denote the set of all functions (of arbitrary arity) by $\mathcal{F}$. Constants are functions of arity zero, while every constant is also a value, i.e, an element of $\mathcal{F} \cap \mathcal{V}$. An n-ary predicate $p : \mathcal{V}^n \to \mathcal{B}$ checks a truth statement on n given values. The set of all predicates (of arbitrary arity) is denoted by $\mathcal{P}$.

[1] All source code is available at https://github.com/reactive-systems/tsltools

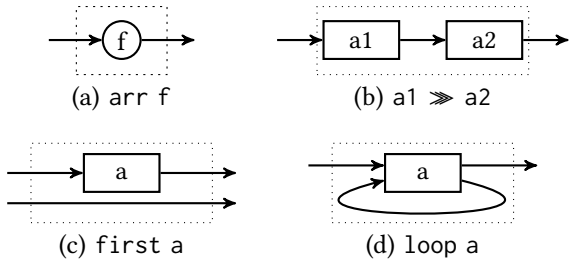

Figure 3. The core Arrow operators. Others, like `second`, are built from these.

2.1 Functional Reactive Programming

FRP is a programming paradigm that uses the abstractions available in functional programming to create an abstraction of time. The core abstraction in FRP is that of a signal

$$Signal\ a = \mathcal{T}ime \to a$$

which produces values of some arbitrary type a over time. The type a can be an input from the world, such as the current position of the mouse, or an output type, such as some text that should be rendered to the screen. Signals are also used internally to manipulate values over time, for example if the position of the mouse should be rendered to the screen.

Arrows There are many incarnations of FRP, which use various abstraction to manipulate signals over time. One popular abstraction for FRP is a Monad, but a weaker abstraction, called Arrows, is also used in many modern libraries [Murphy 2016; Perez et al. 2016]. The Arrow abstraction describes a computation connecting inputs and outputs in a single type [Hughes 2000]. Hence, an Arrow type also allows us to describe reactive programs that process inputs and produce outputs over time.

Arrowized FRP was introduced to plug a space leak in the original FRP work [Elliott and Hudak 1997; Liu and Hudak 2007]. By using the Arrow abstraction introduced in [Hughes 2000], which describes in a single type inputs and outputs, we can also describe reactive programs that process inputs and produce outputs over time. At the top level, an Arrowized FRP program will have the form

$$SF\ Input\ Output = Signal\ Input \to Signal\ Output$$

which is a signal function type, parametrized by the type of input from the world and the type of output to the world. The core Arrow operators, shown in Fig. 3, are used to compose multiple arrows into larger programs.

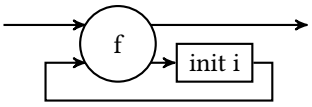

Figure 4. `loopD i f`: a special loop from CCA that is initialized with a user provided value `i`.

The abstractions used in different implementations of FRP vary in expressive power. Arrowized FRP has a smaller interface than Monadic FRP [Lindley et al. 2011] preventing particular constructs that caused the aforementioned space leak. This is also useful when choosing a core language to synthesize, as we are able to simulate Arrowized FRP programs in most Monadic libraries.

CCA We target a restricted set of arrows called Causal Commutative Arrows (CCA) [Liu et al. 2011; Yallop and Liu 2016]. Specifically, CCA adds additional laws to arrows that constrain their behavior and the type of state they may retain. Of particular interest is that CCA also introduces a special initialization operator, `init`. This `init` operator allows for `loopD`, a loop that includes initialization, as shown in Fig. 4.

We use CCA as a minimal language for synthesis. Our synthesis is able to support any FRP library which is at least as powerful as CCA. Because CCA is again restricted in its interface, there are more libraries that can simulate CCA FRP than Arrowized FRP in general. This makes our synthesis procedure possible for an even wider range of application scenarios. We revisit the implications of CCA as our choice of computation abstraction in Section 4.1.

2.2 Reactive Synthesis

The synthesis of a reactive system concerns the process of automatically generating an implementation from a high-level specification. The reactive system acts as a deterministic controller, which reads inputs and produces outputs over an infinite amount of time. In contrast, a specification defines all input/output behavior pairs that are valid, i.e., allowed to be produced by the controller.

In the classical synthesis setting, time is discrete and inputs and outputs are given as vectors of Boolean signals. The standard abstraction treats inputs and outputs as atomic propositions $I \cup O$, while their Boolean combinations form an alphabet $\Sigma = 2^{I \cup O}$ of alphabet symbols. This fixes the behavior of the system to infinite sequences $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ of alphabet symbols σ_t. At every time t signals appearing in the set σ_t are enabled (`true`), while signals not in σ_t are disabled (`false`). The set of all such sequences is denoted by Σ^ω, where the ω-operator induces the infinite concatenation of alphabet symbols of Σ. A deterministic solution links exactly one output sequence $\beta \in (2^O)^\omega$ to every possible input sequence $\alpha \in (2^I)^\omega$, i.e., it is a total function $f : (2^I)^\omega \to (2^O)^\omega$. A specification describes an arbitrary relation between input sequences $\alpha \in (2^I)^\omega$ and output sequences $\beta \in (2^O)^\omega$.

2.3 Connections between FRP and Reactive Systems

A first inspection reveals that FRP fits into the definition of a reactive system, as given in Section 2.2: an FRP program reads an infinite stream of input signals and finally produces a corresponding infinite output stream. Nevertheless, FRP does not fit into the classical setting used for reactive systems, as the input and output streams in FRP are allowed to have arbitrary types.

To solve this problem, one could restrict FRP to just streams of enumerative types, which then are reduced to a Boolean representation. However, this would drop the necessity of almost all interesting features of FRP and it is questionable whether this restricted notion of FRP would give any benefits against Mealy/Moore automata or circuits, which are already used for reactive systems. Additionally, it just creates an exponential blowup and does not provide any insights into the core problem.

Hence, it is more reasonable and interesting to ask whether it is possible to natively handle streams of arbitrary type within reactive systems. Recall that FRP includes functional behavior, defined using standard functional paradigms, but also a control structure, defined via arrows and loops. We will target synthesis of the control structure, leaving the functional level synthesis to tools such as Myth [Kuncak et al. 2010; Osera and Zdancewic 2015] or Synquid [Polikarpova et al. 2016].

2.4 Temporal Stream Logic

We use Temporal Stream Logic (TSL) for specifying the control flow behavior of functional reactive programs [Finkbeiner et al. 2019]. TSL has been especially designed for synthesis and describes control flow properties with respect to their temporal behavior. If a TSL specification is realizable it can be turned into a Control Flow Model (CFM), an abstract representation of the FRP network that covers all possible behavior switches.

Temporal Stream Logic builds on the notion of *updates*, such as $[\![\mathtt{y} \leftarrowtail \mathtt{f\ x}]\!]$, which expresses that the result of mapping the pure function `f` to some input stream `x` is piped to the output stream `y`. The execution of an update is coupled with predicate evaluations guiding the control flow decisions of the network. In combination with Boolean and temporal operators, the logic allows for expressing even complex temporally evolving FRP networks using only a short, but precise description of the temporal behavior.

TSL specifications implicitly induce an architecture as shown in Fig. 5. As a basis, the syntax of TSL uses a term based notion, built from input streams $\mathtt{i} \in \mathbb{I}$, output streams $\mathtt{o} \in \mathbb{O}$, memory cells $\mathtt{c} \in \mathbb{C}$, and function and predicate literals $\mathtt{f} \in \mathbb{F}$ and $\mathtt{p} \in \mathbb{P}$ with $\mathbb{P} \subseteq \mathbb{F}$, respectively. The purpose of cells is to memorize data values, output to a cell at time $t \in \mathcal{T}ime$, to provide them again as inputs at time $t + 1$.

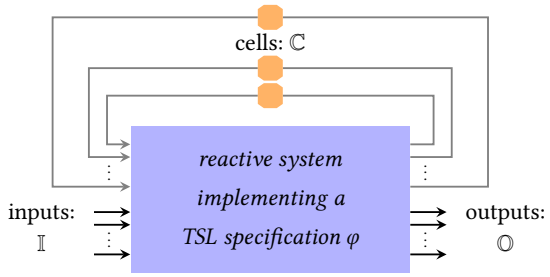

Figure 5. The TSL system architecture.

We differentiate between function terms $\tau_F \in \mathcal{T}_F$ and predicate terms $\tau_P \in \mathcal{T}_P$, built according to the following grammar

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

where $\mathtt{s_i} \in \mathbb{I} \cup \mathbb{C}$ is either an input stream or a cell. In a TSL formula φ, predicate and function terms are then combined to updates with Boolean connectives and temporal operators (as well as the standard derived Boolean and temporal operators)

$$\varphi \;:=\; \tau_P \mid [\![\mathtt{s_o} \leftarrowtail \tau_F]\!] \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\, \mathcal{U}\, \varphi$$

where $\mathtt{s_o} \in \mathbb{O} \cup \mathbb{C}$ is either an output signal or a cell.

For giving semantics to TSL formulas φ, a universally quantified assignment function $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$ is used. The assignment function fixes an implementation for each predicate and function literal, as well as input streams $\iota\colon \mathbb{I} \to \mathcal{S}$. We will only give an intuitive description of the semantics here. For a fully formal definition, the interested reader is referred to [Finkbeiner et al. 2019]. Intuitively, the semantics of TSL can be summarized as follows:

Predicate terms are evaluated to either `true` or `false`, by first selecting implementations for all function and predicate literals according to $\langle\cdot\rangle$, and then applying them to the inputs, as given by ι, and cells, using the stored value at the current time t. The content of a cell thereby is fixed iteratively by selecting the past values piped into the cell over time. Cells are initialized using a special constant provided as part of $\langle\cdot\rangle$.

Function terms are evaluated similar to predicate terms, except that they can evaluate to any value of arbitrary type.

Updates are used to pipe the result of function term evaluations to output streams or cells. Updates, as they appear in a TSL formula, are typed as Boolean expressions. In that sense, update expressions are used in TSL formulas to state that a specific flow is executed at a specific point in time, where the expression evaluates to `true`, if it is executed and to `false`, otherwise. The semantics of TSL ensure that different updates to the same output or cell are always mutual exclusive, *e.g.*, the expression $[\![\mathtt{o} \leftarrowtail \mathtt{f\ x}]\!] \wedge [\![\mathtt{o} \leftarrowtail \mathtt{g\ x}]\!]$ is never satisfied.

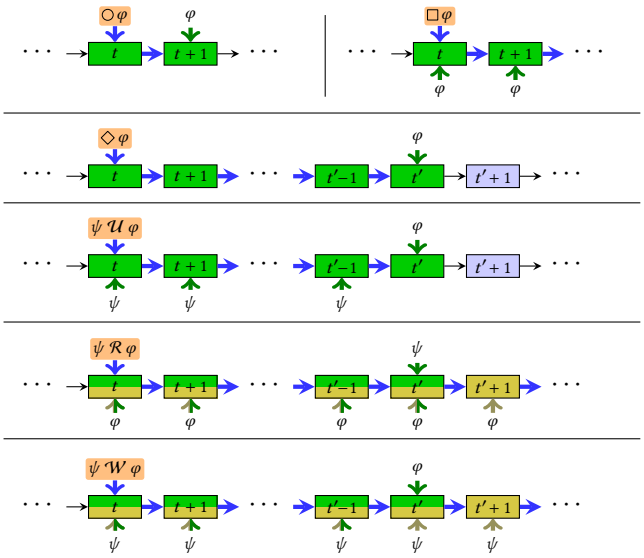

Figure 6. Temporal behavior of the operators *next*, *always*, *finally*, *until*, *release*, and *weak until*. In case of *release* and *weak until*, the formula is either fulfilled by satisfying the top behavior (green) or the bottom behavior (yellow). The blue arrows on the time axis indicate the temporal scope of the operators over time.

Boolean operators like *negation* $[\neg]$ and *conjunction* $[\wedge]$, and ***temporal operators*** like *next* $[\bigcirc]$ and *until* $[\mathcal{U}]$ have standard semantics. We use the standard derived operators, *e.g.*, *disjunction* $[\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)]$, *implication* $[\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi]$ and *equivalence* $[\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)]$. For the temporal operators, intuitive behavior is given in Fig. 6, including derived operators like *release* $[\varphi\,\mathcal{R}\,\psi \equiv \neg((\neg\psi)\,\mathcal{U}\,(\neg\varphi))]$, *finally* $[\Diamond\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi]$, *always* $[\Box\varphi \equiv \mathit{false}\,\mathcal{R}\,\varphi]$, and the *weak* version of *until* $[\varphi\,\mathcal{W}\,\psi \equiv (\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)]$. The precedence order of the listed operators matches the listed order, except that $\Box$ and $\Diamond$ have higher precedence that $\mathcal{U}$ and $\mathcal{R}$.

The synthesis problem of creating a CFM $\mathcal{M}$ satisfying a TSL specification φ then is formalized by

$$\exists \mathcal{M}.\ \forall \iota.\ \forall \langle\cdot\rangle.\ \mathcal{M}\wr\iota, \iota \models_{\langle\cdot\rangle} \varphi$$

where $\mathcal{M}\wr\iota$ denotes the output produced by $\mathcal{M}$ under the input ι. Note that the CFM $\mathcal{M}$ must satisfy the specification for all possibly chosen function and predicate implementations, as selected by $\langle\cdot\rangle$, and all possible inputs ι, which is the reason for the synthesis problem being undecidable in general.

Theorem 2.1 ([Finkbeiner et al. 2019]). *The synthesis problem of TSL is undecidable.*

A useful advantage of TSL in contrast to other specification logics is that function and predicate names, as used by the

specification, are only considered as symbolic literals. That is, TSL formulas are universally quantified over the function and predicate terms that appear in the formula. Therefore, the logic guarantees that synthesized systems satisfy the specified behavior for all possible implementations of these function and predicate literals. The literals are still classified according to their arity, i.e., the number of other function terms they are applied to, as well as by their type: input, output, cell, function or predicate. Thus, they can be considered similar to a function, passed as an argument, of polymorphic type. In this sense, TSL specifications fix the type of the network, but not the type of the data, which is still polymorphic. Type information of the data only needs to be provided after synthesis through the instantiation of function and predicate literals with respective implementations.

If synthesis is successful, then it creates a *Control Flow Model* (CFM) that satisfies the specified behavior. 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, and $\ell \colon V \to (\mathbb{F} \cup \mathbb{L} \cup \mathbb{U})$ assigns each vertex a signal function (either a function $\mathsf{f} \in \mathbb{F}$, a predicate $\mathsf{p} \in \mathbb{P}$, a logic operator in $\mathbb{L}$ lifted to the signal level, or a mutex selector $\mathbb{U}$ lifted to the signal level). The set of logic operators $\mathbb{L}$ contains the standard Boolean operators, and the mutex selectors $\mathbb{U}$ are signal functions, pattern matching on one input signal to select among the other input signals for output. Finally, a CFM also contains a dependency relation

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

relating every output, cell, and vertex to a set of inputs, cells, or vertices. The dependency relations defines the wiring between signal functions. The selector $n \in \mathbb{N}$ argument allows us to specify a specific connection, since a signal function may have multiple inputs. Outputs and cells $\mathsf{s} \in \mathbb{O} \cup \mathbb{C}$ always have only a single input signal stream, so the first selector has some non-bottom value ($\delta(s, 0) \not\equiv \bot$) and any larger selector is undefined ($\forall m > 0.\ \delta(\mathsf{s}, m) \equiv \bot$). In contrast, for vertices $x \in V$ the number of input signals $n \in \mathbb{N}$ match the arity of the assigned function or predicate $\ell(x)$. This means $\forall m \in \mathbb{N}.\ \delta(x, m) \equiv \bot \leftrightarrow m > n$. We only consider valid CFMs, where 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. As an example, a CFM would contain a circular dependency, if given $x, y \in V$, $\delta(x, 0) = y$ and $\delta(y, 0) = x$. Such a CFM, if rendered as an Arrow, would enter an infinite loop, and, in the best case, generates the runtime error `<<loop>>`.

3 System Design with TSL

We demonstrate the advantages of using TSL as a specification language for the development of FRP applications using the example application of a kitchen timer, as presented in

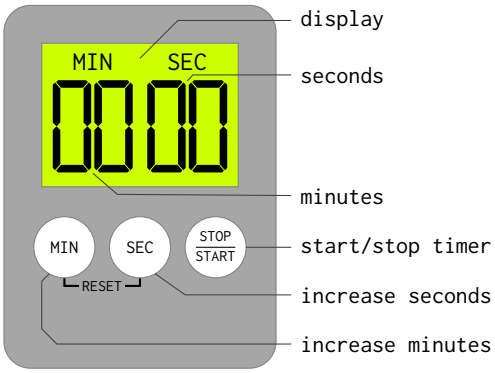

Figure 7. The kitchen timer concept.

Fig. 7. The timer consists of three buttons, a screen displaying the currently set time and a buzzer to produce an alarm. The button values are provided as Boolean input streams to the system, which deliver `true`, as long as a button is pressed, and `false` otherwise. In addition, there is an input stream providing the time passed since the last execution of the network, which is utilized to synchronize the time displayed, with the clock of the application framework.

Similar to the button inputs, the system must output a Boolean data stream to control the buzzer, which is turned on whenever the output is `true`. The second output the system must provide delivers the data to be displayed on the screen, where the data type is fixed by the utilized application framework.

We consider the following list of requirements to be implemented by the timer:

1. Whenever the `MIN` and `SEC` buttons are pressed simultaneously, the timer is reset, meaning the time is set to zero and the system stays idle until the next button gets pressed.
2. If only the `MIN` button is pressed and the timer is not currently counting up or down, then the currently set time is increased by one minute.
3. If only the `SEC` button is pressed and the timer is not currently counting up or down, then the currently set time is increased by one second.
4. As long as no time greater than zero has been set and the system is idle: if the `START/STOP` button is pressed and the timer is not already counting up or down, then the timer starts counting up until it is stopped by any button pressed.
5. If a time has been set and the `START/STOP` button is pressed while the timer is not currently counting up or down, then the timer starts counting down until it is stopped by any button pressed.
6. The timer can only be started by pressing start.
7. The timer can always be stopped by pressing any button while counting up or down.

8. It is possible to start the timer and to set some time simultaneously.
9. The buzzer beeps on any button press and after the counter reaches zero while counting down.
10. The display always shows the time currently set.

While it requires a certain amount of engineering for finding the right control behavior, especially for fixing the additional state to manage the different modes, when directly implementing the application on top of an FRP library, it is an easy task to specify the control behavior with TSL. We first fix possible operations on `time`, used as a cell for holding the currently set time.

$$\begin{aligned}
\textsc{countup} &:= [\![\texttt{time} \leftarrowtail \texttt{countup time dt}]\!] \\
\textsc{countdown} &:= [\![\texttt{time} \leftarrowtail \texttt{countdown time dt}]\!] \\
\textsc{incmin} &:= [\![\texttt{time} \leftarrowtail \texttt{incMinutes time}]\!] \\
\textsc{incsec} &:= [\![\texttt{time} \leftarrowtail \texttt{incSeconds time}]\!] \\
\textsc{idle} &:= [\![\texttt{time} \leftarrowtail \texttt{time}]\!]
\end{aligned}$$

The used literals `countup`, `countdown`, `incMinutes`, and `incSeconds` represent pure functions that update the value of `time` accordingly, while the input signal `dt` delivers the time difference since the last execution of the network. By the semantics of TSL it is already ensured that assignments to the same cell are mutually exclusive, i.e., it can never be the case that time is counting up and the minutes are increased at the same time.

Next, we fix the control flow behavior of `time`. In our case, we need a predicate to check whether the time currently set is zero or not

$$\textsc{zero} := \texttt{eq time zero}$$

where `zero` is a constant function of the same type as `time`. We also fix some sub-properties, that are useful to express conditions regularly appearing in the main specification later. In our case these are

$$\begin{aligned}
\textsc{reset} &:= \texttt{btnMin} \wedge \texttt{btnSec} \\
\textsc{counting} &:= \textsc{countup} \vee \textsc{countdown} \\
\textsc{anykey} &:= \mathit{press}\,\texttt{btnMin} \vee \mathit{press}\,\texttt{btnSec} \\
&\quad \vee \mathit{press}\,\texttt{btnStartStop} \\
\textsc{start} &:= \mathit{press}\,\texttt{btnStartStop} \\
&\quad \wedge \neg \mathit{press}\,\texttt{btnMin} \wedge \neg \mathit{press}\,\texttt{btnSec} \\
\textsc{start\&min} &:= \mathit{press}\,\texttt{btnStartStop} \wedge \mathit{press}\,\texttt{btnMin} \\
&\quad \wedge \bigcirc(\neg\texttt{btnSec} \wedge \bigcirc \neg\texttt{btnSec}) \\
\textsc{start\&sec} &:= \mathit{press}\,\texttt{btnStartStop} \wedge \mathit{press}\,\texttt{btnSec} \\
&\quad \wedge \bigcirc(\neg\texttt{btnMin} \wedge \bigcirc \neg\texttt{btnMin})
\end{aligned}$$

The literals `btnMin`, `btnSec`, and `btnStartStop` represent the input signals for the three buttons, respectively. The function *press* is used as a helper function for improved readability and is defined as

$$\mathit{press}\,x := \neg x \wedge \bigcirc x$$

The additional conditions START&MIN and START&SEC are used for realizing requirement 8. This is all we need for implementing the invariants of the aforementioned requirements:

$$\begin{aligned}
\psi_1 &:= \textsc{reset} \leftrightarrow [\![\texttt{time} \leftarrowtail \texttt{zero}]\!] \\
\psi_2 &:= \neg\textsc{counting} \wedge \mathit{press}\,\texttt{btnMin} \wedge \bigcirc \neg\texttt{btnSec} \\
&\quad \leftrightarrow \bigcirc \textsc{incmin} \\
\psi_3 &:= \neg\textsc{counting} \wedge \mathit{press}\,\texttt{btnSec} \wedge \bigcirc \neg\texttt{btnMin} \\
&\quad \leftrightarrow \bigcirc \textsc{incsec} \\
\psi_4 &:= \textsc{zero} \rightarrow \\
&\quad \big((\textsc{idle} \wedge \textsc{start} \\
&\quad\quad \rightarrow \bigcirc \mathit{tillAnyInput}\,\textsc{countup}) \\
&\quad\quad \mathcal{W}\,(\textsc{incmin} \vee \textsc{incsec})\big) \\
\psi_5 &:= \textsc{incmin} \vee \textsc{incsec} \rightarrow \\
&\quad \big((\neg\textsc{counting} \wedge \textsc{start} \\
&\quad\quad \rightarrow \bigcirc \mathit{tillAnyInput}\,\textsc{countdown}) \\
&\quad\quad \mathcal{W} \bigcirc \textsc{zero}\big) \\
\psi_6 &:= \bigcirc\neg\textsc{counting} \wedge \bigcirc\bigcirc\textsc{counting} \\
&\quad \rightarrow \bigcirc\textsc{start} \vee \textsc{start\&min} \vee \textsc{start\&sec} \\
\psi_7 &:= \textsc{counting} \wedge \textsc{anykey} \wedge \bigcirc\neg\textsc{reset} \\
&\quad \rightarrow \bigcirc \mathit{tillAnyInput}\,\textsc{idle} \\
\psi_8 &:= \neg\textsc{counting} \wedge (\textsc{start\&min} \vee \textsc{start\&sec}) \\
&\quad \rightarrow \bigcirc\bigcirc \mathit{tillAnyInput}\,\textsc{countdown} \\
\psi_9 &:= ([\![\texttt{beep} \leftarrowtail \texttt{true}]\!] \oplus [\![\texttt{beep} \leftarrowtail \texttt{false}]\!]) \wedge \\
&\quad \big(\bigcirc\,(\textsc{countdown} \wedge \textsc{zero}) \vee \textsc{anykey} \\
&\quad\quad \leftrightarrow \bigcirc [\![\texttt{beep} \leftarrowtail \texttt{true}]\!]\big) \\
\psi_{10} &:= [\![\texttt{screen} \leftarrowtail \texttt{display time}]\!]
\end{aligned}$$

The function *tillAnyInput* is a helper function to denote that a condition must be satisfied until either the system is reset or any button gets pressed. It is defined as:

$$\begin{aligned}
\mathit{tillAnyInput}\,x &:= (x \wedge \neg\textsc{anykey}) \\
&\quad\quad \mathcal{W}\ (\textsc{reset} \vee x \wedge \textsc{anykey})
\end{aligned}$$

The final specification is given by $\varphi = \psi_{init} \wedge \square \bigwedge_{i=1}^{10} \psi_i$, where ψ_{init} adds some remaining initial conditions. The full specification, using our plain text specification format, is also given in Fig. 8. Note that the various $\bigcirc$ operations in the formulas ψ_i are always necessary, since the "being pressed" condition requires a change of the input, which is only observable by comparing the currently provided value with the previous one.

For the development of such specifications, the designer also gets feedback from the synthesis engine. For example, the condition of ψ_2 requires `btnSec` to not be pressed in order to increase the minutes of the counter. Without this condition, the synthesis engine would return an unrealizabilty result, since increasing minutes would conflict with setting the time to zero on a potential reset, which also requires `btnMin` to be pressed.

```
COUNTUP   = [ time <- countup time dt ];
COUNTDOWN = [ time <- countdown time dt ];
INCMIN    = [ time <- incMinutes time ];
INCSEC    = [ time <- incSeconds time ];
IDLE      = [ time <- time ];

ZERO = eq time zero();

RESET = btnMin && btnSec;
COUNTING = COUNTUP || COUNTDOWN;
ANYKEY = press btnMin ||
  press btnSec || press btnStartStop;
START = press btnStartStop &&
  !press btnMin && !press btnSec;
STARTANDMIN = press btnStartStop &&
  press btnMin && X !btnSec && X X !btnSec;
STARTANDSEC = press btnStartStop &&
  press btnSec && X !btnMin && X X !btnMin;

xor x y = !(x <-> y);
press x = !x && X x;
tillAnyInput x =
  (x && !ANYKEY) W (RESET || x && ANYKEY);

initially guarantee {
!COUNTING && (X COUNTING -> START);
!INCSEC && !INCMIN;
[ beep <- false ];
}

always guarantee {
RESET
 <-> [ time <- zero() ];
!COUNTING && press btnMin && X !btnSec
 <-> X INCMIN;
!COUNTING && press btnSec && X !btnMin
 <-> X INCSEC;
ZERO
 -> ((IDLE && START -> X tillAnyInput COUNTUP)
      W (INCMIN || INCSEC));
INCMIN || INCSEC -> ((!COUNTING && START
  -> X tillAnyInput COUNTDOWN) W X ZERO);
COUNTING && ANYKEY && X !RESET
  -> X tillAnyInput IDLE;
!COUNTING && (STARTANDMIN || STARTANDSEC)
  -> X X tillAnyInput COUNTDOWN;
X !COUNTING && X X COUNTING
  -> X START || STARTANDMIN || STARTANDSEC;
X (COUNTDOWN && ZERO) || ANYKEY
  <-> X [ beep <- true ];
xor [ beep <- true ] [ beep <- false ];
[ dsp <- display time ];
}
```

Figure 8. The complete plain text specification of the kitchen timer: The `initially guarantee` section specifies requirements that must hold at time $t = 0$. Properties of the `always guarantee` section must hold for all $t \in \mathbb{N}$.

```
control =
  rec
-- gather values from the previous cell value
    ∀c ∈ ℂ. c ← δ(c, 0)
-- gather applications of 𝔽 and ℙ
    ∀v ∈ (V ∩ (𝔽 ∪ ℙ)). v ← (δ(v, 0), …, δ(v, n))
-- compute control signals from cs, vs, and ctrls
    ∀ctrl ∈ (V ∩ 𝕃). ctrl ← (δ(ctrl, 0), …, δ(ctrl, n))
-- use control signals to select from 𝔽 and ℙ
     applications
    ∀m ∈ (V ∩ 𝕌). m ← (δ(m, 0), …, δ(m, n))
-- output signals take the signals from either the
     cs, vs, or ms as specified by δ
  return ∀o ∈ 𝕆. o ← δ(o, 0)
```

Figure 9. The general code template for the control block of the synthesized FRP program. The exact syntax for `rec`, assignment, and outputting signals varies across different FRP abstractions.

After the development of the specification is finished, the synthesis automatically creates a CFM that satisfies the specified control behavior. In the next step the CFM then is specialized towards the specific application context.

4 Code Generation

We present a system for FRP program generation from synthesized CFMs. The user initially provides a CFM that was synthesized from a TSL specification over a set of predicate and function terms. The user specifies a target FRP abstraction, and receives an executable Haskell FRP program in the library of their choosing.

Our approach takes a multi-stage approach, whereby the TSL specification is used to generate a control flow model (CFM). The CFM is an abstract representation of the temporal changes that the FRP program must implement in order to satisfy the TSL specification. In particular, a CFM maps the input signals through various function and predicate terms to the output signals. We only consider valid CFMs, where for every cycle created, by mapping an output back to an input, there is at least one *cell*. A cell is a memory unit with delay, *i.e.*, at one moment a value may be stored, and at the next that value is retrieved. The concept of cells is analogous to ArrowLoop [Paterson 2001], or registers in hardware.

The synthesis from TSL to CFM is the most computationally expensive and may result in unrealizability, in which case synthesis terminates with no solution. We omit a detailed description of the generation of the CFM from a TSL specification, and instead direct the reader to [Finkbeiner et al. 2019]. Here we focus on how the generality of the CFM is utilized to generate framework-independent FRP code. If a satisfying CFM is found during the first stage, a user specifies a target FRP abstraction (Applicative, Monad, Arrow) that is used to generate the FRP program code from the CFM.

```
control
  :: _ signal   -- FRP abstraction
  ⇒ _           -- cell implementation
  → (_ → _)     -- functions and predicates
  → _           -- initial values
  → signal _    -- input signals
  → signal _    -- output signal
```

(a) The general template of the type signature for the control block of the synthesized FRP program.

```
control
  :: Applicative signal
  ⇒ (∀ p. p → signal p → signal p)
  → (a → Bool)   -- press
  → (b → c)      -- display
  → (b → b)      -- increment
  → b            -- initial value: count
  → c            -- initial value: screen
  → signal a     -- button (input)
  → ( signal b   -- count (output)
    , signal c   -- screen (output)
    )
```

(b) An example instantiation of the type signature for the control block of the button (as described in the introduction) as it has been specialized for Applicative FRP.

Figure 10. The control block follows a general type signature template across FRP abstractions.

Given a CFM that satisfies the TSL specification, we convert it into a template for our FRP program. The code implementing the CFM is given in Fig. 9. The CFM is transformed via a syntactic transformation into an FRP program in the abstraction of the user's choice, as a function that is parameterized over the named function and predicate terms, as shown in Fig. 10. The user then provides implementations of the function and predicate terms that complete the construction of the FRP program based on the generated template.

The CFM transformation is modularized to fit any FRP library that is at least as powerful as CCA [Gélineau 2016; Murphy 2016; Patai 2010; Perez et al. 2016; Ploeg and Claessen 2015]. The key insight is that first-order control along with a loop describes the expressive power of both CCA and the CFM model generated from the techniques of [Finkbeiner et al. 2019]. Although many FRP libraries support more powerful operations than CCA, *e.g.*, `switch` in `Yampa`, we do not need to utilize these in the synthesis procedure, and thus can generalize synthesis to target any FRP library that is at least as expressive as CCA.

Recall that in TSL, output signals can be written at the current time t, and be read from at time $t+1$. To implement this in the FRP program, we use the concept of a cell in the CFM. In the translation, we allow a space for the user to provide an implementation of the cell that is specialized to their FRP library of choice (as shown in Fig. 10). In the case of CCA, this is the `loopD` combinator. The `loopD` combinator pipes the output values back to the input to allow them to be read at time $t+1$. Since a system may require output values at $t=0$, the user must also provide initial values to $\mathbb{O}$.

4.1 Properties

In the translation of the CFM, we use Casual Commutative Arrows (CCA) as the target conceptual model. Understanding the implications of using CCA as an underlying model to connect TSL to FRP allows us to gain insight into the expressive power and limitations of using TSL synthesis to construct FRP programs. One interesting note about this is that CCA does not allow the `arrowApply` function, enforcing a static structure on the generated program. The `arrowApply` (also called `switch`) function is a higher-order arrow that allows for dynamically replacing an arrow with a new one that arrives on an input wire. While `switch` is a very expressive operation, it also comes with drawbacks. First, dynamically evolving networks cannot provide runtime guarantees for memory requirements in general, while static networks do. Second, the behavior of a dynamically evolving network is hard to grasp in general, which especially makes them unamenable for verification. Third, the use of dynamic networks is largely impractical for FRP applications with restricted resources, as for example applications that are executed on embedded devices [Sawada and Watanabe 2016] or are implemented directly in hardware [Baaij 2015]. An insight provided by prior work on CCA [Liu et al. 2011] was that, in general, the expressive power of higher-order arrows makes automatic optimization more difficult. Furthermore, for most FRP programs, first-order switch is more than enough [Winograd-Cort and Hudak 2014].

For a full description of the formal properties of TSL synthesis, see the work of [Finkbeiner et al. 2019]. In summary the synthesis procedure is sound, but not complete. From a programming languages design perspective, this means that "compilation" (synthesis) of a specification may not terminate, but when it does terminate, it will generate code that satisfies the specification.

With respect to the synthesis procedure, this is a fundamental restriction related to TSL. With TSL, every update term $[\![x \leftarrowtail y]\!]$ is lifted to an arrow that updates x with y over time. Since in TSL updates are fixed by the specification, so too must the arrow structure be fixed in synthesis. Note that having a fixed arrow structure disallows higher-order arrows, but higher-order functions can still be passed along wires. As an example, we may have a function term `app :: (a → b) → a → b` and signals `f :: a → b` and `x :: a`. A simple specification making use of higher order functions then could state that the system should always apply the incoming higher-order function to the incoming value: $\square[\![\mathtt{x} \leftarrowtail \mathtt{app(f,x)}]\!]$.

```
control
  :: (MonadFix monad, Applicative signal)
  ⇒ (∀ p. p → signal p → monad (signal p))
                          ...
  → signal a → monad (signal b, signal c)
```

(a) The Monadic control for the button from the introduction.

```
control
  :: (Arrow signal, ArrowLoop signalfunction)
  ⇒ (∀ p. p → signalfunction p p))
                          ...
  → signalfunction a (b, c)
```

(b) The Arrowized control for the button from the introduction.

```
control
  :: HiddenClockReset domain gated synchronous
                          ...
  → Signal dom a → (Signal dom b, Signal dom c)
```

(c) The Applicative (specialized to `ClaSH`) control for the button from the introduction.

Figure 11. The abbreviated type signatures of the synthesized control blocks for each FRP framework abstraction.

Additionally, a key difference between arrows and circuits is that arrows are able to carry state that tracks the application of each arrow block. By using CCA, a user may write TSL specifications about stateful arrows that are still handled correctly by the synthesis procedure. To this end, we only synthesize programs that obey the *commutativity* law [Liu et al. 2011; Yallop and Liu 2016] restated below that ensures that arrows cannot carry state influencing the result of composed computations.

$$first\ f \ggg second\ g = second\ g \ggg first\ f$$

Imagine an arrow with a global counter to track data of a buffer. Since addition is commutative, this arrow respects the commutativity law. However, non-commutative state is possible as well. For example, when building GUIs with Arrowized FRP [Winograd-Cort 2015], the position of each new UI element depends on the order of the previously laid out elements. Due to the commutativity of the Boolean operators, the commutativity of CCA is a necessary precondition for synthesis of a TSL specification. Specifically, the commutativity of logical conjunction allows the solution to update signals in any arbitrary order. Thus, the correctness of the TSL synthesis relies on commutativity of composition, which is naturally modeled with CCA's commutativity law.

4.2 Example: Kitchen Timer

We revisit the Kitchen Timer application introduced in Section 3 to show the concrete code that is generated. From the TSL specification, we first generate a CFM using our TSL synthesis toolchain together with the LTL synthesizer

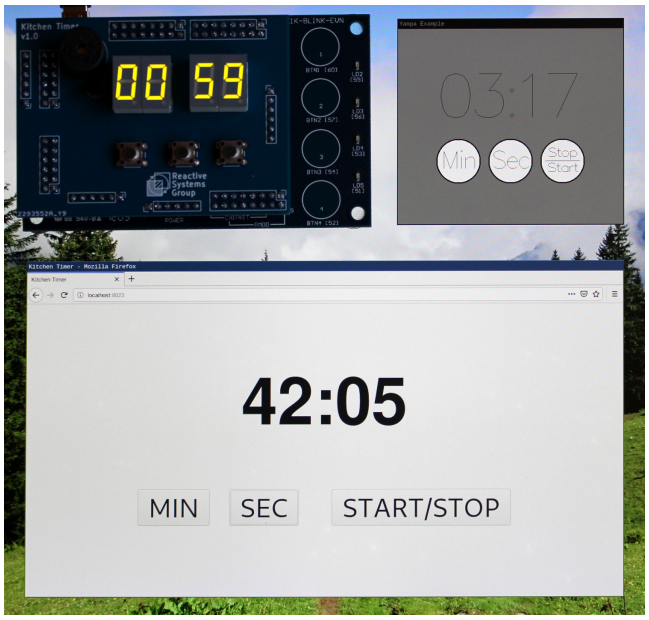

Figure 12. Timer applications: the hardware application built with `ClaSH` is on the top left. The top right shows the desktop application built using `Yampa` and the at bottom the web application built using `Threepenny-GUI`. All are synthesized from the same CFM.

`strix` [Meyer et al. 2018] [version `18.04`]. The resulting CFM utilizes six additionally synthesized cells and consists of 1188 vertices. This CFM is then transformed into a control structure for each of the different application domains. In Fig. 11, we show how the template described in Section 4 is specialized to each of the three application domains: the desktop program is built with `Yampa` [version `0.13`] and the web app with `Threepenny-GUI` [version `0.8.3.0`]. Both have been built using stack[2] on `lts-13.17` [`ghc-8.6.4`]. For building the hardware implementation, we first use the functional hardware description language `ClaSH`[3] to generate verilog-code, which then is turned into the `blif` format using the open synthesis suite `yosys`[4]. Afterwards, the generated `blif`-file is placed using the place-and-route tool `nextpnr`[5]. The packaged result is then uploaded to an iCEblink40HX1K Evaluation Kit Board from Lattice Semiconductor, featuring an ICE40HX1K FPGA with 100 IO-pins and 1280 logic cells, additionally equipped with the required hardware components. The interfaces to the corresponding timer applications are depicted in Fig. 12.

Note that synthesis only needs to be executed once. Afterwards, code is generated from the resulting CFM and combined with function implementations and initialization

[2] https://www.haskellstack.org
[3] https://github.com/clash-lang/clash-compiler [commit: `fff4606`]
[4] https://github.com/YosysHQ/yosys [commit: `70d0f38`]
[5] https://github.com/YosysHQ/nextpnr [commit: `5344bc3`]

Table 1. Synthesis and compilation times for creating the different timer applications from the TSL specification.

Executed Tool	Time (sec)
Synthesis → `strix`	4.965
Compilation	
Desktop → `Yampa`	19.403
Web → `Threepenny-GUI`	18.344
Hardware	
→ `ClaSH`	11.218
→ `yosys`	6.405
→ `nextpnr`	7.276

procedures for each of the frameworks. Both, the desktop and the web application only require GHC for compilation, while for hardware, we need multiple translation steps. The respective synthesis and compilation times of the different tools are depicted in Table 1. The full code of the Arrow module, generated from the CFM realizing the specification of the introduction, is given as an example in Fig. 13.
The specification, the building framework, and a list of the required hardware components can be found at:

https://github.com/reactive-systems/KitchenTimer

`Yampa` uses Arrowized FRP which easily fits into the general interface for Arrows that we provide (Fig. 11b). Likewise, `Threepenny-GUI` uses a Monadic FRP (where the signals themselves are Applicative) which also easily fits into our general interface for Monadic FRP (Fig. 11a). Finally, for `ClaSH`, we use a mostly Applicative interface, that is specialized to handle the peculiarities of hardware (which needs explicit clocks, as opposed to more traditional FRP frameworks). If we wanted to support other libraries with explicit clocks, for example, as presented in [Bärenz and Perez 2018], we would need a specialized module - although the customization is limited mostly to the type signature generation as shown in Fig. 11c.

Each control block requires the user to provide a cell implementation. Both `Yampa` and `ClaSH` provide native implementations of the concepts, as shown in Fig. 14. Although `Threepenny-GUI` does not provide the exact implementation of a cell, as we require in our synthesized control block, it can be easily implemented using the available primitives of the library.

5 Related Work

There are various lines of work that are related to our approach. While we draw inspiration from these research directions, each one, on its own, addresses a different type of problem.

```
{-# LANGUAGE Rank2Types, Arrows #-}
module Example (control) where
import Control.Arrow

control
  :: (Arrow sig, ArrowLoop sig)
  => (∀ poly. poly → sig poly poly)
  → (a → Bool) -- event
  → (b → c) -- display
  → (b → b) -- increment
  → b → c -- initial values: count, screen
  → sig
      a -- input: click
      (b, c) -- output: (count, screen)
control cell pEvent fDisplay
  fIncrement iCount iScreen =
    proc sClick → do
      rec
        cCount <- cell iCount -< oCount
        cScreen <- cell iScreen -< oScreen
        w3 <- arr fDisplay -< cCount
        w4 <- arr fIncrement -< cCount
        b5 <- arr pEvent -< sClick
        (cout0, cout1, cout2, cout3) <-
          controlCircuit cell -< b5
        oCount <- countSwitch -<
          ((cCount, cout0), (w4, cout1))
        oScreen <- screenSwitch -<
          ((cScreen, cout2), (w3, cout3))
      returnA -< (oCount, oScreen)

countSwitch
  :: Arrow sig ⇒ sig ((a, Bool), (a, Bool)) a
countSwitch =
  proc ((s0, b0), (s1, _)) → do
    r0 <- arr ite -< (b0, s0, s1)
    returnA -< r0
  where
    ite (b, t, e) = if b then t else e

screenSwitch
  :: Arrow sig ⇒ sig ((a, Bool), (a, Bool)) a
screenSwitch =
  proc ((s0, b0), (s1, _)) → do
    r0 <- arr ite -< (b0, s0, s1)
    returnA -< r0
  where
    ite (b, t, e) = if b then t else e

controlCircuit
  :: (Arrow sig, ArrowLoop sig)
  ⇒ (Bool → sig Bool Bool) -- cell
  → sig Bool (Bool, Bool, Bool, Bool)
controlCircuit cell =
  proc cin0 → do
    returnA -< (not cin0, cin0, False, True)
```

Figure 13. Generated Arrow code for the intro example.

```
-- yampa
iPre :: SF a a

-- clash
register
  :: HiddenClockReset domain gated synchronous
  ⇒ a → Signal domain a → Signal domain a

-- reactivebanana / threepennygui
cell
  :: MonadMoment / MonadIO m
  ⇒ a → Behavior a → m (Behavior a)
cell v x = stepper v (x <@ allEvents)

-- reflex
cell
  :: Reflex t
  ⇒ b → Behavior t b → Behavior t b
cell v x = hold v (tag x allEvents)
```

Figure 14. The implementations for a cell in the CFM is commonly found across FRP libraries, or easliy re-implemented.

5.1 Temporal Types for FRP

FRP is a programming paradigm for computations over time, and, hence, a natural extension is to investigate type systems to be able to reason about time. A correspondence between LTL and FRP in a dependently typed language was discovered simultaneously by [Jeffrey 2012; Jeltsch 2012]. In this formulation, FRP programs are proofs and LTL propositions are reactive, time-varying types that describe temporal properties of these programs. In establishing the connection between logic and FRP, these LTL types are also used to ensure causality and loop-freeness on the type level.

Dependent LTL types are a useful extension to FRP that provides insight into the underlying model of FRP, but does not lend itself to control flow synthesis. In the work of Jeffery and Jeltsch, the types describe the input/output change over time for each arrow. Using these LTL types, only arrows adhering to sensible temporal orderings (*e.g.* computations only depend on past values) will be well typed. However, as with any other FRP system, the temporal control flow of function applications in the program is fixed by the code. A similar approach was used by [Krishnaswami 2013] to make a temporal type system that ensures there are no space-time leaks in a well typed FRP program. Work on reasoning about FRP using temporal logics also includes [Sculthorpe and Nilsson 2010], although this setting considered dynamic network structure, as allowed by higher-order arrows. While the above works apply temporal types for functional correctness, type extensions have also been used to encode fairness properties [Cave et al. 2014], to ensure that any well-typed system will eventually perform actual work.

In contrast, we use the logic TSL, for a fine-grained description of function application behavior which cannot be expressed within pure LTL. The synthesis procedure of TSL determines a temporal control flow of functions, where the TSL specifications determines the transformations to be applied at each point in time. In addition to the logical specifications, the synthesis is also constrained by the types of functions appearing in the specification. Since the types of all functions are fixed at all times, the type system can be lifted to the specification. If the specification is well typed, synthesis is guaranteed to yield a well typed program.

One connection to our work however is the implications of the fact that the Curry-Howard correspondence extends to FRP and LTL. In the aforementioned work, LTL propositions are types for FRP programs. If a proof of a TSL proposition can be interpreted as a program, one might expect that there is some corresponding type system to TSL. We leave such explorations to future work.

5.2 Synthesis of Reactive Programs

A distinguishing feature of our approach is the connection to an actual programming paradigm, namely FRP. Most reactive synthesis methods instead target transition systems or related formalisms such as finite state machines. The idea to synthesize programs rather than transition systems was introduced in [Madhusudan 2011]. In his work, an automaton is constructed that works on the syntax tree of the program, which makes it possible to obtain concise representations of the implementations, and to determine how many program variables are needed to realize a particular specification. Unlike our FRP programs, Madhusudan's programs only support variables on a finite range of instances.

Another related approach is the synthesis of synchronization primitives introduced in [Bloem et al. 2012] for the purpose of allowing sequential programs to be executed in parallel. Similar to our synthesis approach, uninterpreted functions are used to abstract from implementation details. However, both the specification mechanism (the existing program itself is the specification) and the type of programs considered are completely different from TSL and FRP.

5.3 Logics for Reactive Programs

Many logics have been proposed to specify properties of reactive programs. Synthesis from Signal Temporal Logic [Raman et al. 2015] focuses on modeling physical phenomena on the value level, introducing continuous time and resolving to a system of equations. The approach allows for different notion of data embedded into the equations. While more focused on the data level, the handling of continuous time might provide inspiration for future extensions to explicitly handle continuous time.

Another logic that has been proposed, Ground Temporal Logic [Cyrluk and Narendran 1994], is a fragment of First Order Logic equipped with temporal operators, where it is

not allowed to use quantification. Satisfiability and validity problems are studied, with the result that only a fragment is decidable. However, specifications expressed in Ground Temporal Logic, as well as their motivations, are completely different from our goals.

5.4 Reasoning-based Program Synthesis

Reasoning-based synthesis [Kuncak et al. 2010; Osera and Zdancewic 2015; Solar-Lezama 2013; Vechev et al. 2013] is a major line of work that has been mostly, but not entirely, orthogonal to reactive synthesis. While reactive synthesis has focused on the complex control aspects of reactive systems, deductive and inductive synthesis has been concerned with the data transformation aspects in non-reactive, sequential programs. Our work is most related to Sketching [Solar-Lezama 2013]. In Sketching the user provides the control structure and synthesizes the transformations while, in TSL we synthesize the control and leave the transformations to the user.

The advantage of deductive synthesis is that it can handle systems with complex data. Its limitation is that it cannot handle the continuous interaction between the system and its environment, which is typical for many applications, such as for cyber-physical systems. This type of interaction can be handled by reactive synthesis, which is, however, typically limited to finite states and can therefore not be used in applications with complex data types. Abstraction-based approaches can be seen as a link between deductive and reactive synthesis [Beyene et al. 2014; Dimitrova and Finkbeiner 2012].

Along the lines of standard reactive synthesis, our work is focused on synthesizing control structures. We extend the classic approach by also allowing the user to separately provide implementations of data transformations. This is useful in the case where the value manipulations are unknown or beyond the capability of the synthesis tool. For example, a user may want to synthesize an FRP program that uses closed source libraries, which may not be amenable to deductive synthesis. In this case, the user can only specify that certain functions from that API should be called under certain conditions, but cannot and may not want to reason about their output.

6 Conclusions

In this work we have presented a detailed account of how to transform Control Flow Models into framework-independent FRP code. With this transformation, we utilize TSL synthesis as presented in [Finkbeiner et al. 2019] to build a complete toolchain for synthesizing Functional Reactive Programs. Using TSL specifications prior to manually programming improves designing the underlying control flow. The developer is immediately notified about conflicts in the current design and supported by the feedback returned from synthesis for resolving them.

So far, we have used a discrete time model in our formalization, however, the behavior of the kitchen timer is in fact sampling rate independent (Continuous Time FRP). Sampling rate independence is guaranteed in TSL as long as the next operator is not used. However, the relation between TSL with the next operator and Continuous Time FRP still needs to be explored.

In another direction, the usual way in FRP to distinguish between continuous and discrete behaviors is to use signals and events. So far we have embedded data into signals. It is open to future work how to utilize events natively. Future directions for improvements to usability include integrating FRP synthesis more tightly with programming, *e.g.*, by allowing specifications to be used inline with QuasiQuoters [Mainland 2007].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1302327, the European Research Council (ERC) Grant OSARES (No. 683300), and the German Research Foundation (DFG) as part of the Collaborative Research Center Foundations of Perspicuous Software Systems (TRR 248, 389792660). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the these agencies.

References

Heinrich Apfelmus. 2012. Reactive-banana. *Haskell library available at http://www. haskell. org/haskellwiki/Reactive-banana* (2012).

Heinrich Apfelmus. 2013. Threepenny-gui. https://wiki.haskell.org/Threepenny-gui. (2013).

C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. Ph.D. Dissertation. DOI: http://dx.doi.org/10.3990/1.9789036538039 eemcs-eprint-23939.

Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 145–157. DOI: http://dx.doi.org/10.1145/3242744.3242757

Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A Constraint-based Approach to Solving Games on Infinite Graphs. In *POPL*. ACM, 221–233.

Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of reactive (1) designs. *J. Comput. System Sci.* 78, 3 (2012), 911–938.

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming *(POPL '14)*. ACM, New York, NY, USA, 361–372. DOI: http://dx.doi.org/10.1145/2535838.2535881

Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. ACM, 7–18.

David Cyrluk and Paliath Narendran. 1994. Ground temporal logic: A logic for hardware verification. In *CAV*. Springer, 247–259.

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs *(PLDI '13)*. ACM, New York, NY, USA, 411–422. DOI: http://dx.doi.org/10.1145/2491956.2462161

Rayna Dimitrova and Bernd Finkbeiner. 2012. *Counterexample-Guided Synthesis of Observation Predicates.* Springer Berlin Heidelberg, Berlin, Heidelberg, 107–122.

Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 263–273.

Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Temporal Stream Logic: Synthesis beyond the Bools. *CoRR* abs/1712.00246 (2019). arXiv:1712.00246 http://arxiv.org/abs/1712.00246 (accepted at CAV 2019).

Samuel Gélineau. 2016. FRPzoo - Comparing many FRP implementations by reimplementing the same toy app in each. https://github.com/gelisam/frp-zoo. (2016).

Caleb Helbling and Samuel Z. Guyer. 2016. Juniper: A Functional Reactive Programming Language for the Arduino *(FARM 2016).* ACM, New York, NY, USA, 9. DOI: http://dx.doi.org/10.1145/2975980.2975982

John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1 (2000), 67–111.

Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs, Koen Claessen and Nikhil Swamy (Eds.). ACM, 49–60. DOI: http://dx.doi.org/10.1145/2103776.2103783

Wolfgang Jeltsch. 2012. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *ENTCS* 286 (2012), 229–242.

Gangyuan Jing, Tarik Tosun, Mark Yim, and Hadas Kress-Gazit. 2016. An End-To-End System for Accomplishing Tasks with Modular Robots. In *RSS*.

Ayrat Khalimov, Roderick Paul Bloem, and Swen Jacobs. 2014. Parameterized Synthesis Case Study: AMBA AHB. In *SYNT 2014*, Susmit Jha Krishnendu Chatterjee, Rüdiger Ehlers (Ed.).

Neelakantan R Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 221–232.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete functional synthesis. *ACM Sigplan Notices* 45, 6 (2010), 316–329.

Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *ENTCS* 229, 5 (2011), 97–117.

Hai Liu, Eric Cheng, and Paul Hudak. 2011. Causal commutative arrows. *J. Funct. Program.* 21, 4-5 (2011), 467–496. DOI: http://dx.doi.org/10.1017/S0956796811000153

Hai Liu and Paul Hudak. 2007. Plugging a space leak with an arrow. *ENTCS* 193 (2007), 29–45.

Parthasarathy Madhusudan. 2011. Synthesizing reactive programs. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 12.

Geoffrey Mainland. 2007. Why It's Nice to be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop.* ACM, 73–82.

Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. 2018. Strix: Explicit Reactive Synthesis Strikes Back!. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science)*, Hana Chockler and Georg Weissenbacher (Eds.), Vol. 10981. Springer, 578–586. DOI: http://dx.doi.org/10.1007/978-3-319-96145-3_31

Tom E Murphy. 2016. A livecoding semantics for functional reactive programming. In *Functional Art, Music, Modelling, and Design.* ACM, 48–53.

Peter Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 619–630.

Gergely Patai. 2010. Efficient and compositional higher-order streams. In *International Workshop on Functional and Constraint Logic Programming.* Springer.

Ross Paterson. 2001. A New Notation for Arrows. In *International Conference on Functional Programming.* ACM Press, 229–240. http://www.soi.city.ac.uk/~ross/papers/notation.html

Ivan Perez. 2017. GALE: a functional graphic adventure library and engine. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design, FARM@ICFP 2018, Oxford, UK, September 9, 2017.* 28–35. DOI: http://dx.doi.org/10.1145/3122938.3122944

Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional reactive programming, refactored. In *International Conference on Functional Programming.* ACM.

Ivan Perez and Henrik Nilsson. 2017. Testing and debugging functional reactive programming. *PACMPL* 1, ICFP (2017), 2:1–2:27. DOI: http://dx.doi.org/10.1145/3110246

Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of Reactive(1) Designs. In *VMCAI.*

Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 302–314.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 522–538. DOI: http://dx.doi.org/10.1145/2908080.2908093

Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M. Murray, and Sanjit A. Seshia. 2015. Reactive Synthesis from Signal Temporal Logic Specifications. In *HSCC.* 239–248. DOI: http://dx.doi.org/10.1145/2728606.2728628

Mark Santolucito, Donya Quick, and Paul Hudak. 2015. Media Modules: Intermedia Systems in a Pure Functional Paradigm. In *ICMC 2015, Denton, TX, USA.* http://hdl.handle.net/2027/spo.bbp2372.2015.077

Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki (Eds.). ACM, 36–44. DOI: http://dx.doi.org/10.1145/2892664.2892670

Neil Sculthorpe and Henrik Nilsson. 2010. Keeping calm in the face of change. *Higher-Order and Symbolic Computation* 23, 2 (2010), 227–271.

Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications *(OOPSLA 2016).* ACM, New York, NY, USA, 864–880. DOI: http://dx.doi.org/10.1145/2983990.2984011

Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495.

Ryan Trinkle. 2017. Reflex-FRP. https://github.com/reflex-frp/reflex. (2017).

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2013. Abstraction-guided synthesis of synchronization. *STTT* 15, 5-6 (2013), 413–431.

Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN programming using algorithmic policies. *CCR* 43, 4 (2013), 87–98.

Daniel Winograd-Cort. 2015. *Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming.* Ph.D. Dissertation. Yale University.

Daniel Winograd-Cort and Paul Hudak. 2014. Settable and Non-interfering Signal Functions for FRP: How a First-order Switch is More Than Enough. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14).* ACM, New York, NY, USA, 213–225. DOI: http://dx.doi.org/10.1145/2628136.2628140

Jeremy Yallop and Hai Liu. 2016. Causal commutative arrows revisited. In *Proceedings of the 9th International Symposium on Haskell.* ACM, 21–32.