Property-Based Testing for Functional Reactive Programming in Async Rattus using Linear Temporal Logic

Thesis – KISPECI1SE

Christian Emil Nielsen, Mathias Faber Kristiansen

{cemn,matkr}@itu.dk

Supervisor: Patrick Bahr

June 2025

Abstract

Bugs in software are inevitable, and testing is an important tool to determine if programs satisfies their specification. Reactive programs are programs that react to input from their environment, such as graphical user interfaces. Async Rattus, a domain specific language embedded in Haskell, is a functional reactive programming language for building asynchronous reactive programs. The language introduces a signal data structure, for modeling asynchronous streams of time-varying values. These pose challenges in testing due to the number of possible sequences in which signals can produce values when combined in parallel, thereby rendering example based testing impractical. Property-based testing, where predicates are tested on arbitrarily generated inputs, may be useful in solving these challenges. Moreover, reasoning about signals requires a specification language that can express temporal behaviour of signals. There are currently no testing libraries that suggest a practical approach to test Async Rattus programs. To this end, we present PropRatt, a proof-of-concept property-based testing library. The library provides: (1) a specification language based on linear temporal logic for expressing temporal predicates; (2) a type to model Async Rattus program executions over time; and (3) arbitrary signal generators. QuickCheck, a property based testing library for Haskell, is leveraged to combine these components to form and check properties. We demonstrate the utility of PropRatt through a case study of testing a graphical user interface, in which we suggest how the strategy for generating signals can be modified to better model specific domains. Finally, we discuss limitations such as the inability of testing certain properties in finite time.

Contents

1 Introduction

This section presents the motivation for this project, as well as it outlines the scope and delimitations.

1.1 Motivation

Testing is an important tool to determine if systems live up to their specification. This is especially important in safety critical systems, where the risk of system failures is unacceptable. One class of programs that are relevant to test, are reactive programs. These programs are designed to react upon inputs from their environment during execution, such as Graphical user interface (GUI). Functional reactive programming (FRP) is a paradigm for writing reactive systems declaratively, by treating time-varying values as first-class entities. By modeling the input as streams of time-varying values, FRP provides an intuitive abstraction for reactive programming.

Async Rattus is a FRP language that presents these streams as *signals*. In contrast to synchronous FRP systems, where all signals advance in lockstep, signals in Async Rattus produce values independently of each other, which resembles asynchronous behavior [1]. This aspect of the language introduces complexity in reasoning about program correctness, as it requires a model of how multiple signals produce values in parallel over time.

An effective approach to verifying correctness of Async Rattus programs must therefore:

- Incorporate a specification language that can reason about signals using temporal predicates.
- Model a potentially-infinite execution in fixed, bounded time, such that a representative subset of possible program interleavings can be tested.

1.2 Scope and delimitations

The work presented in this thesis explores the feasibility of property-based testing for Async Rattus. Our focus is on testing signals that evolve asynchronously when combined with other signals. Specifically, we aim to test the combinator functions of the `AsyncRattus.Signal` module, by reasoning about the correctness of inputs compared to outputs for these [2]. Some of these combinators return signals wrapped in computational contexts (such as the `IO` type). We deliberately exclude all these signal functions from this testing library at its current point of implementation.

2 Background

The following sections highlight key concepts and domains that underpin this thesis.

2.1 Functional Reactive Programming

FRP is a programming model that facilities writing reactive programs in a functional style. Reactive programs are programs that must react to external input from their environment. Examples of these include GUI and programs that's must react to hardware sensor readings. The key concept behind FRP is to model said input as data streams which can then be composed using declarative combinators from the functional programming paradigm. Given that any changes in state propagate predictably through the system, the resulting programs are easier to reason about.

Early approaches to FRP distinguished between two kinds of data streams, namely `behaviours` and `events`. The former modelling continous time-varying values and the latter modelling events as a sequence of event occurrences. Conceptually, a behaviour can be thought of as a function from time to values. Historically, an efficient implementation of behaviours has proven to be difficult. A naive implementation would require a behaviour to retain all values from the past, leading to implicit space leaks [3]. Moreover, continuous semantics demand frequent sampling of signals, incurring unnecessary computational costs. To address these limitations, many modern FRP systems adopt a discrete-time model. In such systems, updates to values are triggered exclusively by discrete events, enabling computations to be deferred until change is required [4].

More recent FRP implementations have unified behaviours and events into a new abstraction called a signal, which encapsulates both current state and the potential for discrete updates over time [4]. Signals simplify the model by always maintaining a current value that may change in response to events, effectively merging the continuous and discrete semantics into a single construct. We have described how continuously sampling a value of a signal can be inefficient. This is also called a pull-based (demand-driven) system. Instead, the use of signals support a push-based (data-driven), wherein values are only recomputed on discrete events. The choice between push- and pull-based evaluation strategies significantly impacts the performance and responsiveness of FRP systems. The pull-based model, while conceptually aligned with functional programming's lazy semantics, can be inefficient in reactive contexts that require constant monitoring of dynamic inputs [5].

Recent research has explored the use of type systems to express and enforce temporal constraints. In particular, modal type operators have been introduced to annotate values with temporal information, enabling the compiler to statically verify properties such as causality, productivity, and memory safety [1]. These techniques allow FRP implementations to maintain strong operational

guarantees without sacrificing expressiveness or introducing significant syntactic complexity [3].

2.2 Property-Based Testing

Property-based testing (PBT) is a testing methodology in which one formulates logical properties that are tested against arbitrarily generated inputs. The properties are a higher-level abstraction compared to example-based testing, allowing them to align with the specification of the program. This methodology is especially useful when working with units under test that contain complex input domains, creating a combinatorial problem that is impractical to craft test input manually.

When a test fails, the PBT framework initiates a process known as *shrinking*. Shrinking refers to the systematic reduction of the failing input to simpler forms, based on a predefined strategy that is suitable for the input type. The goal is to identify the minimal input that still reproduces the failure, referred to as the smallest possible counterexample. Having small counterexamples helps the developer in finding the underlying issue by pinpointing the exact input that causes the test to fail. Note that the definition of "small" is type-dependent. For instance, in the case of a list of numbers, smaller could refer to a shorter list, but it may also involve replacing values with smaller values according to their natural ordering.

2.2.1 QuickCheck

QuickCheck is a testing library for Haskell that implements PBT [6]. At its core, QuickCheck takes user defined properties and tests these over a series of arbitrarily generated inputs. For example, consider the property of reversing a list twice, which should be equal to the original list:

```haskell
prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse (reverse xs) == xs
```

When executing the test, QuickCheck generates 100 arbitrary `[Int]` inputs, and the framework checks that the property defined holds for all these cases. To generate and shrink inputs, QuickCheck uses the `Arbitrary` typeclass:

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

The arbitrary function returns a `Gen a` type to generate arbitrary values of type `a`. The `Gen` type implements the `Monad`, `Applicative`, and `Functor` typeclasses, which makes it easy to compose simple generators into more complex ones. This composability allows users to define generators that are not only syntactically convenient, but also encoding domain-specific invariants of the type

in the generation and shrinking implementation. For instance, to generate a pair of a bounded integer and a corresponding list of that many elements:

```haskell
genBoundedList :: Gen (Int, [Bool])
genBoundedList = do
  n <- (choose (1, 10) :: Gen Int)
  bs <- (vectorOf n arbitrary :: Gen Bool)
  return (n, bs)
```

Using Haskell `do` notation provides a convenient and succinct syntax to compose multiple dependent computations in a sequential, easy to read manner.

2.3 Async Rattus

Async Rattus is an Embedded domain specific language (eDSL) for FRP within Haskell. The languages uses modal types to express and enforce when computations occur [1]. Its type system tracks temporal behavior by assigning modalities that distinguish immediate values from those arriving at future time steps. This system introduces, for example, the "later" modality (denoted as $\bigcirc$) to defer computations to future timesteps, and the "box" modality (denoted as $\square$) to preserve data for later use. These modalities are crucial not only for expressing temporal constraints at the type level, but also for efficient memory management. By clearly defining which values are required immediately and which are postponed, the system enables the garbage collector to be more aggressive. Specifically, it only needs to retain the present value and the associated deferred computation encapsulated by the $\bigcirc$ modality, thereby freeing memory associated with data that is no longer necessary. This design promotes both correctness and efficiency, making Async Rattus particularly suited for building responsive and resource-aware reactive systems.

The modal type system of Async Rattus delivers essential guarantees for reactive software. Productivity follows from its design: Each discrete step in time produces a result, preventing the system from degrading performance over time. Causality is enforced at the type level by prohibiting any dependency on values that arise in the future, such that any value produced at a given time step is derived solely from current or past values. In Async Rattus, whenever we defer a computation with the $\bigcirc$ modality, the computation of this future value is not yet calculated and is instead linked to a specific clock. The clock is a set of channels, and whenever a program receives input data on one of these channels, a value is produced.

The primary data type of the language is a *signal*. The signal data type, is the Async Rattus implementation of FRP data streams. A signal is implemented as a recursive data type where each element is accompanied by a delayed computation representing the next value:

Signal Type Definition

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

Figure 1: Async Rattus signal type definition. `:::` denotes infix construction, and the `O` operator corresponds to the later modality.

The head of the signal is available immediately, while the tail is a delayed computation of type `O (Sig a)`. Crucially, this delay is not arbitrary. Its execution is controlled by the clock, tied to the later modality and inferred by the type system. Each delay is essentially a promise that the associated computation will occur once an input is received on a channel within that clock. In other words, the delayed computation is calculated, turning it into a present value. We call this a tick of a clock, denoted as $\checkmark_{cl}$.

To illustrate, consider the following example. Assume that a button in a GUI produces the next value of a signal. The delayed computation of this signal has the clock channel x. Whenever the button is pressed, a value is produced for the channel x. This action turns the delayed computation of the signal, into a value in the present. This mechanism distinguishes Async Rattus from purely synchronous systems. In the synchronous counterpart language, Rattus, every element of a data stream is produced in relation to one global clock [3].

To highlight this difference, we use the zip function as an example. The zip function, takes two data streams as input, and produces a single stream output, that consists of pairs of values from the original two data streams, as shown in Table 2.3.

Time step	xs	ys	zip xs ys
t_1	1	'a'	(1, 'a')
t_2	2	'b'	(2, 'b')
t_3	3	'c'	(3, 'c')
t_4	4	'd'	(4, 'd')

Table 1: Example trace of zipping two streams.

In contrast, Async Rattus allows signals to be associated with independent clocks. When multiple signals are composed to a single signal, using a combinator function such as zip, the function must carefully coordinate their clocks. If only one of the clocks tick, then only that signal produces a new value, and its new value is combined with the most recent value of the other signal. This can lead to pairings where one element remains unchanged over multiple time steps, highlighted in Table 2.3.

Time step	xs	ys	zip xs ys
t_1	1	'a'	(1, 'a')
t_2		'b'	(1, 'b')
t_3	2	'c'	(2, 'c')
t_4	3		(3, 'c')
t_5	4		(4, 'c')
t_6		'd'	(4, 'd')

Table 2: Example trace of zipping two asynchronously ticking signals.

The primitives *delay* and *adv* (advance) are operations used to manage deferred computations. The delay keyword acts as the constructor for the later modality, deferring the evaluation of a computation until the associated clock ticks. Conversely, the adv keyword serves as the eliminator for later, retrieving the value when the clock permits its evaluation. These operations are restricted by the typing rules of the language, which dictate that an advance only can be used in the scope of a delay. This means that we cannot use a future value in the present, and the type checker thereby enforces the temporal order and preserves causality. For example, an attempt to prematurely use a delayed value in the present violates the temporal order, and will be rejected at compile time:

```
-- Not allowed: advancing a delayed value too early
subtractOne :: O Int -> Int
subtractOne x = adv x - 1
```

Subtracting 1 from a value guarded by the ◯ modality is not possible in the current time step. The value doesn't exist yet, and consequently will not compile. Instead, we need to make use of the `delay` primitive:

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

The function must return an argument of type `O Int`, as the program must wait upon the integer input to arrive and only then subtract 1. Similar care has to be taken when we attempt to define a higher-order function on signals:

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

Note that `map` requires that the function is available for application at any point in the future. This makes it possible to define a higher-order function that builds a chain of references to values of prior time steps, causing an implicit space leak. It is therefore important that values that must be moved to the future, are time-invariant and always accessible. We refer to values of these types as *stable*. In Async Rattus, non-stable types can be made stable using the box modality

□ [1]. Box must be applied to the function type, to correctly implement the `map` function.

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

Using box here ensures that the function has no temporal dependencies. Consequently, the box modality must be eliminated using *unbox* before it can be applied.

2.4 Linear Temporal Logic

To reason about signals, we cannot solely use propositional logic. Given that the values a signal produces are inherently time-dependent, we need a way to express and verify properties over time, not just at a single instant. Propositional logic cannot quantify *when* a statement should hold, so it fails to capture how predicates evolve over time. Consider the following example:

$$p \Rightarrow q \quad \text{where}$$

$$p : \text{The patient takes their medication}$$
$$q : \text{The patient's symptoms improve}$$

This fails to express when the symptoms will improve, or for how long the patient must take the medication. Instead, we need a temporal logic let us say quantify *when* the statements should hold, and how their notion of truth behaves dynamically over time. Linear temporal logic (LTL) extends propositional logic by introducing temporal connectives that allow one to specify how propositional statements evolve over time. Using LTL connective **X** (Next), the prior example can be refined as follows:

$$p \Rightarrow X\ q$$

$$p : \text{The patient takes their medication}$$
$$X\ q : \text{The patient's symptoms improve the } \textbf{next} \text{ day}$$

The time frame in which they can expect improvement is now explicit and clear. For the same reason, this demonstrates how LTL is also a suitable logic for reasoning about signals, as it can express properties that describe how signals evolve over time.

2.4.1 Syntax

LTL introduces the following temporal connectives [7] [1]:

- **X** ϕ ("neXt")
  ϕ holds at the next state.
  Example: $X\,p$ means "p holds in the next state."

- **F** ϕ ("Eventually")
  ϕ holds at some future state.
  Example: $F\,q$ means "q will eventually hold."

- **G** ϕ ("Globally")
  ϕ holds at all future states.
  Example: $G\,p$ means "p always holds."

- ϕ **U** ψ ("Until")
  ϕ must hold continuously until ψ becomes true, and ψ must eventually hold.
  Example: $p\,U\,q$ means "p holds until q holds, and q eventually holds."

Using these temporal operators, provides a syntax for reasoning about the state of a program over time.

2.4.2 Safety and Liveness Properties

Using LTL to define properties gives rise to two classes of properties: *safety* and *liveness* properties. The distinction between these are important when implementing LTL as a means of testing reactive systems, as liveness properties inherently cannot be checked in finite time.

Safety properties assert that *something bad never happens*. These properties are violated by a finite prefix of an execution, meaning that once violated, no continuation of the execution can make the property hold again. The example introduced previously,

$$p \Rightarrow X\ q$$

is a safety property, because a counterexample can be found in a finite prefix of an execution, and from that point on the property can never become true again.

Liveness properties assert that *something good eventually happens*. Unlike safety properties, liveness properties cannot be refuted by any finite execution prefix, because the "good" event might still occur in the future. If we modify the previous example to use **F** (eventually) instead of **X** (next), we obtain a liveness property:

$$p \Rightarrow F\ q$$

[1] ϕ and ψ denotes LTL formulas.

Where q now expresses that the patient's symptoms *eventually* improve. This is impossible to check in finite time, given that even if the symptoms haven't improved, it is possible they may do so tomorrow.

The distinction between safety and liveness properties are important because it highlights an inherent limitation when logic is applied in finite-time reasoning. We elaborate further on the relevance in 6.2.

2.5 Domain Specific Languages

A Domain specific language (DSL) is a language tailored to express and solve problems in a specific domain. It provides a higher level of abstraction that closely mirrors the problem space, improving both the clarity of code and the safety of domain-specific computations [8]. An eDSL is a DSL implemented within a general-purpose host language. The advantage of this approach is that all of the tooling and ecosystem surrounding the host language can be reused, instead of making it up from scratch. By virtue of being limited in scope, a DSL can include domain specific rules in the language in ways a general-purpose language cannot. PBT frameworks often incorperate their own eDSL in their host language to express properties. For instance, QuickCheck contains a notion of explicit universal quantification ($\forall$), implication ($\Rightarrow$) and conjunction ($\wedge$):

```
1  prop_dsl =
2    forAll (choose (0, 100)) $ \n ->
3      (n >= 50 ==> even n) .&&. (n <= 100)
```

By embedding this logic into the host language, all of the benefits from the host language are still present, all while using a notation that is highly expressive within the domain of property-based testing.

2.5.1 Degree of embedding

Embedding a DSL involves choosing a strategy for how tightly the DSL is integrated with its host language. A common approach is to deeply embed the DSL. In this strategy, the structure of the DSL is explicitly represented by data types in the host language, such that the structure of the data type defines an Abstract syntax tree (AST) of the language.

In contrast, shallow embedding maps DSL constructs directly to the host-language. This approach results in more concise code, simpler interfaces, and better reuse of the host language features. However, this integration can make it difficult to enforce domain-specific constraints, as the structure of the statements cannot be inspected. Selecting between shallow and deep embedding involves balancing ease of implementation with the need for flexibility in the interpreter. In many cases, the most effective solution is a hybrid design, leveraging the benefits given from the host language, while preserving the structure that a deep embedding provides [8].

2.5.2 Generalized algebraic data types

Generalized algebraic data types (GADTs) extend Haskell's Algebraic data types (ADTs) by allowing constructors to specify their type. In a conventional ADTs, all constructors of a data type must return the same type, limiting how much can be expressed in the type system. By contrast, this extension lifts that restriction, enabling type annotations on a per-constructor basis. This additional flexibility allows for encoding richer invariants directly in the type declarations. It is especially useful for modeling data structures whose valid forms vary depending on type-level information. For instance, consider the canonical example of a simple expression language that supports integer and boolean literals, along with addition and equality:

```haskell
-- ADT
data Expr
  = LitInt Int
  | LitBool Bool
  | Add Expr Expr
```

In this ADT version, nothing prevents the construction of a nonsensical expression such as `Add (LitBool True) (LitInt 5)`.

```haskell
-- GADT
{-# LANGUAGE GADTs #-}
data Expr a where
  LitInt  :: Int -> Expr Int
  LitBool :: Bool -> Expr Bool
  Add     :: Expr Int -> Expr Int -> Expr Int
  Eq      :: Expr Int -> Expr Int -> Expr Bool
```

By defining the `Expr` type using the GADT syntax, the type parameter `a` tracks the type of value that an expression produces. The `Add` constructor ensures it only operates on integer expressions and produces an integer, while `Eq` enforces comparison between integer expressions and returns a boolean. Ill-typed expressions like `Add (LitBool True) (LitInt 5)` are now ruled out by the type system at compile time.

Using a GADT to create an eDSL, the language can be effectively illustrated as an ASTs. The term abstract in AST reflects the fact that the concrete syntactic details of the language are omitted, and only the structure necessary to evaluate the language remains. For example, the prior expression type can be represented as an AST, where each constructor corresponds to a node in the tree. The constructor `Add x y` represents addition of the two child nodes in the tree. Leaf nodes such as `LitInt n` represent integer literals (see Figure 2).

Up to this point we have worked with literals and constants, which is of limited use when making a specification language given that the values cannot be substituted upon evaluation. To represent variables, we can extend the `Expr a` type to include the constructor:

```
1  Var :: String -> Expr a
```

We can now make an expression that references variables by name, storing values for these in a seperate data structure. We call this data structure an *environment*. The environment can be be thought of a mapping from variables to their corresponding values. To evaluate an expression like `Add (Var "x") (LitInt 2)`, we would need to look up the value of x in the environment.

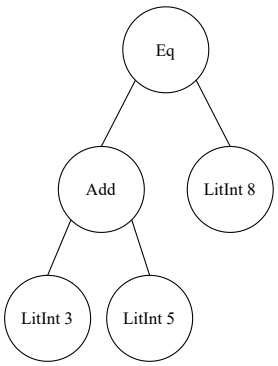

Figure 2: Tree representation of a value of type `Expr Bool`

The representation of `Expr` is useful because it defines a structure that can be evaluated in a principled manner. The same expression can be evaluated on separate environments, and each node can carry its own evaluation semantics, possibly changing the state of the environment. They are also composeable, any AST can be a subtree to larger more complex tree.

2.6 Type Level Programming

We define type level programming (TLP) as a set of techniques that encode logic in the type system, thereby catching violations of the type at compile-time rather than runtime. The key benefits of this approach, as relevant to this thesis, include:

1. Encoding domain invariants: Capturing domain rules at the type level effectively unifies the understanding of the domain with the implementation.

2. Enabling complex types: Richer types can be achieved by lifting logic to the type level, which may lead to a more natural model of the domain.

To illustrate these ideas, consider a function that retrieves the head element of a list. In a conventional list type, this operation is inherently partial due to the potentially-empty list, and consequently may throw an error at run time:

Working with Non-Empty Lists

```haskell
-- Defines a NonEmpty list
data NonEmpty a = a :| [a]

-- Get the head of a list (may throw runtime error!)
head :: [a] -> a
head (x : _) = x

-- Get the head of a NonEmpty
headNE :: NonEmpty a -> a
headNE (x :| _) = x
```

Figure 3: Using `NonEmpty` to enable safe head access

While the example is rather trivial, it effectively illustrates that by narrowing the domain of a function with a stronger argument, we can sift out nonsensical programs before they are ever run. We implement and expand further on TLP and the corresponding Haskell constructs as they become relevant in Section 4.2.

3 Design

This section clarifies the design choices made during the creation of PropRatt. Implementations of these decisions will be elaborated upon in Section 4.

3.1 Shallow or deep embedding

In designing PropRatt as an eDSL, we face a choice between a shallow and deep embedding. A shallow embedding could expose users directly to Async Rattus's primitives, requiring manual management of *delay*, *adv*, and clock coordination strategies. We prefer to express high-level abstractions directly in the DSL, as this improves the readability of specifications and avoids the confusion of encoding abstract specifications of a program through detailed low-level implementations. Therefore, we choose to adopt a deep embedding, encapsulating all clock coordination, deferred computation, and concurrency within the evaluation of the DSL. This approach preserves the expressive power needed to articulate high-level specifications while shielding users from implementation complexity.

3.2 Comparing multiple signals

PropRatt must support testing of signal combinators by enabling direct comparison of values produced by signals. To illustrate the need for this capability, consider the prefix-sum example. Given a finite sequence $a_1, a_2, \ldots, a_n$ of natural numbers, the prefix-sum s is defined by

$$s_k \;=\; \sum_{i=1}^{k} a_i$$

so that each value of the signal s accumulates all numbers from previous time steps. One property to reason about the correctness of such signal, could be that s grows strictly monotonically:

$$\mathit{Always}\,\big(p < \mathit{Next}\ p\big),$$

where p denotes the signal's value at the current timestep, and *Next* p refers to its value in the next time step. This predicate is not possible exclusively using LTL. Since this property refers directly to values of different time steps, the specification language must contain constructs to compare values across time.

Expanding the prefix-sum example we could write a more specific property that compares values from different signals. We want to be able to observe how signal values evolve in relation to each other, to ensure that the output values change correctly and occur precisely at the intended time steps. This allows us to reason about the correctness of combinator functions by comparing signal values of the input, with signal values of the output.

A specification could be expressed as:

It should always be true that, the next value of the output signal should be equal to the sum of the current element of the output signal and the next value of the input signal.

The specification language should have syntax to express:

$$G\left(X\,q = q + X\,p\right) \quad \begin{array}{l}\text{where } p \text{ is the current value of the input signal,}\\ \text{and } q \text{ is the current value of the output signal.}\end{array}$$

Because we deeply embed the language, we need to provide explicit language constructs that allow users to select and manipulate individual signals within the state of the test. This allows us to compare the values produced by the original signal to the values produced by the output signal at each discrete time step throughout the evaluation of the test.

3.3 Modelling signals in parallel

To effectively test Async Rattus programs, we must model the state of an execution in a way that captures the asynchronous nature of signals. As illustrated in Table 2.3 and discussed in Section 2.3, recall that signals in Async Rattus do not produce values in lock-step. That is, each signal may emit a value at different points in logical time, depending on the clock to which its delayed computation is tied. The state of the test framework must therefore support some mechanism that advances the state of the program under test, where each signal may or may not produce a new value. Effectively allowing for simulation of scenarios where multiple clocks may tick simultaneously or independently.

This suggests that the advancement of signals and the generation of their clocks cannot be completely arbitrary, as this could result in signals never producing values, or always producing values, resulting in a poor test coverage. Intuitively we may assume that allowing all signals under test an equal chance of ticking at any given time seems reasonable, as it would be maximally fair to all signals. But it might lead to interleavings of signals that don't accurate reflect how the system would act, as the frequency in which signals ticks largely depends on the domain. We work with this assumption throughout our implementation, but reflect on this design point further in Sections 5 and 6.3.

3.4 Relative temporal order of values

A signal always holds a current value, but this value may persist across multiple time steps of the program, if no new value is produced by its associated deferred computation. That is, a signal may continue to carry the same value infinitely often if its clock does not tick. This leads to an important requirement for testing: the ability to query whether a value produced by a signal was produced at the current timestep, or whether it was carried over from a prior one.

We want to be able to specify if a signal carries a value from its prior time step, ie. *stutters*. To express the property of stuttering, the specification language must have a notion of a *tick* (✓), where $\checkmark_{cl(a)}$ denotes a tick of clock a. Using this notation, we can formulate a property as follows:

> Let a and b be signals, and a_1 and b_1 be clocks tied to each signals' delayed computation[2]. Then,
>
> $$G\left(X\left(\checkmark_{cl(a_1)} \wedge \neg \checkmark_{cl(b_1)}\right) \rightarrow \left(b = X\, b\right)\right)$$

can be read as: "It should always be true that, if in the next time step a produces a value and b does not, then the current value of signal b is equal to the next value of signal b." This temporal predicate effectively conveys whether b is a stuttering of a. To support the evaluation of such properties, the framework must maintain some notion of whether each value in a signal was produced in the current time step or carried over from a prior timestep.

3.5 Shrinking

QuickCheck implements shrinking functionality for base types, to find smallest possible counterexamples when a test fails. To leverage QuickCheck when testing signals in Async Rattus, we must create a custom implementation and shrinking strategy. To illustrate how shrinking of a signal should work, assume we wish to test, that the values produced by the signal of type `Sig Int` are strictly smaller than three:

```
1  -- Pseudo syntax
2  inputSignal = (0 ::: 2 ::: 4 ::: never) :: Sig Int
```

The result of shrinking a signal that fails this property should be the signal containing only one element of the value 3:

```
1  -- Pseudo syntax
2  shrunkSignal = (3 ::: never) :: Sig Int
```

[2]This is a simplification because all delayed computations may have distinct clocks.

4 Implementation

PropRatt enables PBT of signal combinators in Async Rattus with the following components:

- A declarative specification language to write temporal predicates.
- A stateful model that abstracts parallel signal execution.
- Arbitrary typeclass instances for the signal type.

The temporal predicates are evaluated with a model of program behavior in its environment. The signals in the model can be generated arbitrarily, supplied by the user, or composed from both sources. This flexibility enables users to generate input signals, pass them to the function under test, and integrate the function's output to the model. In the temporal predicate, users are then able to refer to the values produced by signals in the model. Users can also advance the state of the model by using the LTL inspired temporal constructs of the specification language, thereby allowing users to reason about how multiple signals behave over time.

In the following sections, we explain how the key components of PropRatt work together to form a QuickCheck property. Along the way, we present the rationale behind important implementation choices. To motivate and illustrate these components, we use the `zip` function as a recurring example. The type signature of the function is defined in Figure 4.

Zip type signature

```
zip :: (Stable a, Stable b) => Sig a -> Sig b -> Sig (a :* b)
```

Figure 4: The `zip` function returns a pair of values, one from each input signal, at every time step. The types must be stable such that they can be moved safely to the future. `:*` denotes a strict pair constructor.

For the purpose of this example, assume that the inputs to `zip` are signals of type `Sig Int` and `Sig Char`. The output will be a signal of type `(Int :* Char)`, as illustrated in the sample trace execution of Table 2.3. To test the behavior of `zip`, we want to assert that the output at each time step reflects the corresponding values from the two input signals. This property can be expressed in the following pseudo-temporal logic syntax:

$$\mathbf{G}\big(\texttt{fst zs} = \texttt{xs} \wedge \texttt{snd zs} = \texttt{ys}\big)$$

where `xs` and `ys` are the input signals, and `zs` is the result of `zip xs ys`.

This specification reads: "It is always the case that the first value of the zipped signal equals the value from the first signal, and the second value equals the value from the second signal." The first step towards testing this property of `zip` is to transcribe the above pseudo-syntax to a real specification language.

4.1 Specification language

To express temporal predicates, we introduce the semantics of our specification language. The types `Pred`, `Expr` and `Lookup`, illustrated in Figure 5 defines the DSL constructs in its entirety.

Specification language

```
data Pred (ts :: [Type]) (t :: Type) where
  Tautology     :: Pred ts t
  Contradiction :: Pred ts t
  Now           :: Expr ts Bool -> Pred ts Bool
  Not           :: Pred ts t -> Pred ts t
  And           :: Pred ts t -> Pred ts t -> Pred ts t
  Or            :: Pred ts t -> Pred ts t -> Pred ts t
  Until         :: Pred ts t -> Pred ts t -> Pred ts t
  Next          :: Pred ts t -> Pred ts t
  Implies       :: Pred ts t -> Pred ts t -> Pred ts t
  Always        :: Pred ts t -> Pred ts t
  Eventually    :: Pred ts t -> Pred ts t
  After         :: Int -> Pred ts t -> Pred ts t
  Release       :: Pred ts t -> Pred ts t -> Pred ts t

data Expr (ts :: [Type]) (t :: Type) where
  Pure    :: t -> Expr ts t
  Apply   :: Expr ts (t -> r) -> Expr ts t -> Expr ts r
  Index   :: Lookup ts t -> Expr ts t
  Ticked  :: Lookup ts t -> Expr ts Bool

data Lookup (ts :: [Type]) (t :: Type) where
  Previous  :: Lookup ts t -> Lookup ts t
  Prior     :: Int -> Lookup ts t -> Lookup ts t
  First     :: Lookup (Value t ': x) t
  Second    :: Lookup (x1 ': Value t ': x2) t
  Third     :: Lookup (x1 ': x2 ': Value t ': x3) t
  Fourth    :: Lookup (x1 ': x2 ': x3 ': Value t ': x4) t
  -- etc. We support up to index Ninth
```

Figure 5: `Pred`, `Expr` and `Lookup` GADTs. These represent all the language constructs available in the specification language. Each type respects the types of the signals being referred to (tracked by the `ts :: [Type]` parameter) and the return type `t :: Type`. The return type of `Now` is constrained to be `Bool`, such that any evaluation of an expression returns true/false.

At a high level overview, the types achieve the following:

- `Pred` combines propositional logic connectives (e.g., `And`, `Not`, `Implies`) with temporal operators from LTL, such as `Next`, `Until`, `Always`, and `Eventually`.
- `Expr` encodes compound expressions (arithmetic, comparison, and function application) using values from the model.
- `Lookup` provides type safe indexing to retrieve values produced by signals in the model.

The `Now` constructor reflects an expression that should be evaluated in the current timestep. An `Expr` is a potentially compound statement that compares values from signals of potentially different types by accessing the current state of the system using the `Lookup` type. Using the presented language constructs of the specification language, we can now express the predicate introduced in Section 4.

Verbose representation of a specification

```
Always $
  Now $
      (Apply
          (Apply (Pure (==))
          (Apply (Pure fst') (Index First)))
      (Index Second))
`And`
  Now $
      (Apply
          (Apply (Pure (==))
          (Apply (Pure snd') (Index First)))
      (Index Third))
```

Figure 6: By nesting `Apply` nodes and lifting functions such as `(==)` and `fst'` to the context of an `Expr`, the language supports function application entirely within the specification.

In Figure 6 the `Pure` constructor lifts functions to the context of the expression being evaluated. By writing `Index First`, the value of the first signal is in the model is retreived in a type-safe manner using the `Lookup` type. While the specification is correct, it is also extremely verbose to the point where it is impractical to write. It would be better if we instead are able to inject idiomatic Haskell code that represents the function application we are trying to achieve. To this end, we implement typeclasses for the `Expr` type.

Applicative functor instance for Expr

```
instance Functor (Expr ts) where
  fmap :: (t -> r) -> Expr ts t -> Expr ts r
  fmap f (Pure x)     = Pure (f x)
  fmap f (Apply g x)  = Apply (fmap (f .) g) x
  fmap f (Index lu)   = Apply (Pure f) (Index lu)
  fmap f (Ticked lu)  = Apply (Pure f) (Ticked lu)

instance Applicative (Expr ts) where
    pure :: t -> Expr ts t
    pure = Pure
    (<*>) :: Expr ts (t -> r) -> Expr ts t -> Expr ts r
    Pure f <*> x = fmap f x
    Apply f g <*> x = Apply (Apply f g) x
    (<*>) _ _ = error "Expr: unsupported constructor for application."
```

Figure 7: Applicative and functor instances for `Expr`.

The implementation of applicative and functor type allows native Haskell functions to be lifted into the embedded language. We can now construct expressions compositionally using familiar idioms such as `(<$>)` (fmap) and `(<*>)` (sequence), instead of introducing new syntax. The applicative instance defines how function application is encoded as a nested tree of `Apply` nodes. When `<*>` is used on an `Apply`, the implementation nests constructions of an `Apply` node:

```
Apply (Apply f g) x
```

And applying a function to a value stored in a `Ticked` or `Index` node results in a new `Apply` node:

```
fmap f (Ticked lu) = Apply (Pure f) (Ticked lu)
```

We can now apply the functions `fst'` and `snd'` which extracts the values from a pair. The predicate from Figure 6 is now more concise and readable, see Figure 8.

Zip predicate

```
Always $
        Now ((fst' <$> (Index First)) |==| (Index Second)
        `And`
        Now ((snd' <$> (Index First)) |==| (Index Third)
```

Figure 8: Zip predicate from Figure 6
transcribed to our specification language.

Note that the constructors may not be of same type, making it impossible to

make an `Eq` instance for `Expr`. Instead, we compare values using the applicative style:

```
1  (|==|) :: (Applicative f, Eq t) => f t -> f t -> f Bool
2  x |==| y = (==) <$> x <*> y
```

Now that we are able to express predicates in our language, we need to compose signals under test into some state, such that we can access the values when evaluating. For this particular example, we need to have the `First`, `Second` and `Third` signal available to interact with using the `Lookup` language constructs. For this we turn to the implementation of the system state.

4.2 Modelling system state as a type

One might initially assume that an Async Rattus program can simply be modeled as a list of signals `[Sig a]`. Then, an evaluator function could be written for the specification:

```
1  evaluate :: Pred ts t -> [Sig a] -> Bool
```

This representation is insufficient because the evaluator would have no way to advance all signals in a way that mimics how signals behave when they produce values in parallel. A more appropriate type would be `Sig [a]`, representing a signal that in each time step yields a list of values produced by a list of signals `[Sig a]`. A value of the type `Sig [a]` could then be constructed as follows:

```
1  flatten :: [Sig a] -> Sig [a]
```

Here, the resulting signal produces a list where the *n'th* value in `Sig [a]` corresponds to the value produced by the *nth* signal of `[Sig a]`, as illustrated in Table 3.

Time	`Sig` a_1	`Sig` a_2	`Sig` a_3	`flatten [`s_1`,` s_2`,` s_3`]`
t_0	a_0	a_{10}	a_{20}	$[a_0, a_{10}, a_{20}]$
t_1	a_1	a_{11}	a_{21}	$[a_1, a_{11}, a_{21}]$
t_2	a_2	a_{12}	a_{22}	$[a_2, a_{12}, a_{22}]$
⋮	⋮	⋮	⋮	⋮

Table 3: Illustration of `flatten :: [Sig a] -> Sig [a]`. Each row corresponds to a time step, and each column represents a constituent signal. The output signal produced by `flatten` is shown on the right.

This remains too restrictive: all signals must have the same type, namely *a*. We therefore need a data structure that can represent a list of signals of different types.

4.2.1 Heterogenous lists

A *heterogeneous list* (HList) is a special kind of list in which the values can have different types. While a regular list contains terms (or values) of a specific type, an HList carries a type-level list that describes the type of each individual value.

To define such structure, we use the *DataKinds* language extension that promotes data constructors to the type level. For instance, the cons operator `:` is lifted to `':`, such that we can construct a type-level list. To understand how this works, it's helpful to understand the idea of a *kind*.

In Haskell, a kind classifies types in much the same way that a type classifies values. For example, the list type constructor `[]` has kind `Type -> Type`, which means it takes a type, such as `Int`, and produces a new type `[Int]`. On the other hand, a value constructor like `True` corresponds to a single type `Bool`, and therefore has kind `Type`. When list constructors are promoted as shown previously, they become kind-polymorphic. This enables us to create type-level lists of types such as `'[Int, Maybe Int, Char]`.

HList type definition

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}

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

infixr 5 %:
(%:) :: x -> HList xs -> HList (x ': xs)
(%:) = HCons
```

Figure 9: HList definition, where `'[]` and `':` are promoted type-level list constructors for the empty list and list construction, respectively.

Using the definition in Figure 9, we can construct a list of different types and enjoy the benefits that come with type-safety. Consider the example of zipping two signals, introduced in Section 4. In this case we must be able to hold three signals of types `(Int :* Char)`, `Int`, and `Char` in our model. An HList can effectively model this list of different types:

```haskell
((1 :* 'a') :% 1 :% 'a' :% HNil) :: HList '[(Int :* Char), Int, Char]
```

Any function that operates on an HList must do so in a type-safe manner. For instance the function `first (HCons h _) = h` must be explicitly annotated with the type signature `first :: HList (a ': _) -> a` to help the compiler infer the return type of the function. This approach quickly breaks down in attempts to generalize the function, as shown in Figure 10. The challenge lies in the implication of a run-time term to a type that must be checked at compile-time. Instead, we would like a function from a type to a (run-time) term. We expand on this idea and implement it using a type class in Section 4.4.1.

Index function

```
-- Returns the n'th element of the list.
index :: HList (a ': as) -> Int -> ???
```

Figure 10: The index function illustrates the need for functions from types to terms, as the `Int` argument determines which type to extract from the HList. The `Int` argument is only known at run-time, therefore the compiler cant infer the return type of the function at compile time.

Using the data type definitions introduced up to this point, we are able to express the type `Sig (HList (a ': as)` as the state signal of our test.

4.2.2 Value definition

Recall the notion of a stuttering, introduced in Section 3.4. To improve the specification we have worked on up to this point, we may wish to express that `zip` also exhibits this stuttering behaviour, written in a temporal predicate in Figure 11.

Stuttering predicate

```
Always $ Next (
            (Not (Now (Ticked Second))
             `And`
            (Now (Ticked Third)))
             `Implies`
            (Now ((fst' <$> Index First) |==|
                  (fst' <$> Index (Previous First))))
            )
```

Figure 11: A predicate that the output of zip stutters, where `First` is the output of zip, and `Second` and `Third` are input signals in the model.

To express this, we need to have notion of whether a signal produced its value in the current timestep, or carried it over from a prior timestep. We use the `Ticked` constructor to express this, along with `Previous`. These constructors introduces two additional requirements to the model:

1. We must retain a history of values produced for all signals.
2. There must be a way to distinguishing new and carried-over values.

To accommodate these requirements, we define the *Value* data type illustrated in Figure 12. This type implements a flag that conveys if the value was produced in the current timestep, alongside a list of values that a signal has produced.

Value definition

```haskell
{-# LANGUAGE GADTs #-}
import AsyncRattus.Strict

newtype HasTicked = HasTicked Bool deriving Show
data Value a where
  Current :: !HasTicked -> !(List a) -> Value a
```

Figure 12: Value definition, where `List` is a strict variant of a list, imported from `AsyncRattus.Strict`.

Using `Value`, we can define the type of the model as shown in Figure 13. This structure sufficiently models the state of execution of an Async Rattus program, providing the context in which predicates are evaluated. As seen in Table 4, the model carries a flag and a history of values produced by the signal up to a given state.

State type definition

```haskell
newtype State ts t = Sig (HList (Value t ': ts))
```

Figure 13: The type encapsulating signals in a state of the test. This type is a signal, that at each time step holds an HList of values.

Table 4 illustrates an example of the data type, holding two signals of different types. While the signal of characters produces a value in every timestep, the signal of integers, carries over its former value in time step t_1. This is a visual representation of adding the two input signals from Table 2.3, to a `State` data structure.

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

Table 4: Visual representation of a sample value of type `Sig (HList '[Value Int, Value Char])`. Each row corresponds to a discrete time step, and each cell displays the current value of the signal at that time. `%:` denotes infix HList construction, T denotes that the latest value of a list was produced within the current time step and F denotes the latest value was carried over from the prior time-step.

4.3 Evaluation semantics

We now turn to the implementation details of the evaluation of the language. This section describes how the specification language introduced in Section 4.1 is evaluated, when tested over a state of the type introduced in Section 4.2.

Temporal predicate evluation function

```
1  evaluate' :: (Ord t) => Int -> Pred ts t -> Sig (HList ts) -> Bool
2  evaluate' timestepsLeft formulae sig@(x ::: Delay cl f) =
3    if IntSet.null cl
4      then evaluateSingle timestepsLeft formulae sig
5      else timestepsLeft <= 0 || case formulae of
6              Tautology       -> True
7              Contradiction   -> False
8              Now expr        ->
9                case evalExpr expr x of
10                 Pure b -> b
11                 _ -> error "Unexpected error during evaluation."
12             Not phi         -> not (eval phi sig)
13             And phi psi     -> eval phi sig && eval psi sig
14             Or phi psi      -> eval phi sig || eval psi sig
15             Until phi psi   -> eval psi sig ||
16                                (eval phi sig &&
17                                  evaluateNext (phi `Until` psi) advance)
18             Next phi        -> evaluateNext phi advance
19             Implies phi psi -> not (eval phi sig && not (eval psi sig))
20             Always phi      -> eval phi sig &&
21                                 evaluateNext (Always phi) advance
22             Eventually phi  -> (eval phi sig ||
23                                  evaluateNext (Eventually phi) advance) &&
24                                  not (timestepsLeft == 1 &&
25                                        not (eval phi sig))
26             Release phi psi -> (eval psi sig && eval phi sig) ||
27                                 (eval psi sig &&
28                                  evaluateNext (phi `Until` psi) advance)
29             After n phi     -> if n <= 0
30                                 then eval phi sig
31                                 else evaluateNext (After (n - 1) phi) sig
32       where
33         evaluateNext = evaluate' (timestepsLeft - 1)
34         eval = evaluate' timestepsLeft
35         advance = f (InputValue (IntSet.findMin cl) ())
```

Figure 14: Evaluator function for `Pred` does post-order traversal of the data structure, returning the result if the predicate is false or has evaluated all time steps left. Constructors such as `Next` and `Always` advance the system state using LTL semantics.

To produce counterexamples in finite time, the evaluation of the predicates are bound by the `Int` argument, specifying how many discrete time steps that should be checked. The temporal operators evaluates the expression by using the Haskell primitive boolean operators, recursively evaluating and advancing

the state of the `Sig (HList (Value t': ts))` argument according to their semantics. The anonymous `advance` function defined in the where clause (line 35) forces the delayed computation by emulating a tick on the smallest channel of the clock of the state signal. This approach to advancing the state ensures the model gives all signal an equal probability of producing a value, and the intuition behind why we advance on the smallest channel is elaborated in Section 27. The conditional logic on lines 3-5 are specific to shrinking and consequently explained in Section 4.6.

The `Now` constructor calls the `evalExpr` evaluator function, and returns the boolean value produced by the `Expr` context. The semantics of the temporal operators differ slightly from those outlined in 2.4. This is due to the inherent constraint of having to check a property in finite time. For instance, the until semantics `Until p q` holds if q or `p` holds currently or in future steps. This is a "weak" until, as it does not promise the arrival of `q`. We defer discussion of this limitation to Section 6.2.2.

Expr evaluation function

```
1  evalExpr :: Expr ts t -> HList ts -> Expr ts t
2  evalExpr (Pure x) _      = pure x
3  evalExpr (Apply f x) hls = (($) <$> evalExpr f hls) <*> evalExpr x hls
4  evalExpr (Index lu) hls  =
5    case evalLookup lu hls of
6      Just' (Current _ (h :! _)) -> pure h
7      Just' (Current _ Nil)      -> error "History not found for signal."
8      Nothing'                   -> error "Signal not found."
9  evalExpr (Ticked lu) hls = pure (evalTicked lu hls)
```

Figure 15: Evaluation of Expr

The `evalExpr` function interprets an `Expr ts t` against a snapshot of the state at a current point in time. Line 3 of Figure 15 recursively evaluates the apply nodes, "fmapping" function application and sequencing another recursive call on the argument to the apply node. Eventually, the recursive calls folds down to a single pure `Expr ts t`, that can be returned. Line 4 looks up the head of the current history for the specified signal, where empty histories throw runtime errors. These branches should be unreachable, as we implement checks to prevent these in Section 4.3.1. Finally, `Ticked lu` uses a helper function `evalTicked` that returns a boolean value, wrapping this in an `Expr` context. The `evalLookup` function interprets a `Lookup ts t` against the snapshot of the state, provided by the `Expr`.

Lookup evaluation

```haskell
evalLookup :: Lookup ts t -> HList ts -> Maybe' (Value t)
evalLookup lu hls = case lu of
  Previous lu' ->
    case evalLookup lu' hls of
      Just' (Current b history) ->
        case history of
          _ :! xs -> Just' (Current b xs)
          Nil     -> Nothing'
      Nothing' -> Nothing'
  Prior n lu'  -> case evalLookup lu' hls of
    Just' v  -> nthPrevious n v
    Nothing' -> Nothing'
  First         -> Just' (first hls)
  Second        -> Just' (second hls)
  Third         -> Just' (third hls)
    -- ...
  Ninth         -> Just' (ninth hls)

nthPrevious :: Int -> Value t -> Maybe' (Value t)
nthPrevious n curr@(Current b history)
  | n <= 0    = Just' curr
  | otherwise =
      case history of
        _ :! xs -> nthPrevious (n - 1) (Current b xs)
        Nil     -> Nothing'
```

Figure 16: Lookup evaluation function. The return type `Maybe' (Value t)` signals that the value may not exist. `Maybe'` is strict variant of Maybe.

The `evalLookup` function implements the functions for accessing values in the state signal. `First` corresponds to accessing the value at the first index of the HList at this timestep, and so on. At last, we also introduce the evaluator function for extracting the `HasTicked` value from a signal in the state. This is shown in Figure 17.

Ticked evaluation

```haskell
evalTicked :: Lookup ts t -> HList ts -> Bool
evalTicked lu hls = case lu of
  Previous _ -> errorTickedPast
  Prior _ _  -> errorTickedPast
  First      -> extract $ first hls
  Second     -> extract $ second hls
  Third      -> extract $ third hls
  Fourth     -> extract $ fourth hls
  Fifth      -> extract $ fifth hls
  Sixth      -> extract $ sixth hls
  Seventh    -> extract $ seventh hls
  Eighth     -> extract $ eighth hls
  Ninth      -> extract $ ninth hls
  where
    errorTickedPast = error
                "Cannot check if signal has ticked in the past."
    extract (Current (HasTicked b) _) = b
```

Figure 17: Ticked evaluation function. Returns a boolean, indicating whether the specified signal has produced a value in the current timestep.

We haven't experienced a need for keeping historical `HasTicked` values in our data type and we therefore throw an error if we try to access a past value of `HasTicked` using `Previous` or `Prior`. In all other cases, the value is deconstructed to extract the boolean flag of `HasTicked` of the `Value`.

4.3.1 Scope check

An attentive reader might have noticed that it's possible to inspect the past before the model has taken enough steps to make that past available. Consider the following example:

```haskell
Always (Now (Previous (Index First) |==| (Index First)))
```

This formula tries to compare the current and previous values of the first signal right from the very beginning. But at the first time step, the past doesn't exist yet! This is precisely why the functions `evalExpr` and `evalTicked` are partial: they may encounter expressions that refer to undefined past values, especially during the early steps of evaluation. The constructors `Previous` and `Prior` could have alternatively been implemented purely in terms of the future, which aligns more closely with LTL semantics. That is, instead of looking into the past as we do, we could instead advance the state for this branch of the AST by implementing an operator like `NextValue` in the `Lookup` type:

```haskell
Always (Now ((Index First) |==| NextValue (Index First)))
```

We choose to implement `Previous` and `Prior` as it offers a language that is closer to the specification of certain signal combinators of Async Rattus. This approach emerges from our approach of examining Async Rattus combinator functions, to define a language. For instance, the signal combinator `buffer` of Async Rattus, takes a signal as input, and returns the same signal, but always one time step behind the original signal:

```
1  -- Buffer takes an initial value and a signal as input and returns a signal
2  -- that is always one tick behind the input signal.
3  buffer :: Stable a => a -> Sig a -> Sig a
4  buffer x (y ::: ys) = x ::: delay (buffer y (adv ys))
```

`buffer` suggests we should be able compare the value of the buffered signal with the previous value of the input signal. To ensure predicates are well-formed, we require that all uses of `Previous` or `Prior` occur within the scope of a sufficient number of `Next` constructors. This guarantees that enough time has passed for the historical references to be valid. This rule is enforced by invoking `checkScope` prior to property evaluation.

Lookup evaluation

```haskell
checkScope :: Pred ts t -> Bool
checkScope p = checkPred p 0

checkPred :: Pred ts t -> Int -> Bool
checkPred predicate scope =
  valid scope &&
  case predicate of
    Tautology       -> valid scope
    Contradiction   -> valid scope
    Now expr        -> valid (checkExpr expr scope)
    Not p           -> checkPred p scope
    And p1 p2       -> checkPred p1 scope && checkPred p2 scope
    Or p1 p2        -> checkPred p1 scope || checkPred p2 scope
    Until p1 p2     -> checkPred p1 scope && checkPred p2 scope
    Next p          -> checkPred p (scope + 1)
    Implies p1 p2   -> checkPred p1 scope && checkPred p2 scope
    Release p1 p2   -> checkPred p1 scope && checkPred p2 scope
    Always p        -> checkPred p scope
    Eventually p    -> checkPred p scope
    After n p       -> checkPred p (scope + n)
  where
    valid s = s >= 0

checkExpr :: Expr ts t -> Int -> Int
checkExpr expr scope =
  case expr of
    Pure _         -> scope
    Apply fun arg -> min (checkExpr fun scope) (checkExpr arg scope)
    Index lu       -> checkLookup lu scope
    Ticked lu      -> checkLookup lu scope

checkLookup :: Lookup ts t -> Int -> Int
checkLookup lu scope =
  case lu of
    Previous lu'  -> checkLookup lu' (scope - 1)
    Prior n lu'   -> checkLookup lu' (scope - n)
    _             -> scope
```

Figure 18: Functions to check that that `Previous` and `Prior` constructors are always used in the scope sufficient amount of `Next` operators to avoid looking too far into the past.

This function performs a traversal of the predicate, passing an integer argument to indicate if any subtree of the AST has a `Previous` outside the scope of a `Next`. These functions could possibly be lifted to the type system as well in future.

4.4 Constructing a model instance

In the previous sections, we introduced the language constructs, data structures and evaluation logic. To be able to make a QuickCheck property, we still lack

the ability to:

- Generate arbitrary signals, and
- Use these signals to make an instance of the model

The functions described in this section enable us to construct signals of the `State` type defined in Section 4.2. In doing so, we must employ some type-level programming to work with the model.

4.4.1 Flatten

We want to implement a convenience function that returns a `State`. Semantically it should work as `flatten` described in 4.2, but lifted with HLists to the following type signature:

```
flatten :: HList (Sig a ': as) -> Sig (HList (Value a ': as)).
```

Flatten 19 is implemented as a multi-parameter typeclass with a single function, as shown in Figure 19.

Flatten typeclass

```
1    class Flatten sigs vals | sigs -> vals where
2      flatten :: HList sigs -> Sig (HList vals)
```

Figure 19: Flatten typeclass.

Because we work with an HList, we can't expect the compiler to infer the return type without guidance. Therefore, using the functional dependency from *sigs* to *vals*, we promise that the *sigs* type uniquely determines *vals*. This constraint is needed to ensure GHC that there exists a mapping from the supplied types of the `HList`, and it can safely try to resolve instances when given a `HList sigs`. Using typeclasses in this way can be thought of as implementing a function from a type to a term. For each type in the HList, the compiler is able to resolve which instance that should be used for a supplied type, mapping it to a term of a different type.

Recursive case instance flatten

```haskell
-- Type constraints omitted for brevity.
instance Flatten (Sig a ': as) (Value a ': bs) where
  flatten :: HList (Sig a : as) -> Sig (HList (Value a : bs))
  flatten (HCons h t) = prepend h (flatten t)

prepend :: Sig t -> Sig (HList ts) -> Sig (HList (Value t ': ts))
prepend _ _ = -- Omitted for brevity
```

Figure 20: Recursive flatten instance traverses the `HList` argument, prepending the signal to a flattened state signal structure. See Figure 22 for implementation of *prepend*.

Prepend takes a signal and a state signal, and merges them into a new version of the state signal. The state signal passed to prepend in this instance, is a recursive call to flatten itself. This essentially means recursively traversing all input signals in the HList and prepending each one to the system state. We must also provide a base case for when the HList is empty, as shown in Figure 21.

Base case instance flatten

```haskell
instance Flatten '[] '[] where
  flatten :: HList '[] -> Sig (HList '[])
  flatten HNil = emptySig

emptySig :: Sig (HList '[])
emptySig = -- Omitted for brevity
```

Figure 21: Base case for the flatten instance, returns an empty `Sig (HList '[])`, ensuring that recursive calls to flatten (and prepend) type checks.

Given that *prepend* expects as first argument a value of type `Sig a`, we must ensure that any elements of the HList passed to *flatten*, exclusively holds signals. To ensure that each signal can be integrated into the state signal, it must be possible to traverse the type-level list of types and ensure the compiler that we have an instance of *flatten*, capable of managing this type. To achieve this, we apply the *Flatten* constraint in the instance for the non-empty *HLists*. This makes sure that the compiler is informed that every element of the HList can be flattened and prepended to the state signal. Additional constraints of `Stable a =>` and `Falsify bs =>` are needed for the instance too. The need for these arises from the implementation of *prepend* and are therefore introduced in the following section.

4.4.2 Prepend

The `prepend` function handles the logic of adding a new signal to the state signal. The first element of the new signal x is wrapped in the `Value` type, and added as the first element of the HList y, creating the new HList of the first time step of the state signal. As signals always produce a value in their first timestep, we default the value of `HasTicked` to `True`. The list is constructed as a strict list, using the `:!` constructor.

Prepend function

```
prepend :: Sig t -> Sig (HList ts) -> Sig (HList (Value t ': ts))
prepend (x ::: xs) (y ::: ys) =
  HCons (Current (HasTicked True) (x :! Nil)) y  :::
    prependAwait (x :! Nil) xs y ys
```

Figure 22: Implementation of prepend.

`prependAwait` represent all the recursive calls of `prepend`. Instead of advancing a single signal using `adv`, we instead make use of the Async Rattus `select` [1]. The `select` operator handles the advancement when combining two delayed computations, comparing the clocks of the two, and returning either that, the first, the second or both delayed computations produced a new value.

PrependAwait

```
prependAwait :: List t -> O (Sig t) -> hls -> O (Sig hls)
                -> O (Sig (HList (Value t ': ts)))
prependAwait x xs y ys  = delay (
  case select xs ys of
     Fst (x' ::: xs')   ys'         ->
        (Current (HasTicked True) (x' :! x) %: toFalse y) :::
            prependAwait (x' :! x) xs' y ys'
     Snd xs' (y' ::: ys')           ->
        (Current (HasTicked False) x %: y') :::
            prependAwait x xs' y' ys'
     Both (x' ::: xs') (y' ::: ys') ->
        (Current (HasTicked True) (x' :! x) %: y') :::
            prependAwait (x' :! x) xs' y' ys')
```

Figure 23: Implementation of prependAwait.

In the first case of the select, only the new signal produced a new value. Therefore we add the value produced, and set the HasTicked value to True. As none of the signals within the state signal produces a new value in this timestep, we must carry over the value(s) from the former time step and set all the `HasTicked` values of these to False. The `toFalse` function recursively goes

through the HList of values and modifies only the HasTicked value within the `Value` type. We once again write this as a new typeclass.

Falsify

```
1  class Falsify ts where
2    toFalse :: HList ts -> HList ts
3
4  instance Falsify '[] where
5    toFalse :: HList '[] -> HList '[]
6    toFalse _ =  HNil
7
8  instance (Falsify ts) => Falsify (Value t ': ts) where
9    toFalse :: HList (Value t : ts) -> HList (Value t : ts)
10   toFalse (HCons (Current _ x) t) =
11                              Current (HasTicked False) x %: toFalse t
```

Figure 24: Implementation of Falsify typeclass and toFalse instances

In the second case of the select, only the state signal produces a new value. That is, at least one of the signals represented in the state signal has produced a new value in the current timestep, while the new signal does not. In this case, we carry over the former value of the new signal and set the `HasTicked` value to `False`.

The `Both` case resembles the case where both the new signal and the state signal produces a new value. When a value is produced in one of the three cases, a delayed computation is added to this value to continue the signal structure of the new state signal. This delayed computation is a recursive call to `prependAwait`, handling the next time step of the new state signal, based on when the next values are produced. For the `prepend` and `prependAwait` functions, we implement them with the following constraints:

```
(Stable t, Stable (HList ts), Falsify ts)
```

The stable constraints are needed, because we potentially move values from a current time step into future time steps. This happens in the case where the new signal doesn't update, or the state signal doesn't update. Therefore, we have the constraint on both a single value `t` and `HList ts`, such that we know the types are stable and safe to move into the future. Finally, we can use flatten and prepend to initialize our test state, as shown in Figure 25.

Property-based test of zip

```
prop_zip :: Property
prop_zip = forAll genDouble $ \(ints, chars) ->
        let zipped      = zip ints chars
            state       = prepend zipped
                            $ prepend ints
                            $ flatten (HCons chars HNil)
            predicate   = Always $
                    Now ((fst' <$> (Index First)) |==| (Index Second))
                    `And`
                    (Now ((snd' <$> (Index First)) |==| (Index Third)))
            result      = evaluate predicate state
        in result
  where
    genDouble = do
      ints <- (arbitrary :: Gen (Sig Int))
      chars <- (arbitrary :: Gen (Sig Char))
      return (ints, chars)
```

Figure 25: Flatten and prepend calls can be chained to produce a state for the test. In this example, a state signal with 3 signals, corresponding to the state signal of the example introduced in Section 4.

All that is left to do is to generate arbitrary signals that can be passed as arguments to the property.

4.5 Arbitrary Signals

To leverage QuickCheck to test signals, we must extend its generator framework to support the `Sig a` type. We do this by implementing the `Arbitrary` instance for signals, see Figure 26.

Arbitrary Instance for `Sig a`

```
instance (Arbitrary a) => Arbitrary (Sig a) where
  arbitrary :: Gen (Sig a)
  arbitrary = arbitrarySig 100
```

Figure 26: Limiting signals to length 100 for bounded testing.

Here, we generate signals of length 100. The default of 100 signal values provides a balance between covering a wide range of asynchronous behaviors, while maintaining practical test performance.

Signal Generator

```
1  arbitrarySig :: (Arbitrary a) => Int -> Gen (Sig a)
2  arbitrarySig n = do
3    if n <= 0
4      then error "Cannot create empty signals"
5      else
6        go n
7        where
8          go 1 = do
9            x <- arbitrary
10           return (x ::: never)
11         go m = do
12           x <- arbitrary
13           len <- chooseInt (1, 3)
14           cl <- genClock len
15           xs <- go (m - 1)
16           let later = Delay cl (\_ -> xs)
17           return (x ::: later)
```

Figure 27: Line 16 constructs a delayed computation of type `D (Sig a)` that is tied to the generated clock `cl`. The unused argument in the lambda function `(\_ -> xs)` represents the value that would have arrived externally as input on one of the channels of the clock. Instead, the function returns the recursive call to generate the tail of the signal. `genClock` is defined in Figure 28.

In each recursive case of the function, an element `x` is generated with `arbitrary` and a clock `cl` is generated using `genClock`. The `arbitrary` keyword here is inherited from QuickCheck and can safely be used since the constraint `(Arbitrary a) =>` promises `a` derives an arbitrary instance.

Clock generator

```
genClock :: Int -> Gen Clock
genClock n = case n of
    1 -> do
      x <- chooseInt (1,3)
      return (IntSet.fromList [x])
    2 -> frequency [
        (1, return (IntSet.fromList [1,2])),
        (1, return (IntSet.fromList [1,3])),
        (1, return (IntSet.fromList [2,3]))
        ]
    3 -> return (IntSet.fromList [1,2,3])
    _ -> error "Partial function doesnt support n > 3"
```

Figure 28: Clock generator strategy. The function `genClock n` returns a `Clock` containing n channels. For $n = 1$, it selects one channel at random from {1, 2, 3}. For $n = 2$, it selects two distinct channels with equal probability for all possible pairs. For $n = 3$, it returns all three channels. The function is partial and raises an error if $n > 3$.

The clock for each delayed computation is represented by a set of integers. To simulate asynchronous behavior, we arbitrarily assign each delayed computation a clock drawn from subsets of $\{1, 2, 3\}$. During evaluation of the model, we then advance the system state by selecting the smallest clock from the union of all delayed computations' clocks. This strategy is used in the evaluator illustrated in Figure 14. This rule makes it possible to produce both synchronous ticks, where signal updates at the same time, but also asynchronous sequences in which some signals wait multiple time steps before advancing.

4.5.1 Heterogenous List generator

As presented in Figure 25, the test state is initialized by passing in an `HList` of signals. Doing so required quite some boilerplate using flatten and prepend. Using the arbitrary signal generator, introduced in section 4.5, we can now simplify the manual work needed to initiate such tests, by implementing a generator of type `Gen (HList (Sig a ': as))`.

HList generator instances

```haskell
class HListGen (ts :: [Type]) where
  generateHList     :: Gen (HList (Map Sig ts))

instance HListGen '[] where
  generateHList = return HNil

instance (Arbitrary (Sig t), HListGen ts) => HListGen (t ': ts) where
  generateHList = do
    x <- arbitrary
    xs <- generateHList @ts
    return (x %: xs)
```

Figure 29: Recursively generating arbitrary signals for each type in the type list.

For each element of the input list of types, we use `arbitrary` to generate the signal, which is safe to do by using the constraint `Arbitrary (Sig t)`. By leveraging Haskell type families, users can declare the element type they wish to generate, and the type family implementation is then able to resolve this at the type level to its corresponding signal type. This enables the compiler to select the correct generator instance for that specific signal type, while simplifying how we generate signals. We implement a type-level map function so that the type of our HList is `HList (Map Sig ts)`. The Map type family applies the Sig constructor to each element of the type list, and ensures that we generate signals of that type.

HList generator

```haskell
type family Map (f :: Type -> Type) (xs :: [Type]) :: [Type] where
  Map f '[] = '[]
  Map f (x ': xs) = f x ': Map f xs
```

Figure 30: Using type families, we can apply functions at the type-level.

While this would work, the type of the generator would have to be annotated explicitly in the test. It also lacks any way to generating an HList with a single signal, because the type parameter `[Type]` says it must be a list of types. Instead, the type applied should be kind-polymorphic.

ToList typeclass

```
type family ToList (a :: k) :: [Type] where
  ToList (a :: [Type]) = a
  ToList (a :: Type)   = '[a]
```

Figure 31: The ToList type family is a type level function that turns a type parameter `a` of some kind `k` to a list of types. By pattern matching on the type of `a`, we can create a singleton type-level list if a has kind `Type`, otherwise if the kind is already `[Type]`, return a. This functionality is enabled by the `PolyKinds` pragma, which allows for polymorphism on kind level.

We can now leverage `HListGen`, `Map` and `ToList` to assemble a single convenience function to generate multiple signals.

generateSignals function

```
generateSignals :: forall a. HListGen (ToList a) =>
Gen (HList (Map Sig (ToList a)))
generateSignals = generateHList @(ToList a)
```

Figure 32: The `TypeApplications` pragma provides a particularly succinct syntax to apply the type parameters directly to `generateHList` using `@`, removing the need to supply any values for type inference.

Final zip property

```
prop_zip :: Property
prop_zip = forAll (generateSignals @[Int, Char]) $ \signals ->
        let zipped      = zip (first signals) (second signals)
            state       = prepend zipped $ flatten signals
            predicate   = Always $
                    Now ((fst' <$> (Index First)) |==| (Index Second))
                    `And`
                    (Now ((snd' <$> (Index First)) |==| (Index Third)))
            result      = evaluate predicate state
        in result
```

Figure 33: The zip property in its final form

This implementation now allows us to call a single function to generate an HList of signals, directly supplying the types to generate with a particularly succinct syntax, as seen in Figure 33. The only piece of the puzzle left is to define a shrinking strategy.

4.6 Shrinking signals

Since the signal data structure in our implementation closely resembles a list, we draw significant inspiration from QuickCheck's shrinking strategy for lists, rather than reinventing the wheel [6]. However, because signals are by definition non-empty and involve time-dependent computations, we must adapt the standard approach to account for these properties. To do this, we convert signals into a shrinkable signal structure, that keeps both values and the clocks tied to each time steps' delayed computation with the type:

```
type TSig a = [(a, IntSet.IntSet)]
```

This allows us to reuse large parts of the shrink implementation for lists. We avoid shrinking the clocks of each time steps delayed computation, as the shrunken signals should reflect the same clock strategy as introduced in Section 4.5. We can now take the signal provided to the shrink function, turn it into a `TSig a`, shrink it, and convert it back into a `Sig a`, using the values and clocks saved in the structure. To achieve this, we define following functions:

Conversion functions to and from TSig

```
fromTSig :: TSig a -> Sig a
fromTSig [] = error "Testable signals are non-empty"
fromTSig [(x, _)] = x ::: never
fromTSig ((x, cl) : xs) =
  if IntSet.null cl
    then x ::: never
    else x ::: Delay cl (\_ -> fromTSig xs)

toTSig :: Sig a -> TSig a
toTSig (x ::: (Delay cl f)) =
  if IntSet.null cl
    then [(x, IntSet.empty)]
    else (x, cl) : toTSig (f (InputValue (IntSet.findMin cl) ()))
```

Now that we're able to convert signals into our desired data type, we turn to the design and implementation of the corresponding shrink function of Figure 34.

shrinkSignal function

```
shrinkSignal :: (a -> [a]) -> Sig a -> [TSig a]
shrinkSignal shr sig@(_ ::: (Delay cly _)) =
  if IntSet.null cly
    then shrinkOne testableSignal shr
    else concat [ removes k n testableSignal
                | k <- takeWhile (>0) (iterate (`div`2) n) ]
    ++ shrinkOne testableSignal shr
  where
    n = sigLength sig
    testableSignal = toTSig sig
```

Figure 34: The shrinkSignal implementation, given a shrinking function and a signal, to create a list of shrink candidates of type `[TSig a]`

The shrinking strategy of `shrinkSignal` is split into two steps:

- Segment removals
- Value shrinking

The first step removes (initially large) contiguous segments of progressively smaller sizes. The `removes` function decomposes the input list into pairs of prefix and suffix sublists, each removing a contiguous segment. The prefix and suffix are then concatenated to produce combinations with removed segments. This process is recursively applied to the sublists, with the `(++)` operator used to reconstruct simplified versions of the original list. In doing so, `removes` produces a list of shrink *candidates*.

Removing segments

```
removes :: Int -> Int -> TSig a -> [TSig a]
removes k n tupleLs =
  if k >= n
    then []
    else let xs1 = take k tupleLs
             xs2 = drop k tupleLs
         in xs1 : xs2 : map (xs1 ++) (removes k (n-k) xs2)
```

Figure 35: Removes produces the combinations of prefixes and suffixes, removing a segment of size k. Argument n is the length of the supplied list.

QuickChecks implementation of shrinking defines the smallest counterexample as the empty list [6]. As signals always has at least one element, we define the smallest possible counterexample of a signal to be one element.

To illustrate the shrinking process, we use the following example:

Test predicate

```
inputSignal = (0 ::: 2 ::: 4 ::: never) :: Sig Int
predicate   = Always $ Now ((Index First) < (Pure 3))
```

Figure 36: A `Sig Int` and a predicate to test that values are strictly smaller than 3. The syntax is pseudo code, not displaying the clocks of each delayed computation.

Following the logic of the `shrinkSignal` function, the first iteration of the shrink produces the segment removals. The result of removing these yields the lists:

```
1  [[0], [2,4], [0,2], [0,4]]
```

This list is converted to signals and fed back to the test, continuing the iterative approach of shrinking. However, we also want to shrink individual values, to find smaller counterexamples. For instance, the smallest possible counterexample in this example must be the signal of one element with the value 3. This values is not present in the signal, and to find this smallest counterexample, we must shrink values too. We implement the shrinking of individual elements in the `shrinkOne` function.

shrinkOne function

```
shrinkOne :: TSig a -> (a -> [a]) -> [TSig a]
shrinkOne [] _ = error "Testable signals are non-empty"
shrinkOne [(x, cl)] shr = [ [(x', cl)] | x' <- shr x ]
shrinkOne ((x, cl) : xs) shr = [ (x', cl) : xs
                                 | x'  <- shr x ] ++
                                     [ (x, cl) : xs'
                                      | xs' <- shrinkOne xs shr ]
```

Figure 37: Shrinking individual elements of the `TSig` type.

The function `(a -> [a])` provided to `shrinkOne`, represents a shrink value for a single element. We leverage the QuickCheck implementation of shrinking individual values of base types. The first iteration of shrinking the signal from Figure 36 is illustrated in 38.

Example use of shrinkSignal

```
[[0],[2,4],[0,2],[0,4],[0,0,4],[0,1,4],[0,2,0],[0,2,2],[0,2,3]]
```

Figure 38: First iteration of shrinking the signal presented in Figure 36

These new candidates are then converted into signals, and evaluated against the property of the test case. QuickCheck applies this evaluation iteratively, traversing the list of shrunken signals from left to right and breaking out as soon as it encounters a signal that violates the predicate. This strategy, combined with the strategy of adding the smallest shrunken signal leftmost in the list, makes the shrinker efficient, by being able to break out early on small failing counterexamples. For the example property in 36, QuickCheck will catch the first failing case of this shrink iteration as `[2,4]`. This is converted into a signal, and is tested against the property continuing the same shrinking procedure, and this cycle repeats until no further reduction is possible or until the shrunken signals no longer fail the property. Each of the iterations of shrinking this example, is illustrated in Figure 39.

```
> Shrinking signal of values [0,2,4]

Shrink iteration 1, candidate [0,2,4]:
  [
   [0], → Passed
   [2,4], → Failed
   ... (unused shrink values ommited from example)
  ]

Shrink iteration 2, candidate [2,4]:
  [
    [2], → Passed
    [4], → Failed
    ... (unused shrink values ommited from example)
  ]

Shrink iteration 3, candidate [4]:
  [
   [0], → Passed
   [2], → Passed
   [3], → Failed
  ]

Shrink iteration 4, candidate [3]:  (No failing cases for this iteration)
  [
   [0] → Passed
   [2] → Passed
  ]

*** Failed! Falsified (after 1 test and 3 shrinks):
    [3]
```

Figure 39: All shrinking iterations for the signal when testing the predicate from Figure 36.

Smaller counterexamples are found in the first three iterations of shrinking. As the fourth iteration only produces new signals that pass the test, we avoid further shrinking of the candidates. Figure 39 demonstrates the shrink iterations

that QuickCheck performs internally to minimize a failing signal. In typical usage, users are shown only the final output, indicating the test failure and the smallest counterexample.

As users may produce multiple signals for testing using the `generateHList` function (see Figure 29), we likewise provide a shrinker for this type, implemented with a typeclass to accommodate HLists.

ShrinkHList typeclass

```
class ShrinkHList as where
  shrinkHls :: HList as -> [HList as]

instance ShrinkHList '[] where
  shrinkHls _ = []

instance (Arbitrary a, ShrinkHList as) => ShrinkHList (a ': as) where
  shrinkHls (HCons x xs) =
    [ HCons x' xs | x'  <- shrink x ] ++
    [ HCons x xs' | xs' <- shrinkHls xs ] ++
    [ HCons x' xs' | x'  <- shrink x, xs' <- shrinkHls xs ]
```

Figure 40: Typeclass and instances for traversing an HList and shrink each signal within.

We use list comprehensions to define three complementary shrinking strategies, each producing a list of HLists. The three strategies involve:

- Shrinking only the head signal.
- Shrinking only the tail signals.
- Shrinking both head and tail signals.

By combining these, we generate shrinks that explore signals that are advanced independently from each-other.

4.6.1 Adaptive evaluation

The evaluator introduced in Section 4.1 runs tests over a default of 100 time steps. However, this number must always be less than or equal to the length of the state signal. If the number of time steps remains fixed, this can lead to a type of out-of-bound error. There are two main situations where this may occur:

- Shrunken signals: during test case shrinking, a generated signal may be shortened to be shorter than the fixed number of time steps evaluated.
- Predicate requirements: some properties might implicitly or explicitly require a minimum number of steps (e.g., temporal operators like “next”).

To avoid these issues, the number of time steps should either adapt to the signal length or ensure that all generated signals are always long enough to support the required evaluation.

minSigLengthForPred function

```
minSigLengthForPred :: Pred ts t -> Int -> Int
minSigLengthForPred predicate acc =
    case predicate of
      Not p           -> minSigLengthForPred p acc
      And p1 p2       -> minSigLengthForPred p1 acc `max`
                         minSigLengthForPred p2 acc
      Or p1 p2        -> minSigLengthForPred p1 acc `max`
                         minSigLengthForPred p2 acc
      Until p1 p2     -> minSigLengthForPred p1 acc `max`
                         minSigLengthForPred p2 acc
      Next p          -> minSigLengthForPred p (acc + 1)
      Implies p1 p2   -> minSigLengthForPred p1 acc `max`
                         minSigLengthForPred p2 acc
      Release p1 p2   -> minSigLengthForPred p1 acc `max`
                         minSigLengthForPred p2 acc
      Always p        -> minSigLengthForPred p acc
      Eventually p    -> minSigLengthForPred p acc
      After n p       -> minSigLengthForPred p (acc + n)
      _               -> acc
```

Figure 41: This function traverses a predicate, and returns the amount of time steps needed to evaluate this predicate once.

To check the number of time steps needed to check a given `Pred ts t`, we traverse the predicate, using the `minSigLengthForPred` function in Figure 41. Using an accumulating argument, we increment at each `Next` operator and add the integer provided to each `After` operator if any are present. Note that when writing predicates, we can use `Next` or `After` in different branches of the AST.

Predicate using two time steps

```
predicate = Always (
             Next ((Now (Index First)) |==| (Now (Index (Previous First)
                `And`
             Next ((Now (Index Second)) |==| (Now (Index (Previous Second))
             )
```

Figure 42: A predicate, using the `Next` operator twice, but as they are in different scopes, only two time steps are needed to evaluate.

For instance, in Figure 42, `Next` is used in two different branches of the AST, namely the left-side and the right-side of the `And` operator. As these are in different scopes of the predicate, we only need one additional time step to

test the predicate once. In other words, the smallest possible signal that fails this property must have a minimum of two time steps. We implement this taking the `max` of each branch, representing the maximum amount of time steps needed to evaluate each branch of the predicate. When the value returned by `minSigLengthForPred` exceeds the number of time steps in the state signal, we choose to pass this test by default. If we allowed these signals to fail, the implementation of shrink would continue reducing them, never finding a valid counterexample, but simply chasing smaller and smaller, untestable inputs. By defaulting signals of too few time steps as passing, we ensure that shrinking stops at the shortest signal that still meets the length requirement and genuinely fails the predicate. This produces the minimal counterexample, reflecting the actual cause of the failing test.

Additionally, we implement a function to evaluate a single timestep, without advancing on the delayed computation of the state signal. For instance, assume that the state signal has only two time steps, at the current state in the process of shrinking, and that the predicate under test is the one described in Figure 42. The evaluator now adapts to only evaluate two time steps, as we don't have anymore available in the state signal. Evaluating `Always` would advance on the state signal twice. However, we also express that we want to use a `Next` value within the predicate. This means that the evaluator will incorrectly try and advance on the state signal thrice. Therefore, we implement `evaluateSingle` as shown in Figure 43, to avoid advancement on a delayed computation, that never emits new values.

evaluateSingle

```
evaluateSingle  :: (Ord t) => Int -> Pred ts t -> Sig (HList ts) -> Bool
evaluateSingle timestepsLeft formulae sig@(x ::: _) =
  timestepsLeft <= 0 || case formulae of
            Tautology       -> True
            Contradiction   -> False
            Now expr        ->
              case evalExpr expr x of
                Pure b -> b
                _ -> error "Unexpected error during evaluation."
            Not phi         -> not (eval phi sig)
            And phi psi     -> eval phi sig && eval psi sig
            Or phi psi      -> eval phi sig || eval psi sig
            Until phi psi   -> eval psi sig || eval phi sig
            Next _          -> True
            Implies phi psi -> not (eval phi sig && not (eval psi sig))
            Always phi      -> eval phi sig
            Eventually phi  -> eval phi sig
            Release _ _     -> True
            After _ _       -> True
          where
            eval = evaluateSingle timestepsLeft
```

Figure 43: A function evaluating a single timestep. Exclusively used for shrunken signals, when the state signal no longer has delayed computations to advance on. Cases that normally advanced this state, is defaulted to return True or only evaluate in the current timestep.

Returning a default True or evaluating parts of the proposition, without advancing the state, stems from the same reason as mentioned earlier in this section. If we cannot find a counterexample in this evaluation, the test must pass, such that the shrinking mechanism provides the correct smallest possible counterexample.

5 Case Study: Testing a timer application

In this case study, we showcase PropRatt's capabilities by testing a real-world reactive application: an externally sourced GUI timer [9].

When the timer is initiated, a counter starts at 0 and increments once per second. A duration may be set by the user at any time using a slider in the GUI, and when the counter hits this value, it must stop incrementing. The user may also reset the counter by pressing a button. The counter will then start counting from 0 again. The program includes a signal that emits the timer's current second count at fixed intervals. We denote this as the counter signal. The full specification for the program is formally defined in [10].

Timer source code

```
everySecondSig :: 0 (Sig ())
everySecondSig = Delay (IntSet.fromList [2]) (\_ -> () ::: everySecondSig)

nats :: 0 (Sig ()) -> (Int :* Int) -> Sig (Int :* Int)
nats later (n :* max) = stop
    (box (\ (n' :* max') -> n' >= max'))
    (scanAwait (box (\ (n' :* max') _ -> (n' + 1) :* max'))
    (n :* max) later)

resetTuple :: (Int :* Int) -> (Int :* Int)
resetTuple (_ :* max) = (0 :* max)

setMax :: Int -> (Int :* Int) -> (Int :* Int)
setMax max' (n :* _) = ((min n max') :* max')

timerState :: Sig () -> Sig Int -> Sig (Int :* Int)
timerState (_ ::: rr) sliderSig@(_ ::: ss) =
    let     resetSig    = mapAwait (box (\ _ -> resetTuple)) rr
            currentMax  = current sliderSig
            setMaxSig   = mapAwait (box setMax) ss
            inputSig    = interleave (box (.)) resetSig setMaxSig
            inputSig'   = mapAwait (box ((nats everySecondSig) .)) inputSig
            counterSig  =
                switchR ((nats everySecondSig) (0 :* currentMax)) inputSig'
    in counterSig
```

Figure 44: The composition of signals produces the timerState signal, that contains a pair of the current timer value and the max duration value. This is the core logic of the timer example.

We use the implementation from the source that creates the counter signal [9]. However, to fit this into our model, we must modify the implementation of the signal, that produces values every second. We create this with a fixed clock channel of 2. This channel ensures that the `everySecondSig` signal ticks at a fixed rate, allowing other signals to tick at the same time and in between the intervals of this signal ticking. The user input signals are arbitrarily generated,

using the same strategies as presented in Section 4.5. The generated signals are passed as arguments to the `timerState` function and the result is prepended to the state signal of the test.

Up to this point, we have worked under the assumption that all signals in a test should have en equal probability of producing a value at any given time step. When applied to this example, the rate at which the reset signal produces a value, would possibly be more often than the use case in practice. This could yield a timer signal that never reaches its max duration, rendering any property involving the max duration futile.

In our tests, we want as much as possible to mimic a sample execution of the program, where the user may not reset as often as the counter signal produces values. We want to allow the timer signal to tick more often than the reset signal, and we therefore introduce a weighted arbitrary generator. This generator is implemented much like in section 4.5. The difference here is the strategy for generating clocks. We now generate clock channels that, with a higher frequency, emits the clock channel 3. When using the strategy of always advancing on the smallest clock during execution, introduced in Section 4, we are now able to simulate that users less often interacts with the reset button.

Weighted clock channel generator

```
genClockChannelWeighted :: Gen Int
genClockChannelWeighted = frequency [(1, pure 1),
                                     (1, pure 2),
                                     (50, pure 3)]
```

Figure 45: Generator with higher frequency of channel 3 simulates fewer signal updates because the evaluation semantics shown in Figure 14 selects the smallest channel to advance in a given clock.

We leverage QuickChecks `frequency` combinator to specify that 3 is generated with a higher probablity of 1 and 2. Additionally, the slider signal handling the maximum value that the counter can increment to, must not be entirely arbitrarily generated. As QuickCheck generates both positive and negative integers, we need to constrain this, as we don't want the maximum value of the timer to be a negative integer.

Constructing the slider and reset signal

```
genPair = do
    slider <- (arbitrarySigWith 100 (chooseInt (0, 100)) :: Gen (Sig Int))
    reset <- (arbitrarySigWeighted 100 :: Gen (Sig ()))
    return (reset, slider)
```

Figure 46: Arbitrary generation of slider and reset signals, using a custom range of integers and a weighted clock channel strategy.

The setup for all tests will be structured as shown in Figure 47.

Setup for timer testing

```
prop_?? :: Property
prop_?? = forAllShrink genPair shrink $ \(reset, slider) ->
        let counterSig  = timerState reset slider
            state       = prepend counterSig $ prepend reset
                          $ singletonH slider
            predicate   = ??
            result      = evaluate predicate state
        in counterexample (show state) result
  where
    genPair = do
      slider <- (arbitrarySigWith 100 (chooseInt (0, 100))) :: Gen (Sig Int)
      reset <- (arbitrarySigWeighted 100 :: Gen (Sig ()))
      return (reset, slider)
```

Figure 47: Setup for testing timer example. `singletonH` is a utility function to produce a singleton HList of a single signal.

The timerState function returns a signal of pairs, representing the counter and the max value. The leftmost value of the pair corresponds to the timer, incrementing at fixed intervals, and the rightmost value corresponds to the maximum value of the timer. With this setup, we test the following properties:

Property: State is correctly initiated At first, we examine a simple specification of the program. For the timer to work as intended, it must be initiated correctly.

Initial state is correct

```
predicate = Now ((fst' <$> (Index First)) |==| (Pure 0))
             `And`
            (Now ((snd' <$> (Index First)) |==| (Index Third)))
```

Figure 48: The initial state of the timer program must have the counter signal start at 0 while the maximum of the pair signal is the same as the maximum set by default in the maximum signal.

The timer should always start its counter at 0, and the pair signal at first index of the state, should reflect this, as well as it should reflect the default maximum value. This property should hold for all timer programs, and represents a basic functional property based test of the program.

Property: Counter Never Exceeds Maximum Value The intended behavior of the timer is that the counter signal must stop counting, as soon as it hits the maximum value. Therefore, it must be true for all programs, that the counter signal is equal to or smaller than the maximum signal. We express this with the following predicate:

Counter Never Exceeds Maximum

```
predicate = Always $
                      Now ((fst' <$> Index First)
                           |<=|
                           (snd' <$> Index First))
```

Figure 49: A predicate expressing the property that the counter never should exceed the maximum value set by the slider.

This property works on the timer state signal exclusively, and verifies that the first value of the pair, namely the counter signal value is always smaller than or equal to the second value of the pair.

Property: Counter is strictly monotonically increasing The timer should always be monotonically increasing. In other words, the counter must continuously increment or stop once the maximum value has been reached. We express this in the following property:

Strictly monotonically increasing

```
Always $
    Next $
        Implies
            (
            (Now (Ticked First)) `And` ((Not (Now (Ticked Second))
            `And`
            (Not (Now (Ticked Third)))))
            )
            (
            Now(
                ((fst' <$> (Index First))
                |>|
                (fst' <$> (Index (Previous First)))))
             )
```

Figure 50: A more complex specification, expressing that the counter signal always strictly monotonically increases its value.

This property asserts that if the counter signal is the only signal that ticks in a given timestep, then its current value must be strictly greater than its value in the previous timestep. In the cases, where the maximum or the reset signal produces a value, the counter signal must use its previous value, when it hasn't produced a new value itself. This is the intended behavior, when maximum or reset is interacted with at time steps, where the counter signal doesn't advance. These properties tested some of the basic properties, ensuring that the behavior of the timer works as intended. Now lets examine a property that could find more subtle bugs, caused if concurrent interactions aren't handled properly.

Property: Concurrent interaction with reset and maximum value updates accordingly We now investigate a more complex scenario involving concurrent signal interactions. The timer's logic must correctly handle concurrent inputs, like updating the maximum and resetting the timer, without violating its invariants. To capture this, we express the following property:

Concurrent events should be handled correctly

```
Always $
    Implies
        (
            (Now ((Ticked Second)))
            `And`
            (Now ((Ticked Third)))
        )
        (
            (Now (((Index Third)) |==| (snd' <$> Index First)))
            `And`
            (Now ((Pure 0) |==| (fst' <$> Index First))))
        )
```

Figure 51: Testing concurrent user interactions.

This property states that, if both the second and third signal updates, representing the interaction with the reset and maximum value, then it implies that the maximum value has been updated, as well as the counter value has been set to 0. If not implemented correctly, concurrency issues like this can be difficult to catch.

All of the properties presented here are tested, yielding no counterexamples. Although these results increase our confidence in the correctness of the timer GUI application, they do not amount to a formal proof of correctness. Nevertheless, the specification language is sufficient in expressing and evaluating the properties we wish to write.

6 Discussion

6.1 Related work

While PropRatt is unique in the way it enables testing of asynchronous behaviors in Async Rattus, similar approaches has been used to test other FRP languages.

QuickStrom, another temporal specification language, also builds on LTL for PBT [11]. For instance, splitting the Next operator into Required Next, Weak Next and Strong Next. It builds upon RV-LTL which extends predicates from returning true/false, to a logic that has four values [12]. This logic determines whether predicates are *presumptively* or *definitely* true/false. Assume that a predicate evaluates to true, but only because no counterexample is found in finite time, then this logic would define it as *presumptively* true instead. To control this behavior, QuickStrom requires the user to specify the exact number of time steps to be checked. We have had similar considerations in designing PropRatt, to be more expressive about why a predicate holds true, but these have been left for future work of the framework.

Not unlike PropRatt, both Yampa and QuickStrom leverage QuickCheck to generate input streams for testing [13] [11]. However, these frameworks operate in purely synchronous settings and users explicitly choose strategies, such as defining length and distribution of the generated data. In contrast, PropRatt is specifically designed for asynchronous FRP, with a focus on reasoning about how signals evolve in parallel with one another over time.

QuickRat is a PBT framework for testing Rattus [3]. While having different design choices of implementation, its builds on some of the same concepts as Yampa and QuickStrom [13] [11]. QuickRat aims at testing synchronous Streams in Rattus, where one global clock determines when streams emits values. Here, two state machines are presented to be used for generating arbitrary signals using LTL predicates. This differs from our implementation, as we arbitrarily generate our input based on types provided by the user, rather than specifically generating them for a use case specified by the LTL predicate.

6.2 Limitations

We now discuss the limitations we have found in our approach.

6.2.1 Dynamic signal combinators

Signal combinators that dynamically switch to another signal at runtime are particularly tricky to test. To effectively test these, we must be able to get answers to the following questions:

- *Did the switch happen at the correct time?*
- *Are the values resulting from the switch correct?*

Figure 52 shows the jump signal combinator, which is an example of such a function. At the moment the signal "jumps", one may wish to test that the jumped signal equals the signal bound in the boxed function argument `Box (a -> Maybe' (Sig a))`. Arbitrarily generating a signal in a pure function presents its own host of challenges, but we would also have to predict when the boxed function returns the `Just'` constructor.

Jump defintion

```
1  jump :: Box (a -> Maybe' (Sig a)) -> Sig a -> Sig a
2  jump f (x ::: xs) = case unbox f x of
3                        Just' xs' -> xs'
4                        Nothing' -> x ::: delay (jump f (adv xs))
```

Figure 52: Jump signal combinator function implementation from the `AsyncRattus.Signal` module. Jump dynamically switches to the signal bound in the function argument if the functions holds.

It is tempting to add the signal that the function returns to the state signal as well, but this could cause prematurely advancements on a signal that normally wouldn't advance until returned by the function. This limitation can be worked around by returning a constant signal in the boxed function, see Figure 53. By knowing the values produced by the signal that is jumped to, the instant in which it jumps can be predicted, and it can be tested that signal is constant from that point on. While this solves the problem, it is not very intuitive nor elegant.

Jump property

```
1  prop_jump :: Property
2  prop_jump = forAllShrink (generateSignals @Int) shrinkHls $ \intSignals ->
3     let  jumpFunc    = box (\n -> if n > 10
4                                    then Just' (const 1)
5                                    else Nothing')
6          jumpSig     = jump jumpFunc (first intSignals)
7          state       = prepend jumpSig $ flatten intSignals
8          predicate   = Always $
9                          Now ((Index First) |==| (Index Second))
10                         `Or`
11                         (Always $ Now ((Index First) |==| (Pure 1)))
12         result      = evaluate predicate state
13       in result
```

Figure 53: Jumped signal should switch once the boxed predicate function returns `Just'`.

Similar challenges are found in signal combinators that takes functions as arguments, guarded by the later modality. Such example is switchS, see 54. Here

the function argument guarded by the later modality presents the problem of having to know *when* the delayed computation arrives, which is tricky to predict because it is tied to the smallest clock strategy. A value of type `O (a -> Sig a)` can be produced as follows:

```
Delay (IntSet.fromList [1,2,3]) (\_ a -> const a),
```

where the signal produced by the function is a constant of the argument supplied. Because the clock channels are known ahead of time, the DSL could possibly be extended to include a construct that allows users to express a statement that is evaluated once the delayed computation ticks. It is however not desirable to expose the internal primitives of Async Rattus needed to construct a `O (a -> Sig a)` that arrives at a specific channel, as the level of abstraction is too low (see Section 3.1).

SwitchS defintion

```
switchS :: Stable a => Sig a -> O (a -> Sig a) -> Sig a
switchS (x ::: xs) d = x ::: delay (case select xs d of
                                      Fst   xs'  d'  -> switchS xs' d'
                                      Snd   _    f   -> f x
                                      Both  _    f   -> f x)
```

Figure 54: SwitchS definition from `AsyncRattus.Signal` module. SwitchS switches once, where the switched signal may depend on the last value of the first signal.

The later modality introduces another problem, wherein if the user wishes to prepend a later signal `O (Sig a)`, to fit the state abstraction from Section 4.2, the delayed computation must be constructed as a signal that contains an arbitrary "dummy" value (because signals always carry a current value, see 1. Figure 55 shows how this is achieved. This yields another awkward DSL rule where values of type `O (Sig a)` must be referenced in the specification under the `Next` constructor to advance the signal once to avoid using the dummy value in the test.

PrependLater definition

```
prependLater :: (Stable t, Stable (HList ts), Falsify ts) =>
O (Sig t) -> Sig (HList ts) -> Sig (HList (Value t ': ts))
prependLater xs (y ::: ys) =
  HCons (Current (HasTicked False) Nil) y ::: prependAwait Nil xs y ys
```

Figure 55: To construct a signal from `O (Sig a)`, the head of the signal must be a dummy value that must be discarded in the test.

6.2.2 Liveness properties

As discussed in Section 2.4.2, liveness properties, by definition, require infinite execution traces. This is because any finite counterexample to a liveness property can always be refuted by a later time stepwhere the property might hold. As illustrated in Figure 14, the evaluation semantics for liveness operators default to true if no counterexample is found within a bounded execution, since all evaluations are inherently limited to finite traces. This limitation significantly limits the practical applicability of many LTL operators.

6.2.3 Absence of bugs

PBT reveals bugs if they are present, but cannot prove their absence. This is also the case for PropRatt.

6.3 Future work

To enrich the DSL we could distiguinish between safety and liveness properties by introducing new type definitions for these, parsing a `Pred` 5 to be either a `SafetyPred` or `LivenessPred`. These types could be constructed via. smart constructors that identify which temporal operators are used and returning the corresponding type. Upon evaluation of a liveness predicate, the user could be informed of the limitations such that they can act accordingly.

Another possible improvement is to allow users to configure or influence the clock strategy. This might involve making several clock generators that can be combined to specify signals that tick at different rates relative to each other. Exposing this control requires intuitive abstraction, such as the following suggestions:

- Policy-based configuration, where users select from predefined signal update strategies (e.g., uniform, weighted, probabilistic, domain-specific presets).
- Declarative signal properties, where signals are annotated with metadata (e.g., "high frequency", "rarely ticks"), and DSL evaluates accordingly.

The challenge lies in how to make such an interface without forcing users to reason about low-level clock management. Providing an interface at too low a level introduces complexity and undermines the abstraction at which properties are written currently.

7 Conclusion

In this thesis, we introduced PropRatt, a specification language and model to facilitate the testing of Async Rattus. PropRatt is implemented as an eDSL, allowing users to express temporal predicates over signals. The LTL-inspired specification language enables the ability to reason about correctness of Async Rattus programs. To achieve this, we introduce a model that effectively reflects how signals of different types behave during an Async Rattus program execution. The model is realized using type-level programming techniques enabled by several Haskell language extensions. We leverage property based testing with QuickCheck, to arbitrarily generate inputs, in order to evaluate predicates over a large set of signal interleavings. We implement shrinking strategies for key data types, to shrink to smallest counterexamples, upon test case failure. Through the testing of a GUI application, we demonstate how to use the library in practice. The case study reveals situations that raise questions about whether our strategies accurately capture user interactions across all domains. We suggest different approaches, that might be useful for specific domains. We discuss limitations of the approach, including the inability to check liveness properties in finite time. Additionally we discuss limitations of not supporting certain combinator functions, specifically when they dynamically switch behavior over time, which the current design of the model cannot elegantly encapsulate.

Acronyms

ADT Algebraic data type. 13

AST Abstract syntax tree. 12, 14, 31

DSL Domain specific language. 12

eDSL Embedded domain specific language. 7, 12, 14, 16, 61

FRP Functional reactive programming. 4, 5, 7

GADT Generalized algebraic data type. 13, 14, 20

GUI Graphical user interface. 4, 5, 8, 51, 56, 61

LTL Linear temporal logic. 10, 11, 16, 21, 57

PBT Property-based testing. 6, 12, 19, 57, 60

References

[1] R. G. Bahr P., Houlborg E. (2023) Asynchronous rattus: A functional reactive programming language with multiple clocks. Accessed: 2025-06-01. [Online]. Available: https://hackage.haskell.org/package/AsyncRattus-0.2.0.2/src/docs/paper.pdf

[2] P. Bahr, "Asyncrattus: A functional reactive programming language," 2020, accessed: 2025-05-15. Specific reference to `src/AsyncRattus/Signal.hs`. [Online]. Available: https://github.com/pa-ba/AsyncRattus

[3] L. D. Jensen, "Property based testing of functional reactive programs using linear temporal logic," Master's Thesis, IT University of Copenhagen, February 2023, accessed: 2025-06-01. [Online]. Available: https://bahr.io/students/Property%20Based%20Testing%20of%20Functional%20Reactive%20Programs%20using%20Linear%20Temporal%20Logic.pdf

[4] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for guis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 411–422, accessed: 2025-06-01. [Online]. Available: https://doi.org/10.1145/2491956.2462161

[5] C. Elliott, "Push-pull functional reactive programming," 2009, pp. 25–36, accessed: 2025-06-01. [Online]. Available: https://doi.org/10.1145/1596638.1596643

[6] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, p. 268–279, Sep. 2000, accessed: 2025-06-01. [Online]. Available: https://doi.org/10.1145/357766.351266

[7] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd ed. Cambridge, UK: Cambridge University Press, 2004, accessed: 2025-06-01.

[8] E. A. Josef Svenningsson. (2015) Combining deep and shallow embedding of domain-specific languages. Accessed: 2025-05-12. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842415000500#ab0005

[9] H. A. Bahr P., Disch J. (2024) Functional reactive gui programming with modal types. Accessed: 2025-06-01. [Online]. Available: https://bahr.io/pubs/files/widgetrattus-paper.pdf

[10] E. Kiss. (2018) 7guis: A gui programming benchmark. Accessed: 2025-05-26. [Online]. Available: https://eugenkiss.github.io/7guis/tasks

[11] W. O. O'Connor L. (2022) Quickstrom: property-based acceptance testing with ltl specifications. Accessed: 2025-05-29. [Online]. Available: https://dl.acm.org/doi/10.1145/3519939.3523728

[12] S. C. Bauer A., Leucker M. (2011) Runtime verification for ltl and tltl. acm transactions in software engineering methodology 20, 4, article 14. Accessed: 2025-05-29. [Online]. Available: https://doi.org/10.1145/2000799.2000800

[13] N. H. Perez I. (2020) Runtime verification and validation of functional reactive systems. Accessed: 2025-05-29. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/runtime-verification-and-validation-of-functional-reactive-systems/875DE7B51D38C418739B441874EB23D2