Rhine: FRP with Type-Level Clocks

Manuel Bärenz
sonnen eServices
Berlin, Deutschland
maths@manuelbaerenz.de

Ivan Perez
National Institute of Aerospace
USA
ivan.perez@nianet.org

	Time	Data
Synchronous	Atomic clocks	Signal functions
Asynchronous	Schedules	Resampling buffers

Table 1. Separation of aspects in Rhine

Abstract

Processing data at different rates is generally a hard problem in reactive programming. Buffering problems, lags, and concurrency issues often occur. Many of these problems are *clock errors*, where data at different rates is combined incorrectly. Techniques to avoid clock errors, such as type-level clocks and deterministic scheduling, exist in the field of synchronous programming, but are not implemented in general-purpose languages like Haskell.

Rhine is a *clock-safe* library for synchronous and asynchronous Functional Reactive Programming (FRP). It separates the aspects of clocking, scheduling and resampling from each other, and ensures clock-safety at the type level. Concurrent communication is encapsulated safely. Diverse reactive subsystems can be combined in a coherent, declarative data-flow framework, while correct interoperability of data at different rates is guaranteed by type-level clocks. This provides a general-purpose framework that simplifies multi-rate FRP systems and can be used for game development, media applications, GUIs and embedded systems, through a flexible API with many reusable components.

CCS Concepts • **Software and its engineering** → **Functional languages**; *Data flow languages*; *Concurrent programming structures*; *Domain specific languages*;

Keywords functional reactive programming, haskell, reactive programming, asynchronous programming

1 Introduction

Complex reactive programs often process data at different rates. For example, in games, rendering is synchronized with the video card refresh rate (e.g., 60Hz), while the game physics may need to run faster, and user input may be received only scarcely and at unpredictable times [13]. Navigation systems combine data from GPS, polled tens of times per second, with data from accelerometers, gyroscopes and magnetometers, polled tens of thousands of times per second.

Coordinating these systems running at different rates is a hard problem in general [19]. Many kinds of bugs can occur, such as accidental synchronisation of independent subsystems, concurrency issues, buffer underruns and overflows, and space and time leaks.

Functional descriptions of reactive systems suffer from similar issues. Functional Reactive Programming [8, 10, 20] frameworks either work synchronously, or leave it to users to coordinate subsystems. Other asynchronous reactive frameworks show similar limitations [25].

Synchronous languages like Lustre [6] and Signal [3] introduce explicit clocks as time-varying boolean expressions at value level. While this approach is suitable for domains like critical systems, their robustness stems from using a limited language to describe clocks, and on using proof techniques that cannot be applied to arbitrary Haskell expressions.

This paper addresses these limitations with Rhine, a framework for explicit coordination of multi-rate FRP systems. Rhine's key strength is that it separates conceptual notions, like signals, data dependency and control, from operational aspects, like clocking, scheduling and resampling.

In Rhine, clock information is expressed at the type level, so that incorrectly clocked programs are *rejected at compile time*. Subsystems running at different rates that need to communicate must be coordinated explicitly and safely, deciding when components become active (*scheduling*) and how data is transferred and adapted between different rates (*resampling*). The mechanisms we provide for each aspect are illustrated in Table 1.

Rhine has been designed as a minimal DSL embedded in Haskell, enabling us to reuse existing libraries to write real-world applications with little or no wrapper code for the

backends. Implementations of clocks, schedules and resampling strategies for standard cases are available, as well as example applications, facilitating adoption.

This paper makes the following contributions:

- Data and clocking aspects, as well as synchronous and asynchronous aspects, are separated clearly, and their compatibility asserted via clock types.
- Features from synchronous languages like deterministic schedules are brought into the Haskell ecosystem.
- Concepts from arrowized and classical FRP are unified. Standard reactive programming techniques are implemented as modular components of a library[1] which easily connects to backends.

Outline

Section 2 revisits synchronous FRP, with emphasis on arrowized FRP, as well as prior work on monadic stream functions, which define reactive constructs with side effects. Section 3 introduces a notion of *type level clocks*, used in Section 4 to introduce clock-safe synchronous FRP in Rhine. Scheduling and asynchronous data flow is covered in Section 5. In Section 6 all concepts are combined to Rhine's main programs. Section 7 gives an overview over the various signal processing features, clocks, schedules and resampling buffers that Rhine has to offer. It also discusses connections to other frameworks, and applications built with Rhine. Qualitative comparisons to other frameworks and discussion of related work are found in Section 8.

2 Background

To make the paper self contained, this section presents the basics of synchronous arrowized FRP as embodied by Yampa [8, 20] and Dunai [24]. We identify the shortcomings that we seek to address in later sections with Rhine, which extends Dunai with clocks and asynchronicity.

Signals and Signal Functions   A central concept in Functional Reactive Programming is that of a *signal*, a value varying with time. Intuitively, signals can be seen as functions with type `time -> a`, where `a` is the signal value type and `time` is a type modelling points in time (for now, `Double`).

Direct implementations of signals typically suffer from time and space leaks [8, 9]. While *classical* FRP frameworks alleviate the issue with a variety of techniques, *arrowized* FRP frameworks such as Yampa [20] avoid the issue conceptually. Taking the "Functional" in FRP literally, the arrowized approach emphasises *signal functions*, which semantically model causal functions from signals to signals, but are implemented in continuation-passing style. Signals are not first class, but exist only as inputs and outputs of signal functions.

Each *tick*, Yampa-style signal functions are fed an input value of type `a` and the time step:

```
data YSF a b = YSF (a -> Double -> (b, YSF a b))
```
The step size, which is the time since the last tick, is not fixed. Rather, the response depends on it continuously, in the form of the second parameter (here as a `Double`). Then, an output sample of type `b` is created, together with a continuation that processes the next tick.

Monadic Synchronous Stream Processing   Knowledge of time, exemplified here as the reader monad `Double ->`, can be seen as a side effect. Dunai [24] generalises Yampa in this aspect: instead of reading the time step each tick, we allow *any* side effect[2] to be executed:

```
data MSF m a b = MSF (a -> m (b, MSF m a b))
```
The abbreviation `MSF` stands for *monadic stream function*, since they are most powerful when `m` is a monad (which will be assumed from now on).

Dunai does not introduce hidden side effects, unlike many other FRP frameworks. The side effects of an MSF are explicit in its type signature, and one may constrain the stream function to be completely pure.

Arrows, Loops and Effects   Monadic stream functions are instances of the `Arrow` type class, which extends `Category`. Thus there is an identity arrow, and pure functions can be lifted to signal functions, which in turn can be composed parallelly and sequentially:

```
id    :: MSF m a a
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 c d
      -> MSF m (a, c) (b, d)
```
Strikingly, not only pure functions can be lifted to `MSF`s, but also Kleisli arrows:

```
arrM :: (a -> m b) -> MSF m a b
```
Additionally, Dunai provides initialised feedback loops.

```
feedback :: c -> MSF m (a, c) (b, c)
         ->        MSF m a      b
```
This allows data to be delayed and memory built up, which enables implementing signal processing components that depend on history, like integrators.

Diverse monads offer many convenient possibilities. Signal functions may share state and environment variables without passing them around explicitly. The list monad allows for branching computations. Dunai programs can be stopped gracefully with `Either e`.

Effects can be combined through *monad transformers*. The functions used to create and handle effects for transformers in standard Haskell can be generalised to useful functions on `MSF`s. For example, escaping an `ExceptT e` effect means handling exceptions of type `e` and thus control flow, which allows us to model streams that terminate, as demonstrated in the following.

---

[1] Available open-source at https://github.com/turion/rhine.

[2] Dependency on time, as a side effect, we will be reintroduced soon.

Control Flow through Exceptions Control flow and termination, which may seem ad hoc in Yampa, become entirely natural with the **ExceptT** monad transformer[3]. If a monadic stream function wishes to terminate with a result value e, it simply produces the side effect **Left** e. For example, we may throw an exception on the input **Just** e:

```
throwMaybe :: MSF (ExceptT e m) (Maybe e) ()
```

If the program should continue, an exception handler has to be supplied that calculates a continuation (which may in turn raise an exception of a different type). With a simple newtype, **MSF**s become monads in the exception type:

```
newtype MSFExcept m a b e
      = MSFExcept (MSF (ExceptT e m) a b)
return :: e  -> MSFExcept m a b e
(>>=)  :: MSFExcept m a b e1
        -> (e1 -> MSFExcept m a b e2)
        ->         MSFExcept m a b e2
```

The function return throws an exception. Bind corresponds to catching an exception and switching to a continuation. For convenience, Dunai offers functions to enter and leave the newtype:

```
try   :: MSF (ExceptT e m) a b
        -> MSFExcept m a b e
safely :: MSFExcept m a b Void -> MSF m a b
```

In colloquial terms, try says "run the **MSF** until an exception occurs", whereas safely says "all exceptions were handled and no exception can occur anymore, thus the **ExceptT** layer can be removed". This leads to a convenient interface for control flow, as will be demonstrated in Section 4.2.

Main Loops Running the main event loop in Yampa requires an unwieldy separation of the program into data-producing sensors and data-consuming actuators (both of which are in **IO**), and a pure signal function. In Dunai, sensors and actuators are just special cases of **MSF**s:

```
type Sensor   a = MSF IO () a
type Actuator b = MSF IO b ()
```

A sensor produces data by means of side effects, and an actuator consumes data while creating side effects.

An **MSF** can be pre- and post-composed with sensors and actuators, yielding a *closed* stream function with no open inputs or outputs, of type **MSF** m () (). Running the main loop[4] is now simply:

```
reactimate :: MSF m () () -> m ()
```

A closed monadic stream function is run indefinitely (or, if in the **Either** monad, until an exception is thrown), and its behaviour is given by the side effects it produces.

Effectful Signal Functions As argued before, knowledge of time is just another side effect, the **Reader** monad. The type of Yampa-style signal functions **YSF** a b is isomorphic to **MSF** (**Reader Double**) a b.

Combining this temporal aspect with arbitrary further effects gives *effectful synchronous signal functions*:

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

As the outermost effect, the time step can be retrieved from the environment, whereas the programmer is free to use any further monads inside the transformer.

Numerical integration, differentiation, and other digital signal processing components can be implemented as effectful signal functions. It is possible to write non-trivial FRP programs – e.g. arcade games – in such a framework [22, 24]. Of course, Yampa programs embed into this approach, by choosing the identity monad for m, but more complex applications are possible with effects.

To run **ESF**s using reactimate, we must supply time information (i.e., sampling times) and pass it to the **ReaderT** environment. One can escape reader layers in Dunai:

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

Instead of implicitly reading the value from the environment, it is passed explicitly as an input. Using runReaderS, we can turn a top-level signal function into an **MSF** in which the sampling times are part of the input stream:

```
mainESF              :: ESF m        ()  ()
runReaderS mainESF :: MSF m (Double, ()) ()
```

We then need to pre-compose such an MSF manually with a stream of time steps of type **MSF** m () **Double**, resulting in an **MSF** m () () which we can run using reactimate.

Shortcomings Let us illustrate the limitations of synchronous FRP in practice with an example that will lead us through the rest of the article.

Imagine we want to create a simple simulation of the game of "fetch" with a virtual dog. The user decides when to throw the ball at a random start velocity, the simulation of the ball is executed, and the status is presented on the screen. Conceptually, we would like to express data dependencies in a reactive system as follows:

```
mainESF :: ESF IO () ()
mainESF = startVel >>> ball >>> statusMsg

-- To be defined elsewhere
startVel    :: ESF IO ()        StartVel
ball        :: ESF m  StartVel Ball
statusMsg   :: ESF IO Ball       ()
type Ball     = ...
type StartVel = ...
```

However, imagine that, as it could occur in real-world applications, different subsystems have to run at different frequencies. The user can trigger an event (i.e., throw the ball at a

---

[3]Recall that **ExceptT** e m a is isomorphic to m (**Either** e a).
[4]Dunai doesn't insist on controlling the main loop, though, and can be stepped from other frameworks as well.

random velocity) by pressing the return key at any time. The simulation of the simple physical system runs in real-time, with, for example, a temporal resolution of 10 milliseconds. The status message the position of the ball is displayed on the console twice per second, and requires exactly 50 simulation steps. But the definition of `mainESF` forces all subsystems to tick simultaneously, and thus to run at a wrong rate.

Adjusting the synchronous arrowized FRP implementation above to account for these multi-rate requirements is error-prone and unsatisfactory, for a number of reasons. Since signal functions are synchronous, *all* subsystems would need to be sampled at the same times, a severe restriction. To run all systems in the same thread, one needs to manually downsample the simulation state by counting the steps and accumulating the data in a list, and only emitting it for every 50th step. (If instead simulation and status message run in different threads and communicate through a buffer, the sampling ratio would not be stable, and the buffer may be underrun or overflowed.) Furthermore, listening to user input from the standard input is blocking and has to be performed in a background thread, while polling a shared variable in the foreground thread.

Such systems perform unnecessary recomputation [11]. The user input and the output are executed as frequently as the physical simulation, which is unnecessary and may add a performance delay. Finally, supplying the stream of time steps to the simulation manually mixes signals (a conceptual notion) and sampling times (an implementation detail).

All the mentioned problems arise from forcing implementation details of scheduling, resampling and communication onto the programmer, who just wants to declare the data flow conceptually. Rhine solves these problems by extending Dunai's MSFs with clocking information and clock-safe combinators.

The example is implemented in this article in less than a hundred lines of literate Haskell[5]. As we develop the example, the four aspects from Table 1 are separated. In particular:

- Each of the three subsystems is declared individually, without worrying about resampling or scheduling.
- All subsystems run safely under their own clock.
- Simulation and status message are synchronised deterministically, while the standard input events are scheduled safely in a separate thread.
- Data is transferred between the different systems without buffering mishaps.

The direction of the data flow will be clearly visible, and the overview over the whole program can be kept.

3 Clocks

Here, we introduce clocks and time domains. In the next sections, we associate clocks to signal functions, preventing

accidental clock errors, and we provide clock-safe mechanisms to connect signal functions on different clocks.

In Rhine, a *running clock* is an effectful stream of time stamps and *tags*, which may contain further information about the clock state or the nature of the tick:

```haskell
type RunningClock m time tag
  = MSF m () (time, tag)
```

A running clock may produce side effects such as blocking, or polling the system clock or another device. A running clock is said to *tick* at the time stamps it outputs. In the case of real-time clocks, it is the obligation of the clock implementation to correctly observe the system time, and to wait, if necessary, until the desired time has been reached.

Time Domains and Tags The type `time` in a running clock is a *time domain*, it represent points in time. Since different situations may call for different implementations of the `time` type, Rhine parametrises over it with a type class:

```haskell
class TimeDomain time where
  type Diff time
  diffTime :: time -> time -> Diff time
```

The type family `Diff time` represents time durations. We can calculate the duration between two sampling points with `diffTime`. From now on, it will always be assumed that type variables called `time` are instances of `TimeDomain`. Typical instances are `Double`, `UTCTime` or even `Integer`.

The additional data supplied by the tags are useful in real world applications, as they may specify *why* a clock ticked (e.g., the occurrence of a particular event), or *how* it ticked (e.g., whether an attempt at soft real-time was successful).

Clock Types In this section, we will see how Rhine's clocks supply the time steps to the synchronous subsystems in a *clock-safe* way, i.e. such that every subsystem will be supplied with the correct time steps, and subsystems with different clocks cannot be synchronised by accident.

The key idea is to let the type checker assert the clock-safety. Thus, a type class `Clock` is supplied[6], and its instances will be called *clock types*:

```haskell
class Clock m cl where
  type Time cl
  type Tag  cl
  initClock
    :: cl -> RunningClock m (Time cl) (Tag cl)
```

For the remainder of the article, it will be assumed that any type variable called `cl` is an instance of `Clock m`.

Two concepts have to be distinguished:

The clock type `cl` specifies all relevant properties of the clock, in particular its time domain, when and how fast it ticks, in which monads its side effects can take place, and what kind of `tag`s it produces.

---

[5]To be made available under https://github.com/turion/rhine on publication.

[6]The actual implementation is only slightly more involved to allow for more complex initialisation actions of the clock.

The clock value, i.e. a value of type `cl`, holds all information necessary to *run* the clock, such as event sockets, device addresses, or implementation choices. If no information is required, the clock is a singleton.

A running clock supplies absolute times. The time differences to the respective last ticks can be calculated using `diffTime`. For convenience, the absolute time and the time since the clock initialisation will also be calculated, and supplied in a data type together with the tag:

```
data TimeInfo cl = TimeInfo
  { sinceTick :: Diff (Time cl)
  , sinceInit :: Diff (Time cl)
  , absolute  :: Time cl
  , tag       :: Tag cl
  }
```

Example Three clocks in the `UTCTime` time domain are present. The simulation rate at 100 steps per second constitutes a clock, and so does the status display rate at two ticks per second. The standard input events (when the user presses the return key) are ticks of a clock as well. It neither ticks at regular intervals, nor approximates temporal continuity, but it does tick every time when the subsystem processing the input line needs to become active. This point of view unifies events and behaviours, as explained in the next section.

We define the clock types as type synonyms:

```
type EventClock  = StdinClock
type SimClock    = Millisecond 10
type StatusClock = Millisecond 500
```

`StdinClock` is a singleton. `Millisecond n`, parameterised by a type-level natural number representing the step size, is implemented in the library as a "rescaled" clock:

```
newtype Millisecond (n :: Nat) = Millisecond
  (RescaledClock IO (FixedStep n) UTCTime)
```

A value of type `RescaledClock` is an effectful translation of a clock into another time domain. The intermediate clock `FixedStep n` is *pure*: it successively outputs the tick values 0, n, 2*n etc. in the `Integer` time domain, without any side effects. Rescaling to `UTCTime` by means of `IO` means to wait for every tick until the required time span has passed, and then emit the current real time. Rhine supplies an implementation, `waitClock :: Millisecond n`.

4 Clock-safe Synchronous FRP

The fundamental components of Rhine's signal networks are called *clocked signal functions*. They are effectful synchronous signal functions that are aware of the time information of a particular clock type:

```
type ClSF m cl a b
  = MSF (ReaderT (TimeInfo cl) m) a b
```

As a key contribution, and the basis for clock-safe coordination, two clocked signal functions can only be composed if their clock types agree.

The extra functionality in clocked signal functions is access to time information, such as the time since the last tick[7]:

```
sinceTickS :: ClSF m cl () (Diff (Time cl))
```

A closed synchronous signal function can be run together with a clock value of the correct type:

```
reactimateCl :: cl -> ClSF m cl () () -> m ()
```

Internally, the clock is run, and the resulting stream of type `TimeInfo cl` (which is calculated from the timestamps), supplies the environments for the `ReaderT (TimeInfo cl)` effect in the signal function.

Example In Rhine, it is easy to build a whole program bottom-up by first creating the synchronous subsystems, and joining them later.

Let us start with the display of a status message. We represent the position and the velocity of the ball with a three-dimensional vector of `Double`s:

```
type Ball    = (Double, Double, Double)
type BallVel = (Double, Double, Double)
```

We would like to print a ball position to the console, and annotate its clock type to ensure that it is run at the correct speed. One can simply lift Kleisli arrows to signal functions with a library function that generalises `arr` and `arrM`:

```
arrMCl :: (a -> m b) -> ClSF m cl a b
```

The status message display then becomes an *actuator* since it consumes data and converts it to an effect:

```
statusMsg :: ClSF IO StatusClock Ball ()
statusMsg = arrMCl $ \(x,y,z) ->
  printf "%.2f %.2f %.2f\n" x y z
```

4.1 Unifying Events and Behaviours

We can describe events and behaviours in terms of clocked signal functions. They are merely distinguished by their clock types. This is a drastic simplification over classical FRP.

Behaviours In the original spirit of FRP, behaviours describe a value that continuously changes with time, independently of the sampling strategy. Continuity is approached by sampling at arbitrary times, thus a *behaviour signal function* is defined as a clocked signal function which is *clock-polymorphic* over a given time domain:

```
type BehaviourF m time a b
  = forall cl. time ~ Time cl => ClSF m cl a b
```

A typical example of a `BehaviourF` is numerical integration:

```
integral
    :: (VectorSpace v, Groundfield v ~ Diff time)
    => BehaviourF m time v v
```

---

[7]The other `TimeInfo` fields can be accessed in a similar manner.

The incoming signal samples are weighted with the time differences and added up, requiring the **VectorSpace** instance. The precision and step size of the numerical method can be adjusted arbitrarily by selecting different clocks. More examples of signal processing components implemented in Rhine can be found in Section 7.1.

Events   Event sources constitute clocks as well, where the clock ticks whenever an event is emitted. The original event data is available as `tagS :: ClSF m cl () (Tag cl)`, and further events can be created by composition with signal functions. Two events occur simultaneously[8] if they have the same clock, which is immediately visible in the type.

Example   The motion of the ball consists of two modes: a stationary position at the origin while waiting, and free fall. Let us implement free fall first. It is a good example of a behaviour, since the trajectory of the freely falling ball should conceptually be independent of the sampling rate. We will only require the simulation to happen in real-time, and thus constrain the time domain to **UTCTime**.

```
freeFall :: Monad m
         => BallVel -- ^ The start velocity
         -> BehaviourF m UTCTime () Ball
freeFall v0 = arr (const (0, 0, -9.81))
             >>> integralFrom v0 >>> integral
```

The acceleration is constant gravity. We integrate once to get the velocity, and again, to get the ball position. The type signature, polymorphic in the monad `m`, reflects that there are no side effects other than awareness of time.

The console standard input is clearly an event source. We will discard the event data (the line that was entered by the user) and generate the start velocity of the ball randomly:

```
startVel :: ClSF IO StdinClock () BallVel
startVel = arrMCl $ const $ do
  velX <- randomRIO (-10, 10)
  velY <- randomRIO (-10, 10)
  velZ <- randomRIO (  3, 10)
  return (velX, velY, velZ)
```

Every time the return key is pressed, a new start velocity is produced, thus `startVel` is a *sensor*. It is called precisely when the event occurs, so we need not manually resample it as **Maybe** values and provide **Nothing** when the event is absent. This is a key difference between Rhine and pull-based frameworks like Yampa, in which an absent event still carries an empty value and results in an unnecessary recomputation of the whole network.

---

[8]Non-simultaneous events can be treated with the tools from Section 5.

4.2   Exception Handling and Control Flow

Rhine mimics Dunai's exception handling interface. The newtype **ClSFExcept** `m cl a b e` encapsulates signal functions on the clock `cl` throwing exceptions of type `e`. Exceptions can again be thrown with `throwMaybe`, the monad interface can be entered with `try`, and left with `safely`.

Example   To introduce the control flow that switches between the two modes, we use exceptions:

```
waiting :: Monad m => ClSF (ExceptT BallVel m)
           SimClock (Maybe BallVel) Ball
waiting = throwMaybe >>> arr (const zeroVector)
```

The waiting mode, constantly outputs the coordinate origin, but on an input event **Just** v0 it throws v0 as an exception.
   To implement the falling mode, we use arrow notation:

```
falling :: Monad m
        => BallVel -- ^ Initial velocity
        -> ClSF (ExceptT () m) SimClock
           (Maybe BallVel) Ball
falling v0 = proc _ -> do
  pos <- freeFall v0 -< ()
  let (_, _, height) = pos
  throwMaybe        -< guard $ height < 0
  returnA           -< pos
```

The input is ignored, since the ball can only be thrown from the waiting mode. The current position of the ball is calculated and returned, but before the height would become negative, a singleton exception is thrown.
   Since **ClSFExcept** is a monad, we can specify the control flow via **do**-notation:

```
ballModes :: ClSFExcept IO SimClock
             (Maybe BallVel) Ball void
ballModes = do
  v0 <- try waiting
  once_ $ putStrLn "Catch!"
  try $ falling v0
  once_ $ putStrLn "Caught!"
  ballModes
```

The ball is in the waiting mode until it throws the new start velocity as an exception. At this point the dog owner shouts "Catch!" to the dog once, using a library function which executes one monadic action within the current tick:

```
once_ :: Monad m => m e -> ClSFExcept m cl a b e
```

The new start velocity is then passed to the falling mode, which continues until it throws an exception just before the height becomes negative. We assume a perfect dog that always catches the ball (with its owner shouting "Caught!"), and returns it without delay. The simulation restarts. Finally, the type checker verifies that no exception is left unhandled:

```
ball :: ClSF IO SimClock (Maybe BallVel) Ball
ball = safely ballModes
```

4.3 Clock Safety

In our example, we have not yet addressed how the ball velocity events emitted by the standard input subsystem are transported into the simulation subsystem. Rhine ensures that synchronous components on different clocks do not communicate directly. The arrow combinators require the clock types to match:

```
(>>>) :: ClSF m cl a b -> ClSF m cl b c
      -> ClSF m cl a c
```

One might try to naively connect the two subsystems:

```
userBall = startVel >>> arr Just >>> ball
```

But this would accidentally synchronise the systems, and sample the ball system only when a standard input event occurs, leading to big lags. By Rhine's design, such an attempt will result in a clock type error:

```
Couldn't match type `Millisecond 10'
  with `StdinClock'
```

The type error is very clear and helpful. Furthermore, the clock types serve as useful documentation of the processing rates of the different subsystems.

Nevertheless, the two subsystems will have to communicate eventually. This is addressed in the next section.

5 Asynchronicity

To coordinate subsystems on different clocks safely, we introduce *schedules*, which determine how two clocks can be combined, and *resampling*, which addresses how signal functions at different rates can communicate.

5.1 Schedules

The *scheduling problem* is the question of when to execute the ticks of several differently clocked synchronous subsystems. There is no universal solution for it, and different situations call for different schedules: some subsystems may be scheduled in a deterministic way with a fixed resampling ratio, while others might have to make use of concurrency.

Consequently, Rhine gives the programmer the freedom to implement their own schedules, while supplying default implementations to cover typical use cases.

Schedules   A *binary schedule* for two clocks cl1 and cl2 has the semantics of a universal clock such that cl1 and cl2 are its subclocks. Its tag specifies which subclock ticks:

```
data Schedule m cl1 cl2
  = Time cl1 ~ Time cl2 => Schedule
  { initSchedule :: cl1 -> cl2
      -> RunningClock m (Time cl1)
          (Either (Tag cl1) (Tag cl2))
  }
```

Depending on the clocks, a binary schedule might not exist (e.g., if the side effects required by the clocks are incompatible) or not be unique (e.g., if certain ticks of the clocks have

to occur simultaneously, and an order has to be chosen). Schedules can be polymorphic in the clocks (such as the concurrency schedule mentioned before), or polymorphic in the side effects (such as deterministic schedules).

Example   The simulation system has to be *deterministically* scheduled with the status message, in the sense that one status message has to emitted after exactly 50 simulation steps. We want to ensure the determinism of the schedule by tracking its side effect as a type parameter. The intermediate **FixedStep** n clocks, on which **Millisecond** n is based, have pure, deterministic schedules in the library.

```
scheduleFixedStep
  :: Schedule m (FixedStep n1) (FixedStep n2)
```

Also contained in the library is a rescaling of this schedule to the **Millisecond** n clocks, which reintroduces the waiting side effects already present in the clocks, but adds no further effects. The sampling ratio is thus still deterministic:

```
scheduleMillisecond
  :: Schedule IO (Millisecond n1)
                 (Millisecond n2)
```

On the other hand, it is of course impossible to predict how many steps may pass between two return key presses, so there is no deterministic schedule for the standard input clock and the simulation clock. The simplest solution is to fork separate threads for the two clocks and to collect the ticks from a shared variable in the foreground thread. In other words, we rely on Haskell's concurrency mechanisms to solve this scheduling problem. Rhine provides a clock-polymorphic schedule, introducing **IO** as side effect:

```
concurrently :: Schedule IO cl1 cl2
```

It encapsulates all concurrent communication between the clocks, and thus eliminates typical pitfalls, since no threads or shared variables need to be created by the programmer.

Clock Trees   A binary schedule for two clocks cl1 and cl2 with side effects in m, together with values for the individual clocks, gives a value of type **SeqClock** m cl1 cl2. As will be shown in the next section, sequential clocks are used to type signal networks that are *sequentially* composed of two subsystems, where the subsystem under cl1 produces data that the subsystem under cl2 consumes.

It is also possible to compose signal networks *parallelly*, and again a schedule is needed to clock such compositions. Such a schedule, and values for the individual clocks, yield a value of type **ParClock** m cl1 cl2.

Sequential and parallel clocks are both called *combined clocks*, and they tick whenever any of their constituents ticks. All other clocks are called *atomic*.

The constituents of combined clocks may themselves be composed of further clocks, resulting finally in an ordered tree with atomic clocks at the leaves and clock-safe schedules on the nodes, called the *clock tree*. Rhine supplies type

families **In** and **Out**, which calculate the clocks at which data enters, or leaves the system, respectively.

5.2 Resampling

Scheduling describes the *clock aspect* of running several systems at different rates, whereas resampling describes the *data aspect*. One fundamental advantage that Rhine has over any other FRP framework in Haskell known to us, is the separation of these two concerns. The resampling problem can be treated separately from the scheduling problem, allowing for modular, reusable solutions. Of course, the two problems are often intertwined, and some data can only be resampled under the assumption of specific clocks. This can be encoded as clock type constraints in Rhine.

Like scheduling, there is no single solution to the resampling problem. The choice is again with the programmer, and popular resampling techniques (such as buffers or interpolations) are already given in the library.

Resampling Buffers   The fundamental building block of resampling provided by Rhine is a *resampling buffer*:

```
data ResBuf m cl1 cl2 a b = ResBuf
  { put :: TimeInfo cl1
        -> a -> m (   ResBuf m cl1 cl2 a b)
  , get :: TimeInfo cl2
        ->       m (b, ResBuf m cl1 cl2 a b)
  }
```

They are the connecting link between two neighbouring subsystems. The schedule decides which subsystem ticks. If the left subsystem produces data, it is stored in the resampling buffer, together with a timestamp from the schedule, via put. If the right subsystem requires data, it is retrieved (again timestamped) from the resampling buffer via get. The implementation is akin to MSFs, in that continuation passing style is used, but resampling buffers are fundamentally asynchronous: input and output never happen simultaneously.

Clock-polymorphic buffers accept put and get calls at any time. Buffers requiring a specific pattern of puts and gets (e.g. to meet space requirements) can constrain the clocks in the type signature. Resampling buffers can range from digital signal resampling such as linear, cubic or sinc interpolation, to buffering and queueing strategies like FIFO queues.

Example   Data has to be resampled in two places: at the boundary between the standard input and simulation subsystems, and between the simulation and status subsystems.

From standard input to simulation, we will simply use a FIFO queue from the library:

```
fifo :: ResBuf m cl1 cl2 a (Maybe a)
```

It is clock-polymorphic and side-effect free. Incoming data is queued in a sequence, and the oldest value, if present, is output, with **Nothing** representing an empty buffer.

From the simulation to the status message, we can use deterministic scheduling and a buffer provided by the library that collects the values in vectors of statically known length[9]:

```
downsampleMillisecond
  :: ResBuf m (Millisecond k)
       (Millisecond (n * k)) a (Vector n a)
```

In our example, the type-level natural numbers k and n * k are known at compile time, and the size of the buffer n is statically derived. With the operator >>-^ that composes a resampling buffer with a clock-matching **ClSF**, we build our own buffer which – for simplicity – picks the first of all simulation results that occurred since the last status clock tick:

```
downsampleSimToStatus
  :: ResBuf IO SimClock StatusClock Ball Ball
downsampleSimToStatus = downsampleMillisecond
                  >>-^ arr head
```

Knowing that exactly 50 samples will accumulate for every tick of the status clock, we can safely apply head to them, without having to handle the case of an empty vector.

6 Signal Networks and Main Loops

We finally combine all of the previous ideas: *signal networks*, which are clock-safe combinations of clocked signal functions with matching resampling buffers, and *Rhines*, which are signal networks with a specific clock value.

Signal Networks   Asynchronous *signal networks* consist of signal functions and resampling buffers that tick under a given combined clock. They are implemented as trees of the same shape as the clock tree:

```
data SN m cl a b where
  Synchronous
    :: ( cl ~ In cl, cl ~ Out cl)
    => ClSF m cl a b -> SN m cl a b
  Sequential
    :: SN m            clab          a b
    -> ResBuf m (Out   clab) (In clcd)   b c
    -> SN m                        clcd      c d
    -> SN m (SeqClock m clab      clcd) a     d
  Parallel
    :: SN m            cl1       a b
    -> SN m              cl2 a b
    -> SN m (ParClock m cl1 cl2) a b
```

In a signal network of type **SN** m cl a b, m is the type of side effects, cl is the type of the whole clock, and a and b are input and output, respectively.

A signal network forms a tree that matches the topology of its clock. **Synchronous** signal functions under the corresponding atomic clocks sit on the leaves. **Sequential**ly combined clocks allow the tree to branch, and such nodes

---

[9]Rhine uses the implementation from the vector-sized [15] package.

are tagged by clock-safe resampling buffers. The combined clock will decide which subnetwork becomes available, and whether to store data in, or retrieve data from, the buffer. Two signal networks can also be combined *temporally* in **Parallel**. They will be activated in turns, depending on which constituent of the parallely combined clock ticks. No data needs to be transferred between them (thus no buffers are necessary), but their input and output data types must match.

Rhines An asynchronous signal network, together with a value of the correct clock type is called a *Rhine* (since many streams flow in it):

```
data Rhine m cl a b = Rhine
  { sn    :: SN m cl a b
  , clock :: cl
  }
```

A closed Rhine can be run as a main loop:

```
flow :: Rhine m cl () () -> m ()
```

The main loop waits for the next tick of the top schedule, and then navigates through the tree to step the corresponding synchronous signal function on the leaf, putting data into, and getting data from the neighbouring resampling buffers.

Example The operator @@ combines a signal function with a clock value of the correct type. With this syntactic sugar, we create the synchronous subsystems:

```
startVelRh :: Rhine IO StdinClock () BallVel
startVelRh = startVel @@ StdinClock

ballRh :: Rhine IO SimClock (Maybe BallVel) Ball
ballRh = ball @@ waitClock

statusRh :: Rhine IO StatusClock Ball ()
statusRh = statusMsg @@ waitClock
```

With further syntactic sugar, we can join the subsystems:

```
simToStatus :: ResamplingPoint IO SimClock
                      StatusClock Ball Ball
simToStatus
 = downsampleSimToStatus -@- scheduleMillisecond

ballStatusRh
   :: Rhine IO (SeqClock IO SimClock StatusClock)
       (Maybe BallVel) ()
ballStatusRh = ballRh >--simToStatus--> statusRh
```

A **ResamplingPoint** is a mere bookkeeping tool that pairs a resampling buffer and clock-matching schedule. The operator -@- combines a resampling buffer and a schedule to a *resampling point*. The operators >-- and --> compose two **Rhine**s along a resampling point, building up the clock tree and the signal network tree simultaneously.

The complete Rhine program can finally be run:

```
main :: IO ()
main = flow $ startVelRh
  >-- fifo -@- concurrently --> ballStatusRh
```

```
0.00 0.00 0.00
0.00 0.00 0.00

Catch!
1.09 1.51 3.08
2.39 3.31 4.49
3.68 5.11 3.44
Caught!
0.00 0.00 0.00
[...]
```

In this example, the main program has to be terminated manually, but one can easily add a graceful exit by lifting the Rhine to the **ExceptT () IO** monad and throwing () in the standard input subsystem when the user enters "q".

7 A Brief Tour through the Library

7.1 Signal Functions and Signal Processing

Rhine provides utilities to manipulate the monad transformer stack of a clocked signal function, such as lifting into transformers, and entering and leaving **ReaderT r** layers. In particular, it provides control flow via exceptions, like Dunai.

Using knowledge of time as an extra side effect, standard digital signal processing tools are implemented in terms of **ClSF**s, such as edge detectors, numerical integration and differentiation, moving averages, and high and low passes.

7.2 Clocks

Real Time Apart from the **Millisecond** n clocks, which were presented in the example, other implementations of soft-real-time clocks (polling the system time) exist, notably the **Busy** clock which tries to tick without delay.

Rhine provides an **AudioClock** for soft-real-time audio synthesis and analysis. The computation of the audio samples happens in buffers, and is aligned with system time at the end of each completed buffer. The type is parametrised by a data kind representing the sampling rate, so accidental pitch changes due to missing resampling cannot occur.

Deterministic Clocks The **FixedStep** n clock, with the type level natural number n, has already been mentioned in the example. It is generalised by the deterministic **Periodic** clocks, which are parametrised by a type level *list* of natural numbers. These clocks tick in periodic steps, for example **Periodic '[1, 2]** ticks at the times 1, 3, 4, 6, and so on.

Deterministic clocks can be used to *test* Rhine programs, a notable feature in the FRP world [26]. Since the clocks require no side effects, a simulation can be run at arbitrary speed. This is useful to debug Rhine programs with QuickCheck [7], where we can run a whole signal network with randomly generated test data, and automatically check its behaviour. Another practical use case is batch processing of big files.

Event Clock As explained in Section 4.1, an event source is a clock. Rhine offers a general purpose **Event** clock, which

encapsulates a `Chan` across which the events are communicated. Users can emit events from any place in the signal network with the library method `emit`. Like the standard input clock from the example, it blocks until an event becomes available, and then ticks, storing the event data in the `Tag`.

Concurrent data   While Rhine supplies concurrent scheduling, data is processed in a single thread by default. This is undesired when performing expensive computations or blocking effects that take longer than their allotted time step.

A standard solution is to place the computation in a separate thread, and be notified when the data is ready. In Rhine, the `Event` clock can provide such a notification: the `Chan` over which it communicates can be shared across several main Rhines running in separate threads. In one of them, the time-consuming computation can take place, and the fully evaluated result is emitted as an event. In the thread where the `Event` clock ticks, it is received and made available to signal functions under this clock.

This technique is especially useful when connecting to backend libraries with their own notions of events and main loops. It shares similarities with *wormholes* [32].

Selection Subclocks   For a given reference clock, a sub-clock can be defined by selecting certain tags:

```haskell
data SelectClock cl a = SelectClock
  { mainClock :: cl
  , select    :: Tag cl -> Maybe a
  }
```

The main clock is run, but a tick for the `SelectClock` occurs only when `select` applied to the current tag of the main clock is `Just a`. The value a is then the tag of the subclock[10].

7.3   Schedules

Concurrent Scheduling   As in the example, any two clocks in `IO` may be scheduled concurrently. Further schedules for `ReaderT r IO`, `ExceptT e IO` and `WriterT w IO` exist that synchronise the effects across threads.

Deterministic Schedules   The `FixedStep` and `Periodic` clocks using type level natural numbers can be scheduled deterministically without side effects. Likewise, there exist deterministic schedules between a selection subclock and its reference clock, and also between two selection subclocks.

Universal Scheduling   To provide a declarative interface to effectful clocks that can be automatically scheduled, a new monad transformer `ScheduleT diff` is defined as a free monad which takes a time difference argument `diff` and adds the functionality of formally waiting for a certain time before resuming with the effectful computation.

Two *arbitrary* clocks in the `ScheduleT` monad can always be scheduled, using a single, polymorphic[11] schedule:

```haskell
schedule :: Schedule (ScheduleT diff m) cl1 cl2
```

If a concrete blocking effect in the encapsulated monad is provided, the `ScheduleT` transformer can be escaped. For example, we can transform formal waiting operations into `IO` delays, interpreting `Int`s as milliseconds:

```haskell
runScheduleIO :: ScheduleT Int IO a -> IO a
```

7.4   Resampling Buffers

The library provides a general purpose buffer that keeps the newest input value in its internal state (defaulting to an initialisation value), and returns it on a `put` call:

```haskell
keepLast :: a -> ResBuf m cl1 cl2 a a
```

In the special case when `cl2` ticks much faster than `cl1` (thus approximating continuity), `keepLast` models a *zero-order hold*. Other interpolation methods, (e.g. first-order holds, cubic and sinc) are also implemented.

The unbounded FIFO queue has been demonstrated in the example already. There exists a bounded version which discards old values on overflow, and analogous LIFO variants.

Utilities   Clock polymorphic buffers which are unaware of the passage of time can be built from *asynchronous Mealy machines*, consisting of effectful getter and setter functions:

```haskell
data AsyncMealy m s a b = AsyncMealy
  { amPut :: s -> a -> m     s
  , amGet :: s        -> m (b, s)
  }
timelessResamplingBuffer
  :: AsyncMealy m s a b -> s
  -> ResamplingBuffer m cl1 cl2 a b
```

We have already seen combinators that post-compose `ResBuf`s with clock-matching `ClSF`s in Section 5, and a combinator for pre-composition exists as well. One can also parallelly compose clock-matching resampling buffers.

Avoiding Leaks   `ResBuf`s consume a potentially unpredictable number of input values to produce one output value. All `put` calls can be collected in a list[12], and returned when `get` is called:

```haskell
collect :: ResBuf m cl1 cl2 a [a]
```

But collecting the values in a list can lead to a space leak, and processing the whole list at once can create a time leak. Therefore, the implementation of `ResBuf` in terms of continuation passing style is justified, since it allows for strict, leak-free `put` implementations. For example, if all we would do is folding the list of collected values, we can do so efficiently with the following buffer from the library instead:

---

[10]Two different selection subclocks of the same main clock that produce the same tag type have the same type at first, so it is customary to distinguish them by newtyping the tag.

[11]As a further restriction, both clocks need to operate on the same time domain, with time difference type `diff`.

[12]This list does not contain the time stamps of the `put` calls, but a generalised version exists in the library that does.

```
foldBuffer
  :: (a -> b -> b) -- ^ The folding function
  -> b             -- ^ The initial value
  -> ResBuf m cl1 cl2 a b
```

The folding function is applied strictly on each `put` call, so if `b` can be stored with constant size and computed in constant time, the buffer will not have space leaks or time leaks.

7.5 Signal Network Combinators

The library provides numerous utilities and combinators for signal networks and Rhines. Signal networks can be composed in parallel, albeit in two different ways. First, if the clock types match, the **SN**s can be composed by processing the data in parallel:

```
(****) :: SN m cl  a       b
          -> SN m cl     c       d
          -> SN m cl (a, c) (b, d)
```

At one tick of `cl`, the corresponding component of the first network is called before the second. Alternatively, if the two **SN**s are on different clocks, but have the same input type, they can be composed as well, although clock-parallelly:

```
(++++)
  :: SN m              cl1     a         b
  -> SN m                cl2 a           c
  -> SN m (ParClock m cl1 cl2) a (Either b c)
```

At one tick of **ParClock** m cl1 cl2, one subnetwork becomes active, depending on whether `cl1` or `cl2` has ticked.[13]

7.6 Wrappers for Other Frameworks

Since Rhine is parametrised over monads, it is easy to integrate it with existing backends. Often, one can simply call the backend functions via `arrMCl`.

For backends that have their own notion of events (such as OpenGL or SDL), it is usually straightforward to connect them to Rhine's **Event** clock, or write a new clock from scratch. An example implementation for SDL exists.

If the backend has a pure interface, it is usually possible to attach pure Rhine code to it. To connect to Gloss [18], Rhine provides a side-effect-free Gloss clock[14]. Signal functions under this clock can be simulated in Gloss, which facilitates developing simple OpenGL applications in Rhine.

7.7 Example Applications

This article is written in literate Haskell and can be executed. The code is less than 100 lines, with no hidden boiler plate.

The rhine-examples package[15] contains many small examples that demonstrate how to write signal functions, use different clocks, run several main loops communicating through events, and handle control flow with exceptions. A presentation and tutorial are also available[16]. A user-contributed web application in form of a stock market game compiling with GHCJS can be found online[17].

A more extensive example[18] implements an interactive graphical animation of a solar power system, with Rhine's Gloss bindings. The behaviour of the system is literally defined as a pure **BehaviourF**, with user actions as input and system state as output. Two backends are supplied, a pure GUI using Gloss, and an effectful one via the console.

8 Related Work

Most other FRP Haskell frameworks do not supply type-level clocks, making it hard to reason about the speeds of the different subsystems. To our knowledge, the single exception is CλaSH [2], which only provides fixed-rate logical subclocks of a main clock. CλaSH [2] targets hardware circuits and does not support **IO**. Rhine provides arbitrary clocks and works with standard Haskell.

Most frameworks do not explicitly distinguish clocks and data. In particular, they do not separate the scheduling aspect from the resampling aspect. In Rhine, this separation allows for modularity and reusability of Rhine's components.

Classic (non-arrowized) FRP frameworks inherit the separation into events and behaviours from Fran [10]. Rhine unifies and generalises the two concepts. Most classic frameworks also hide **IO** primitives under their abstractions, typically **IORefs** such as in Sodium [4], Elerea [21] and Reactive Banana [1], or even `unsafePerformIO` such as in FRPNow [27]. Rhine is completely pure, its abstractions are parametrised over monads. This is very useful since it is then possible to simulate signal networks, e.g. for testing purposes, without user interaction. Additionally, one can reason about the behaviour of the reactive program, since there are no hidden side effects. For example, Gloss provides a pure interface, and Rhine can in fact connect to it. The benefits of **IO** can be brought back by simply instantiating the framework with components using **IO**.

Dunai, Netwire [31], varying [29] and Auto [16] have a monad type parameter to allow reasoning about the possible side effects, but they are all synchronous in nature.

Pipes [12] and Conduit [30] are asynchronous and also quantify over monads. They are powerful in their domain, but lack any treatment of time and cannot be regarded as FRP frameworks. Finally, they have no notion of clocks, and lack the separation of scheduling and resampling.

---

[13] The input types must match, as opposed to the more familiar operator `+++` from **ArrowChoice** which is additive in the input type: due to the separation of clock and data aspects in Rhine, the parallel clock chooses which subsystem becomes active, while in an **ArrowChoice** instance, the decision depends on the incoming data. Similarly, there is no **Category** instance, so no **Arrow** instance for signal *networks*. We can compose them sequentially and parallelly, but an identity signal network cannot exist.
[14] http://hackage.haskell.org/package/rhine-gloss

[15] Available in https://github.com/turion/rhine.
[16] https://github.com/turion/rhine-tutorial
[17] https://github.com/fphh/rhine-ghcjs
[18] https://github.com/turion/sonnendemo

Outside the Haskell ecosystem, languages like Lucid Synchrone [6], Lustre [14], Signal [3, 17] and Multi-rate Esterel [28] introduce notions of clocks and supply useful clock combinators. With the exception of Lucid Synchrone, clocks are defined at value level as boolean signals. In contrast, we do not assume a global reference clock, and use type-level information to determine clock compatibility. Also, these systems are designed for analysis and compilation of critical embedded control software, whereas ours is interoperable with Haskell and has been designed to facilitate simulation.

The hybrid language Zélus [5] includes constructors to define point-wise, stateful/discrete-time and continuous-time signals. The determination of the sampling point at which to sample each signal is done by using ODEs and numerical solvers to detect points of zero-crossing. In contrast, Rhine does not solve an ODE to detect when a signal should be sampled, and relies on the clock implementation. Additionally, Rhine signals may include type-level information that indicates if they are discrete-time or continuous-time, whereas in Zélus, this information is required by construction and determines, via a subtyping relation between these time or clock domains, how signals can be combined.

9 Summary

We have seen that annotating synchronous subsystems of a signal network with type level clocks prevents clock errors and guides through the development of an FRP application. The separation of data flow, clocking, scheduling and re-sampling aspects allowed us to combine modular, reusable components in a clock-safe manner. Behaviours and events became special cases of clocked signal functions, furthermore deterministic and concurrent schedules as well as a multitude of resampling and queueing strategies could be unified. The top-level programs could be declared in an intuitive data flow fashion, and easily connect to different backends.

As future work, writing more backends for GUI and multimedia would lower the bar for real-life FRP development more. Bigger example applications would help us better determine how well Rhine's abstractions scale.

We plan to extend Rhine to compile FRP system descriptions to languages with predictable memory usage, like ATS and C, to use our proposal for embedded systems. In this setting, we plan to use recent work on annotating monadic stream functions with fault tolerance information [23] to indicate, at the type level, possible faults in Rhine networks.

Acknowledgements

References

[1] Heinrich Apfelmus. 2011. Reactive-banana. https://github.com/HeinrichApfelmus/reactive-banana.

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*. IEEE Computer Society, 714–721.

[3] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of computer programming* 16, 2 (1991), 103–149.

[4] Stephen Blackheath. 2012. Sodium. https://github.com/SodiumFRP/sodium.

[5] Timothy Bourke and Marc Pouzet. 2013. Zélus: A Synchronous Language with ODEs. In *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*. Philadelphia, USA, 113–118.

[6] Paul Caspi and Marc Pouzet. 2000. *Lucid Synchrone, a functional extension of Lustre*. Technical Report. Université Pierre et Marie Curie, Laboratoire LIP6.

[7] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the International Conference on Functional Programming (ICFP 2000)*. ACM.

[8] Antony Courtney and Conal Elliott. 2001. Genuinely Functional User Interfaces. In *Haskell Workshop*. 41–69.

[9] Conal Elliott. 1998. Functional implementations of continuous modeled animation. In *Principles of Declarative Programming*. Springer, 284–299.

[10] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, 263–273.

[11] Conal M Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. ACM, 25–36.

[12] Gabriel Gonzalez. 2012. Pipes. https://github.com/Gabriel439/Haskell-Pipes-Library.

[13] Jason Gregory. 2014. *Game Engine Architecture, Second Edition* (2nd ed.). A. K. Peters, Ltd., Natick, MA, USA.

[14] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[15] Joe Hermaszewski. 2016. vector-sized. https://github.com/expipiplus1/vector-sized.

[16] Justin Le. 2015. Auto. https://github.com/mstksg/auto.

[17] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. 2003. Polychrony for system design. *Journal of Circuits, Systems, and Computers* 12, 03 (2003), 261–303.

[18] Ben Lippmeier. 2010. Gloss. https://github.com/benl23x5/gloss.

[19] Thomas D. C. Little and Arif Ghafoor. 1990. Synchronization and storage models for multimedia objects. *IEEE journal on selected areas in communications* 8, 3 (1990), 413–427.

[20] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM*

SIGPLAN Workshop on Haskell (Haskell '02). ACM, 51–64.

[21] Gergely Patai. 2011. Efficient and Compositional Higher-order Streams. In *Proceedings of the 19th International Conference on Functional and Constraint Logic Programming (WFLP'10)*. 137–154.

[22] Ivan Perez. 2017. Back to the Future: Time Travel in FRP. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, 105–116.

[23] Ivan Perez. 2018. Fault Tolerant Functional Reactive Programming (Functional Pearl). *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 96 (July 2018), 30 pages.

[24] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, 33–44.

[25] Ivan Perez and Henrik Nilsson. 2015. Bridging the GUI Gap with Reactive Values and Relations. In *Haskell Symposium*. 47–58.

[26] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 2 (Aug. 2017), 27 pages.

[27] Atze van der Ploeg and Koen Claessen. 2015. Practical Principled FRP: Forget the Past, Change the Future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, 302–314.

[28] Basant Rajan and RK Shyamasundar. 2000. Multiclock ESTEREL: A reactive framework for asynchronous design. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE, 201–209.

[29] Schell Scivally. 2015. varying. https://github.com/schell/varying.

[30] Michael Snoyman. 2011. Conduit. https://github.com/snoyberg/conduit.

[31] Ertugrul Söylemez. 2016. Netwire. https://github.com/esoeylemez/netwire.

[32] Daniel Winograd-Cort and Paul Hudak. 2012. Wormholes: Introducing Effects to FRP. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, 91–104.