Under consideration for publication in J. Functional Programming

Runtime Verification and Validation of Functional Reactive Systems

IVAN PEREZ
National Institute of Aerospace, Hampton, VA, 23666, USA
and
HENRIK NILSSON
School of Computer Science, University of Nottingham, Hottingham, NG8 1BB, UK
(*e-mail:* ivan.perez@nianet.org)

Abstract

Many types of interactive applications, including reactive systems implemented in hardware, interactive physics simulations, and games, raise particular challenges when it comes to testing and debugging. Reasons include de-facto lack of reproducibility and difficulties of automatically generating suitable test data. This paper demonstrates that certain variants of Functional Reactive Programming (FRP) implemented in pure functional languages can mitigate such difficulties by offering referential transparency at the level of whole programs. This opens up for a multi-pronged approach for assisting with testing and debugging that works across platforms, including assertions based on temporal logic, recording and replaying of runs (also from deployed code), and automated random testing using QuickCheck. When combined with extensible forms of Functional Reactive Programming that allow for constrained side effects, this allow us to not only validate software simulations, but to analyse the effect of faults in reactive systems, validate the efficacy of fault tolerance mechanisms, and perform software- and hardware-in-the-loop testing. The approach has been validated on non-trivial systems implemented in several existing FRP implementations, by means of careful debugging using a tool that allows the test or simulation under scrutiny to be controlled, moving along the execution time line, and pin-pointing of violations of assertions on PCs as well as external devices.

Contents

1 Introduction

Testing software thoroughly is hard in general (Whittaker, 2000). Testing reactive systems raises specific additional difficulties due to their interactive and real-time aspects (Lewis *et al.*, 2010). For example, just specifying system input so as to ensure adequate test coverage is daunting. As a result, much of the testing is often left to final users. Another difficulty is the lack of reproducibility: in general, the exact same input may produce different results at different times. This is due to the random nature of real-world signals, the deliberate random elements in interactive applications, and the effects of interacting with a constantly changing outside world at points in time that are difficult to control exactly due to real-time considerations. Consequently, the conditions that led to a bug can be hard to replicate.

These challenges apply generally, regardless of what language is used to implement a system. Thus, while using a pure functional language can offer benefits at the level of unit testing thanks to referential transparency, just using a pure language offers few if any additional benefits over commonly used languages for reactive system programming when it comes to testing a system as a whole.

Functional Reactive Programming (FRP) (Elliott & Hudak, 1997; Nilsson *et al.*, 2002; Courtney *et al.*, 2003) helps express interactive software declaratively in a way that arguably brings the benefits of pure functional programming to the *system level*. In this paper, we demonstrate that *Arrowized* FRP in particular provides enough structure to address whole-program testing and debugging challenges such as those outlined above. Our work applies to FRP programs in general, but we put a particular emphasis on reactive hardware systems, physical simulations, games, and related applications in this paper, as these tend to highlight the difficulties of interest here.

There are two sources of uncertainty in FRP programs that may affect system-level referential transparency. First, some FRP implementations use IO inside core definitions to increase versatility and performance. This makes those systems susceptible to the problems discussed earlier. Second, even if we run a simulation twice with the same initial state and user input, we may obtain different outputs if simulations are sampled at different points in time, due to differences in system load and OS scheduling decisions outside our control, for example. This is demonstrated by the force-directed graph simulation depicted in Figure 1, which does not depend on user input, yet converges to different stable configurations depending on the sampling step, due to small differences in floating point calculations.

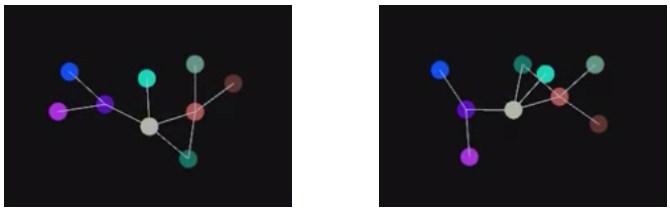

Fig. 1. Two runs of the same graph layout algorithm in which nodes repel each other, unless directly connected, in which case they also attract. Even with identical initial conditions, two executions converge to different stable configurations, demonstrating that pure FRP *systems* exhibit non-determinism.

Pure Arrowized FRP (Courtney & Elliott, 2001; Nilsson *et al.*, 2002) completely separates all side effects and time sampling from the data processing, providing referential transparency across executions. In this variant we can truly run a program twice with the same input, poll it at the same times, and obtain the same output, enabling a form of testing unparalleled by other languages and paradigms. Given the same architecture, the results will be guaranteed by the type system and the compiler to be the same, *even* across different devices. This is a key aid for system development, especially on external platforms and for applications that run on remote devices, since users often find bugs that developers cannot reproduce.

In this paper we explore how Arrowized FRP enables testing and debugging to be approached systematically in pure functional languages. This paper extends on prior work by Perez & Nilsson (2017), revisiting prior contributions and including novel work, as follows:

- We extend FRP constructs with a notion of temporal predicates based on Linear Temporal Logic, equipped with an evaluation function, and demonstrate how they can be used to express temporal properties.
- We provide random input data generators, and demonstrate how they can be used effectively to test FRP systems using QuickCheck.
- We present a *causal* subset of Past-time Linear Temporal Logic that can be used to insert temporal assertions into FRP programs for revealing bugs during execution.
- We present an extension to an FRP implementation that allows users to record and replay input traces, which can be used to remotely control and debug running applications.

- We demonstrate how our system allows for software- and hardware-in-the-loop testing of reactive systems.
- We show that, by combining this work with prior work on fault-tolerant FRP, we can analyze the effects of faults in reactive systems and the effectiveness of fault tolerance mechanisms.

Our proposal is applicable for multiple existing FRP implementations, including Yampa (Nilsson & Courtney, 2003) and Dunai (Perez & Bärenz, 2016). It is capable of recording, replaying, manipulating and visualizing execution traces from real users, as well as counter-example traces generated by QuickCheck. Our system lets designers and developers traverse an input trace and find points where assertions are violated, moving back and forth along the execution timeline and performing hot-swapping of the application to verify whether changes to the system fix existing bugs.

2 Background

In the interest of making this paper sufficiently self-contained, we summarize the basics of FRP and Monadic Stream Functions in the following. For further details on FRP, Arrowized FRP (AFRP), and Monadic Stream Functions, see earlier papers (Elliott & Hudak, 1997; Nilsson *et al.*, 2002; Courtney *et al.*, 2003; Perez *et al.*, 2016). This presentation draws heavily from the summaries in Perez (2018); Perez & Nilsson (2017); Courtney *et al.* (2003).

2.1 Functional Reactive Programming

FRP is a programming paradigm to describe hybrid systems that operate on time-varying data. FRP is structured around the concept of *signal*, which conceptually can be seen as a function from time to values of some type:

$$Signal\ \alpha\ = Time \rightarrow \alpha$$

Time is (notionally) continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of an animation would be *Signal Picture* for some type *Picture* representing static pictures. Signals can also represent input data, like the mouse position.

Additional constraints are required to make this abstraction executable. First, it is necessary to limit how much of the history of a signal can be examined, to avoid memory leaks. Second, if we are interested in running signals in real time, we require them to be *causal*: they cannot depend on other signals at future times. FRP implementations address these concerns by limiting the ability to sample signals at arbitrary points in time.

The space of FRP frameworks can be subdivided into two main branches, namely Classic FRP (Elliott & Hudak, 1997) and Arrowized FRP (Nilsson *et al.*, 2002). Classic FRP programs are structured around signals or a similar notion representing internal and external time-varying data (originally built around a notion of *behaviours* and *events*). In contrast, Arrowized FRP programs are defined using causal functions between signals, or *signal functions*, connected to the outside world only at the top level. In Classic FRP, signals

(or behaviours and event streams) are first-class entities; in Arrowized FRP, they are not. While FRP is conceptually continuous, implementations still execute by sampling inputs at discrete points in time. To give a concrete example, signal functions are commonly represented like::

```
newtype SF a b = SF (Time -> a -> (b, SF a b))
```

The function representing a signal function is invoked on a sample of the input at a discrete point in time, yielding an output sample as well as a new version of the signal function to be used at the next time step. Such a representation ensures that signal functions are causal by construction, meaning that they in effect can carry a state *forward* in time.

2.2 Monadic Stream Functions

Monadic Stream Functions (MSFs) are an abstraction for Functional Reactive Programming that supports discrete and continuous time, and both Classic and Arrowized variants. The initial observation was that there often is quite a bit of plumbing in Arrowized FRP networks; e.g. to thread access to global resources through to all points of use or to do logging. Of course, monads are great for abstracting away this type of plumbing when that is desired, suggesting that signal functions ought to be parametrised over a monad. The resulting notion is very flexible: for example, given the signal function representation above, time can be seen as being part of an environment, rather than being hard-wired into the fundamental abstraction at a specific type, as `(Time ->)` is just a reader monad.

In this section we briefly introduce MSFs, as well as some of the common use cases and effects obtained when combined with different monads. We refer the reader to Perez *et al.* (2016); Pérez (2018) for further details on MSFs, and to Perez & Bärenz (2016) for an implementation.

2.2.1 Fundamental Concepts

MSFs are defined by a polymorphic type `MSF` and an evaluation function that applies an `MSF` to an input and returns, in a monadic context, an output and a continuation:

```
newtype MSF m a b

step :: Monad m => MSF m a b -> a -> m (b, MSF m a b)
```

The type `MSF` and the `step` function alone do not represent causal functions on streams. It is only when we successively apply the function to a stream of inputs and consume the side effects that we get the unrolled, streamed version of the function. Causality, or the requirement that the n-th element of the output stream only depend on the first n elements of the input stream, is obtained as a consequence of applying the MSF continuations *step by step*, or sample by sample.

For the purposes of exposition, we will use the following function to apply an MSF to a *finite list* of inputs, with effects and continuations chained sequentially. This is merely a debugging aid, not how MSFs are actually executed:

```
embed :: Monad m => MSF m a b -> [a] -> m [b]
```

2.2.2 Composing Monadic Stream Functions

Programming with MSFs consists of defining monadic stream functions compositionally using a library of primitive stream functions and a set of combinators. MSFs are `Arrows` (Hughes, 2000), and so `Arrow` combinators can be used to create them and compose them. Some central `Arrow` combinators are `arr` that lifts an ordinary function to a stateless signal function, composition `>>>`, and parallel composition `&&&`. In our framework, they have the following types:

```
arr   :: (a -> b)  -> MSF m a b
(>>>) :: MSF m a b -> MSF m b c -> MSF m a c
(&&&) :: MSF m a b -> MSF m a c -> MSF m a (b,c)
```

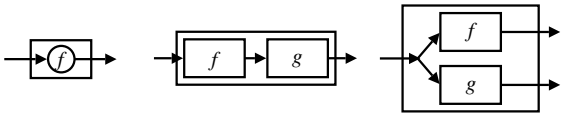

Fig. 2. Basic MSF combinators.

We can think of streams and monadic stream functions using a simple flow chart analogy. Line segments (or "wires") represent streams, with arrow heads indicating the direction of flow. Boxes represent MSFs, with the input stream flowing into the input port of the box and the output stream flowing out of the output port. Figure 2 illustrates the aforementioned combinators using this analogy. Through the use of these and related combinators, arbitrary MSF networks can be expressed.

2.2.3 Arrow Notation

Writing arrow code using the basic combinators quickly becomes challenging as the expressions get larger. Paterson's arrow notation (Paterson, 2001) mitigates this concern. At least for basic arrow programming, it can be understood as a textual rendition of the above flow chart analogy. From our perspective, that means that signal function networks networks can be described directly, and in particular that signals effectively can be named, despite signals not being first class values. Note that the notation is *syntactic sugar* that gets translated into plain combinator expressions. We review basics of Paterson's notation here is it will be used occasionally in the following to simplify the presentation of arrow code.

Using Paterson's notation, an expression denoting a signal function has the form:

```
proc pat -> do
    pat1  <-  sfexp1  -<  exp1
    pat2  <-  sfexp2  -<  exp2
    ...
    patn  <-  sfexpn  -<  expn
    returnA -< exp
```

The keyword `proc` is analogous to the λ in λ-expressions, `pat` and `pati` are patterns binding signal variables pointwise by matching on instantaneous signal values, `exp` and `expi` are expressions defining instantaneous signal values, and `sfexpi` are expressions denoting signal functions. The idea is that the signal being defined pointwise by each `expi` is fed into the corresponding signal function `sfexpi`, whose output is bound pointwise in `pati`. The overall input to the signal function denoted by the `proc`-expression is bound pointwise by `pat`, and its output signal is defined pointwise by the expression `exp`. The signal variables bound in the patterns may occur in the signal value expressions, but *not* in the signal function expressions `sfexpi`.

2.2.4 Depending on the Past

MSFs must remain causal and leak-free, and so we introduce limited ways of depending on past values. To remember the past by producing an extra value or *accumulator* accessible in future iterations, we use:

```
feedback :: c -> MSF m (a,c) (b,c) -> MSF m a b
```

This combinator takes an initial value for the accumulator, runs the MSF, and feeds the new accumulator back for future iterations.

Example The following calculates the cumulative sum of its inputs, initializing an accumulator and using a feedback loop:

```
sumFrom :: (Num n, Monad m) => n -> MSF m n n
sumFrom n0 = feedback n0 (arr add2)
  where
    add2 (n, acc) = let n' = n + acc in (n', n')
```

A counter, for example, can be defined as follows:

```
count :: (Num n, Monad m) => MSF m () n
count = arr (const 1) >>> sumFrom 0
```

2.2.5 Monads

MSFs can be combined with different monads for different effects. We provide a general function `arrM` to lift a Kleisli arrow, applied point-wise to every sample:

```
arrM :: Monad m => (a -> m b) -> MSF m a b
```

The use of monads with MSFs provides great versatility. For example, we can make certain values available in an environment in a `Reader` monad, without having to route them down manually as inputs to other MSFs. This is particularly useful in interactive applications, in which settings are adjusted interactively within the application (e.g., game difficulty, graphics quality) or depend on dynamic aspects (e.g., screen size), yet large parts of the application can treat those values as "constants". For example, we can pass the screen size to an application by putting it in an environment:

```
data Env = Env { windowWidth  :: Int
               , windowHeight :: Int
               }
```

An MSF that rotates the input mouse position by 180 degrees with respect to the center of the screen can use that implicit environment:

```
rotateMousePos180 :: MSF (Reader Env) (Int, Int) (Int, Int)
rotateMousePos180 = proc (x,y) -> do
  winW <- arrM (ask windowWidth)  -< ()
  winH <- arrM (ask windowHeight) -< ()
  returnA -< (winW - x, winH - y)
```

Any MSF can use `rotateMousePos180` without having to manually route down the environment, which is only explicit in the type.

It is possible to "flatten" an MSF by removing the monadic effect, by means of what are called *MSF running functions*. This normally requires extra inputs, extra outputs, or extra continuations. For example, the running function for the reader monad is defined as:

```
runReaderS :: MSF (ReaderT r m) a b -> r -> MSF m a b
```

We can use this function to build an MSF that removes the `Reader` monad from the monad stack by providing a specific environment (i.e., screen size):

```
rotateMouse1024x768 :: MSF m a b
rotateMouse1024x768 = runReaderS (rotateMousePos180) (Env 1024 768)
```

The behaviour is, as we expect:

```
ghci> embed rotateMouse1024x768 [(10, 10), (100, 100)]
[(1014, 758), (924, 668)]
```

Following the same pattern as before, the associated execution function for an `MSF` on the `Maybe` monad would have type:

```
runMaybeS :: Monad m => MSF (MaybeT m) a b -> MSF m a (Maybe b)
```

The evaluation function `step`, instantiated for the `Maybe` monad, would have the type `MSF Maybe a b -> a -> Maybe (b, MSF Maybe a b)`, indicating that it may produce *no continuation*. The function `runMaybeS` applied to an `MSF` outputs `Nothing` continuously once the internal `MSF` produces no result. "Recovering" from failure requires an additional continuation:

```
catchM :: Monad m => MSF (MaybeT m) a b -> MSF m a b -> MSF m a b
```

`Either c` is another form of `MSF` that may terminate (in this case, with a result `Left cr` for some `cr` of type `c`). Recovering from failure requires, once again, an additional MSF, which in this case may be constructed from the value being provided in the `Left` branch of the result. Note that `Either c = ExceptT c Identity`:

```
catchS :: Monad m
       => MSF (ExceptT e m) a b
       -> (e -> MSF m a b)
       -> MSF m a b
```

This closely resembles the signature of Yampa's *switch* combinator (Nilsson *et al.*, 2002):

```
switch :: SF a (b, Event c) -> (c -> SF a b) -> SF a b
```

In Yampa, the SF in the first argument is used by default to transform an incoming signal of type `a` into a signal of type `b` and a discrete signal of type `c` (note that `Event` is isomorphic to `Maybe`, i.e., `data Event e = NoEvent | Event e`, encoding the fact that samples may not exist at specific points). By default, the `switch` combinator outputs the value in the first component of the output signal, of type `b`, so long as the event in the second component of the output does not hold a value. If the second component holds a value, the `b` is discarded and the second argument to `switch` is immediately applied to the value in the `Event` to construct a new signal function that is turned on at that time and used in place of the original `SF` for the present and all future inputs. Because only the `b` or the `c` will be used at any sampling time, but not both, this makes the output type a sum of both, i.e., `Either`. Therefore, switching emerges for free in MSFs by virtue of combining them with the `Either` monad.

Another possibility is to use a list monad, which gives rise to constructs that maintain *dynamic collections* of monadic stream functions connected in *parallel* (Fig. 3).

The first-class status of MSFs, in combination with the extensibility provided by different monad transformers and their running functions makes MSFs unusually flexible for describing reactive systems.

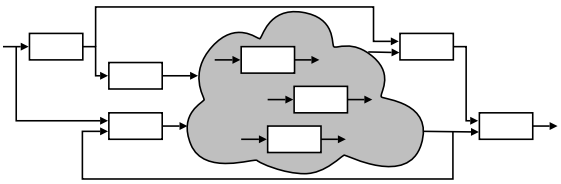

Fig. 3. System of interconnected signal functions with varying structure

2.2.6 *FRP using Monadic Stream Functions*

The framework described so far is inherently discrete and focused on functions, not streams. However, different elections of inputs, outputs and monads lets us realize streams, monadic streams, Classic FRP signals and Arrowized FRP Signal Functions.

FRP and, in particular, Arrowized FRP, are frameworks built around notions of values varying over continuous *time*. We can reintroduce time explicitly by making it available in a global environment using the `Reader` monad. This allows us to bridge the ideas in implementations like Yampa, in which Signal Functions (SFs) are pure, and MSFs, building the former in terms of the latter:

```
type SF a b = MSF (Reader DTime) a b
```

The type `DTime` conceptually represents real positive time between successive samples. (A common implementation choice, that also is adopted in the following, is to represent `DTime` as a positive `Double`.) The type constructor `SF` above, together with the associated

`step` function, is isomorphic to Yampa's internal representation of initialized Signal Functions. This makes it possible to run Yampa simulations on top of an intermediate layer that implements an API-compatible version of FRP on top of MSFs.

Introducing time allows us to define FRP constructs that depend on time explicitly. For example, integrals and derivatives are important for application domains like physics simulations, games and multimedia, and they have well-defined continuous-time semantics. Their types, just like in Yampa, are as follows (`VectorSpace` is a class in an auxiliary library capturing vector spaces, `v` represents the type of vectors, and `s` the type of scalars):

```
integral   :: VectorSpace v s => SF v v
derivative :: VectorSpace v s => SF v v
```

Both `integral` and `derivative` are signal functions, transforming one signal into another: the output signal from `integral` is the integral of the input signal from when integration started up until the present point in time; the output signal from `derivative` is the slope of the input signal at all points in time. Here, just as in typical FRP implementations, both are computed numerically over the sequence of samples representing the input signal.

Time-aware primitives like the above make specifications highly declarative. For example, the velocity v and position p at time t of a falling mass with initial velocity v_0 and position p_0 are given by the following equations of motion:

$$\begin{aligned} v(t) &= v_0 + \int_0^t -9.8\,dt \\ p(t) &= p_0 + \int_0^t v(t)\,dt \end{aligned}$$

Using Paterson's Arrow notation, these equations can be transliterated into FRP with very little syntactic noise yielding a signal function for the time-varying position parametrised over the initial position and velocity:

```
fallingMass :: Double -> Double -> SF () Double
fallingMass p0 v0 = proc () -> do
  v <- arr (+v0) <<< integral -< (-9.8)
  p <- arr (+p0) <<< integral -< v
  returnA -< p
```

Even without the arrow notation, the rendering into FRP is straightforward:

```
fallingMass :: Double -> Double -> SF () Double
fallingMass p0 v0 = arr (const (-9.8))
  >>> integral >>> arr (+ v0)
  >>> integral >>> arr (+ p0)
```

Signal functions (`SF`) are completely pure: provided that we execute them with the exact same input signal (i.e., with the exact same input samples obtained at the exact same *sampling times*), we will obtain an entirely predictable output. To allow for signal processing with actual side effects, we introduce Extensible Signal Functions (`ESF`):

```
type ESF m a b = MSF (ReaderT DTime m) a b
```

Note that the monad is explicit in the type constructor `ESF`. If a monad with no real-world side effects is chosen, the result is again a completely pure and reproducible simulation. We will use this later on to add additional context and controlled side effects.

Monadic Stream Functions can also be used with monads that depend on external input. If we combine this idea with MSFs that do not depend on their input, we obtain MSFs that produce a stream of outputs in a monadic context, or *Monadic Streams*. This allows us to also implement classic forms of FRP where signals are first class entities by representing signals as MSFs in the `IO` monad with a unit input type. Because these Monadic Streams or Signals are functors and applicatives, we can use a convenient applicative syntax to combine them and compose them. At the same time, because they are Monadic Stream Functions, we can use `Arrow` combinators and all the `MSF` functions presented so far, allowing us to merge the key ideas from reactive programming and FRP, both Classic and Arrowized.

3 A Temporal Logic for FRP

Temporal Logic is a branch of Modal Logic in which propositions are quantified over time (Emerson, 1990; Pnueli, 1977): just like the value of a signal in FRP varies over time, so does the truth value of a proposition in temporal logic. Temporal logic thus constitutes a suitable framework for expressing temporal properties of changing and reactive systems for test and verification. Our immediate interest here is testing. In the following, we introduce a temporal logic for (arrowized) FRP to that end, aiming at striking a balance between allowing temporal properties to be expressed at a conceptual level, while also allowing implementation aspects (specifically sampling) to be considered where needed. As a starting point, we first give a brief introduction to temporal logic in general, albeit with a presentation that is geared towards our specific setting and objectives.

3.1 Temporal Logic

3.1.1 The Semantic Domain

In Temporal Logic the nature of time (bounded vs unbounded, discrete vs continuous, etc.) plays a central role, determining both which modalities can be defined and which propositions hold. However, the fundamentals remain largely the same irrespective of these choices, allowing us to treat time mostly in the abstract, only assuming that it is linear (totally ordered and progressing towards the future) with a starting point denoted 0. In FRP (as originally conceived), time is conceptually continuous, but discretized during execution. We will return to the implications of this in more detail when we consider the FRP-specific temporal logic later.

Guided by the intuitive understanding of temporal logic, and in the spirit of FRP, we consider a temporal proposition to be a time-dependent Boolean function; that is, a Boolean whose value changes over time:

$$TProp = Time \rightarrow Bool$$

The semantics of temporal logics is typically given in terms of sets of states and transitions between these (Emerson, 1990). For our purposes, an FRP-style denotational se-

mantics is simpler. In the following, we thus give denotational definitions for the various temporal operators in a style that follows the denotational semantics for FRP given in Wan and Hudak's seminal paper *Functional Reactive Programming from First Principles* (Wan & Hudak, 2000).

3.1.2 Point-wise Operations

The standard logical operators, like conjunction and disjunction, are defined pointwise. We spell out the function names (instead of using operators) and use curried definitions to align the definitions here with the later FRP-specific logic, which is embedded in Haskell.

$$
\begin{aligned}
&\mathit{neg} : \mathit{TProp} \to \mathit{TProp} \\
&\mathit{neg}\ \phi = \lambda t\ .\ \neg(\phi\, t)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{and} : \mathit{TProp} \to \mathit{TProp} \to \mathit{TProp} \\
&\mathit{and}\ \phi\ \psi = \lambda t\ .\ \phi\, t\ \vee\ \psi t
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{or} : \mathit{TProp} \to \mathit{TProp} \to \mathit{TProp} \\
&\mathit{or}\ \phi\ \psi = \lambda t\ .\ \phi\, t\ \wedge\ \psi t
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{impl} : \mathit{TProp} \to \mathit{TProp} \to \mathit{TProp} \\
&\mathit{impl}\ \phi\ \psi = \lambda t\ .\ \phi\, t\ \Longrightarrow\ \psi t
\end{aligned}
$$

The meaning of these point-wise operations should be obvious. For instance, $\mathit{neg}\ \phi$ is true at a time t if ϕ is not true at that time.

3.1.3 Temporal Modalities

To make temporal logic meaningfully temporal, operators that refer to other points in time than the present are needed. The well-known Priorean operators Global, Future, History and Past (Prior, 1967) constitute one example. We introduce four temporal operators in that vein:

always	at every point in the future
eventually	at some point in the future
history	at every point in the past
ever	at some point in the past

Their definitions are:

$$\begin{aligned}&\mathit{always} : \mathit{TProp} \to \mathit{TProp}\\&\mathit{always}\ \phi = \lambda t\ .\ \forall t' \in \mathit{Time}\ .\ t' \geq t \Longrightarrow \phi\, t'\end{aligned}$$

$$\begin{aligned}&\mathit{eventually} : \mathit{TProp} \to \mathit{TProp}\\&\mathit{eventually}\ \phi = \lambda t\ .\ \exists t' \in \mathit{Time}\ .\ t' \geq t \wedge \phi\, t'\end{aligned}$$

$$\begin{aligned}&\mathit{history} : \mathit{TProp} \to \mathit{TProp}\\&\mathit{history}\ \phi = \lambda t\ .\ \forall t' \in \mathit{Time}\ .\ t' \leq t \Longrightarrow \phi\, t'\end{aligned}$$

$$\begin{aligned}&\mathit{ever} : \mathit{TProp} \to \mathit{TProp}\\&\mathit{ever}\ \phi = \lambda t\ .\ \exists t' \in \mathit{Time}\ .\ t' \leq t \wedge \phi\, t'\end{aligned}$$

As an example, the property "ϕ eventually becomes true forever" can be expressed as $\mathit{eventually}\,(\mathit{always}\,\phi)$, and the property "$\phi$ will always be true in the future, but has not always been true in the past" can be expressed as $\mathit{ever}\,(\mathit{neg}\,\phi)$ '*and*' $\mathit{always}\,\phi$, where we allow ourselves to use the Haskell-convention for infix application of two-argument functions.

Note that these operators have counterparts with strict inequalities, in which the value of the predicate argument ϕ at the current time t is not taken into account. However, the above versions fit our needs better in the following.

Two modalities *next* and *until* are sometimes taken as primitive in terms of which further modalities such as those above are defined (Emerson, 1990, Sec. 3.2). The modality *next* qualifies a proposition to pertain to the next time step, implying a setting where time is discrete. As time is conceptually continuous in FRP, we will not use *next* as such in our FRP-specific temporal logic. Nevertheless, it is helpful to understand its semantics to provide context for the alternatives that we will use:

$$\begin{aligned}&\mathit{next} : \mathit{TProp} \to \mathit{TProp}\\&\mathit{next}\ \phi = \lambda t\ .\ \phi\,(N\,t)\end{aligned}$$

where $N\,t$ is the smallest $t' \in \mathit{Time}$ such that $t' > t$. Until is defined as:

$$\begin{aligned}&\mathit{until} : \mathit{TProp} \to \mathit{TProp} \to \mathit{TProp}\\&\mathit{until}\ \phi\ \psi = \lambda t\ .\ \exists t' \in \mathit{Time}\ .\ t' \geq t \Longrightarrow (\psi\, t' \wedge \forall t'' \in \mathit{Time}\ .\ t'' < t' \Longrightarrow \phi\, t'')\end{aligned}$$

That is, $\mathit{until}\,\phi\,\psi$ means that ϕ holds at every point until ψ does.

As was remarked above, the modality *next* presupposes a setting with discrete time and is thus not suitable for FRP. Nevertheless, there is a need to qualify propositions to pertain to some specific future point in time also in a setting where time is notionally continuous. To that end, we introduce the modalities *imminently* and *after*:

$$\begin{aligned}&\mathit{imminently} : \mathit{TProp} \to \mathit{TProp}\\&\mathit{imminently}\ \phi = \lambda t\ .\ \lim_{t' \to t^+}\ .\ \phi\, t'\end{aligned}$$

$$\begin{aligned}&\mathit{after} : \mathit{Time} \to \mathit{TProp} \to \mathit{TProp}\\&\mathit{after}\ \Delta t\ \phi = \lambda t\ .\ \phi\,(t + \Delta t)\end{aligned}$$

The modality *imminently* thus refers to a point that is not now, but immediately thereafter, while *after* refers to a point Δt into the future. The former is defined in terms of the right limit of a proposition as the time approaches a given point in time from above. Again, this follows the style of denotational semantics in Wan & Hudak's paper (Wan & Hudak, 2000), and it is understood that the meaning is $\perp$ in case there is no convergence to a definitive truth value. Both *imminently* and *after* work in settings of continuous as well as discrete time, but in a discrete-time setting the limit construct in the semantics of *imminently* simply yields the next point in time meaning that the semantics of *imminently* coincides with that of the modality *next* in this case.

Note that these definitions require refinement in a setting where time is bounded from above, as they would otherwise refer to points outside the designated time domain. In the setting of FRP, this is not a concern at the conceptual level, as time is unbounded from above. However, when it comes to execution and testing, we need to account for the fact that the time domain, as defined by the points at which observations are made, necessarily becomes discrete and finite. We will return to this and related points in the next section, where we develop a testable temporal logic for FRP.

3.2 Testable Temporal Predicates

The semantic domain of temporal logic, as given above, is bigger than that of FRP. While FRP Signal Functions must be *causal*, functions over temporal propositions may not be so. For example, modalities like *always* make the present depend on the future and thus cannot be implemented faithfully in Arrowized FRP as Boolean-carrying signals. There are also efficiency concerns: even if we only consider modalities that look to the present or the past, *history*, for example, depends on *all the past*, which could lead to memory leaks.

We can address these problems by limiting our attention to testing after-the-fact on a given, discretized time-bounded input signal, what we can think of as the *observed input*, letting this implicitly define the time domain, and adapting the semantics of the temporal modalities accordingly.

In this section, we use this idea to present a testable encoding of a version of Linear Temporal Logic (LTL). In Section 5, we complement this approach with an encoding of a variant of Past-time Linear Temporal Logic (ptLTL) that can be implemented efficiently in FRP in the form of Boolean-carrying Signals, to define monitors or temporal assertions checked on the fly as applications run live.

The following encoding of LTL, framed as temporal predicates on an input signal of type `a`, allows us to look into the present and the future. We redeploy a selection of the modalities from the previous section, turned from propositions into predicates by abstracting over the input. In addition we introduce a basic value constructor `SP`, for Signal Predicate, to allow the full power of FRP to be used to define basic predicates on signals.

```
data TPred a where
  SP         :: SF a Bool -> TPred a
  And        :: TPred a   -> TPred a -> TPred a
  Or         :: TPred a   -> TPred a -> TPred a
  Not        :: TPred a   -> TPred a
  Implies    :: TPred a   -> TPred a -> TPred a
```

```
Always     :: TPred a   -> TPred a
Eventually :: TPred a   -> TPred a
Until      :: TPred a   -> TPred a -> TPred a
Imminently :: TPred a   -> TPred a
After      :: Time      -> TPred a -> TPred a
```

The semantics is as given for the corresponding temporal propositions in the previous section, with a couple of exceptions as the time domain is now discrete and bounded. The semantics of the pointwise operations carries over unchanged, as these do not depend on the nature of time. The semantics of the propositions *always*, *eventually*, and *until* also carries over unchanged, except time is now quantified over a finite number of time points.

In contrast, the semantics for the temporal predicates `Imminently` and `After` needs a bit of refinement. For a discrete time domain, as discussed above, the limit construct used to define the semantics of the proposition *imminently* simply degenerates to N, the next point in time, so were it not for the fact that time now also is bounded, the semantics of `Imminently` would be like that of *next*. However, because time now is bounded by the period over which input was observed, there is a possibility that predicates constructed using `Imminently` and `After` may refer to time points outside the time domain of the specific input signal under consideration.

The question, then, is how to evaluate temporal predicates at such time points: are they true or false? The honest answer is that we do not know, suggesting that a three-valued logic should be used. But in practice, as unknown is "contagious", one would have to be quite careful to avoid stating properties in such a way that unknowns do not mask useful information. Thus, for simplicity, we define both `Imminently` and `After` to be true for time points outside the current domain. This is also consistent with the setting of testing: tests pass in the absence of evidence of failure.

Consequently, for the most part, temporal properties can be stated at a conceptual level, as if time were continuous and unbounded from above, as long as we keep in mind that testing on a specific discretized input signals might be successful even if a property actually does not hold in continuous, unbounded time. For example, if we claim that something is always true, but, as it happens, is is only *almost* always true, it may take many runs before we find a failing case. This is to be expected: after all, we are testing, not proving properties. Our chances of successfully identifying properties that do not hold increase greatly if we employ techniques for automated, principled testing, such as QuickCheck, which we explore in Section 4.

Nevertheless, sometimes properties that would hold if time were continuous may fail under adverse sampling conditions. For example, imagine a control system designed to keep a position of an object within certain boundaries as long as disturbances are within predetermined bounds (e.g., an inverted pendulum). Such a system would often be provably correct. However, if a discretized realisation were sampled sufficiently sparsely, it would fail to maintain the position within the expected bounds.

One way to address this is to only test on discretized input signals with a sufficiently high sampling rate. This is our current main approach, but it has the drawback that the sampling assumptions are not captured by the stated temporal properties. An alternative is to introduce predicates on the "quality" of the input signal sampling, such as a minimal

difference between sample times or a sufficiently high running average of the sampling frequency. Obviously, such predicates break FRP's continuous-time abstraction, but does allow stating that sampling meeting certain criteria implies desired correctness properties. This is already possible using `SP`, but such properties could also be stated declaratively at the level of temporal predicates. An advantage of the latter would be that system-wide minimal quality requirements could be inferred. This remains future work.

There are many different approaches to implementing FRP, some of which may be less susceptible to problems like that outlined above. The point here is not the pros and cons of various implementation approaches, but rather that effectively employing property-based testing for FRP may necessitate going beyond supporting stating ideal properties and also cater for operational aspects. The specifics of the latter of course depend on the nature of the underlying implementation.

3.3 Evaluation of Temporal Predicates

We can think of a `TPred a` for some type `a` as a possibly non-causal function that takes a Signal of type `a` defined on a time domain and returns a Temporal Proposition (a Signal of type `Bool`) on the same time domain. To evaluate a `TPred`, we thus need to provide an input signal. In our implementation, we provide both the signal and the time domain in the form of a finite input or sample *stream*:

```
type SignalSampleStream a = (a, [(DTime, a)])
```

`SignalSampleStream` represents the signal starting from the first sample, always at time 0, and a subsequent streams of samples samples spaced by strictly positive delays.

We can now give an evaluation function that takes a Temporal Predicate and a sample stream and evaluates the temporal predicate at the initial time:

```
evalT :: TPred a -> SignalSampleStream a -> Bool
```

When we evaluate a `TPred` by providing a stream of input samples, we are effectively restricting the time domain to a subset defined implicitly by the times of the samples in the input stream, as discussed earlier. Note that the function `evalT` only lets us query the value of a `TPred` at the initial time, but we can always refer to specific future times using `After`.

3.4 Examples

Let us now introduce an example of an FRP program, define temporal assertions and verify them. The presentation should be reasonably self-contained, but as the main focus of this paper is not FRP programming as such, the reader may wish to consult a suitable reference on FRP such as (Nilsson *et al.*, 2002) for details, in particular in relation to events and switching.

We start with a simple animation of a falling ball, considering only the vertical axis:

```
fallingBall :: Double -> SF () Double
fallingBall p0 = proc () -> do
  v <- integral -< -9.8
```

```
  p <- integral -< v
  returnA -< (p0 + p)
```

To check, for example, that the position of the falling ball at any time is lower than the initial position, one can define the following property:

```
ballFellLower :: Double -> TPred ()
ballFellLower p0 = SP (fallingBall p0 >>> arr (\p1 -> p1 <= p0))
```

We now test this property in a session, using an input stream `stream01` of 42 samples spaced by 0.1 seconds, carrying no data (that is, a stream `((), [(0.1, ()), (0.1, ()), ...])` with 42 samples in total).

```
> evalT (ballFellLower 100) stream01
True
```

However, with the given definition we are only checking this proposition at time 0. In order to check that the proposition always holds, we define:

```
ballFallingLower :: Double -> TPred ()
ballFallingLower p0 = Always (ballFellLower p0)
```

Testing it now tests it for every position in the stream:

```
> evalT (ballFallingLower 100) stream01
True
```

To obtain further guarantees, we may want to check that the new position of the ball is always lower than the previous one. We can express that idea by looking at the *derivative* of the position and ensure it is always strictly negative:

```
ballTrulyFalling :: Double -> TPred ()
ballTrulyFalling p0 =
   Always (SP (fallingBall p0 >>> derivative >>> arr (\v -> v < 0)))
```

However, this predicate does not hold at time 0 as the derivative is not defined there.[1] We could explicitly initialize the derivative to some specific negative value, but a more appealing approach is to say that the derivative should always be negative immediately after time 0:

```
ballTrulyFalling' :: Double -> SF () Double
ballTrulyFalling' p0 =
  Imminently
    (Always
```

[1] Mathematically, differentiability of functions of one variable is defined on *open* intervals, as differentiability at some point p requires the function to be defined everywhere in some neighbourhood around p. To avoid depending on future time points, `derivative` in FRP-implementations is typically a numerical approximation of the mathematical notion of a *left derivative*, requiring a function to be defined everywhere in some neighbourhood to the *left* of a point p to be differentiable at that point. Either way, in FRP, `derivative` is not defined at time 0 as FRP's signals are not defined prior to time 0.

```
       (SP (fallingBall p0 >>> derivative >>> arr (\v -> v < 0)))

> evalT (ballTrulyFalling' 100) stream01
True
```

Let us now consider the case that the ball changes direction and bounces up when it hits the floor. To implement this behaviour, we use a mechanism known as *switching*, implemented by the combinator `switch`, in which a signal function is applied to a signal until a condition holds, and a different signal function is applied from that point on (Nilsson *et al.*, 2002). Switching can also be used to restart a signal function with different initial conditions, as demonstrated by this example.

```
bouncingBall :: Double -> Double -> SF () (Double, Double)
bouncingBall p0 v0 =
  switch (fallingBall'' p0 v0 >>> (identity &&& hit))
         (\(p0', v0') -> bouncingBall p0' (-v0'))

fallingBall'' :: Double -> Double -> SF () (Double, Double)
fallingBall'' p0 v0 = proc () -> do
  v <- arr (v0 +) <<< integral  -< -9.8
  p <- arr (p0 +) <<< integral  -< v
  returnA -< (p, v)

hit :: SF (Double, Double) (Event (Double, Double))
hit = arr (\(p0, v0) -> if (p0 <= 0 && v0 < 0)
                        then Event (p0, v0)
                        else NoEvent)
```

The signal function `bouncingBall` starts with a ball falling down, until it hits the floor, and restarts the signal from the point of collision, inverting the direction of the velocity. The signal function `fallingBall''` is the same as `fallingBall`, except that it takes an initial velocity and outputs the current velocity. The signal function `hit` creates an `Event` when the ball hits the floor, that is, when the position is not positive and the ball is going down. The `switch` combinator uses the information in the `Event`, if present, to switch on the second signal function, which, in this case, refers to `bouncingBall` with the velocity negated.

If we were to translate the property `ballFellLower` to this new definition, we could assume that, with perfect elasticity and if the initial velocity is 0, the ball would never bounce higher than the initial position:

```
ballLower :: Double -> TPred ()
ballLower p0 =
  Always (SP (bouncingBall p0 0 >>> arr (\(p1,v1) -> p1 <= p0)))
```

If we now test this predicate with a stream we see that it does not hold in our program. To give the ball enough time to bounce back up, we use a stream of 42 samples spaced by 0.5 seconds.

```
> evalT (ballBouncingLower 100) stream05
False
```

If we print the ball position at every time, we obtain:

```
[ 100.0,                 100.0,               97.55
,  92.65,                85.3 ,               75.5
,  63.25,                48.55,               31.400000000000006
,  11.800000000000011,  -10.249999999999986,  14.25000000000001
,  36.30000000000001,    55.900000000000006,  73.05
,  87.74999999999999,    99.99999999999999,   109.79999999999998
, 117.14999999999998,   122.04999999999997,   124.49999999999996
, 124.49999999999996,   122.04999999999997,  117.14999999999996,  ...
```

The ball, dropped from 100 points with no vertical velocity, bounces up to 124.5 points. This is not the result of small floating-point inaccuracies or rounding errors, which could be accounted for by introducing a margin of error in the equality for floating-point numbers. Instead, it is caused by errors introduced by the implementation of integral using the rectangle rule. We could address this particular issue by providing a more accurate integral, or by manually capping `bouncingBall` to be lower than `p0`. This, however, is out of the scope of this paper.

Another problem with this simulation is that the simulated physics is not time-continuous and that the ball is not pulled out of the floor when it collides. Thus, the ball will temporarily be rendered as if it had penetrated the floor. This can be seen in the previous trace, with the ball position being -10.25 on the 11th sample, which the following property detects:

```
ballOverFloor :: Double -> TPred ()
ballOverFloor p0 =
  Always (SP (bouncingBall p0 0 >>> arr (\(p1, v1) -> p1 >= 0)))
```

```
> evalT (ballOverFloor 100) stream05
False
```

For a proposal on how to address this problem by implementing continuous physics in Arrowized FRP, see Perez *et al.* (2016).

4 QuickChecking

In the previous section we saw how to express and test Temporal Properties of an FRP program. The input streams, however, were manually generated, which limits the coverage of our tests. As explained in Sec. 3.2, this limits the conclusions we can draw from testing, unless we can use a more principled approach to explore the space of possible inputs. For instance, if we test the predicate `ballOverFloor` dropping the ball from a slightly higher height, the property would not be violated for a stream `stream05'` with 21 samples spaced by 0.5 seconds, but it would be for one stream with 42 samples or one with 0.1-second delays instead:

```
> evalT (ballOverFloor 110.24999999999999) stream05'
True
```

```
> evalT (ballOverFloor 110.24999999999999) stream05
False
```

QuickCheck (Claessen & Hughes, 2000) is a tool that combines random data generation with a property language in order to test code more thoroughly. Properties are defined using combinators to express conditions on values. For example, a property stating that reversing a list twice leaves it unchanged could be defined and tested as follows:

```
propReverseTwice :: [Int] -> Property
propReverseTwice xs = reverse (reverse xs) == xs
```

```
> quickCheck propReverseTwice
+++ OK, passed 100 tests.
```

If we state an incorrect predicate, for instance, that `reverse` is the identity function on lists, QuickCheck finds and prints a counter-example that invalidates our assertion (in this case, the predicate is false for the input `[1,0]`):

```
propReverseOnce :: [Int] -> Property
propReverseOnce xs = reverse xs == xs
```

```
> quickCheck propReverseOnce
*** Failed! Falsifiable (after 3 tests and 1 shrink):
[1,0]
```

The confidence we can place in tests against randomly generated data depends on the nature of such data. We may want to constrain data to meet a function's precondition or mimic expected user input. Pre-conditions can be defined by means of *filters*, like the following, which states that the property defining how the functions `head` and `tail` relate to one another only makes sense for non-empty lists:

```
propHeadTail :: [Int] -> Property
propHeadTail xs = not (null xs) ==> (head xs) : (tail xs) == xs
```

Filtering data can make the search inefficient when most of the data generated does not meet the preconditions. Additionally, randomly-generated data may not explore the corner cases of our solution. To address these concerns, QuickCheck defines a language of generators, consisting of a series of types, classes and combinators operating on values that can be randomly generated.

In this section we demonstrate how to use the Temporal Language described in Section 3 to test temporal properties of FRP programs with QuickCheck. We do so by providing a series of input stream generators, together with combinators to constrain the kinds of streams generated. In Section 8 we demonstrate how this approach helped us find bugs in real systems.

4.1 Stream Generators

Generating suitable inputs for our tests requires that we provide random, but reasonable, input signal values and sampling times, constructing what we called a sample stream or *input stream*.

Our basic definition of input streams are signal samples paired with strictly positive time deltas. The data corresponding to each input is application-specific, but we can provide general-purpose generators to create random sampling times.

In order to generate lists of time samples, we use the following general function:

```
generateStream      :: Arbitrary a
                    => Distribution
                    -> Range
                    -> Length
                    -> Gen (SignalSampleStream a)
```

where `Gen` is QuickCheck's type constructor for value generators. This function takes three parameters whose types are defined as follows:

```
data Distribution = DistConstant
                  | DistNormal (DTime, DTime)
                  | DistRandom
type Range        = (Maybe DTime, Maybe DTime)
type Length       = Maybe (Either Int DTime)
```

The type `Distribution` represents how the values for the time deltas are chosen within the ranges specified. The case `DistConstant` represents the case in which the time delta is random, but the same for all samples in the stream. `DistNormal` represents a normal (Gaussian) distribution with a given average and sigma coefficients. The case `DistRandom` represents a more general scenario, where the time deltas are uniformly distributed in the given range, if the range has an upper bound (the smallest time delta is considered a lower bound if there is none), and otherwise rely on QuickCheck's standard generator for values of type `DTime` (positive `Doubles`). The type `Range` describes possible lower and upper boundaries for the generated time deltas. The smallest representable positive double-precision floating point number is considered the lower bound in the range if none is specified. The type `Length` describes the length of the stream, either in number of samples or in time length. This parameter may be nothing, in which case the function `generateStream` initially picks a random number of samples for the generated stream.

The following auxiliary function allow us to generate streams with an auxiliary generator that depends on the sample number and the absolute time:

```
generateStreamWith :: Arbitrary a
                   => (Int -> DTime -> Gen a)
                   -> Distribution
                   -> Range
                   -> Length
                   -> Gen (SignalSampleStream a)
```

We provide additional facilities to make code shorter:

```
uniDistStream :: Arbitrary a
              => Gen (SignalSampleStream a)
uniDistStream =
  generateStream DistRandom (Nothing, Nothing) Nothing
```

```
uniDistStreamMaxDT :: Arbitrary a
                   => DTime -> Gen (SignalSampleStream a)
uniDistStreamMaxDT maxDT =
  generateStream DistRandom (Nothing, Just maxDT) Nothing

fixedDelayStream :: Arbitrary a
                 => DTime -> Gen (SignalSampleStream a)
fixedDelayStream dt =
  generateStream DistConstant (Just dt, Just dt) Nothing
```

4.2 Streams

The previous API lets us generate random streams, but in order to pre-feed existing data or express complex properties such as that a signal function behaves the same regardless of how often it is sampled, we need additional ways to constrain streams. Recall that streams in our framework are defined as follows:

```
type SignalSampleStream a = (a, [(DTime, a)])
```

Streams can be concatenated, as well as merged. Because streams start at time zero, stream concatenation requires an additional time difference (`DTime`) to separate the last sample of the first stream from the first sample of the second stream. Merging needs an auxiliary function to determine what to do if the two streams provide a sample for the same time:

```
sConcat :: SignalSampleStream a
        -> DTime
        -> SignalSampleStream a
        -> SignalSampleStream a

sMerge  :: (a -> a -> a)
        -> SignalSampleStream a
        -> SignalSampleStream a
        -> SignalSampleStream a
```

We also provide clipping functions that allow us to drop samples before or after a specific time:

```
sClipAfterFrame  :: Int   -> SignalSampleStream a
                 -> SignalSampleStream a
sClipAfterTime   :: DTime -> SignalSampleStream a
                 -> SignalSampleStream a
sClipBeforeFrame :: Int   -> SignalSampleStream a
                 -> SignalSampleStream a
sClipBeforeTime  :: DTime -> SignalSampleStream a
                 -> SignalSampleStream a
```

These functions work in the intuitive way, and do not re-adjust the time deltas to make up for samples being removed. For example, the function `sClipAfterTime` drops any sample that takes place after the given time, leaving the stream unchanged if the time is after the last sample in the stream.

4.3 Examples

We can define the QuickCheck property quantified over input streams as follows:

```
propTestBallOverFloor =
  forAll myStream (evalT (ballOverFloor 110.24999999999999))
  where  myStream :: Gen (SignalSampleStream ())
         myStream = uniDistStream
```

QuickCheck finds a counter-example in only four tries:

```
> quickCheck propTestBallOverFloor
*** Failed! Falsifiable (after 4 tests):
((),[(4.437746115172792,()),(1.079898766664353,())
    ,(3.0041170922472684,())])
```

The counter-example generated by QuickCheck contains very large time deltas. In a realistic scenario, with a screen refresh rate of 60Hz standard on PC and phones, time deltas would approximate $0.016s$. We can test the previous property with this different generator and see how it behaves in ideal conditions:

```
propTestBallOverFloorFixed =
  forAll myStream (evalT (ballOverFloor 110.24999999999999))
  where  myStream :: Gen (SignalSampleStream ())
         myStream = fixedDelayStream (1/60)

> quickCheck propTestBallOverFloorFixed
+++ OK, passed 100 tests.
```

The fact that this test passes is, however, a result of exploring few and small traces. Before, our ball was not bouncing until several seconds into the simulation, and with a time delta of barely 16ms, it takes several hundred samples to reach the floor. If we explore more cases and larger input streams, we again find situations in which the program misbehaves:[2]

```
> quickCheckWith (stdArgs {maxSuccess = 1000, maxSize = 300})
                 propTestBallOverFloorFixed
*** Failed! Falsifiable (after 897 tests):
((),[(1.6666666666666666e-2,()),(1.6666666666666666e-2,())
    ,(1.6666666666666666e-2,()),...
```

[2] This hints to a bigger problem with our physics simulation: Because we do not use information about the objects in the system and their positions, velocities and accelerations, to determine *when* to sample the simulation, there is always a chance that we may miss the exact time when the ball hits the floor, and only sample too early or too late. This is known as *tunneling* or the *bullet-through-paper* effect, and is discussed further in Sec. 8 and in Perez *et al.* (2016).

While the generator `fixedDelayStream` simulates ideal conditions, in realistic scenarios we cannot expect every frame to last exactly $1/60$ seconds. In practice, we find it useful to have more control over how small or uniform delta times are, by means of `uniDistStreamMaxDT`, setting instead a maximum time delta and accepting that, if, for example, a simulation runs slower, the physics may not be as realistic.

We can use the given interface to generate sampling streams prefixed with real user input, and have QuickCheck generate random samples after a particular point in time when we suspect a bug may exist. The following generator completes a run after 1 minute for approximately 10 additional seconds with 16ms delays:

```
tenAdditionalSecondsAfterMinute :: SignalSampleStream ()
                                -> Gen (SignalSampleStream ())
tenAdditionalSecondsAfterMinute userStream =  do
  qcStream <- generateStream (DistNormal (0.016, 0.002))
                             (Nothing, Nothing)
                             (Just (Right 10))
  let clippedUserStream = sClipAfterTime 60 userStream
  return (sConcat clippedUserStream 0.016 qcStream)
```

In Section 8 we show how we used QuickCheck to test properties of more complex applications, and explain how it helped us find and address a fundamental problem in a game.

4.4 Testing Abstract Properties

Our approach to purely functional reactive testing is also useful to test properties of FRP implementations, general properties of Signal Functions, such as statelessness, or check that the Temporal Logic is sound by testing tautologies.

We begin with a simple test of equality between two signal functions. This is useful for checking that an optimized implementation fulfills a specification:

```
alwaysEqual :: Eq b => SF a b -> SF a b -> TPred a
alwaysEqual sf1 sf2 =
  Always (SP ((sf1 &&& sf2) >>> (arr (uncurry (==)))))

propTestAlwaysEqual =
  forAll myStream (evalT (alwaysEqual (arr (**2)) (arr (^2))))
  where  myStream :: Gen (SignalSampleStream Double)
         myStream = uniDistStream
```

We can use this generic predicate to test if two specific signal functions perform the same transformation:

```
> quickCheck propTestAlwaysEqual
+++ OK, passed 100 tests.
```

This approach can be used to test underlying properties of reactive frameworks like Yampa and Dunai. For instance, we could check whether certain arrow laws (Hughes, 2000; Paterson, 2001) hold for signal functions:

```
propArrowIdentity =
  forAll myStream (evalT (alwaysEqual identity (arr id)))
  where  myStream :: Gen (SignalSampleStream Double)
         myStream = uniDistStream
```

```
> quickCheck propArrowIdentity
+++ OK, passed 100 tests.
```

The above property expresses, for a limited input domain, that the identity for signal functions (`identity`) is always equal to lifting (`arr`) the identity function (`id`). Testing more complex laws requires that we generate, again, suitable random data. Consider, for instance, the following attempt at testing that composition with `identity` leaves a signal function unchanged:

```
propCompositionRightIdentity =
  forAll myStream
    (evalT (alwaysEqual (arr (**2) >>> identity) (arr (**2))))
  where  myStream :: Gen (SignalSampleStream Double)
         myStream = uniDistStream
```

```
> quickCheck propCompositionRightIdentity
+++ OK, passed 100 tests.
```

This test is rather limited, since it only checks that a specific `SF`, `arr (**2)`, has a right identity for `>>>`. We can use QuickCheck's function generators to quantify our predicate over functions from `Int` to `Int`:

```
propTestAlwaysEqualIntFunctions =
   forAll dataStream $ \s ->
     forAll streamFunction $ \f ->
       evalT (alwaysEqual (arr (apply f) >>> identity)
                          (arr (apply f)))
             s

  where dataStream :: Gen (SignalSampleStream Double)
        dataStream = uniDistStream

        streamFunction :: Gen (Fun Int Int)
        streamFunction = arbitrary
```

This property holds as we would expect:

```
> quickCheckWith (stdArgs {maxSuccess = 1000, maxSize = 300})
                 propTestAlwaysEqualIntFunctions
+++ OK, passed 1000 tests.
```

5 Debugging

The previous facilities allow us to treat the program as a closed box and test its behaviour from the outside against a complete execution trace. This section introduces two facilities

to debug programs as they run: temporal assertions, checked during runtime, and tools for analysing the progress of an FRP simulation. This will help us determine not only whether programs fail, but also exactly when and where. For simplicity we use the definition of Signal Functions without a monad, introduced in Section 2, to which we refer as Yampa's Signal Functions. Our system, however, also works for general Monadic Stream Functions, as will be exemplified in Sections 6 and 7.

5.1 Temporal Assertions

Some of the temporal constructs presented in previous sections require arbitrary amounts of past or non-causality (depending on the future). Modalities like `always` and `eventually`, which look into the future, only become decidable once we reach the end of the input stream. Until that point, they are semi-decidable: `always` can be falsified, if we find that the condition does not hold at some point, while `eventually` can be verified, if we find out that the condition holds at some point.

To monitor temporal assertions as programs execute without the need for a multi-valued logic, we are limited in the language to causal modalities we can implement efficiently. Non-causal temporal propositions will need to be transformed into causal ones, and asked with respect to a later point in time. For example, if one examines a limited trace with sampling times $[t_0, t_1, \ldots, t_n]$ it is clear that if a condition holds for all points greater or equal than t_0, then it holds for all points earlier or equal to t_n.

5.1.1 Implementation of Temporal Logic inside FRP

We introduce the type `SPred` as a signal carrying a Boolean, which represents causal Temporal Predicates that can be defined as signal functions.

```
type SPred a = SF a Bool
```

Point-wise operators like `not` or `and` have straightforward implementations by lifting the existing Haskell implementation of those logical operators over Booleans to the signal function level:

```
notSP   sp      = sf >>> arr not
andSP   sp1 sp2 = (sp1 &&& sp2) >>> arr (uncurry (&&))
orSP    sp1 sp2 = (sp1 &&& sp2) >>> arr (uncurry (||))
implySP sp1 sp2 = orSP sp2 (notSP sp1)
```

Temporal modalities that refer to the past can be easily described using signal function combinators. We implement `history`, which checks a condition at every point and becomes `False` forever as soon as the internal condition becomes `False`, as follows:[3]

```
history :: SPred a -> SPred a
history sf = feedback True $ proc (a, last) ->
```

[3] We use the combinator `feedback` for consistency with prior sections. In Yampa, this function is called `loopPre`.

```
  b <- sf -< a
  let cur = last && b
  returnA -< (cur, cur)
```

Similarly, we can define `ever`, which checks if a condition ever held.

```
ever :: SPred a -> SPred a
ever sf = feedback False $ proc (a, last) ->
  b <- sf -< a
  let cur = last || b
  returnA -< (cur, cur)
```

We can also insert simple predicates into signal functions, for example:

```
ballAboveFloor :: SF () (Double, Bool)
ballAboveFloor = proc () -> do
  ballPos <- bouncingBall -< ()
  let aboveFloor = ballPos >= 0
  returnA -< (ballPos, aboveFloor)
```

This temporal language, based on Past-time Linear Temporal Logic (ptLTL), complements the definitions in section 3 based on (future-time) Linear Temporal Logic (LTL). In particular, the type `SPred` matches the type of the argument required by the `SP` value constructor of the `TPred` data type. Both approaches can be used in combination, making for a very expressive temporal logic language.

Definitions like the one above produce Boolean signals but, in order to report these violations (e.g., to print them or to record them in a log), they need to be passed as output of the signal function, affecting the types and definitions of the function that use that signal function, and so on, all the way up to the top-level signal function. This makes this approach suboptimal when we try to debug programs with minimal changes. Low-level workarounds with `Debug.Trace` are not portable to mobile platforms, and `unsafePerformIO` might hinder referential transparency across executions unless introduced with care.

Using the full power of Monadic Stream Functions, we could, instead, define a debugging monad that logs assertions that are violated and the times when that happens. We use the type synonym `ESF`, introduced in Section 2, to extend signal functions with additional monads using `MSF` combinators:

```
type AssertionId = String
type DebuggingMonadT = WriterT [(AssertionId, DTime)]

assert :: AssertionId -> ESF (DebuggingMonadT m) Bool ()
assert assertionId = proc (val) -> do

  let optionallyLog (t1, v1) = when v1 (tell (assertionId, t1))

  t  <- localTime          -< ()
  () <- arrM optionallyLog -< (t, val)

  returnA -< ()
```

While this also changes the types in our program, we only need to make this change once to introduce the debugging monad regardless of how many assertions we introduce. We could then use assertions as follows:

```
ballAboveFloorM :: ESF (DebuggingMonadT m) () Double
ballAboveFloorM = proc () -> do
  ballPos <- bouncingBall -< ()
  ()      <- assert "Ball must be above the floor" -< (ballPos >= 0)
  returnA -< ballPos
```

5.2 Time-travel Debugger

Debugging systems using the facilities provided so far is technically possible, but can be cumbersome. In many cases, visual inspection is needed to understand whether a problem has occurred and why.

With pure, Arrowized FRP, users can record the inputs and sampling times while they play and send them to developers indicating the time when a bug manifested. Developers can later replay these traces and move forward to that time, run additional tests, introduce assertions and visualize the problem with total reliability.

To facilitate this task, we have created an FRP time-travel debugger for Yampa. Our system consists of two components: a extension to Yampa's main execution function that allows controlling and recording simulations, and a Graphical User Interface that connects to running Yampa applications via the network and allows controlling the debugger. This implementation uses internal Yampa definitions, and it does not currently work for full Monadic Stream Functions. We discuss this further in Section 10.

5.2.1 Yampa time-travel debugger

We have extended Yampa's main execution function with a communication channel to send messages to and receive commands from an external debugger. Our implementation also carries additional state, saving the history of all the inputs and sampling times, as well as simulation preferences.

At every main loop iteration, our program checks for incoming commands. Features supported by our system include saving the input trace to a file, loading or substituting input samples, communicating the contents and time of an input sample, pausing, stopping and playing the simulation, moving or skipping steps forwards and backwards, playing until a certain condition is met and indicating when assertions are violated.

The communication with the remote debugger takes place via two sockets: a synchronous one to receive commands and send responses, and an asynchronous one to notify interesting events to the remote debugger. The remote debugger listens on the asynchronous channel and, when it detects an event, or when instructed by the user, uses the synchronous socket to send commands and obtain results. This lets us implement the remote debugger in a reactive way, with less knowledge of the internals of how signal function execution is implemented, as described in Perez & Nilsson (2015). The FRP simulator runs locally on the device where we want to test the system, which can be a developer's computer, but also can be an external device.

5.2.2 FRP Time-travel Debugger GUI

The second component of our system consists of a Graphical User Interface (Figure 4) that allows us to connect to, follow and control an application running remotely on a phone, a tablet, or a computer. Apart from executing the commands provided by the extended FRP execution function, this GUI enables two advanced use cases.

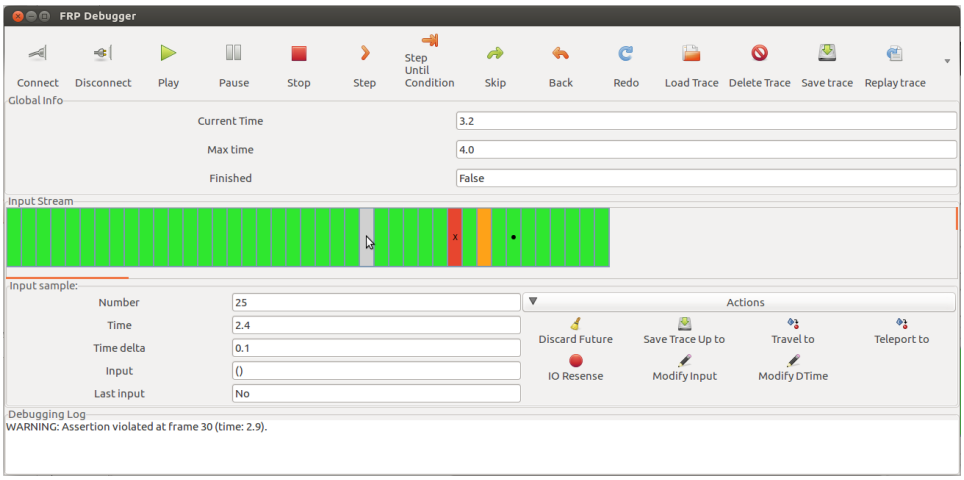

Fig. 4. Screenshot of our FRP system debugger, with the central area showing a loaded input stream being used to execute a simulation on an external device. The screenshot shows a sample in which a temporal assertion was violated (red), the current sample according to the FRP program running on a phone (orange), a breakpoint (dot), and the sample selected by the user (grey).

First, because we can save and reload traces and they are fully referentially transparent, developers can run the traces provided by users and visualize, with absolute guarantee, what the user saw, provided that the bug was not in the Input/Output layer of the system. Once a bug is detected, they can go back in time, step by step or as far back as desired, and find the points when assertions were violated. As the FRP program, running on a phone or other device, follows the debugging GUI as it moves along the trace, it will actually show the animation going backwards, forwards and jumping steps as instructed, which is an excellent visual aid for developers. Furthermore developers can hot-swap the application, that is, make changes to the program, recompile it, restart it on the phone, and take it back to the same point in time to see if the bugs persist, all without having to close the debugger. This is discussed further in Sec. 10.

Second, and aided by this first feature, we can take the user traces and feed them to QuickCheck in order to find bugs. When users see the effect of a bug in a game, we can use the Stream Manipulation API presented in Section 4 to instruct QuickCheck to take only a portion of that stream and add random samples to it, to try and find an earlier point in time at which the bug already manifests. Because QuickCheck generates new input traces as counter-examples, we can save them in files, load them in the debugger and on the phone, and visualize the issue. So, in effect, we can see QuickCheck play.

As an example, let us show how we can use this approach to debug the input stream provided by QuickCheck invalidating the property `propTestBallOverFloor` (Section 4).

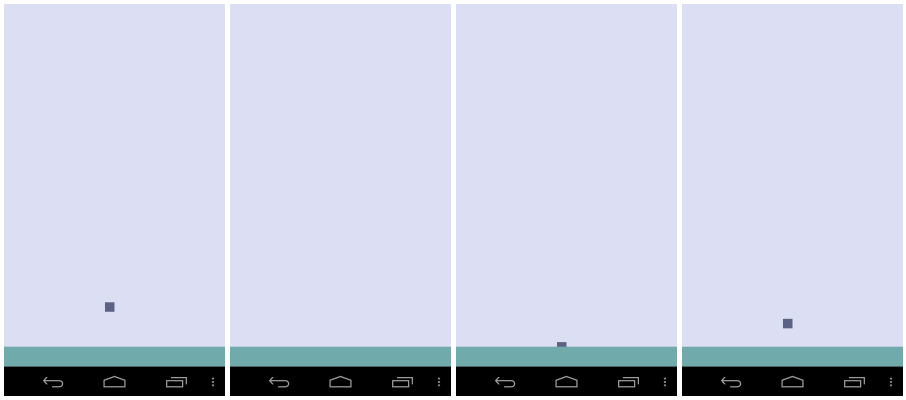

Fig. 5. Screenshots of a sample simulation running on an external device being remotely controlled using the FRP debugging GUI, executing the counter-example generated by QuickCheck, step-by-step. The ball is under the floor after 1 frame and takes 2 frames to come back to the screen area. Frames #2 and #3 produce assertion violations while the ball is below the floor.

If we connect an external device running this application with the debugger, we can use the GUI to load the counter-example input stream generated by QuickCheck and visualize on the device the point at which our assertion fails (Figure 5).

The fact that we can rely on Haskell's purity and explicit, strong types to obtain the referential transparency needed to debug deterministically supports the idea that pure Functional Programming is a very good fit for developing many kinds of applications, including reactive systems, physics simulations, multimedia systems, and games.

6 Testing Hardware and External Systems

FRP can be used to program software simulations and games, as well as to model reactive systems that can later be implemented in hardware or in a different language. While simulations can help us understand how a system may behave under some conditions, the real-world realisation may differ from expectations. In order to provide any guarantee of reliability, we need to be able to compare the realization (software or hardware) with a model, and to introduce monitors to detect property violations.

Because MSFs are inherently extensible, we can trivially replace one component in a large MSF network by an external system implementing the same functionality, what is known as *software-* and *hardware-in-the-loop* simulation.

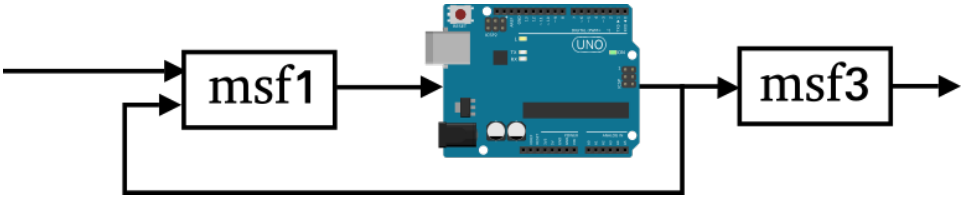

Fig. 6. A depiction of an MSF that combines hardware (represented by the Arduino board) and software components (represented by the black rectangles) to perform hardware-in-the-loop testing.

We can verify that the new MSF with the external component behaves as expected in multiple ways, such as by running it in parallel with the old model and comparing the outputs, or by running a battery of tests against on it. In both of these cases, we can take advantage of the randomized testing and temporal languages introduced earlier to produce simpler, more comprehensive tests with lower production and maintenance costs.

For simplicity, the temporal languages described in earlier sections were based on signal functions without side effects, as defined in Yampa. This and the following section assume to have a similar language for Monadic Stream Functions, in which `TPred` has an additional monad argument (i.e., `data TPred m a`), the value constructor `SP` takes a Monadic Stream Function as argument (i.e., `SP :: MSF m a Bool -> TPred m a`), and the evaluation function `evalT` returns a Boolean in a monadic context (that is, `evalT :: TPred m a -> SignalSampleStream a -> m Bool`). Due to the way that MSFs work, the type `SignalSampleStream` in this context is defined as the type synonym `SignalSampleStream a = [(DTime, a)]`, and the generators in our implementation simply generate 0 always for the time delta of the first sample. The implementation of `SP` for Monadic Stream Functions requires an `ESF`, which always includes time in a `Reader` environment and is required to give semantics to modalities such as `After`. As time is not essential in this section, we omit that detail. This simplification does not invalidate our claims.

Determinism and external systems Our approach to testing reactive systems depends greatly on tests being referentially transparent. While determinism is guaranteed by the compiler for *pure* or monad-polymorphic MSFs, the same is not true for MSFs that enclose or communicate with external software or hardware components.

To apply the same testing approach to MSFs with stateful components in-the-loop, we need to have available a mechanism to "reset" their state. Such a mechanism is likely to be very dependent on the specific component, and there is no simple way to capture that requirement in our language. In this section we assume to have a resetting operation available, like, for example, a function `reset :: IO ()` that resets the state of the system under test.

6.1 Software in the Loop Testing

We can now take a simple implementation and replace one component by a piece of hardware that implements the same functionality.

Let us demonstrate with an example of an adder, an MSF that adds all incoming numbers and outputs the sum so far, which we can implement with our language by keeping the current sum in an accumulator and feeding that value back to the same MSF, as follows:

```
adder :: MSF m Int Int
adder = feedback 0 (arr (dup . uncurry (+)))
```

We can implement similar functionality in a different programming language, for example, by saving the current sum in a file, parsing the input as a number, adding it to the value in the accumulator, and printing it both to the file and to stdout. The specifics of how to write that program are irrelevant; here, we simply assume to have it available as

`externalAdder`. The connection between that external program and an MSF can be done using IO actions that execute the program, passing the input and parsing the result. We can create a trivial connection by means of the function `System.Process.readProcess`, executing `externalAdder` in a separate process, passing the MSF input as input via stdin and obtaining the standard output as a `String`:

```
adderC :: MSF IO Int Int
adderC = arrM $ \a ->
  fmap read $ readProcess "./externalAdder" [show a] ""
```

This example assumes that we are only testing whether `externalAdder` actually adds the numbers correctly, but it assumes that it always returns *some* valid number as output. Failure to do so could make `adderC` crash (e.g., throw an exception). It would be trivial to modify the definition to handle incorrect outputs, as well as other possible exceptions and errors (e.g., the program not existing in the current path).

Using the `Always` temporal operator, we can check whether both are equivalent. Because one of the components is in the IO monad (`adderC`), the whole monadic function is `IO`, and so is the evaluation of that MSF with a specific input stream. Note that, in between tests, we need to reset the accumulator for the external process, which we do by calling `reset` before `evalT`:

```
pred stream = do
  reset
  evalT (Always $ SP ((adder &&& adderC) >>> arr (uncurry (==))))
        stream
```

We evaluate this `IO` property using the auxiliary QuickCheck function `ioProperty`, necessary to test properties with side effects:

```
ghci> quickCheck $ ioProperty pred
+++ OK, passed 100 tests.
```

Introducing an error in the external implementation makes that error manifest in our testing framework. For example, if we add twice the input in `externalAdder` (or provide it with double the number), our framework immediately detects the error.

Although this section does not explore resetting in depth, the existence of such an operation is key to making tests reproducible. If, for example, we do not reset the external component in between tests, both the counter-example provided by QuickCheck and, potentially, its own shrinking mechanisms, would not work as expected. If we try to use a counter-example provided by QuickCheck after manually re-setting the component, the example might work, violating our expectation that tests should be reproducible.

6.2 Hardware in the Loop Testing

The same mechanism that allows us to communicate with an external program in place of an `MSF` also allows us to test hardware in the loop. For example, we can run the same algorithm in an Arduino, and communicate with the device via USB, exposed to both the Arduino and the PC as a serial port.

The connection of the Arduino with an MSF network requires two additions, akin to what we discussed for external software: a communication layer, to exchange data between the computer and the device, and a resetting mechanism, to return the device to its initial state. These additional layers must be implemented both on the Arduino and on the MSF (Haskell) side.

Implementation in Arduino In order to simplify the exposition, we discuss the Arduino implementation in three parts: the adder itself, the communication layer, and the resetting mechanism. Structuring programs this way can help test each part independently, to make sure that errors are not due to incorrect communication with the device.

The implementation of an adder in Arduino language is as trivial as in MSFs:[4]

```
unsigned long accum = 0;

unsigned long step (unsigned long val) {
  accum += val;
  return (accum);
}
```

The additional layer required to communicate with Arduino needs to perform two functions: detect when the Arduino needs to be restarted (to restore its original state), and pass the incoming data to the `step` function and its result back to the computer. We can implement the communication as follows:

```
void resetUponRequest ();

void setup() {
    Serial.begin(9600);
}

void loop() {
    if (Serial.available() > 0) {
        resetUponRequest();

        unsigned long input  = (unsigned long)(Serial.parseInt());
        unsigned long result = step(input);
        Serial.println(result, DEC);

        while (Serial.available()) {
            Serial.read(); // clear buffer
        }
    }
}
```

[4] For simplicity, this code uses C's `unsigned long` type, which is different from Haskell's `Int` and not directly compatible. The Haskell code that corresponds to this implementation uses `Word8` as input and output types.

The types of the `input` and the `result` are specific to this application, but the general idea is applicable to other cases as well.

To make tests reproducible, it is crucial that the Arduino behave the same after a reset. Different mechanisms exist to reset the device depending on the model. In our example, tested on an Arduino Duemilanove, we reset the device in `resetUponRequest` when a character `*` is sent via the serial port (by calling a function at address $0x0$):

```
// Declare reset function at address 0
void(* resetFunc) (void) = 0;

// Reset the arduino when a '*' is sent via serial
void resetUponRequest () {
    char maybeEOL = Serial.peek();
    if (maybeEOL == '*') {
        Serial.end();
        resetFunc();
    }
}
```

Connecting to external system via MSFs In this example, the Arduino is connected to the USB as a serial port, which can be accessed from Haskell as a file handle using standard IO operations like `hGetLine :: Handle -> IO String`. We assume to have an operation available that executes an IO action that depends on the handle used to communicate with the device:[5]

```
withArduino :: (Handle -> IO a) -> IO a
```

This operation also resets the device at the beginning by sending an initial '*' character and introducing a short delay before the execution of the auxiliary action to give the Arduino time to reset.

A simple wrapper allows defining an MSFs that communicate with the Arduino using the handle provided by `withArduino`:

```
adderArduino :: Handle -> MSF IO Int Int
adderArduino handle = arrM $ \i -> do
  hPutStr h $ show i ++ "\n"
  read <$> hGetLine h
```

Testing with Hardware in the Loop The encoding of this property is rather trivial and, like before, simply compares the results of both implementations, side by side. We pass the handle to communicate with the Arduino as an argument, needed by `adderArduino`:

```
propA h =
  Always $
    SP ((adder &&& adderArduino h) >>> arr (uncurry (==)))
```

[5] Such an operation should open the device in binary mode, reset it by sending in a `*` character, execute the operation, and close it at the end. The function `openSerial` from the package `serial` can be used to open the device at the right speed of 9600 baud for communication with a PC.

We use `ioProperty` to run the effectful evaluation, with `withArduino` and `evalT`, using QuickCheck:

```
pred = \stream -> ioProperty $ withArduino $ \h ->
  evalT (propA h) stream
```

Because Arduino's `parseInt`, used in the implementation before, cannot handle negative integers, we only test this property for all streams of positive integers. We use the primitives defined in Section 4 to generate those streams:

```
positiveSignalStream :: Gen (SignalSampleStream Int)
positiveSignalStream = generateStreamWith (\_ _ -> genPos)
                         DistRandom (Nothing, Nothing) Nothing
  where
    genPos :: Gen Int
    genPos = do
      Positive x <- arbitrary
      return x
```

Like before, the standard test works as expected:

```
ghci> quickCheck $ forAll positiveSignalStream pred
+++ OK, passed 100 tests.
```

Introducing a small fault in the Arduino results in the test failing on the machine. For example, if we modify the Arduino program to include `adder += 2 * val;`, we obtain the following:

```
ghci> quickCheck $ forAll positiveSignalStream pred
*** Failed! Falsifiable (after 3 tests and 6 shrinks):
[(0.0,25),(1.3866281957553916,26)]
```

This results in perfectly reproducible executions, provided that the hardware behaves according to specification and that both resetting and communication with the device are implemented correctly. In the following we shall explore how to deal with potential hardware failures, and how QuickCheck and FRP can help us introduce reliable fault tolerance mechanisms.

7 Fault Analysis, Injection and Tolerance in Reactive Systems

In normal conditions, we work under the assumption that the hardware works perfectly, that the values obtained from input devices are accurate, and that the memory is never corrupted. However, this is not true in practice, and critical systems that carry out important operations or function over long periods of time in hostile environments without maintenance, like a satellite, warrant extra levels of reliability.

In particular, in prior sections, nothing tells us *whether data is inaccurate*, *how inaccurate it can be*, and what *kinds of faults* may have lead to such incorrect results. If we expect systems to fail, it is unclear what conclusions we can draw in practice from a battery of tests having succeeded under ideal conditions.

The use of techniques to minimize the impact of system failures is grouped under the umbrella term of *fault tolerance* (Avižienis, 1967, 1976). In fault tolerance, a distinction is made between a *fault* (a potential error that has not manifested yet), an *error* (a defect that has already affected the internal state of the system, but whose effect has not had an impact on the service yet), and a *failure* (an error that has affected the service provided). The goal of fault tolerance is, therefore, to minimize faults and to detect and correct errors before they result in system failure.

Redundancy can be used either to compare results from multiple units and disable those that fail, or to average them and lower the impact of minor deviations (a process called *voting*). Apart from adding extra redundancy, and to limit the possibility of multiple redundant units failing for the same reason, we may choose to divide our architecture into independent subsystems, which may be both physically and electrically isolated, constituting independent *Fault Containment Regions (FCRs)*.

The combination of these fault tolerance mechanisms helps determine which faults may still affect system operations, and defines the *fault model*. Total reliability is never possible, so our fault model is likely to determine both the kinds of faults that are handled and also how many simultaneous faults may be tolerated. For instance, a system may be able to deal with a value not being available, but not be able to deal with a value being incorrect. Others are capable of handling Byzantine errors, those in which different subsystems have different incorrect values for the same conceptual element, whereas others may only be able to do so, so long as the error is corrected before another simultaneous error occurs.

The study of how to introduce fault tolerance mechanisms in Functional Reactive Programming systems was first introduced in (Perez, 2018), from which we borrow part of the introduction in this section and the introductory example that follows. We then extend that work to use Temporal Logic and QuickCheck to study the behaviour of a reactive system when faults are present in the system, and to determine if fault tolerance mechanisms can handle faults as expected.

7.1 Faults in Reactive Systems

Imagine that we are building a system to control a satellite and show that it will be able to carry out its mission without colliding with other satellites, falling off its orbit or drifting away.

To move and orient the satellite as required for a mission, we carry out a phase of *attitude determination* (determining the position and direction of the spacecraft) prior to *attitude control* (corrections to position and direction via a series of actuators). To estimate the satellite's attitude, we obtain data from a star tracker, which determines the position and orientation relative to stars whose locations in space we know, and from an inertial measurement unit (IMU), which combines gyroscopes and accelerometers to provide an estimate of the spacecraft's acceleration, linear velocity, orientation and surrounding magnetic fields.

Errors in these calculations can lead to the satellite falling out of its orbit and being lost, or colliding into other spacecraft. It is therefore crucial that we understand how systems can fail and introduce errors in our calculations, how we are dealing with potential faults, and what effect that may have on our system overall.

A schematic definition of our control system follows. Our reactive control `MSF` receives a desired attitude as an input stream, and produces a stream with actions as output. Internally, it gathers data from a star tracker and the IMU. We use a monad, `StateT Universe m`, to represent the fact that the satellite's position affects what stars it sees and its inertial measurements, where `Universe` represents the state in the transformer that changes as satellites and other elements move over time. This abstract `Universe`, which captures the changing environment in which the simulation takes place, can also include a clock, under the assumption that time can be considered global and absolute in our simplified example. The use of `feedback` allows us to calculate the new attitude based on the last known attitude and the information gathered from the sensors:

```
controlSystem :: Monad m
              => Attitude
              -> MSF (StateT Universe m) Attitude [Action]
controlSystem initialAttitude = proc (desiredAttitude) -> do
  stars        <- starTrackerSense  -< ()
  inertialInfo <- imuSense          -< ()

  attitude     <- feedback
                  initialAttitude
                  (arr (dup . calculateAttitude))
                                  -< (stars, inertialInfo)

  let actions = calculateActions desiredAttitude attitude

  returnA -< actions

-- Predefined elsewhere
type Action        = ...
type Attitude      = ...
type StarInfo      = ...
type InertialInfo  = ...

starTrackerSense   :: Monad m
                   => MSF (StateT Universe m) () [StarInfo]
imuSense           :: Monad m
                   => MSF (StateT Universe m) () InertialInfo
calculateAttitude  :: (([StarInfo], InertialInfo), Attitude)
                   -> Attitude
calculateActions   :: Attitude -> Attitude -> [Action]

dup                :: a -> (a, a)
```

For the purposes of simulating our example, this control system is part of a larger program that executes the actions of the control system, affecting its position in the `Universe`, and simulates other spacecraft as well.

```
simulation :: Monad m
           => MSF m (DTime, ()) Universe
simulation = runReader $ runStateS initialUniverse $ sat1 >>> sat2
  where
    sat1 = controlSystem >>> actuators
    sat2 = ... -- different expression
```

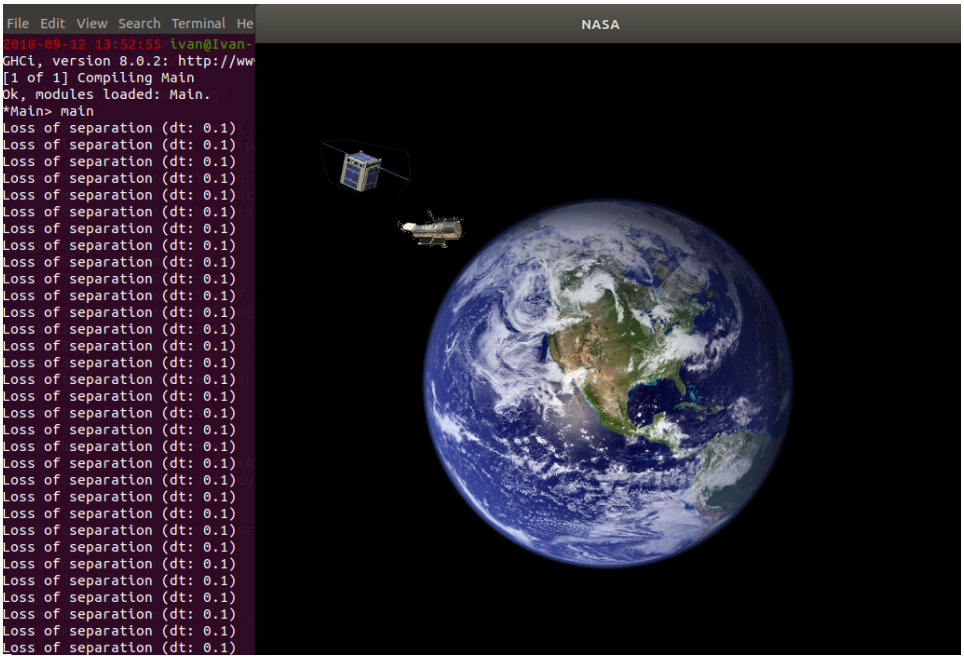

Fig. 7. A schematic simulation of two satellites moving around the Earth (not to scale), as implemented below, extended with runtime verification to detect and report loss of separation (i.e., two satellites passing each other too closely.

Temporal Properties To understand if our simulation behaves correctly, let us express some of the properties we would desire our system to have. We will first encode basic properties, and then express more complex properties that take advantage of temporal logic combinators.

A basic property would be that the main satellite, whose control system we have programmed, remains in orbit around the Earth, never getting too close to the other satellite. We can express this property by saying that the distance between the satellites is greater than a threshold:

```
propNoLossOfSeparation = Always $ SP $
  simulation
    >>>
     (\s -> let (p1x,p1y) = position (sat1 s)
```

```
            (p2x,p2y) = position (sat2 s)
            distance  = abs (sqrt((p1x-p2x)^2 + (p2x - p2y)^2))
        in distance > 110)
```

Testing this property is not entirely straightforward. If we do not sample frequently enough, therefore adjusting the speed and the propulsion to correct the velocity, the satellites will not avoid each other:[6]

```
ghci> quickCheck
        $ forAll uniDistStream $ evalT propNoLossOfSeparation
*** Failed! Falsifiable (after 5 tests):
[(0.0,()),(13.67361148360696,()),(3.247598098323074,())]
```

Instead, we use our stream combinator library to produce a stream with a limit to the sampling rate:

```
ghci> quickCheck
        $ forAll (generateStream DistConstant (Just 0.5, Just 0.5)
                    Nothing)
        $ evalT propNoLossOfSeparation
+++ OK, passed 100 tests.
```

We can be more conservative in the sampling rate, assuming that small deviations from the perfect rate will exist, and checking whether our simulation is correct:

```
ghci> quickCheck
        $ forAll (generateStream DistRandom (Nothing, Just 0.5)
                    Nothing)
        $ evalT propNoLossOfSeparation
+++ OK, passed 100 tests.
```

We can even use this idea to try to find the highest sampling rate at which a problem manifests. For example, incrementing the maximum time delta by 0.30 results in:

```
ghci> quickCheck
        $ forAll (generateStream DistRandom (Nothing, Just 0.8)
                    Nothing)
        $ evalT propNoLossOfSeparation
*** Failed! Falsifiable (after 83 tests):
[(0.0,())
,(0.6337967097559944,())
,(0.6552991075588447,())
,(0.38588618564678595,())
,(0.17043590981341944,())
,(0.7511450842519329,())
 ... (multiple times) ...
]
```

[6] In the subsequent sessions, we manually adapt the layout of the code and introduce new lines for presentation purposes.

A more sophisticated property that we might want to check is that, even with a relatively low sampling rate or if the sampling rate drops temporarily, the satellite will always correct its course towards the safe orbit:

```
ghci> quickCheck
        $ forAll (generateStream DistRandom (Nothing, Just 0.79)
                    Nothing)
        $ \stream -> evalT propNoLossOfSeparation
                       (sConcat stream (sConcat [(0.8, ())] stream))
*** Failed! Falsifiable (after 63 tests):
[(0.0,())
,(0.730513278034849,())
,(0.7438086357750868,())
,(1.8471818310137077e-2,())
,(0.5480596195116617,())
... (multiple times) ...
]
```

If we decrease the sampling rate sufficiently enough, we see that the system becomes more robust to this kind of fault:

```
ghci> quickCheck
        $ forAll (generateStream DistRandom (Nothing, Just 0.78)
                    Nothing)
        $ \stream -> evalT propNoLossOfSeparation
                       (sConcat stream (sConcat [(0.8, ())] stream))
+++ OK, passed 100 tests.
```

This kind of search is, of course, not exhaustive. As it is often the case with QuickCheck, finding counter examples is meaningful (i.e., the property can definitely be violated), while passing the tests renders an inconclusive decision (i.e., the property might still be violated with an input not tested). We can never obtain full confidence in a property by means of testing alone for very large or infinite input spaces, we can increase our confidence in an implementation by parameterizing our tests by increasing the number of tests and by improving the random input generators. The study of how much of the input has actually been explored remains as future work, as will be detailed in Section 10.

It is worth noting that this example exhibits the same bullet-through-paper effect that we discussed before (Sec. 4). However, this example further assumes that both the physics simulation and the computers on the satellites run on the same clock, which is obviously inaccurate. From the point of view of the user, a change in the clock rate will imply both that the physics simulation is more accurate, and that the computers are sampling (and correcting) more frequently, but not necessarily help explain which one is at fault. The use of different clocks for FRP simulations and their correct coordination has been the subject of study of Bärenz & Perez (2018). Due to that solution being based on Monadic Stream Functions, the ideas discussed in this paper are directly applicable to simulations with multiple clocks.

7.2 Fault Injection and Correction

To test whether our system is robust to different kinds of faults, we are going to modify its internal connections to inject the kinds of faults we know may occur. We shall do so by adding monadic stream functions that alter values depending on predefined fault rates, fixed for the simulation and provided in a global environment.

We first define the fault profile, or the settings used to inject faults, with a type that captures fault rates for different subparts, as well as a seed that we can use to inject faults in a predictable way.

```
data FaultSettings = FaultSettings
  { seed      :: StdGen -- ^ Random number generator seed
  , sense1FR  :: Double -- ^ Sensing failure rate
  , sense1F   :: Bool   -- ^ Has a fault actually been introduced?
  }
 deriving (Show, Generic)
```

It would be trivial to extend this fault profile to include failure rates for other subparts, such as communication failures, other sensors, the planning system, and the actuators and propulsion systems.

We can now re-define the satellite position sensor to alter the data inserting a fault. Let us first insert a random fault that depends on the fault rate defined in the `FaultSettings` given above. We modify the monad to provide both the state of the system and the fault profile or settings in a combined simulation state `SimState`, which is a pair of `Universe` and `FaultSettings`:

```
starTrackerSense' :: Monad m
                  => MSF (StateT SimState m) () StarInfo
starTrackerSense' = arrM $ \_ -> do
    (univ,ft) <- get
    let (failT, seed') = randomR (0,1) (seed ft)
        fail = failT < sense1FR ft

    -- Record that there's a fault
    put (univ, ft { seed = seed', sense1F = fail })

    -- Produce wrong value
    if fail
      then return incorrectStarInfo
      else return $ sat1 univ
  where
    incorrectStarInfo = -- fixed bad position and orientation
```

If we use this in place of our original sensor, QuickCheck can quickly find a fault rate for which the satellite does not meet the properties we saw before:

```
ghci> quickCheck
        $ forAll (generateStream DistRandom (Nothing, Just 0.5)
```

```
                    Nothing)
        $ evalT propNoLossOfSeparation'
*** Failed! Falsifiable (after 82 tests):
([(0.0,())
 ,(0.32561791284270414,())
 ,(0.2939231037169424,())
 ,(0.2822720676219349,())
 ,...
 ]
,FaultSettings {seed = 25 2, sense1FR = 0.8833330397836819
                           , sense1F = False})
```

Note that, for a sampling rate for which our system worked perfectly before, the introduction of faults results in properties being violated. We can use a more realistic failure rate of 0.01 for this kind of device to determine how inaccurate the position may be, and run the properties again for two maximum time deltas. Our tests succeed for a maximum time delta of 0.5, like before, but they fail for a max delta of 0.7, which was working perfectly when failures were not present in the system:

```
*** Failed! Falsifiable (after 33 tests):
([(0.0,()),(0.47233347953205923,()),(0.32903943890981263,()),...]
,FaultSettings {seed = 12 1, sense1FR = 1.0e-2, sense1F = False})
```

To detect and correct faults present in our system, we can use multiple redundant units in parallel and some mechanism to either average their results or detect and ignore the ones that seem incorrect.

For example, if we modify `controlSystem` to use three redundant `starTrackerSense'` in parallel and average the position and orientation received from them, we see that not only is it smoother, but it is also more robust. For a sampling rate of up to 0.7, which our last example failed for, the new run now passes the test without problems. If we increase the sampling rate to a maximum time delta of 0.37*s* with fault rates of close to 0.05, the tests pass correctly, indicating that the fault tolerance mechanisms we have introduced are increasing the robustness of the calculation.

The approach presented in this section is limited in that we included one random seed that determines if or when faults are introduced in the system, but exposes no way for QuickCheck to control it in order to shrink randomly generated streams and produce shorter or simpler counter-examples. The exploration of shrinking with the techniques proposed in this paper remains an open problem.

8 Experience

The approach to testing described in this paper has been used to create a testing extension to the FRP implementation Yampa, and to implement a suite of unit tests for that library (Perez, 2017c). Similar facilities have also been added to the library dunai, that implements Monadic Stream Functions (Perez, 2017b).

While the creation of more suitable generators for signal functions is still work in progress, the properties we have implemented provide better coverage of the input test space than Yampa's previous set of unit tests.

The combination of Temporal Logic, QuickCheck and the debugging GUI application has also been used to debug existing demonstration applications like Haskanoid,[7] professional iOS/Android games like Magic Cookies![8,9] and other titles currently under development.[10]

In this section, we explore how the combination of QuickCheck and Temporal Logic helped find bugs in the game Haskanoid. We cannot expect game developers to mirror realistic user-input using just QuickCheck generators. To produce complex inputs, the use of the additional recording facilities provided would be useful. We expand on this point in Sec. 8.4.

8.1 Haskanoid

To demonstrate properties that QuickCheck can detect and how it has helped find bugs in real applications, we provide examples using an existing cross-platform open-source Haskell implementation of the popular Breakout that runs on Windows, Linux, Mac, iOS, Android and can be controlled with devices like Kinects, Wiimotes and accelerometers.

The user controls a paddle which moves only sideways. A ball is initially attached to the paddle. When the user clicks the mouse, the ball is propelled upwards, bouncing against walls, blocks and the paddle itself. Blocks disappear when hit three times. The player wins when all blocks disappear, and loses if the ball hits the floor.

The full system implements more complex features like levels, lives and sound. Here we present a simplified version to test the application physics and simple, yet important, invariants.

8.2 Objects

Objects in Haskanoid are defined as signal functions that, depending on some input, present their new state and any effects that could affect other objects or their own existence (i.e. requests to create new objects or kill themselves). This idea is described in more detail in Courtney *et al.* (2003).

```
type ObjectSF = SF ObjectInput ObjectOutput

data ObjectInput = ObjectInput
  { userInput    :: Controller
```

[7] `http://github.com/ivanperez-keera/haskanoid`
[8] `https://itunes.apple.com/us/app/magic-cookies/id1244709871`
[9] `https://play.google.com/store/apps/details?id=uk.co.keera.games.magiccookies`
[10] The development of the mobile extensions to the framework presented in this paper and their use to test these games have been carried out by Keera Studios Ltd., a start-up founded by the first author.

```
  , collisions   :: Collisions
  , knownObjects :: Objects
  }

data ObjectOutput = ObjectOutput
  { outputObject :: Object
  , objectDied   :: Event ()
  }
```

Any object can depend on the user input (`Controller`), previous collisions, and the state of other objects. Objects produce an internal state with physical properties and indicate whether the object has died. This is used by blocks hit by the ball, to signal that they must be eliminated.

A simplified definition for objects, used for for the ball, the paddle, blocks and the walls, follows. All 2D-suffixed types correspond to tuples of `Doubles`.

```
data Object = Object
  { objectName      :: String
  , objectKind      :: ObjectKind
  , objectPos       :: Pos2D
  , objectVel       :: Vel2D
  , objectAcc       :: Acc2D
  , objectHit       :: Bool
  , objectDead      :: Bool
  , collisionEnergy :: Double
  }
```

8.3 Testing physics simulations

In this simulation, the position of the ball in the output can be extracted using a function called `ballPos`. In the following example, the top-level function `mainSystem` takes user input (`Controller`) and returns a system state, which includes a collection of `ObjectOutputs`. Like in previous occasions, in this case we were interested in knowing whether the ball could overlap with the wall, or pass through blocks exhibiting an effect called *tunneling* or *bullet through paper* (Gregory, 2014).

```
propBallInScreenAreaW = forAll myStream $ evalT
  Always
    (SP (mainSystem
           >>> arr systemObjects
           >>> arr (\p -> isInRange (0, systemWidth)
                                    (fst (ballPos p)))))
  where myStream :: Gen Controller
        myStream = uniDistStream

> quickCheck propBallInScreenAreaW
*** Failed! Falsifiable (after 31 tests):
```

```
(Controller { controllerPos = (8.49338899008, 29.07069391684)
            , controllerClick = False},
[ ( 17.68729810519
  , Controller { controllerPos = (28.12964084927,  12.4975777242)
               , controllerClick = True})
, ( 29.88005602026
  , Controller { controllerPos = (51.41061779016,   3.0186952000)
               , controllerClick = False})])
```

In this trace, QuickCheck is telling us that the input controller (e.g. mouse) needs to start at a particular position with the main button released, moved to another position with the button pressed, and moved elsewhere with the button released. However, the delays between one sample and the next are unrealistically large: one new frame every 15 seconds. To make it a bit more realistic we decide to use a stream generator that guarantees shorter delays. Let's assume that our simulation runs at around 25FPS (`denseStream` is an auxiliary generator that creates streams with very small time deltas following a normal distribution):

```
propBallInScreenAreaW' = forAll myStream $ evalT
    Always
      (SP (mainSystem
       >>> arr systemObjects
       >>> arr (\p -> isInRange (0, systemWidth) (fst (ballPos p)))))
  where  myStream :: Gen Controller
         myStream = denseStream 0.04

> quickCheck propBallInScreenAreaW'
+++ OK, passed 100 tests.
```

If we try this property with even higher framerates (lower deltas), it seems to pass all tests. But that is only an illusion. If, instead of increasing the number of deltas, we choose to increase the number of tests, we start seeing that not even 0.04 is low enough:

```
> quickCheckWith (stdArgs {maxSuccess = 100000})
                 propBallInScreenAreaW'
*** Failed! Falsifiable (after 3443 tests):...
```

There is one artificial way in which we can prevent the ball from going out of the screen: with a speed cap. If we cap the maximum speed and introduce a margin of error in the detection of collisions, then the collision with the ball against any wall will be detected before they overlap.

Nevertheless, this artificial cap can easily be tested for by checking the magnitude of the ball's velocity. This test should work regardless of the sampling rate and detecting that the ball moves too fast (one condition) is easier than detecting that the ball is out of the screen, because it was moving too fast (two conditions).

Introducing such a test in this system was particularly useful, as it showed that the velocity was not being capped right after launching it or after collisions, which implied

that, if the paddle hit the ball with a quick, sudden movement, it could temporarily "insert" it into the wall. Before adding the second speed cap, we saw that it was still possible to interact with the application in a way that the velocity would be too fast:

```
propVelInRange = forAll myStream $ evalT
    Always
      (SP (mainSystem
             >>> arr systemObjects
             >>> arr (\p -> isInRange (0, 800) (norm (ballVel p)))))
  where myStream :: Gen Controller
        myStream = denseStream 0.004

> quickCheck propVelInRange
*** Failed! Falsifiable (after 67 tests):
(Controller { controllerPos = (130.5942684124, 40.2324487584)
            , controllerClick = False},
[ ( 3.9907922435e-3
  , Controller { controllerPos = (61.5220268505,  108.729668432)
               , controllerClick = True})
, ( 2.1598144494e-3
  , Controller { controllerPos = (66.5487565898,   50.682566812)
               , controllerClick = False})])
```

In this trace, the mouse is moved quickly to the left and clicked, propelling the ball with a velocity of thousands of pixels per second. By adding the cap in other places in the code, we guarantee that the velocity is within the expected range, making the ball more likely to remain within the application area even with lower sampling rates. This solution is, unfortunately, ad hoc and not reusable, and it might not behave properly in other situations. A general solution for this problem, called *tunneling*, is to implement a continuous-collision detection system (CCD).

8.4 Discussion

The approach explored in this section discusses how QuickCheck can be used to detect property violations. Using the generators provided, it is possible to explore simple properties in games, models and simulations, as shown in this and earlier sections on fault tolerance. In recent work, we have used the same approach to evaluate the resilience of swarms of satellites (Perez *et al.*, 2019), showing that this approach works for more complex properties.

We would not expect game developers to use the low-level API to try to replicate user input faithfully. For more complex inputs, especially in interactive games, the recording facilities provided are crucial. The ability to transport the game to a point of the simulation where players have witnessed a problem helps replicate the conditions, and the existing API can help detect if the problems seen are a consequence of violations of basic properties at a later stage. For most games, the generators provided cannot be used in practice to generate

whole streams as long as recordings from actual players (which can be minutes or even hours), but they can be used to complement those streams.

It has been suggested that a random input generation method that is guided by the parts of the properties that are being invalidated could help find violations faster. This would not be possible with the information available at present in QuickCheck, and we consider this future work.

9 Related work

The link between FRP and Temporal Logic has been studied extensively. Jeffrey (2012) describes how LTL can be used to encode FRP's construct types, making any well-typed FRP program a proof of an LTL tautology. Jeffrey (2014) also combines LTL with FRP using dependent types to define streams of types, which themselves type reactive programs and form a constructive temporal logic, exploiting the Curry-Howard isomorphism between TL and FRP. The author goes on to define a combination of FRP with past-time LTL, in which combinators like "always" mean "so far", making all modalities executable and causal.

Jeltsch (2012) has also described the Curry-Horward isomorphism between Temporal Logic and FRP, in which FRP programs implement Temporal Logic propositions. The author shows that behaviours and events can be generalized in terms of the Until operator from LTL (Jeltsch, 2013). He then gives categorical semantics for LTL with until and FRP with behaviours and events. This work is used to define Concrete Process Categories (CPCs), which serve as categorical models of FRP with processes. The author defines a new semantics, a new temporal logic for CPCs, which captures causality.

Sculthorpe (2011) has described an encoding of Priorean Temporal Logic in a denotational model used to describe FRP signals. In his approach, properties of both the time domain and FRP signals and signal functions can be described as temporal predicates, that may or may not hold, depending on the time domain. Sculthorpe also shows how properties of signal functions, like being temporally decoupled or stateless may be captured using temporal logic. There are several differences between Sculthorpe's approach and ours. First, we use a different kind of time, which is bounded because our simulations can actually end. Also, because we are interested in executing or checking these properties live, we cannot depend on the past or the future in the way that Sculthorpe's semantics does. Dependencies on the past lead to leaks, dependencies on the future cannot be determined in the present (which is how Yampa is evaluated).

The link between TL and FRP has been discussed by Winitzki (2014), who explains the meaning of modalities in Temporal Logic with discrete unbounded time in terms of streams and samples. The author also lists features desired for implementing Temporal Logic currently unavailable in existing FRP implementations.

The use of Temporal Logic to specify properties of reactive systems is frequent in other domains. Hughes *et al.* (2010) use Temporal Logic to specify parts of an asynchronous messaging server, and use QuickCheck to prove properties under timing uncertainty. Tan *et al.* (2004) provide a metric to understand how well a property specified using Temporal Logic has been tested by a test suite.

Giannakopoulou & Havelund (2001) presents an approach at verifying a program's behaviour against LTL specifications, by translating an LTL formulae into a finite-state

automata that monitors the program's behaviour. This approach is based on the next-free variant of Linear Temporal Logic with Until, as it is guaranteed to be unaffected by *stuttering* (receiving the same input several times in succession).

Havelund & Rosu (2001) have also used Linear Temporal Logic to monitor programs by connecting to them running live via the network. In this work the authors use finite-trace LTL, a variant of infinite-trace LTL in which always means at every point *in the trace*, and next defaults to true at the end of the trace. Like in our case, formulae that always hold in limited-trace LTL do not hold in infinite-trace LTL and vice-versa, an aspect explored in also in some detail by Sculthorpe (2011).

Nejati *et al.* (2005) discuss the construction of a model that approximates a program of interest, and use CTL to verify properties of the model. The authors use a three-valued logic in order to indicate when a model needs further refinements and determine whether a CTL property holds.

In the context of game programming, the idea of recording and replaying programs deterministically was proposed, among others, by John Carmack (Carmack, 1998, p. 55), who implemented a single entry point for all input events to a game, and set up a journaling system that recorded everything the game received, including the time. Carmack also pointed out that reproducing bugs is key to fixing them, and that it is important to be able to roll back time in order to find when a bug is first introduced, even if its effect is only visible later in the execution.

Execution-replay systems have also been studied in the context of general imperative programming and especially parallel and distributed systems Cornelis *et al.* (2003); Ronsse *et al.* (2000). Much of the focus in those areas has been towards dealing with low-level issues to guarantee determinism of replays. In contrast, we rely on Haskell's strong type system to guarantee the freedom from side effects in FRP programs and hence absolute determinism, letting us focus on how to exploit such guarantees for declarative testing and debugging. Our debugger currently records complete input traces, but our approach could be used to log input to only a subpart of the system, making it more suitable for higher-level debugging, as opposed to logging low-level system calls.

Mozilla's tool *rr*[11] monitors a group of Linux processes, capturing the results of kernel calls and non-deterministic CPU effects. In future replays, memory and register contents are the same, and all system calls return the same values. This tool has been designed for general purpose applications, so it works at a very low level. As a result, it is tied to the Linux kernel, it emulates a single-core machine, and it is difficult to port to other architectures (ARM and Android support, for example, is still an open issue[12]). In contrast, our approach works on all architectures for which there is a Haskell Compiler with a corresponding back-end (currently iOS, Android, web, Linux, Windows and MacOSX). To control a simulation with the debugging GUI, the only platform-specific adaptation required is a way for the Yampa debugger to open two network sockets for communication.

GDB has *Process Record and Replay*,[13] capabilities, which record system calls and the CPU and memory state at each point. Like Mozilla's rr, GDB's replay feature requires

[11] `http://rr-project.org/`http://rr-project.org/
[12] `https://github.com/mozilla/rr/issues/1373`https://github.com/mozilla/rr/issues/1373
[13] `https://sourceware.org/gdb/wiki/ProcessRecord`https://sourceware.org/gdb/wiki/ProcessRecord

adaptations specific for each architecture and system call. An advantage of GDB's implementation is that actions are undoable, which means that we can move backwards along the trace. In contrast, FRP signals only move forward, although our system records intermediate Signal Functions to provide random access to any time point in constant time, and they can always be reproduced deterministically if only the inputs are recorded. Time-reversible FRP combinators remain an open problem.

10 Conclusions and Future work

In this paper we have shown how to approach testing and debugging of FRP programs in variants like Arrowized FRP that completely separate IO from the signal processing. We have used an encoding of Linear Temporal Logic over FRP to describe temporal properties of FRP programs and evaluate them against input streams. We have also shown that this simple approach makes FRP programs easily testable with existing tools like QuickCheck, and we can add temporal assertions to programs in a similar fashion. Our implementations of temporal logic and the compatibility layer for QuickCheck work for two FRP implementations: the pure Arrowized FRP library Yampa, and the Monadic Stream Function library Dunai.

We have extended an FRP implementation with recording and remote control capabilities, and implemented a Graphical User Interface to communicate with applications running remotely on PCs, mobile devices, or external hardware like ARM boards and Arduinos. The referential transparency available in pure FRP implementations like Monadic Stream Functions and Yampa helps developers reproduce the exact same situation witnessed by beta-testers, and move back and forth along the trace adding new assertions or visualizing the simulation along on an external device to pinpoint possible bugs. These traces can be fed to QuickCheck for additional help in finding possible property violations, and the counter-examples found by QuickCheck can be loaded back into a phone for future study and to visualize them. So, in practice, we can truly see QuickCheck interact with our applications.

We have shown that the testing facilities, when used in combination with Monadic Stream Functions, enable hardware- and software-in-the-loop testing, and have provided examples for both cases. We have also exemplified how to inject faults with different rates in MSFs, and how to use QuickCheck and the temporal logics provided to evaluate fault tolerance mechanism.

We would be interested in finding systems for which the temporal languages provided are insufficient to express desired properties declaratively and concisely. Our encoding of temporal properties is based on Linear Temporal Logic, but we can answer questions pertaining to multiple possible futures by means of the quantification provided by QuickCheck with `forAll` and the stream generators we provide. It is unclear whether our formalisation of temporal logic on top of FRP is as expressive as linear temporal logic, or how it compares otherwise.. Nevertheless, the use of a different formalism like Computation Tree Logic (Clarke & Emerson, 1982) and a comparison with our approach remain as future work. In the context of real-time systems, we might need to use a metric logic to deal with the notion of continuous time.

A question we have not answered is how well the input space is explored. In this respect, further improvements could be made by adding Signal Function generators, increasing the confidence in abstract properties about Signal Functions, like the Arrow laws that they are expected to fulfill. We have also yet to explore further shrinking strategies for input traces, in order to help QuickCheck find minimal counter-examples.

Our FRP debugger has been implemented for Yampa, but the same approach could be used with different Arrowized FRP variants like Dunai (Perez *et al.*, 2016). An advantage of using Dunai over Yampa would be that the former only has one type for Monadic Stream Functions, which can be parameterised over the time domain (Perez, 2017a), making it more versatile while reducing code duplication in our debugger. Also, Dunai introduces monads, which can be used to implement safe debugging facilities, as shown in Section 5. Additionally, a monad would allow us to introduce the recording system inside a Signal Function, instead of at the top-level like we do in Yampa. This would let us record high-level input, like declarative actions, instead of low-level input, like user clicks and mouse movement, which would be more useful as small variations to the User Interface are more likely to invalidate the latter. At present, the need to record system inputs at the top-level for our approach to work is a limitation to hot-swap running applications, since a bug fix that requires modifying the user interface may lead to a completely different application state, rendering the recorded input trace useless for the purposes of debugging.

We have also not studied the performance cost of using our solution, or how much it could affect the behaviour of a program and the detection of bugs. In order for a system to be fully reproducible, users need to record the complete input trace. It is in principle possible, for instance, for sampling times to never be fast enough when debugging is enabled for certain bugs to appear, what is known as a *heisenbug* (Gray, 1986) (a bug that is only detectable when debugging tools are not enabled). In some cases we have reduced the debugging facilities introduced in a binary to only saving the inputs to a file, to minimize the overhead and hence the possibility of introducing heisenbugs. If a bug is detected with this minimal debugger and the input is logged accordingly, then it will be completely reproducible with the full debugger.

A side feature facilitated by our framework is that we could stream input traces over to another machine to visualize a simulation as it is being executed, or use them to replay a simulation in a machine with higher visual specifications or to record a video. We could also replay these simulations later on in higher quality or record videos of the output for publication online, even if they were produced in devices with lower specifications.

Our system currently requires that the complete input be recorded. If we could serialize/de-serialize Signal Functions, then we could in principle start the recording only when desired. This would, of course, affect the meaning of our language of Temporal Properties and the modalities admitted, since we would be discarding part of the past and might not be able to answer questions about such past anymore.

FRP is defined in continuous time, but implementations are inherently discrete. Implementations like Yampa include Signal Functions that diverge in the limit (as the time delta becomes smaller). It would be interesting to show that the behaviour of Signal Functions is irrespective of the sampling rate, and that it is bounded in the limit. This would require using stream refinement or partitioning to express the corresponding QuickCheck properties.

It is possible to use the techniques presented in this paper to explore the system behaviour when components misbehave. We could parameterise the system by fault injection rates, and program it with explicit fault injection features to introduce faults based on the given settings. Alternatively, having a variant of FRP in which changes to an FRP network are first-class would allow us to use QuickCheck to automatically generate those changes, and find (small) modifications that make the system fail. This would help us determine which levels of fault tolerance need to be introduced to make the network robust to certain kinds of faults.

It would be interesting to combine our system to provide debugging capabilities for distributed systems. In particular, in the context of games, it is crucial that multiple players see the same world in order to provide a smooth simulation.[14] Differences between each player's worlds manifest as inconsistencies during gameplay (e.g., a player being hit by the opponent and then recovering), which are exacerbated by network lag. In this context, distributed consensus might provide the necessary requirements for determinism and reliable testing and debugging.

In terms of implementation, we have implemented this for Yampa and included some properties of its own battery of unit tests, but not all of them. Similarly, it would be interesting to include properties of the logic itself in the battery of tests, in the form of tautologies that must hold, to guarantee the consistency of the temporal logic(s) implemented.

Acknowledgements

The authors would like to thank Paolo Capriotti, Graham Hutton, Jan Bracker, Jonathan Thaler, Csaba Hruska, Alwyn Goodloe, Kerianne Hobbs, and the anonymous referees for their constructive comments and helpful suggestions.

Fig. 6 uses parts of an image by Philipp Henkel, licensed under the Creative Commons Attribution-Share Alike 4.0 International license, and is consequently licensed under the same terms.

Ivan Perez's contributions to the presented work were partly funded under NASA Cooperative Agreement 80LARC17C0004. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, either expressed or implied, of any of the funding organizations. The first author is also the founder of Keera Studios Ltd., as described in Sec. 8.

Bibliography

Bibliography

Avižienis, Algirdas. (1967). Design of fault-tolerant computers. *Pages 733–743 of: Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*. AFIPS '67 (Fall). New York, NY, USA: ACM.

Avižienis, Algirdas. (1976). Fault-tolerant systems. *Ieee trans. computers*, **25**(12), 1304–1312.

[14] See `https://www.aorensoftware.com/blog/2011/01/28/determinism-in-games/` and `http://bulletphysics.org/Bullet/phpBB3/viewtopic.php?f=12&t=6231`

Bärenz, Manuel, & Perez, Ivan. (2018). Rhine: Frp with type-level clocks. *Page 145157 of: Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018. New York, NY, USA: Association for Computing Machinery.

Carmack, John. (1998). *John Carmack archive - .plan*. `http://fd.fabiensanglard.net/doom3/pdfs/johnc-plan_1998.pdf`.

Claessen, Koen, & Hughes, John. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. *Pages 268–279 of: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: ACM.

Clarke, Edmund M., & Emerson, E. Allen. (1982). Design and synthesis of synchronization skeletons using branching time temporal logic. *Pages 52–71 of:* Kozen, Dexter (ed), *Logics of Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Cornelis, F., Georges, Andy, Christiaens, M., Ronsse, Michiel, Ghesquiere, T., & De Bosschere, K. (2003). A taxonomy of execution replay systems. *Proceedings of international conference on advances in infrastructure for electronic business, education, science, medicine, and mobile technologies on the internet*.

Courtney, Antony, & Elliott, Conal. (2001). Genuinely Functional User Interfaces. *Pages 41–69 of: Proceedings of the 2001 Haskell Workshop*.

Courtney, Antony, Nilsson, Henrik, & Peterson, John. (2003). The Yampa arcade. *Pages 7–18 of: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. Haskell '03. New York, NY, USA: ACM.

Elliott, Conal, & Hudak, Paul. 1997 (June). Functional reactive animation. *Pages 163–173 of: International Conference on Functional Programming*.

Emerson, E. Allen. (1990). Handbook of theoretical computer science (Vol. b). Cambridge, MA, USA: MIT Press.

Giannakopoulou, D., & Havelund, K. 2001 (Nov). Automata-based verification of temporal properties on running programs. *Pages 412–416 of: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*.

Gray, Jim. (1986). Why Do Computers Stop and What Can Be Done About It? *Pages 3–12 of: Symposium on Reliability in Distributed Software and Database Systems*.

Gregory, Jason. (2014). *Game engine architecture, second edition*. 2nd edn. Natick, MA, USA: A. K. Peters, Ltd.

Havelund, Klaus, & Rosu, Grigore. (2001). Monitoring programs using rewriting. *Pages 135–143 of: Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE.

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

Hughes, John, Norell, Ulf, & Sautret, Jérôme. (2010). Using temporal relations to specify and test an instant messaging server. *Pages 95–102 of: The 5th Workshop on Automation of Software Test, AST 2010, May 3-4, 2010, Cape Town, South Africa*.

Jeffrey, Alan. (2012). LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. *Pages 49–60 of: Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*. PLPV '12. New York, NY, USA: ACM.

Jeffrey, Alan. (2014). Functional reactive types. *Pages 54:1–54:9 of: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. CSL-LICS '14. New York, NY, USA: ACM.

Jeltsch, Wolfgang. (2012). Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electron. notes theor. comput. sci.*, **286**(Sept.), 229–242.

Jeltsch, Wolfgang. (2013). Temporal logic with "until", functional reactive programming with processes, and concrete process categories. *Pages 69–78 of: Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*. PLPV '13. New York, NY, USA: ACM.

Lewis, Chris, Whitehead, Jim, & Wardrip-Fruin, Noah. (2010). What went wrong: A taxonomy of video game bugs. *Pages 108–115 of: Proceedings of the Fifth International Conference on the Foundations of Digital Games*. FDG '10. New York, NY, USA: ACM.

Nejati, Shiva, Gurfinkel, Arie, & Chechik, Marsha. (2005). Stuttering abstraction for model checking. *Pages 311–320 of: Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*. IEEE.

Nilsson, Henrik, & Courtney, Antony. (2003). *Yampa*. `https://github.com/ivanperez-keera/Yampa`.

Nilsson, Henrik, Courtney, Antony, & Peterson, John. (2002). Functional reactive programming, continued. *Pages 51–64 of: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM.

Paterson, Ross. (2001). A new notation for arrows. *Acm sigplan notices*, **36**(10), 229–240.

Perez, Ivan. (2017a). Back to the Future: Time Travel in FRP. *Proceedings of the 10th ACM SIGPLAN International Haskell Symposium*. Haskell 2017. New York, NY, USA: ACM.

Perez, Ivan. (2017b). *Dunai-test*. `http://hackage.haskell.org/package/dunai-test`.

Perez, Ivan. (2017c). *Yampa-test*. `http://hackage.haskell.org/package/yampa-test`.

Pérez, Iván. (2018). *Extensible and Robust Functional Reactive Programming*. Ph.D. thesis, School of Computer Science, University of Nottingham.

Perez, Ivan. (2018). Fault Tolerant Functional Reactive Programming (Functional Pearl). *Proceedings of the acm on programming languages*, **2**(ICFP), 96:1–96:30.

Perez, Ivan, & Bärenz, Manuel. (2016). *Dunai*. `https://github.com/ivanperez-keera/dunai`.

Perez, Ivan, & Nilsson, Henrik. (2015). Bridging the gui gap with reactive values and relations. *Pages 47–58 of: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. New York, NY, USA: ACM.

Perez, Ivan, & Nilsson, Henrik. (2017). Testing and Debugging Functional Reactive Programming. *Proceedings of the acm on programming languages*, **1**(ICFP), 2:1–2:27.

Perez, Ivan, Bärenz, Manuel, & Nilsson, Henrik. (2016). Functional reactive programming, refactored. *Pages 33–44 of: Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. New York, NY, USA: ACM.

Perez, Ivan, Goodloe, Alwyn, & Edmonson, William. 2019 (July). Fault-tolerant swarms. *IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*.

Pnueli, Amir. (1977). The temporal logic of programs. *Pages 46–57 of: Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE.

Prior, Arthur N. (1967). *Past, present and future*. Vol. 154. Oxford University Press.

Ronsse, Michiel, De Bosschere, Koen, & De Kergommeaux, Jacques Chassin. (2000). Execution replay and debugging. *arxiv preprint cs/0011006*.

Sculthorpe, Neil. (2011). *Towards safe and efficient functional reactive programming*. Ph.D. thesis, University of Nottingham.

Tan, Li, Sokolsky, Oleg, & Lee, Insup. (2004). Specification-based testing with linear temporal logic. *Pages 493–498 of: Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*. IEEE.

Wan, Zhanyong, & Hudak, Paul. (2000). Functional Reactive Programming from first principles. *Pages 242–252 of: ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Whittaker, J. A. (2000). What is software testing? and why is it so hard? *Ieee software*, **17**(1), 70–79.

Winitzki, Sergei. (2014). *Temporal logic and functional reactive programming*. `https://github.com/winitzki/talks/tree/master/frp`.