Property-Based Testing for Asynchronous Functional Reactive Programming Using Linear Temporal Logic

Christian Emil Nielsen, Mathias Faber Kristiansen, and Patrick Bahr

IT University of Copenhagen, Denmark
{cemn,matkr,paba}@itu.dk

Abstract. Functional reactive programming (FRP) is a programming paradigm for implementing software that continuously interacts with its environment and manipulates highly dynamic data. *Asynchronous* FRP, in particular, is very expressive and can be used to implement graphical user interfaces and other reactive systems interacting with data streams and events that are not synchronized. Testing such asynchronous FRP programs is difficult since a program's behaviour depends not only on the concrete data it receives from its environment but also the *relative timing* of when each piece of data arrives.
In this paper, we propose *PropRatt*, a property-based testing framework for asynchronous FRP. The key component of PropRatt is its specification language, which extends basic linear temporal logic with a means to express properties of several concurrent signals. This allows us to express temporal properties that relate data coming from different signals at different points in time. PropRatt is implemented in Haskell and targets a recently introduced asynchronous FRP language embedded in Haskell. We demonstrate the utility of PropRatt through a case study testing a signal combinator library as well as a graphical user interface, in which we suggest how the strategy for generating signals can be modified to better model specific domains.

Keywords: Property-based testing · Functional Reactive Programming · Linear Temporal Logic

1 Introduction

Reactive systems continuously respond to external inputs received from their environments. Examples of these include graphical user interfaces (GUIs) that react to a stream of mouse clicks and keyboard inputs, or control software for robots that continuously read input from hardware sensors. The key idea behind functional reactive programming (FRP) [7] is to model the input from the environment as a *signal* that can be manipulated as any other first-class value. This programming paradigm enables writing reactive systems in a declarative and compositional style. In the model of FRP we are considering here, signals are values of type `Sig a` that represent a stream of values of type `a` that arrive

time t_i	t_1	t_2	t_3	t_4	t_5	t_6	$\cdots$
$s_1 =$	1		2	3	4		$\cdots$
$s_2 =$	`'a'`	`'b'`	`'c'`			`'d'`	$\cdots$
`zip` s_1 $s_2 =$	(1, `'a'`)	(1, `'b'`)	(2, `'c'`)	(3, `'c'`)	(4, `'c'`)	(4, `'d'`)	$\cdots$

Fig. 1. Execution trace for `zip`.

incrementally over time. These signals can then be composed and manipulated using higher-order functions to build complex reactive systems. *Async Rattus* [1] is an asynchronous FRP language embedded in Haskell that enables this programming style. The asynchronous nature of the language allows for signals to update independently with respect to a *local* clock attached to each signal, as opposed to synchronous languages where all signals update according to a *global* clock. This model is favourable for GUIs, where subcomponents frequently need to be recomputed *independently* of most of the rest of the GUI. To illustrate such asynchronous behaviour, consider the `zip` function on signals in Async Rattus (instantiated to signals of integers and characters for the sake of this example):

```
zip :: Sig Int -> Sig Char -> Sig (Int :* Char)
```

A call to `zip` may produce the execution trace illustrated in Fig. 1. For each of the three signals, the figure shows a value whenever the signal updates. For example, at time t_5, the signal s_2 still has value `'c'`, while the other two signals have been updated to the values 4 and (4, `'c'`). The `zip` function combines two signals into a signal of (strict) pairs containing the two last observed values of the two signals. That means, it must update whenever either of the two argument signals updates. For example, at time t_5, only signal s_1 updates and s_2 remains unchanged. Accordingly, `zip` s_1 s_2 updates to the new value (4, `'c'`). As a sidenote, `zip` uses the strict product type `Int :* Char` rather than Haskell's standard lazy pair type `(Int, Char)`, because Async Rattus uses an eager evaluation semantics to make guarantees about the absence of space leaks.

From the trace in Fig. 1 we can see how the number of test cases explodes. Given two input signals s_1 and s_2, there are $3^5 = 243$ possible different outcomes for `zip` s_1 s_2 in the first 6 time steps just due to the different timings of s_1 and s_2! And that is before we even begin to consider the number of different values the signals can take. This combinatorial explosion renders conventional unit testing that uses known input/output pairs highly impractical for sufficient testing.

Property-based testing (PBT) [3] is a tool which may aid in addressing these challenges. Instead of writing test cases as input/output pairs, the user specifies a predicate that must hold *for all* input/output pairs. A testing framework then automatically checks the predicate against many randomly generated inputs, thereby testing a more general property of the system under test. This approach is well-suited for complex or combinatorial input spaces, as it can explore a broad range of inputs that would be impractical to write manually otherwise. To test a representative subset of execution paths, the generation of signals cannot be

completely arbitrary. Instead, it must be carefully constrained to ensure fairness for all signals under test, to avoid a signal never updating during a test run.

However, predicates written exclusively with propositional logic are insufficiently expressive to specify the expected behaviour of signals, as it can only represent a static notion of truth. The value produced by a signal at a given instant is of limited use, since it cannot express how signals evolve dynamically *over time*. This naturally leads us to temporal logic, which allows properties to be formulated with temporal operators. In particular, linear temporal logic (LTL) [16] provides operators such as **G** ("always") and **F** ("eventually") to express statements relative to time. For example, we can express the property that the first component of a zipped signal $s_3 = \texttt{zip}\, s_1\, s_2$ always reflects the current value of s_1 whenever s_1 produces a value:

$$\mathbf{G}(\checkmark s_1 \Rightarrow s_1 = \mathit{fst}(s_3)) \tag{1}$$

Here, $\checkmark s_1$ denotes a tick of the clock tied to s_1, capturing the idea that the property is only relevant when s_1 updates. Using the temporal operators offered by LTL allows for more expressive predicates and, in turn, properties of a system that we wish to test. Exactly how an LTL-based specification language should be designed to express specifications for *asynchronous* FRP has remained an open question so far.

In this paper, we present PropRatt[1], a PBT library for Async Rattus. PropRatt builds on top of QuickCheck [3], a PBT library for Haskell, to generate arbitrary signals and to check temporal properties. The key contributions of this work are as follows:

- We devise an LTL-based specification language to express temporal properties that involve multiple asynchronous signals.
- We implement an API that allows specifications to interact with several signals of heterogeneous types in a type-safe manner using Haskell's advanced type system features.
- We combine this API with an execution model that produces finite well-typed traces of parallel, asynchronous signals.
- We implement a testing harness that integrates the specification language and execution model into QuickCheck.
- We demonstrate the expressiveness of the specification language, its use, and its limitations in practice with a number of examples.

The remainder of this paper is structured as follows: In Sect. 2 we give a short introduction to the Async Rattus language and property-based testing. In Sect. 3 we present the LTL-based specification language, and in Sect. 4 we give an extended example of its use in PropRatt. In Sect. 5, we give an overview of the most important parts of the implementation of PropRatt and its specification language. Finally, we give an overview of related work in Sect. 6 and conclude in Sect. 7 with an outlook on future work.

[1] Available as a Hackage library, including all examples presented in this paper [12]

2 Background

In this section, we give a brief introduction to functional reactive programming in the Async Rattus language and property-based testing.

Async Rattus. Async Rattus is an FRP language embedded in Haskell that enables programming with asynchronous signals. It uses modal types to reflect the temporal availability of expressions and values at the type level. To this end, Async Rattus introduces two type modalities: a *later* modality `O` and a *box* modality `Box`. The later modality classifies data that is available at some future time, while the box modality classifies time-independent data that can be moved unchanged across time. By encoding timing constraints at the type level, Async Rattus guarantees that only time-appropriate values are accessed at each step. For example, a computation cannot accidentally use a value before it is received at some later time. But these modalities also aid resource management: The runtime only needs to keep the current values and the deferred computations awaiting future input, freeing any older data that is no longer needed, thus avoiding space leaks, which are notoriously difficult to debug. The core FRP abstraction provided by Async Rattus is the signal type. A signal of type `Sig a` represents a (possibly infinite) stream of values of type `a`. It is defined in Async Rattus as a recursive type:

```
data Sig a = a ::: O (Sig a)
```

Here we use `:::` as a cons-like constructor operator. A value of type `Sig a` carries a head of type `a` and a tail of type `O (Sig a)`. The head is available right away, while the tail is a delayed computation that becomes available in the next time step, according to some clock.

Each value `d :: O b` consist of a *clock* `cl(d)` and a delayed computation, which produces a value of type `b` whenever the clock `cl(d)` ticks. In the case of a signal, the tail is of type `O (Sig a)` and thus has an associated clock, which determines when a new value of type `Sig a` is available, i.e. when the signal updates. Concretely, a clock is a set of *channels*, and each such channel is an address from which the program can receive input. When an input arrives on one of those channels, we say that the associated clock ticks and the deferred computation is executed. In the case of the tail of a signal, such a tick yields the next value of the signal. Thus, each `O (Sig a)` is essentially a promise to compute the next signal element as soon as the clock ticks. For example, in a GUI application, each button has an associated channel that receives data whenever the button is pressed. If a signal's clock includes the channel associated with a button, then each button press causes the signal to advance by one step. This design lets different signals operate on independent clocks, unlike purely synchronous FRP where all signals share a single global clock.

Async Rattus provides two primitives for working with the later modality: `delay` and `adv` (advance). The `delay` primitive is the constructor for the later modality. It wraps a computation so that it does not execute until the signal's

clock ticks. Conversely, `adv` is the eliminator. It retrieves the value from a delayed computation when its clock allows it. The typing rules of Async Rattus ensures that every `adv` must be nested under the scope of a corresponding `delay`, preventing the use of values that do not exist. For instance, consider the following function, that subtracts one from a delayed integer:

```
subtractOne :: O Int -> Int
subtractOne laterN = laterN - 1
```

This function does not type check since `laterN :: O Int` is not an integer but rather a delayed integer that arrives at some point in the future. We cannot subtract `1` in the current time step. Instead, we have to await the clock of the integer and only then subtract `1`. A valid function could instead use the `delay` and `adv` primitives to delay the computation according to the clock of `laterN`.

```
subtractOne :: O Int -> O Int
subtractOne laterN = delay (adv laterN - 1)
```

In this way, we correctly await the input, before executing the computation, which also changes the type of the function so that it returns a *delayed* integer.

When we bind a variable `x` and use it in the scope of a `delay`, such as in `\x -> delay x`, we effectively move the data referenced by `x` into the future. Async Rattus requires that any value moved into the future be time-invariant. To this end, the language features the notion of *stable types*, which includes basic types like integers and recursive types like lists, but it excludes any types that contain the later modality or function types. Values of stable types are time-independent and consequently can be safely moved arbitrarily far into the future without risk of space leaks. To illustrate, consider a naive `map` implementation on signals:

```
leakyMap :: (a -> b) -> Sig a -> Sig b
```

This type would requite `leakyMap` to move the function of type `a -> b` arbitrarily far into the future in order to apply it to the signal at any time. However, function types are inherently not stable, as function closures may capture temporally-dependent data. Instead, `map` can be defined with the following type:

```
map :: Box (a -> b) -> Sig a -> Sig b
map f (x ::: xs) = unbox f x ::: delay (map f (adv xs))
```

The `Box` modality ensures that the function `f` is of a stable type and thereby temporally-invariant, which allows us to use `f` nested under a `delay`. To apply `f` as a function, we must first unbox it using `unbox :: Box a -> a`.

When constructing boxed values using the introduction form `box`, Async Rattus enforces a restriction similar to that for `delay`: We can only move values into a `box` if they are of a stable type. For example, `\x -> box x` only type checks if `x` has a stable type.

Property-Based Testing. Testing reactive and asynchronous programs written in Async Rattus is challenging because of the enormous space of possible event sequences. Property-based testing (PBT) offers a promising alternative to traditional testing. Instead of writing fixed test cases, the programmer states properties that the program should satisfy, and the testing framework checks them against many automatically generated inputs. This approach aligns well with the high-level nature of specifications and can uncover edge cases that handwritten examples might miss. PBT libraries allow the programmer to write compact specifications, such as the following specification that reversing a list twice produces the original list:

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

A PBT library such as *QuickCheck* can generate random inputs for the argument `xs` and check whether these satisfy the equation `reverse (reverse xs) == xs`. In addition, QuickCheck can also *shrink* such inputs in order to provide the programmer with useful counterexamples if a property fails. The core of this general principle is provided by the `Arbitrary` type class of QuickCheck:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

Given an instance of `Arbitrary [Int]`, QuickCheck can test properties that quantify over lists of integers such as `prop_rev` above: First `arbitrary` generates random values of type `[Int]`. Then it supplies these random values to the function `prop_rev` one by one. Finally, if the property fails for one of the generated inputs, say `xs`, then `shrink xs` will produce smaller lists with which to test `prop_rev` in order to obtain a smaller counterexample.

In the context of reactive systems, PBT can offer the same benefits: It can generate arbitrary input signals to explore the space of behaviours and shrink such signals to simpler counterexamples to help locate the source of errors. However, equational specifications alone are not sufficiently expressive to capture the complex behaviours expressible by asynchronous FRP programs.

3 Specification Language

3.1 Untyped Specification Language

PropRatt introduces a DSL for writing declarative specifications at a high level of abstraction, enabling property-based testing of FRP programs written in Async Rattus. Such programs manipulate signals of different types, e.g. `zip` from Sect. 1 combines an `Int` and a `Char` signal. Therefore, the design of the specification language must be able to account for multiple, heterogeneously typed signals. But before turning to the concrete syntax of the typed specification language as

$$
\begin{aligned}
\textit{Predicate } \varphi &::= T \mid F \mid e \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \\
&\quad \mid \mathbf{X}\,\phi \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi \mid \phi\ \mathbf{U}\ \psi \mid \phi\ \mathbf{R}\ \psi \\
\textit{Expression } e &::= c \mid e_1\ e_2 \mid l \mid \checkmark l \\
\textit{Lookup } l &::= \mathbf{prev}\ l \mid \mathbf{sig}_n
\end{aligned}
$$

Fig. 2. Syntax of PropRatt's specification language.

implemented in PropRatt, we consider an idealized syntax of an untyped version of the language in Fig. 2 in order to discuss the essential ideas of the language.

The specification language consists of three components: a logical language of predicates ϕ, a language of expressions e, and a notation l for looking up signals at a specific (relative) time. The predicate language is exactly LTL with atoms of the form e written in the expression language. For example, we write $\mathbf{G}(\mathbf{sig}_1 > 0)$ to express that the signal identified by $\mathbf{sig}_1$ always has a positive value, and we write $\mathbf{sig}_1 > 0\ \mathbf{U}\ \mathbf{sig}_1 = \mathbf{sig}_2$ to express that the signal $\mathbf{sig}_1$ is positive until both $\mathbf{sig}_1$ and another signal $\mathbf{sig}_2$ have the same value.

In these examples, $\mathbf{sig}_1 > 0$ and $\mathbf{sig}_1 = \mathbf{sig}_2$ are Boolean-valued expressions, which can act as atoms in an LTL formula. Expressions combine constants (denoted c in Fig. 2), such as 0 and $>$, and signal lookups, such as $\mathbf{sig}_1$, using function application (denoted $e_1\ e_2$ in Fig. 2). In the case of operator constants such as $>$, we allow the infix notation $\mathbf{sig}_1 > 0$ instead of the prefix notation $> \mathbf{sig}_1\ 0$. In general, an LTL formula is a predicate over n signals whose current value can be referenced using $\mathbf{sig}_1, \ldots, \mathbf{sig}_n$. We can also reference a previous value of a signal by using **prev**. For example, $\mathbf{G}(\mathbf{X}(\mathbf{prev}\ \mathbf{sig}_1 < \mathbf{sig}_1))$ stipulates that signal $\mathbf{sig}_1$ is strictly monotonically increasing.

Traditionally, LTL works only on a *single* execution trace, whereas PropRatt allows multiple parallel signals, which may produce new values independently of one another according to their own clocks. However, given such parallel, asynchronous signals, we can construct a single execution trace that maintains the relative timing information. We have seen an example of that in the execution trace for `zip`, shown in Fig. 1. It involves three signals, which, if combined, produce a trace where at each time step t_i at least one of the three signals ticks, i.e. produces a new value.

In the expression language, we have access to the timing information of such a combined trace via expressions of the form $\checkmark s$, which are true whenever the signal identified by s has ticked. Returning to the `zip` example from Fig. 1, we have that at time t_5, the expressions $\checkmark\mathbf{sig}_1$ and $\checkmark\mathbf{sig}_3$ are true, since both s_1 and `zip` s_1 s_2 produced a new value, whereas $\checkmark\mathbf{sig}_2$ is false as s_2 did not produce a new value. Using this more formal notation for the specification language, specification (1) is written more precisely as $\mathbf{G}(\checkmark\mathbf{sig}_1 \Rightarrow \mathbf{sig}_1 = \texttt{fst'}\ \mathbf{sig}_3)$, where `fst' :: a :* b -> a` is the first projection of the strict pair type.

To further illustrate the specification language, suppose we have a GUI with the following signals:

```
timer :: Sig Int      -- time to display
```

```
reset :: Sig ()       -- reset button
```

The `timer` signal is meant to increment by one every second and `reset` produces a unit value `()` every time the 'reset' button in the GUI is pressed. Assuming that the signals we wish to test are supplied in the order they appear above, i.e. $\mathbf{sig}_1 = \texttt{timer}$ and $\mathbf{sig}_2 = \texttt{reset}$, we can specify that `timer` increments whenever the clock of `timer` ticks and the reset button is not pressed:

$$\mathbf{G}(\mathbf{X}(\checkmark\mathbf{sig}_1 \wedge \neg\checkmark\mathbf{sig}_2 \Rightarrow \mathbf{prev}\ \mathbf{sig}_1 < \mathbf{sig}_1)) \tag{2}$$

$\mathbf{G}$ and $\mathbf{X}$ are standard LTL temporal operators. The *expression* $\checkmark\mathbf{sig}_1$ checks whether the first signal has updated in the current time step, while $\mathbf{sig}_1$ retrieves the value of that signal. The *lookup* $\mathbf{prev}\ \mathbf{sig}_1$ accesses the value of the timer from the previous time step. The property asserts that at any time ($\mathbf{G}$), whenever a time step passes ($\mathbf{X}$) where the timer signal ticks ($\checkmark\mathbf{sig}_1$) but 'reset' has not been pressed ($\neg\checkmark\mathbf{sig}_2$), then the timer ($\mathbf{sig}_1$) must have increased relative to its previous value ($\mathbf{prev}\ \mathbf{sig}_1$).

Note that the specification language features the LTL connective $\mathbf{F}$, which expresses a *liveness* property. Since liveness properties do not have finite counterexamples, testing them will always succeed. However, when liveness properties occur in a negative position in a larger specification, i.e. in negated form or to the left of $\Rightarrow$, they become safety properties (e.g. $\neg(\mathbf{F}\phi)$ is equivalent to $\mathbf{G}(\neg\phi)$). Thus, such connectives are still useful for writing compositional specification.

3.2 Typed Specification Language

The syntax in Fig. 2 is untyped and to ensure well-typedness we have to give a type system so that only Boolean-valued expressions are used as atoms, and expressions themselves are also well-typed, e.g. $e_1\,e_2$ is only well-typed if e_1 is of type $\tau_1 \to \tau_2$ and e_2 is of type τ_1. Instead of presenting such a type system, we give the encoding of this type system in Haskell.

The concrete syntax of PropRatt's specification language is intrinsically well-typed and uses generalized algebraic data types (GADTs) to represent the typing information. The full definition of the syntax is given in Fig. 3. The type of predicates `Pred` is indexed by a list of type `ts :: [Type]`, which represents the types of the signals over which we want to specify a property. For example, for the property involving the `timer :: Sig Int` and `reset :: Sig ()` signals, this list of types is `'[Int,()]`.[2]

Similarly, expressions are indexed by such a list of types `ts :: [Type]` as well. But they also have an additional type index `t :: Type` that indicates the type of values produced when evaluating such expressions. Using the `Now` constructor, predicates over signal types `ts` may include expressions of type `Bool` over signal types `ts`. In turn, expressions have access to the values and timing information of the signals via `Val` and `Tick`, respectively. In addition,

[2] The quote notation `'` distinguishes a type-level list `'[Int]` from the list type `[Int]`. Similarly, `':` denotes the type-level list cons operation in Fig. 3.

```
data Pred (ts :: [Type]) where
  TT, FF               :: Pred ts
  Now                  :: Expr ts Bool -> Pred ts
  Not, X, G, F         :: Pred ts -> Pred ts
  And, Or, (:=>), U, R :: Pred ts -> Pred ts -> Pred ts

data Expr (ts :: [Type]) (t :: Type) where
  Pure  :: t -> Expr ts t
  App   :: Expr ts (t -> r) -> Expr ts t -> Expr ts r
  Val   :: Lookup ts t -> Expr ts t
  Tick  :: Lookup ts t -> Expr ts Bool

data Lookup (ts :: [Type]) (t :: Type) where
  Prev  :: Lookup ts t -> Lookup ts t
  Sig1  :: Lookup (Value t ': x) t
  Sig2  :: Lookup (x1 ': Value t ': x2) t
        ⋮
```

Fig. 3. Well-typed syntax of PropRatt.

they also have an applicative functor structure provided by `Pure` and `Apply`. These two constructors correspond directly to the two operations `pure` and `<*>` of the applicative functor interface. For example the expression $\mathbf{prev}\,\mathbf{sig}_1 < \mathbf{sig}_1$ from the timer specification (2) above, is written

```
(<) <$> Val (Prev Sig1) <*> Val Sig1
```

where `<$>` is the map operator of applicative functors, defined by `f <$> x = pure f <*> x`. The above expression is of type `Expr '[Int,()] Bool` and thus can be used in a predicate using `Now`. The type of the `Sig1` constructor ensures that the type of the expression `Val Sig1` matches exactly the first signal, and likewise for the remaining constructors `Sig2`, `Sig3`, etc.

Putting all of this together, specification (2) can be expressed as follows:

```
prop :: Pred `[Int,()]
prop = G (X ((Now (Tick Sig1) `And` Not (Now (Tick Sig2)))
         :=> Now ((<) <$> Val (Prev Sig1) <*> Val Sig1)))
```

To make properties more readable, the specification language also provides shorthands, defined in Fig. 4. This includes shorthands for looking up the current value of a signal (`sig1`, `sig2`, etc.), checking whether a signal has ticked (`tick1`, `tick2`, etc.), `Prev` generalized to the expression language (`prev`), common arithmetic operators for the expression language, and common comparison operators for the predicate language (`|<|`,`|==|`, etc.). With these shorthands the above property `prop` can be written more concisely:

```
prop :: Pred `[Int,()]
prop = G (X ((tick1 `And` Not tick2) :=> prev sig1 |<| sig1))
```

```
sig1 :: Expr (Value t ': ts) t
sig1 = Val Sig1

tick1 :: Pred (Value t ': ts)
tick1 = Now (Tick Sig1)

-- sig2, tick2, etc. are defined similarly

prev :: Expr ts t -> Expr ts t
prev (Pure x)   = Pure x
prev (App f x)  = App (prev f) (prev x)
prev (Val lu)   = Val (Prev lu)
prev (Tick lu)  = Tick (Prev lu)

instance Num t => Num (Expr ts t) where
  (+) x y = (+) <$> x <*> y
  -- similar definitions for (-), (*), etc.

(|<|) :: Ord t => Expr ts t -> Expr ts t -> Pred ts
x |<| y = Now ((<) <$> x <*> y)
-- similar definitions for comparison operators (==), (<=), etc.
```

Fig. 4. Shorthand notations for the specification language.

4 Case Study

To evaluate the expressiveness of the specification language, we have compiled two sets of example specifications: A set of specifications for the signal combinators provided by the Async Rattus library and a set of specifications for a simple timer GUI. Here we give a selection of these specifications, but the complete set can be found in the accompanying Haskell package [12].

Signal Combinators. The first set of examples comprises specifications for signal combinators like `zip` and `map`. We have seen one such specification in the form of the LTL formula $\mathbf{G}(\checkmark \mathbf{sig}_1 \Rightarrow \mathbf{sig}_1 = \texttt{fst'}\,\mathbf{sig}_3)$ for the `zip` combinator. This can be written in the typed specification language as follows:

```
-- sig3 = zip sig1 sig2
prop_zip :: Pred '[Int, Char, (Int :* Char)]
prop_zip = G (tick1 :=> (sig1 |==| (fst' <$> sig3)))
```

This property is able to catch the following wrong implementation of `zip`:

```
zipWrong :: (Stable a, Stable b) => Sig a -> Sig b -> Sig (a :* b)
zipWrong (a ::: as) (b ::: bs) = (a :* b) ::: delay (
  case select as bs of
    Fst (a' ::: as') bs' -> zipWrong (a' ::: as') (b ::: bs')
    Snd as' (b' ::: bs') -> zipWrong (a  ::: as') (b ::: bs')
    Both as' bs'         -> zipWrong as' bs')
```

In the `Snd` case, this definition uses the value `b` (rather than `b'`) in the recursive call to `zipWrong`. This is incorrect according to the specification, given that the value is now stale in the presence of the newly produced value `b'`. Upon evaluating the property using QuickCheck the test indeed fails:

```
*** Failed! Falsified (after 2 tests and 14 shrinks):
[!(0 :* 0); !0; !0;
,!(0 :* 0); _0; !1;]
```

Each row of the above counterexample represents a time step, and each column represents a signal with its current value and whether it ticked (`!`) or not (`_`).

Async Rattus can express dynamic dataflows using the `switch` combinator, which takes a signal `s1` and a delayed signal `s2` and produces a new signal that first behaves like `s1` and behaves like `s2` as soon as that delayed signal `s2` arrives. We can express this property as follows:

```
-- sig3 = switch sig1 sig2
prop_switch :: Pred '[Int, Int, Int]
prop_switch = (sig3 |==| sig1) `U` (tick2 `And` G(sig3 |==| sig2))
```

Note that PropRatt's until operator `U` has the semantics of the 'weak until' operator (often denoted $\mathbf{W}$). In an LTL formula $\phi \,\mathbf{U}\, \psi$, the 'strict until' operator requires ψ to become true at some point in the future (and ϕ to be true at all times before that). However, this requirement of ψ becoming true at some point is a liveness property, which cannot be tested using PBT since such properties have only infinite counterexamples. Therefore, by necessity, PropRatt adopts the weak form which does not require ψ to become true. This 'weak until' semantics of `U` matches the semantics of `switch` because the second (delayed) signal passed to `switch` may never arrive.

GUI Application. For the second set of example specifications, we consider a GUI application that extends the timer example from Sect. 3.1 to include additional signals. This GUI application is taken from Disch et al. [6] and implements the timer of Kiss's 7GUIs benchmark [11].

The timer application displays a value of elapsed time, a reset button, and a slider for adjusting a maximum duration. Users may interact with the timer in two ways: pressing the reset button and adjusting the slider which changes the maximum value the timer may reach. Once the timer reaches this maximum, it stops incrementing until the slider is moved again or the reset button is pressed. The GUI processes three inputs: a reset button (`reset :: Sig ()`), a slider to adjust the maximum of the timer (`max :: Sig Int`), and a signal that ticks every second (`sec :: Sig ()`). These three signals are combined to form the signal of type `state:: Sig (Int :* Int)`, which captures the state of the GUI. Its first component holds the current timer value and its second component is the maximum value currently chosen by the user.

Kiss [11] provides a specification of the timer application consisting of several informal properties, which we can now formally express in the PropRatt speci-

fication language. First, the timer must stop whenever it reaches its maximum. That is, the timer value never exceeds the maximum value supplied by the user:

```
-- sig1: state; sig2: reset; sig3: max; sig4: sec
prop1 :: Pred '[(Int :* Int), (), Int, ()]
prop1 = G ((fst' <$> sig1) |<=| sig3)
```

Second, if the timer has not yet reached its maximum, then it must increase every second:

```
prop2 :: Pred '[(Int :* Int), (), Int, ()]
prop2 = G ( ((fst' <$> sig1) |<| sig3) `And` X (Not tick2 `And` tick4)
    :=> X ((fst' <$> sig1) |==| ((+1) . fst' <$> prev sig1)))
```

Third, whenever the user presses the reset button, then in that same time step the timer value must be 0:

```
prop3 :: Pred '[(Int :* Int), (), Int, ()]
prop3 = G (tick2 :=> (pure 0 |==| (fst' <$> sig1)))
```

Finally, implicit in the specification is the property that the timer remains constant unless a second has passed or the reset button was pressed:

```
prop4 :: Pred '[(Int :* Int), (), Int, ()]
prop4 = G (X ((Not tick2 `And` Not tick4)
          :=> ((fst' <$> prev sig1) |==| (fst' <$> sig1))))
```

Testing these properties confirms that the implementation of the timer GUI by Disch et al. [6] does indeed satisfy the specification.

5 Implementation

In Sect. 3, we introduced the key type `Pred`, which defines well-typed temporal specifications. We now turn to the implementation details that connect these specifications to property-based testing with QuickCheck [3].

A specification is tested using the following function:

```
evaluate :: Pred ts -> Sig (HList ts) -> Bool
```

The type `Sig (HList ts)` represents a flattened trace of a program containing data from all signals under test such as the example trace for `zip` in Fig. 1. Understanding this type requires a short detour into how asynchronous signal behaviour is represented. A naive approach might model a trace as a list of independent signals `[Sig a]`. This representation is insufficient as it does not allow us to inspect the timing relationships between signals directly. Instead, we want a *single* signal of lists, where the nth value in a list corresponds to the value produced by the nth signal of the property we are testing. In order to also represent cases where some of the signals do not produce a new value, we can

represent such a single signal with the type `Sig [Maybe a]`. This representation is essentially the idea of the type `Sig (HList ts)` used by `evaluate` above, but instead of a list it uses a heterogeneous list type (`HList`) that allows different signals to have different types:

```
data HList :: [Type] -> Type where
  HNil :: HList '[]
  (:%) :: !x -> !(HList xs) -> HList (x ': xs)
```

Users can construct a heterogeneous list of signals to test:

```
-- assuming s1 :: Sig Int, s2 :: Sig Char
sigsOfZip :: HList '[Sig Int, Sig Char, Sig (Int :* Char)]
sigsOfZip = s1 :% s2 :% zip s1 s2 :% HNil
```

Such a heterogeneous list of signals is then flattened into a single signal that can then be passed to `evaluate`:

```
traceOfZip :: Sig (HList '[Value Int,Value Char,Value (Int:*Char)])
traceOfZip = flatten sigsOfZip
```

Before looking more closely at flattening, we take a look at the `Value` type.

To support operators such as ✓ and **prev**, the trace must maintain past values and indicate whether a signal has emitted a fresh value at the current time step. We collect this data in the `Value` type:

```
newtype HasTicked = HasTicked Bool deriving Show
data    Value a   = Current !HasTicked !(List a)
```

Recall the definition of the `Lookup` type in Fig. 3 that represents lookups of signal values. The type of each `Sig`n constructor enforces that the nth type in the list of types `ts` is of the form `Value t` in the type signature of `evaluate`.

To illustrate this representation, we show the trace from Fig. 1 as a value of type `Sig (HList '[Value Int, Value Char, Value (Int :* Char)])`:

	s_1	s_2	`zip` s_1 s_2
t_0	`(T,[1])`	`:% (T,['a'])`	`:% (T,[(1,'a')]) :% HNil`
t_1	`(F,[1])`	`:% (T,['b','a'])`	`:% (T,[(1,'b'),(1,'a')]) :% HNil`
t_2	`(T,[2, 1])`	`:% (T,['c','b','a'])`	`:% (T,[(2,'c'),(1,'b'),(1,'a')]) :% HNil`
⋮	⋮		

At t_0, both signals produce a new value, whereas at t_1, only the second signal emits a new value while the first carries over its previous one, which is also indicated by the false Boolean flag at t_1.

Flattening of Signals to Traces. To construct traces from individual signals, we define the function `prepend`, which merges a signal `Sig t` with an already-flattened signal of heterogeneous list `Sig (HList ts)`:

```
prepend :: (Stable t, Stable (HList ts), Falsify ts)
    => Sig t -> Sig (HList ts) -> Sig (HList (Value t ': ts))
```

The key idea is that `prepend` *unions* the clocks of the delayed computations from both arguments, which means the resulting signal updates whenever either of the input signals would update. This enables us to explore traces systematically. The `Stable` constraints ensures that the values produced by either signal argument are temporally-independent and safe to move to the future according to the type system of Async Rattus, and `Falsify` is a type class that implements a single method for a `Value` to negate the `HasTicked` flag. To flatten a heterogeneous list of signals to a trace, we recursively apply `prepend` through the `flatten` method defined in the multi-parameter type class `Flatten`:

```
class Stable (HList vals) =>
  Flatten sigs vals | sigs -> vals, vals -> sigs where
    flatten :: HList sigs -> Sig (HList vals)

instance Flatten '[] '[] where
  flatten HNil = emptySig

instance (Stable a, Stable (Value a), Flatten as bs, Falsify bs)
  => Flatten (Sig a ': as) (Value a ': bs) where
    flatten (HCons h t) = prepend h (flatten t)
```

We use a type class to implement `flatten` so that we can express the relation between the argument type `HList sigs` and return type `Sig (HList vals)`. Namely, `sigs` is a list of types of the form `Sig t`, whereas `vals` is the corresponding list of types of the form `Value t`. The type class declaration uses functional dependency annotations to make explicit that `sigs` and `vals` are in a one-to-one correspondence, which helps the type checker to infer types.

Generating Signals. Recall from Sect. 2 that the clock of a delayed computation is represented by a set of channels. For example, the clock $\{\kappa_1, \kappa_2\}$ indicates that it will tick if data arrives on channel κ_1 or channel κ_2. Clocks such as $\{\kappa_1, \kappa_2\}$ and $\{\kappa_2, \kappa_3\}$ may tick simultaneously if data has arrived on a channel that both clocks share, in this case κ_2. Concretely, channels are simply addresses that are represented as integers in Async Rattus, so that clocks are represented using the type `IntSet` of finite sets of integers.

At each step, a signal consists of a current value and a delayed computation that produces the future state of that signal. Hence, in order to generate a signal, PropRatt must generate the clocks of these delayed computations. In turn, the generated clocks need to adequately reflect the asynchronous interaction of multiple signals. To this end, we randomly assign each delayed computation of a generated signal a clock drawn from the non-empty subsets of $\{\kappa_1, \kappa_2, \kappa_3\}$:

```
genClock :: Gen Clock
```

```
genClock = do
  n <- chooseInt (1, 3)
  case n of
    1 -> do x <- chooseInt (1,3)
            return (IntSet.fromList [x])
    2 -> elements $ map IntSet.fromList [[1,2], [2,3], [1,3]]
    3 -> return (IntSet.fromList [1,2,3])
```

After each step, a signal has a new delayed computation, and thus the clock of a signal may dynamically change over time. Moreover, by drawing from a set of three channels, we are able to represent complex asynchronous behaviour where different signals may have identical, disjoint, or overlapping clocks.

To simulate the behaviour of signals producing values asynchronously, we must apply a strategy that at each time step choses a channel κ on which the program has received input and which therefore causes the relevant clocks to tick (namely all clocks θ with $\kappa \in \theta$). This strategy can be deterministic, because the clocks themselves have been randomly generated. Moreover, since PropRatt flattens all signals to a single *trace* signal, we only have to pick a channel from the clock of that single signal. We make use of the fact that channels are represented as integers and simply pick the channel with the smallest integer representation.

In short, this approach allows us to generate random input signals that mimic asynchronous external interaction, combine them with user-supplied signals if desired, and integrate the resulting outputs into a single trace of our program. This single trace is then deterministically traversed by choosing the smallest channel for each step. As a consequence, tests are able to explore different values of the randomly generated inputs as well as different timings of these inputs.

Shrinking. We supply implementations of the `shrink` method given by the `Arbitrary` type class from QuickCheck. We implemented a shrinker for signals by converting signals into a list that preserves the clocks of delayed computations:

```
type TSig a = [(a, Clock)]
```

This representation allows us to shrink signals using a strategy similar to that for lists implemented in QuickCheck: Shrink the signal by iteratively dropping contiguous chunks of values and shrink the remaining values contained in the signal themselves as well. After producing new shrink candidates, we rebuild them as signals and reapply the saved clocks so the delayed-computation strategy is preserved. QuickCheck evaluates shrink candidates iteratively until the property no longer fails. This yields compact failing inputs and correspondingly short traces that help pinpointing errors in the implementation or the specification.

6 Related Work

LTL has long been recognized as a suitable specification language for reactive programs. Jeffrey [9] and Jeltsch [10] independently discovered that LTL can

be seen as a type system for functional reactive programs. Later, Perez and Nilsson [14,15] used LTL as a specification language for property-based testing of functional reactive programs. LTL-based specification languages have also been used for property-based testing of web applications [13]. The work most closely related to PropRatt is the property-based testing library developed for Rattus [5], a synchronous version of Async Rattus.

In all previous work mentioned above, the LTL-based specification language always targets *single*, *synchronous* executions of programs. By contrast, PropRatt specifications are *hyperproperties*, i.e. properties over several parallel execution traces of signals. Variants of LTL for hyperproperties have been suggested [4], but we are not aware of any property-based testing framework that uses these ideas, apart from the thesis of Nielsen and Kristiansen [2] which this paper summarizes. However, we are not the first to devise a specification language for PBT of asynchronous reactive systems. Hughes et al. [8] have proposed *temporal relations* as a specification language for testing the asynchronous behaviour of a communication protocol. A temporal relation is a relation between time intervals and values. Such relations provide a compositional way to specify properties of asynchronous computations.

Sculthorpe and Nilsson [17] embedded an FRP language in the dependently-typed language Agda, which allows safety properties to be formally *proved*. In later work, Sculthorpe and Nilsson [18] devised an LTL-like, shallowly embedded logic in Agda to express and *prove* temporal specifications of FRP programs.

7 Conclusion and Future Work

The focus of the present work is to explore the expressiveness of asynchronous LTL specifications and to demonstrate it with case studies. This leaves considerations of ergonomics of the specification language for future work. Such considerations are important, because the ease with witch specifications can be written, read, and modified do significantly contribute to the utility of a PBT framework. The design of PropRatt as a hybrid DSL with a deeply embedded core allows for further improvements to the specification language. For example, by inspecting the AST of specifications, we can produce a warning when the user tries to test liveness properties, which by their nature do not have finite counterexamples and thus cannot be tested.

In addition, we could extend the language with further connectives. For example, the predicate fragment of the specification language has combinators that move forward in time (such as **X** and **U**), whereas the expression fragment has a combinator that moves backwards in time (namely **prev**). While this design decision simplifies the implementation of an efficient checking procedure for specifications, it can lead to unnatural or inelegant specifications. However, it is possible to extend the specification language with combinators that allow e.g. the expression fragment to also move forward in time. Specifications written in this richer specification language can then be translated into an equivalent specification in the simpler specification language presented in this paper.

References

1. Bahr, P., Houlborg, E., Rørdam, G.T.S.: Asynchronous Reactive Programming with Modal Types in Haskell. In: Gebser, M., Sergey, I. (eds.) Practical Aspects of Declarative Languages. pp. 18–36. Springer Nature Switzerland (2024). https://doi.org/10.1007/978-3-031-52038-9_2
2. Christian Emil Nielsen, Mathias Faber Kristiansen: Property-Based Testing for Functional Reactive Programming in Async Rattus using Linear Temporal Logic. Master's thesis, IT University of Copenhagen (2025)
3. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266
4. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal Logics for Hyperproperties. In: Abadi, M., Kremer, S. (eds.) Principles of Security and Trust. pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15
5. Dannebrog Jensen, L.: Property Based Testing of Functional Reactive Programs using Linear Temporal Logic. Master's thesis, IT University of Copenhagen (2023)
6. Disch, J.C., Heegaard, A., Bahr, P.: Functional reactive GUI programming with modal types. In: Gibbons, J. (ed.) Trends in Functional Programming. pp. 93–114. Springer Nature Switzerland (2026). https://doi.org/10.1007/978-3-031-99751-8_5
7. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the second ACM SIGPLAN international conference on Functional programming. pp. 263–273. ICFP '97, Association for Computing Machinery (Aug 1997). https://doi.org/10.1145/258948.258973, 00000
8. Hughes, J., Norell, U., Sautret, J.: Using temporal relations to specify and test an instant messaging server. In: Proceedings of the 5th Workshop on Automation of Software Test. pp. 95–102. AST '10, Association for Computing Machinery (May 2010). https://doi.org/10.1145/1808266.1808281
9. Jeffrey, A.: LTL Types FRP: Linear-time Temporal Logic Propositions As Types, Proofs As Functional Reactive Programs. In: Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification. pp. 49–60. PLPV '12, ACM (2012). https://doi.org/10.1145/2103776.2103783
10. Jeltsch, W.: Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming. Electronic Notes in Theoretical Computer Science **286**, 229–242 (Sep 2012). https://doi.org/10.1016/j.entcs.2012.08.015
11. Kiss, E.: 7GUIs: A GUI programming benchmark (2014), https://eugenkiss.github.io/7guis/
12. Nielsen, C.E., Kristiansen, M.F., Bahr, P.: `PropRatt` Haskell library package. https://hackage.haskell.org/package/PropRatt (2025)
13. O'Connor, L., Wickström, O.: Quickstrom: property-based acceptance testing with LTL specifications. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 1025–1038. PLDI 2022, Association for Computing Machinery (2022). https://doi.org/10.1145/3519939.3523728
14. Perez, I., Nilsson, H.: Testing and debugging functional reactive programming. Proceedings of the ACM on Programming Languages **1**(ICFP), 1–27 (Aug 2017). https://doi.org/10.1145/3110246

15. Perez, I., Nilsson, H.: Runtime verification and validation of functional reactive systems. Journal of Functional Programming **30** (2020). https://doi.org/10.1017/S0956796820000210
16. Pnueli, A.: The Temporal Logic of Programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. SFCS '77, IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
17. Sculthorpe, N., Nilsson, H.: Safe Functional Reactive Programming Through Dependent Types. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 23–34. ICFP '09, ACM (2009). https://doi.org/10.1145/1596550.1596558, 00039 event-place: Edinburgh, Scotland
18. Sculthorpe, N., Nilsson, H.: Keeping calm in the face of change. Higher-Order and Symbolic Computation **23**(2), 227–271 (Jun 2010). https://doi.org/10.1007/s10990-011-9068-x