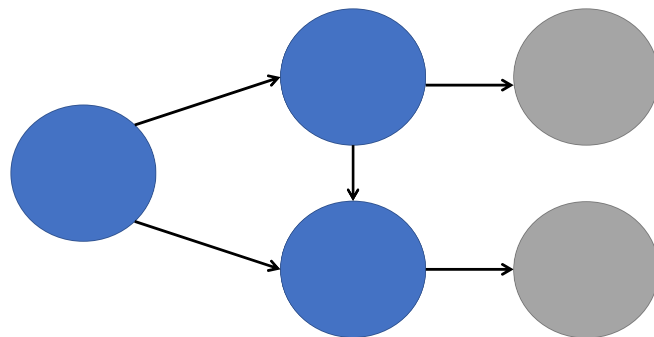
IT UNIVERSITY OF COPENHAGEN

Property Based Testing of Functional Reactive Programs using Linear Temporal Logic

---

Supervisor: Patrick Bahr



By Lukas Dannebrog Jensen, lukj@itu.dk

Submitted: June 1, 2023

Contents

1 Introduction **6**

1.1 Motivation 6

1.2 Key Ideas 8

2 Background **11**

2.1 Functional Reactive Programming 11

2.2 Property Based Testing 13

2.2.1 QuickCheck 14

2.2.2 The Gen Monad 15

2.3 Linear Temporal Logic 18

2.3.1 Syntax 19

2.3.2 Semantics 20

3 Implementation **22**

3.1 Analysis 22

3.2 LTL Syntax 24

3.3 Stream Evaluation 25

3.3.1 Equality 25

3.3.2 LTL evaluation 26

3.3.3 The acceptor 27

3.4 Stream Generation 30

3.4.1 Generating from LTL 32

4 Discussion **35**

4.1 Related Work . . . 35

4.1.1 Evaluating signals . . . 35

4.1.2 Generating signals . . . 35

4.2 Core of QuickRat . . . 36

4.3 Limitations of QuickRat . . . 38

4.3.1 Expressiveness of LTL . . . 38

4.3.2 Absence of bugs . . . 38

4.3.3 Feasibility . . . 38

4.3.4 Liveness and safety . . . 40

4.4 Possible Future Work . . . 40

5 Conclusion **42**

Abstract

Functional Reactive Programming (FRP) is a paradigm, which due to its ability to model time dependent problem domains has become increasingly popular. A general feature of functional programming, is the of this ease which Property Based Testing (PBT) can be applied. In PBT properties of function outputs, which should hold on any given input are specified. Previously Linear Temporal Logic (LTL) has been used to evaluate outputs, LTL is a superset of propositional logic and is commonly used in software verification, which makes it expressive and well understood.

So far testing FRP programs, has been been a challenge because of difficulties in generating suitable test inputs as well as output. In this thesis I present my tool called QuickRat, that together with QuickCheck allows for automatic generation of test inputs for the FRP-language Rattus, which is a Domain Specific Language embedded in Haskell. I will present this novel ability to generate many streams, which all satisfy a LTL-formulas, and how it is applied in PBT.

Problem Statement

For thesis on Property-Based Testing of Reactive Programs
written by Lukas Dannebrog Jensen
supervised by Patrick Bahr
submitted on February 24, 2023

Programming at any levels will inevitably produce some erroneous code. The cost of programming errors found post-release can be everything from a passenger airplane crashing to millions of annoying browser reloads. Therefore testing software to find errors pre-release is hugely important.

In the most common form of testing, called unit testing, the programmer defines a set of test cases that are then checked. Unfortunately generating suitable test cases and expected output to check against is laborious, and difficult. property-based testing (PBT), takes a different approach. In PBT the programmer defines a set of properties, that specify desired behaviors for many if not all different input values, and then tools like QuickCheck (Claessen and Hughes, 2000) generate random test cases and checks fulfillment of the properties. $f_{old}(x) = f_{new}(x) \forall x$ is an example of a property that works for doing regression testing. QuickCheck is implemented as a Domain Specific Language (DSL) in Haskell and utilizes Haskell's type system to come up with suitable test cases. However, some programming paradigms implemented in specific languages does not have tools like QuickCheck at hand.

An example of such a language is the functional reactive programming language Rattus (Bahr, 2022). Rattus is also implemented as a DSL in Haskell. In FRP the basic components are signals. Signals models time-varying values, like prices, physical measurements, disease incidents etc. In functional programming a signal is represented as function that a takes a time and returns a value, this is an issue because a property on a signal that holds now, might not hold tomorrow. "Arrowised" FRP languages solves this by using signal functions instead of signals themselves (Hudak et al., 2003), and there PBT has been be employed (Perez and Nilsson, 2020). Unfortunately signal functions quickly leads to reduced code comprehensibility.

The goal for this thesis is to create a prototype for a PBT tool for Rattus, inspired by the specifications and implementations of QuickCheck in Haskell and the PBT implementation for an Arrowised language. My deliverables will be a repository with the source code for the prototype along with the architectural description, key parts of which will be presented in a thesis document along with examples of usage.

1 Introduction

The repository can be found at https://github.com/LukasWing/QuickRat, while the following thesis contains a presentation of QuickRat and the ideas behind it.

1.1 Motivation

Most people has had an idea for a piece of software at some point. This could be anything from an egg timer app to a program that plays chess or perhaps one that can tell you were to find an empty parking slot. Getting such ideas takes only little effort, but it takes more effort to get it to work once, even more effort to get to work reliably, and even more so to guarantee for correctness of such programs. On this continuous scale of reliability, each program ends up at a value determined by the balance between work effort and correctness. What I will be presented in this thesis is a testing tool called QuickRat, which in a specific domain of programming, can lower the work effort to make programs more reliable. Which might move software higher up the reliability scale.

QuickRat is written in the pure functional programming language Haskell and is intended for testing functions that uses signals as the are defined Functional Reactive Programming (FRP). FRP is a programming paradigm originally described for animations (Elliott & Hudak, 1997). In FRP time varying behavior is the primary concern, to model these behaviors, time dependent functions, called signals. The signature of such functions are $s : \mathbb{R} \to C$, where time is represented as a real number and C can be any co-domain.

Signals models many aspects of the world well, since so much varies with time, and therefore implementing functions that take signals as input or returns them as outputs is of key interest.

Consider for instance a factory with a boilertank, that produces a signal with a pressure sensor reading. For safety reasons we need a function that returns a boolean signal, to a valve, which can relief the pressure. The signature would be

$$\begin{aligned} f &: s_p \to s_v \text{ where} \\ s_p &: \mathbb{R} \to \mathbb{R}_+ \\ s_v &: \mathbb{R} \to \{0, 1\}. \end{aligned}$$

By modelling with signals we get a comprehensible mathematical signatures, which are

best mapped to code written in FRP.

There exists a variety of FRP libraries/languages, one of which is Rattus (Bahr, 2022), which I used in this thesis another is Yampa (Hudak et al., 2003), which I will also mention later on.

Even though signals can be implemented well using FRP, implementation can still have errors and for a safety critical system such as this, we need reliable code. And to make reliable code, it is hard to avoid testing. Testing can be done in many ways, if you see the semantics of a functions as something that takes an input, and then returns an output, then your natural way to test it would be to define one or more inputs, and check that the output is equal to the expected. That is called unit testing. In the boiler example, if the valve act only according to the current reading to determine whether the valve should be open the unit-testing would suffice. But it might be more suitable to have a specification such as:

1. If at some point the the pressure gets above some threshold 10 for two consecutive measurements, then the valve is activated in the following step.
2. Once the valve is activated it is on until the pressure gets below 10.
3. If the pressure never gets above c the valve is never activated.

because such specifications are more focused on temporal properties of f these are very hard and cumbersome to test with traditional unit testing. But a promising solution is to use Property Based Testing.

In PBT the tester does not need manually create the entire set of test-cases. Instead a property that should hold for any outputs from for a subset of possible inputs is given. For instance a simple property of a sorting functions is that the output is actually sorted, and another may be that the size is unchanged. Unfortunately both specifying the input and the properties, when input or output are signals, usually takes a lot of work, because we need to deal with variability over time, rather than static inputs. For so called Arrowised-FRP there has been made an implementation (Perez & Nilsson, 2020), that makes property creation on signals easier. These properties are specified in a time-dependent logic known as Linear Temporal Logic (LTL). LTL is for making formulas, such that any temporal behavior is either accepted or not, just like formulas in propositional logic are true or false for any complete value assignment, the only is change is that the truth assignment varies with time.

While LTL is shown as promising (O'Connor & Wickström, 2022; Perez & Nilsson, 2020), extending its use into the process of generating suitable inputs has not been achieved, neither with Rattus or other FRP libraries. This thesis introduces QuickRat, which tries to accomplish this.

1.2 Key Ideas

QuickRat consists of three parts, which are sketched as the left hand side of 13.

- A definition of LTL syntax, as a type in Haskell (`TPred a`).
- A function that from any LTL formula can create a signal acceptor (`mkAcceptor :: TPred a -> Acceptor a`).
- A function that for any LTL formula can create a signal generator (`mkTransducer :: TPred a -> Transducer a`).

A simple code example which test property 1 and 3 with 100 generated cases is shown in listing 1 (which is captioned as Figure 1).

```
f :: Str Int → Str Bool
f aStr = ...
--- Testing specification 1 and 3 with helper function.
ghci> quickCheck $ ltlProperty
                        f
                        (Atom (>10) `And` Next (Atom (>10)))
                        (Next (Next (Atom id)))
+++ OK, passed 100 tests.

ghci> quickCheck $ ltlProperty
                        f
                        (Always (Atom (<10)))
                        (Always (Atom not)))
+++ OK, passed 100 tests.

-- Testing specification 3 without ltlProperty
prop_f_pBelow10_vAlwaysOff :: Property
prop_f_pBelow10_vAlwaysOff =
    forAll
        (trans $ mkTransducer $ Always (Atom (<10)))
        $ accept (mkAcceptor $ Always (Atom not)) . f
ghci> quickCheck prop_f_pBelow10_vAlwaysOff
+++ OK, passed 100 tests.

```

Figure 1: Property based testing off a pressure valve using QuickRat, using helper and explicit construction of state machines. All code snippets can found in Core or CoreTest.hs, where proper import are also given. '...' denotes that some code has been left out for brevity, 'ghci¿' denotes that what follows is the output of GHC-interactive.

The `Atom` represents some property of signal output, that is true or false, in this case the first atom simply represents whether the output is smaller than ten, and the second atom represents whether output is false. While listing 1 looks simple there are some caveats. Some caveats stems from the fact, that signals are infinite, where checking is only done for a finite part, while others stems from input generation being infeasible, when atoms are only true for a small fraction of tried elements in the streams, I will treat those caveats in the discussion section.

As presented in the listing my implementation depends on a transducer and an acceptor, both of which are state emachines. The acceptor is constructed from an LTL formula and running it can make it reject or accept any signal which satifies according to the formula, using state machines to solve this is specification problem is just a different way of making what has previously done (Perez & Nilsson, 2020), the novelty come with how the acceptor used together with the transducer. The transducer uses the acceptor as a guide to create signals, for now you may think of the transducer, as something that continues

to generate elements in signals, where the elements are is approved by the acceptor to eventually create a signal that will satisfy the original LTL formula. For instance when running the test for specification 3 the transducer uses an acceptor under the hood to make sure to only generate signals with $f : \mathbb{R} \to (-\infty, 10)$, this is also reflected in 13 which I encourage again to look at.

2 Background

2.1 Functional Reactive Programming

FRP is a programming paradigm, where signal is the primary type. Here are some examples of things that are often modelled with signal in FRP.

1. Animations are pictures that vary with time, to run an animation is supply a signal with the time (usually every 25'th second) and display the returned picture.
2. User inputs, can be represented as signal that tells which buttons are pressed, by these in a collection of button Ids.
3. Prices on everything from bitcoins to bacon slices, is representable as signals from time to some currency of interest.

It is tempting to use signals defined as `data Signal a = Time -> a` However, for real-time systems signals raises issues, with both past and future values. The past value concern is that the system is prone to space leaks, if the signal function should be able to return all possible measurements until now. For future values the concern is, that we do not know the measurement in the future, but there is nothing preventing time arguments where that are larger than the current. The future value concern raises issues with causality, namely that some value now might depend on a value, that is not available yet.

Arrowised FRP deals these two concerns, by dealing only with signal indirectly using so called Monadic Stream Functions as their representatives. This approach protects from improper use. unfortunately working through such representatives can be quite be quite which can be appreciated by looking at some basic code snippets from Yampa (Perez & Nilsson, 2020)2.

```
-- Type definition of Signal Function
data SF a b = SF (DTime -> a -> (b, SF a b))
-- Type definition of the Monadic Stream Function
newtype MSF m a b
-- Function to get a value of type b.
step :: Monad m => MSF m a b -> a -> m (b, MSF m a b)

```

Figure 2: Examples displaying complexity of MSF (Perez & Nilsson, 2020)

To be fair, because Monadics Stream Function are instances of Arrow, their exists combinators and special notation (Paterson, 2001), that alleviates this problem, but there will be complexity under the hood, that inevitable will show up in development.

A way to deal with this complexity, is to reallow the developer to work with signals directly, while mitigating prior concerns by the use of a modal type operator `O`, and a type system that enforces the correct use of such. (Bahr, 2022) For example consider the naive and protected implementations and use of a discrete signal function in listing 3. Note that the notionally continuous `Signal a` is replaced by a discrete `Str a` in Rattus, these formulations are similar but the latter allows us to move focus away from timestep size and clocks, which is an entire project in itself, it is however important to note that the head of a stream is the current values, and the tail are values available in only the future.

```
1       -- Naive definition
2       data Str a = a ::: Str a
3       currentVal (h ::: _) = h
4       -- Should be impossible for real-time systems.
5       futureVal (h ::: strTail) = currentVal strTail
6
7       -- Rattus' definition
8       data Str a = a ::: O (Str a)
9       currentVal (h ::: _) = h
10      -- Raise a type error in Haskell
11      futureVal (h ::: delayedStrTail) = currentVal delayedStrTail
12      -- Compiles in Haskell but raises a type error in Rattus.
13      futureVal' (h ::: delayedStrTail) = currentVal (adv delayedStrTail)
14
15
```

Figure 3: Example displaying usage of the modal type operator.

The `futureVal` type error can mitigated since the exists a function `adv :: O a -> a` which will convert the delayed stream to regular stream, but the compiler from Rattus, restricting the use adv. The general principle of these restrictions (Krishnaswami, 2013) is one of causality and is quite simple: It is allowed to use a delayed value, only in expressions whose value will be computed in a later time step. `futureVal'` fails to follow this principle, because it tries to acquire a value in a later time step now.

Unfortunately the principle raises a question, how do we then tell the Rattus compiler that an expression is to be computed in the next time step? The answer is the semantic inverse of `adv` and has the according type `delay :: a -> O a`, to appreciate how they work together consider the following function, which maps every element of the input to

the output stream using f.

```
1        map :: Box (a → b) → Str a → Str b
2        map f (h ::: t) = unbox f h ::: delay (map f (adv t))
3
```

Figure 4: Map definition in Rattus, the Box denotes that f is time independent and safe to use in delay

Here `adv` is used inside the expression whose value will be passed on to `delay`, and thereby the causality principle. `map` also uses the box modality, which marks the inner function as one that safe to use inside delay, because of its time-independency. In practice the Rattus compiler is active on functions or modules using Haskell's compiler directives. There are more things to the type system of Rattus, one of which is how it prevents space-leaks, but going into details of such is beyond the scope of this project. Actually we will deliberately not activate the Rattus compiler, because we generate streams for testing rather than receiving them in real-time.

2.2 Property Based Testing

Take for instance a function `fAvg` for making some kind of average between coordinates og two GPS signals, we expect that it will always return a point in rectangle cornered by those two. If you were to make unittests you would probably check that all the variables in listing 5:

```
1     tAvg1 = isBetween (0,0) (0,1) $ fAvg (0,0) (0,1)
2     tAvg2 = not $ isBetween (0,0) (0,0) $ fAvg (0,0) (0,1)
3     tAvg3 = isBetween (0,0) (1,0) $ fAvg (1,0) (0,0)
4     -- etc.
5
```

Figure 5: Unit test of a GPS-averager

This leads to large amount of time spent producing arbitrary test cases. In some cases the actual test cases might be good and covering, however this is rarely the case, and either way there will inevitably be variability in the quality of such unit tests. Properties are also closely related to models of the problem domain. For instance the property stated in

the example, resembles a verbal description of an average much better than all the test cases.

When testing a property the ideal would be to check that the property holds for all combinations of inputs, libraries such as SmallCheck (Runciman et al., 2008) does this. However this is only feasible, in the sense that the running time is realistic, when the size of the function-domain is very small. As the number of input cases is $O(2^n)$ where n is the input size in bits. So instead of generating values covering the entire function-domain, the goal is to use random generator that picks values from a distribution, such that all possible execution paths are hit. Knowing exactly which distribution to pick from is a science in itself, however you can go a long way by treating each input as isolated, and let the generator only depend on the type.

2.2.1 QuickCheck

Having defined what properties are and a strategy for asserting them we need an actual implementation of it conepts. Many programming languages has one or more tools for performing PBT, examples are FastCheck for JavaScript, FsCheck for F# and JUnit-QuickCheck for Java, they differ in implementation. But they are all inspired from the grandfather of PBT QuickCheck (Claessen & Hughes, 2000). Since QuickCheck is written in and applicable for Haskell, it can take advantage of its type system and pureness. The pureness guarantees that the output of a function only depends on the input, this increases the reliability of the tests. The type system for Haskell ait generating test cases of the correct type, to appreciate this consider listing 6.

```
1   fAvg (x1, y1) (x2, y2) = ((x1 + x2) / 2, (y1 + y2) / 2)
2   ghci> quickCheck (λp1 p2 → isBetween p1 p2 $ fAvg p1 p2)
3   +++ OK, passed 100 tests.
4
```

Figure 6: Testing GPS-averager with QuickCheck

The sought property is expressed as the lambda passed to `quickCheck`. The type of `p1` and `p2` is inferred, from the function declaration to have type Num, but to illustrate how this works, I expand the expression a bit more, and add some type declaration.

```
prop_pointsAreBetween :: (Int, Int) -> (Int,  Int) -> Property
prop_pointsAreBetween p1 p2 = isBetween p1 p2 $ fAvg p1 p2
ghci> quickCheck prop_pointsAreBetween
+++ OK, passed 100 tests.
```

Figure 7: Property Based Testing of fAvg

A property in QuickCheck is implemented as the type `Property` which at its simplest, is just a boolean. The data for the 100 inputs were generated based the argument types for `prop_pointsAreBetween`. The default behavior is that values are generated using the function `arbitrary :: Gen a`. Generating random values within a Haskell function would break its pureness, but QuickCheck solves this problem by its clever way of using monads.

2.2.2 The Gen Monad

Since Gen is a Monad, along with many other types it is worthwhile to step back and go through what a Monad is, this is also worthwile because arrows in AFRP is a more general abstraction of Monads. Monad is a typeclass defined as in listing 8

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    -- Some methods excluded for brevity

```

Figure 8: Partial definition of the Monad typeclass

Monad models types that has both a context `m`, and a polymorphic value. There are two core methods of our concern. `return` takes a value and wraps it in a minimum context. In the case of Gen return `a` is just a generator for values of type `a`. `(>>=)` is also called bind. Bind returns the result of applying the second argument to the value first monad. Note that while the typeclass definition defines an interface with default implementation os some methods, the implementation of, return, bind and methods from Applicative is left for the programmer. There also exists Monad laws such as associativity, but these are not checked by the compiler. A practical way to use Monads in Haskell is with the do-notation, which is syntactic sugar for bind-expressions nested in others. You

can appreciate the do-notation by considering how you would implement a generator for complex numbers `Z`. Such an implementation is done in listing 9 using both `return` and `>>=`, and the do-notation.

```
data Z = Z {a::Float, b::Float}
genComplex = arbitrary >>= (λa → arbitrary >>= (λb → return Z {a=a, b=b}))
genComplex' = do
    a ← arbitrary
    b ← arbitrary
    return Z {a=a, b=b}

```

Figure 9: Constructing a custom Gen monad with and without do-notation.

I have already given example of how to make `Gen Z`. Here I will elaborate a bit more on why that works, and how it can indeed generate random inputs in a pure functional programming language. Consider the following hypothetical impure haskell function `rollDice nSides = randomInt 1 nSides` This function would return different results for identical inputs, which is not pure. But suppose that `rollDice` was called from `main`, where IO is handled, if needed we could just get a list of randombits from IO and make randomInt that read from it.

```
rollDice ::(Int, [Bool]) → Int
rollDice (nSides, randombits) =
     randomInt randombits 1 nSides

```

Figure 10: Fixing with random bits

But the implementation in listing 10 would only produce identical random numbers if called multiple times. To solve the issue we need to return from `randomInt`, information on which part `randomBits` is already used, perhaps by returning a tuple `(aRandomInt, unusedBitList)`. The type signature gets rather messy, we can prevent the by instead of returning the actual value could just produce a computation which was able to generate a random values following some specification.

The type of this computation is called a generator. The generator could then be called where a random-generator is actually available to be passed as an argument to the generator, so the arguments sequence of random values can advanced. Listing 11 shows

the relevant types in QuickCheck for doing just that. `Gen a` is the type of the generator, and `QCGen` is the type of the random-generator that QuickCheck supplies as an argument for `unGen`.

Although we will not directly call `unGen` it is worth noting the seconde integer argurment, which can used to control to size or complexity of the generated values, only some instances of Arbitrary uses the it, an example is `Lists` where it controls the length of the generated lists. In general a failing test case is easier the debug from if the input is simple, therefore QuickChecks will use increasing sizes.

```
newtype Gen a = MkGen {
    unGen :: QCGen → Int → a
}
generate :: Gen a → IO a

-- Usage example
diceRoller :: Int → Gen Int
diceRoller nSides = do
            i ← (arbitrary:: Gen Int)
            return (i `mod` nSides + 1)
ghci> generate (diceRoller 6)
3
ghci> generate (diceRoller 6)
6

```

Figure 11: Minimum example of Gen usage

In practice we do not need to call `generate` explicitly as done in listing 11, this done by the `quickCheck` which uses a supplied generator for the type, that matches input to the property that should hold. If have to use a custom generator like in 11 or 9, you need to pass in the your generator as first argument to the `forAll` function, like shown in 12.

```
forAll :: (Testable prop) ⇒ Gen a → (a → prop) → Property
prop_winner_diceRoll12_alwaysHouse =
    forAll
        (diceRoller 12)
        (λroll → houseWins roll == "House Wins")

```

Figure 12: Using QuickChecks forAll

An important way to reduces input complexity is to shrink test cases. The basic idea is

automate the error isolation, that a programmer would do, upon finding an error. Consider an example where a property failed to hold with the input `[-1, 2, -1, -2, 2, 3, 4, 2]` then a shrink strategy could be to recursively check whether halves of the array would also cause the property to not hold. Perhaps the error only occurs if the array contains two successive negative numbers, and hence succesfull shrinking would reduce the failing case to `[-1,-2]`. In general shrinkings are defined by implementing the `shrink :: a -> [a]` method, or use `forAllWith` which in addition to a Gen and a `(a -> prop)` take a function `a -> [a]` which returns a list of simpler inputs from the failing input.

2.3 Linear Temporal Logic

As I mentioned before, the simple way to unit-test is to test for equality, but for property based testing equality testing is only possible if another semantically equivalent correct function is already known. Most of the times this is not the case, and then usefull properties such as: the stream elements will eventually reach a certain value; the next three elements of the stream is 42; $P(x)$ always holds for the stream elements; any element the will eventually be followed by an x; x is never followed by y etc. is impossible to test with only stream equality. To test such we need a language to specify streams in a flexible way precise enough for testing. One such language is Linear Temporal Logic (LTL). LTL is best thought of as a time dependent superset of propositional logic, because it combines logical operators from propositional logic with modalities, that describes temporal behavior. Consider the following statement in propositional logic; if it rains today or tomorrow I will bring my raincoat.

$$p \vee q \implies r \text{ where}$$
$$p : \text{It rains today}$$
$$q : \text{It rains tomorrow}$$
$$r : \text{I will bring my raincoat}$$

The downside with such a formula is that it does not express that p and q are related since they both denote the fact that it rains, just at different times, surely we can fix it a bit by using predicate logic, but that introduces uneeded flexibility. In LTL the formula can rewritten so it only uses two atomic statements, at the expense of introducing the

next operator ($\mathbf{X}$), which can be read as "in the next time step".

$$\begin{aligned}&p \vee \mathbf{X}p \implies r \text{ where}\\&p : \text{It rains}\\&r : \text{I will bring my raincoat}\end{aligned}$$

This is a simple example, an extension that is tougher to crack in propositional logic is, if it does not rain until the day I go home, I will not bring my raincoat. In LTL this would be

$$\begin{aligned}&\neg p\mathbf{U}h \implies \neg r \text{ where}\\&p : \text{It rains}\\&h : \text{I go home}\\&r : \text{I will bring my raincoat}\end{aligned}$$

where $\mathbf{U}$ is read as "Until".

LTL models the world much more precise for the same reasons as signal does: Values of observations almost always depends on time. Atomic propositions like: p, r and h, has truth values that vary with time. Such atomic propositions are examples of elements in the set `Atoms` which the syntax of LTL uses.

2.3.1 Syntax

The syntax of formula in LTL can be defined recursively from the following:

Definition 1. Syntax of LTL

- if $p \in \texttt{Atoms} \cup \{\top, \bot\}$ then p is an LTL formula
- if ϕ ψ are LTL formulas then $(\sim \phi)$, $(\psi \vee \phi)$, $(\mathbf{X}\phi)$ and $(\psi\mathbf{U}\phi)$ are LTL formulas.

Like some subsets of propositional logic operators are adequate, the operators in 1 are adequate to form a basis for many more operators, some of which will be introduced when needed. According to 1 $((\mathbf{X}\phi) \vee \psi)$ is a formula. However to reduce parentheses yet still preserve dis-ambiguity the following convention applies: The unary connectives binds the strongest; followed by the binary connectives. The convention allows for simplifications such as

$$((\mathbf{X}\phi) \vee \psi) \equiv \mathbf{X}\phi \vee \psi$$

2.3.2 Semantics

Formally LTL can verify systems that can be modelled as transitions systems. A transition system can be represented by a directed graph where each vertex is a state, and each edge is a transition. Each state can be given to a labelling function $L : S \to \mathcal{P}(\texttt{Atoms})$ which takes a state and returns the set of true propositions $p \in \texttt{Atoms}$ for that state. In the previous example the day is a state, the labeling function returns the set of all atoms that where true on that day. A path is an infinite sequence of transitions such as $\pi : s_1 \to s_2 \to s_3 \to s_4 \to \ldots$. The infinity requirement of a path leads to the requirement that no state can 'dead-lock', because these deadlocks can would produces finite streams. But as the digraph representation hints such a system can contain many different paths and $\pi : s_1 \to s_2 \to s_1 \to s_2 \to \ldots$ could be a different path. Luckily for us, the paths we describe comes from streams, where each state appears chronologically and is deterministic, after all a day is always followed by a new one. Hence we only need to consider a single path. However, we need a definition 2 for deciding whether π satisfies an LTL-formula, denoted as $\pi \vDash \phi$, this is in order to obtain a formal semantic which for example tells whether to wear a rain coat, given an actual weather forecast.

Definition 2. Semantics of LTL. To distinguish temporal operators from propositional ones, the latter is written in words.

1. $\pi \vDash \top$
2. $\pi \nvDash \bot$
3. $\pi \vDash p$ iff. $p \in L(s_1)$
4. $\pi \vDash \sim \phi$ iff. $\pi \nvDash \phi$
5. $\pi \vDash \phi \vee \psi$ iff. $\pi \vDash \phi$ or $\pi \vDash \psi$
6. $\pi \vDash \mathbf{X}\phi$ iff. $\pi^2 \vDash \phi$
7. $\pi \vDash \phi \mathbf{U} \psi$ iff. $\exists i \in \mathbb{N}.\pi^i \vDash \psi$ and $\forall j.\pi^j \vDash \phi$ and $j < i$

Consider the following example suppose we have a path $\kappa : 11 \to 12 \to 13 \to \ldots$ where each state is an integer. We also have a labelling function $K(s) = \mathcal{P}(q_{11}) = \{\{\}, \{q_{11}\}\}$ where q_{11} is the proposition *the state is 11.* We then say that $\kappa \vDash q_{11}$ because $q_{11} \in K(1)$. We could also so ask model the property that the next two states are odd numbers, by the formula $\phi = q_{odd} \wedge \mathbf{X} q_{odd}$ where $K(s) = \mathcal{P}(q_{odd})$ and q_{odd} is the proposition

the state is odd. By working our way through the formula with definition 2, we get

$$
\begin{aligned}
& \kappa \models \phi \\
\Leftrightarrow \quad & \kappa \models q_{odd} \wedge \mathbf{X} q_{odd} \\
\Rightarrow \quad & \kappa \models \mathbf{X} q_{odd} \\
\Leftrightarrow \quad & \kappa^2 \models q_{odd} \\
\Leftrightarrow \quad & p_{odd} \in K(2)
\end{aligned}
$$

which is contradiction because 12 is not an odd number, an hence $\kappa \not\models \phi$.

3 Implementation

3.1 Analysis

As mentioned before PBT relies on two core concepts, the ability to generate suitable input values, and to specify properties. To solve both of these I needed a language for specification. LTL is has already been used for evaluating Yampa-streams (Perez & Nilsson, 2020). The syntax described in 3.2 uses theirs, and constructing an evaluator based on 2 is fairly simple, which I will present in the next section, the general idea is to make an evaluator function `TPred a -> Str a -> Bool`, which recursively traverses the formula, down to the atoms, which serves as basecases.

An aim of QuickRat however, was to not only specify properties but also to *generate* streams using LTL. For that stream generation with `evalLTL` becomes infeasible, just like guessing a random a password is, instead we would like generate streams where for each new element guessed, feedback is given on whether the latest element is correct, just like a Frantz Jäger safe from Berlin can guide Egon (Olsen Banden Movies, 1968). To achieve this in practice we use two state machines, an acceptor and a transducer. The general idea is, that the transducer can be created from an acceptor and can be used to make a `Gen (Str a)` monad which can produce input for properties.

The success of the implementation therefore relies on three main types, as sketched in figure 13 and reflected in the next three subsections.

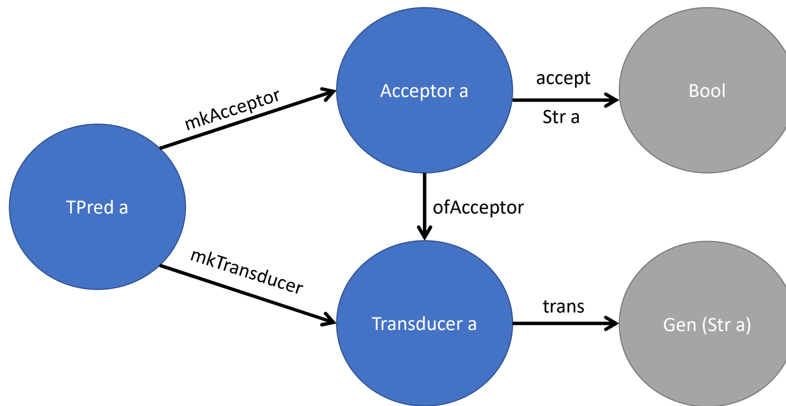


Figure 13: Core types in QuickRat, all types parameterized with an a, assumes that a is an instance of arbitrary, names above arrows are function names, the type accept also takes a stream argument, which is shown beneath the arrow.

Combining these it should be possible to define a function such as the one shown in listing 14. The `ltlProperty` takes the stream function under test, and specification for both in and output in LTL, and returns a property to test.

```
ltlProperty :: (Str a → Str b) → TPred a → TPred b → Property
ltlProperty fUnderTest inputSpec outputSpec =
    forAll
        (trans $ mkTransducer inputSpec)
        $ accept (mkAcceptor outputSpec) . fUnderTest

```

Figure 14: ltlProperty function

While the approach of state machines solves some problems regarding specification and feasibility there are some caveats to these, which we be elaborated on in the discussion.

3.2 LTL Syntax

This syntax is modelled in Haskell as the temporal predicate `TPred a` on stream with elements of type `a`, `TPred a` is a Generalised Algebraic Data Type, where each value constructor can instanciate a value of its return type, the full implementation and two examples are shown in listing 15. Consider for example the case
`And :: TPred a -> TPred a -> TPred a`, the semantics here is that `And` is a value constructor that takes two temporal predicates with type parameter a, and in itself is a temporal predicate with type parameter a. Most important is Atom which takes a predicate on `a`. For instance `Atom odd` corresponds to the statement the first element of the stream is odd.

```
1     data TPred a where
2         Tautology       :: TPred a
3         Contradiction   :: TPred a
4         Atom            :: (a → Bool) → TPred a
5         Not             :: TPred a → TPred a
6         Or              :: TPred a → TPred a → TPred a
7         Until           :: TPred a → TPred a → TPred a
8         Imminently      :: TPred a → TPred a
9         And             :: TPred a → TPred a → TPred a
10        Implies         :: TPred a → TPred a → TPred a
11        Always          :: TPred a → TPred a
12        Eventually      :: TPred a → TPred a
13        After           :: Int → TPred a → TPred a
14
15    firstOddSecondEven :: TPred Int
16    firstOddSecondEven = Atom odd ‘And‘ Imminently (Atom odd)
17
18    rainyExample :: TPred DayInfo
19    rainyExample = (Not (Atom isRainyDay) ‘Until‘ Atom goHome) ‘Implies‘ Not (Atom
      bringRaincoat)
20
```

Figure 15: Full syntax of LTL in QuickRat with examples from background section.

There are some slight deviations from the LTL presented earlier, firstly some additional operators, `And` have been introduced

- Imminently replaces next, but its semantics remain, it is however important to note, that as we are working with a stream of discrete elements the Imminently is concerned with the stream in the next time-step.
- And is just like in propositional logic.

- Implies is also like in propositional logic.
- Always holds if its argument holds for *all* tails of the stream.
- Eventually means that its argument will hold for *some* tail of the stream.
- After takes a count n and a `TPred a` that should hold in the n'th tail, for instance `After 2 somePred == Imminently (Imminently somePred))`.

3.3 Stream Evaluation

Before presenting how the final acceptor-approach works I will present some simpler stream evaluation cases.

3.3.1 Equality

Because a `Str a` is infinite, we can never know whether they are truly equal by comparing values, therefore I implemented a notion of "probably equal", using simple approach to check for equality for elements in a finite part. Doing so requires, that `a` implements `Eq`.

```
1    -- strTake 20 aStr constructs a list from the first 20 elements of aStr.
2    class ProbEq a where
3        (=~=) :: a → a → Bool
4
5    instance (Eq a) ⇒ ProbEq (Str a) where
6        s1 =~= s2 = strTake 20 s1 == strTake 20 s2
7
```

Figure 16: Implementation of probably equal for streams

The implementation of probably equal in listing 16 checks for whether the tip of the stream are equal, I chose the first only 20 elements for convience of my own debugging, but ideally this number would be considered depending on the expected behavior, and making studies to find an generally suitable count is beyond the scope of this thesis. With some notion of equality we can test an outputted stream against some pre-constructed stream, and if we do regression testing, we can sensibly test for the property that `f` produces out similar to `fOld`.

3.3.2 LTL evaluation

With this syntax being defined in listing 15, I wrote an evaluator using for it using 2 2, to aid in debugging the acceptor. Listing 17 shows the most prominent parts of its implementation.

```
evalLTL' :: Int → TPred a → Str a → Bool
evalLTL' checksLeft formulae aStr@(h ::: t) = checksLeft ≤ 0
    || case formulae of
        Tautology       → True
        Contradiction   → False
        Atom headPred   → headPred h
        Or phi psi      → eval phi aStr || eval psi aStr
        Imminently phi  → evalNext phi strTail
        Until phi psi   → eval psi aStr
                          || (eval phi aStr && evalNext (phi `Until` psi) strTail)
        -- And, Not, Always, Implies, After and Eventually are excluded for brevity.
    where
        evalNext = evalLTL' (checksLeft - 1)
        eval = evalLTL' checksLeft
        strTail = adv t

evalLTL :: TPred a → Str a → Bool
evalLTL = evalLTL' 20


```

Figure 17: An excerpt of evalLTL which evaluates a stream according to an LTL formula.

Consider the execution in the bottom corresponding to $\kappa \vDash q_{odd} \wedge \mathbf{X} q_{odd}$ introduced earlier with $\kappa_{oddOdd} = 1 \rightarrow 3 \rightarrow 5 \rightarrow \ldots$. For that execution we get `True`, meaning that κ does indeed satisfy $q_{odd} \wedge \mathbf{X} q_{odd}$. Had we instead used κ as the sequence of natural numbers, we would eventually end up with `True && odd 2` which, as expected would evaluate to false.

So far it all looks good as we looked at an example specified propositions for a finite tip of the stream only, but for formulas with operators that speak of the entire stream get issues. For instance consider the formula modelling the liveness property `(Always (Eventually (Atom id)))` the execution when supplied with constant inacitve thread is in listing 18's bottom, where we get the unexpected result, that there is liveness for a thread that is never active. `evalLTL'` does so because it returns true after having checked only the first 20 elements of the stream. The reason I chose not to return `False` after 20 tests lies in the spirit of testing, where a test passes unless a counterexample falsifies it, for this example we simply cannot tell for an infinite stream, whether it will

eventually become active, by only looking at a finite part of it. This leads to the description of the semantics of evalLTL: evalLTL returns False `only if` the first 20 elements of the stream makes the formula unsatisifiable by the stream.

```
1
2  -- strExtend creates a stream from the elements of the arguments, with the last repeated.
3  -- Assuming left to right evaluation.
4  evalLTL (Atom odd `And` Imminently (Atom odd)) (strExtend [1,3,5,7])
5  ⇝ evalLTL' 20 (Atom odd `And` Imminently (Atom odd)) (strExtend [1,3,5,7])
6  ⇝ eval (Atom odd) (strExtend [1,3,5,7])
7     && eval (Imminently (Atom odd)) (strExtend [1,3,5,7])
8  ⇝ odd 1 && eval (Imminently (Atom odd)) (strExtend [1,3,5,7])
9  ⇝ True && eval (Imminently (Atom odd)) (strExtend [1,3,5,7])
10 ⇝ True && eval (Atom odd) (strExtend [3,5,7])
11 ⇝ True && odd 3
12 ⇝ True
13
14 -- Testing of liveness property.
15 evalLTL' 20 (Always (Eventually (Atom id))) (strExtend [False])
16 ⇝ evalLTL' 20 (Eventually (Atom id)) (strExtend [False]) && ...
17 ⇝ evalLTL' 20 (Atom id) (strExtend [False])
18    || evalLTL' 19 (Eventually (Atom id)) (strExtend [False]) && ...
19 ⇝ False
20    || evalLTL' 19 (Eventually (Atom id)) (strExtend [False]) && ...
21 ⇝ False || False
22    || evalLTL' 18 (Eventually (Atom id)) (strExtend [False]) && ...
23 ⇝ ...
24 ⇝ False || False || ...
25    || evalLTL' 0 (Eventually (Atom id)) (strExtend [False]) && ...
26 ⇝ False || False || ... || True && ...
27 ⇝ True && evalLTL' 19 (Always (Eventually (Atom id))) (strExtend [False])
28 ...
29 ⇝ True && True && evalLTL' 18 (Always (Eventually (Atom id))) (strExtend [False])
30 ⇝ True && True && ... && True
31 ⇝ True
32
```

Figure 18: Evaluation of thread liveness

3.3.3 The acceptor

Because the aim is to construct streams rather than just evaluate, I has to come up with code that given a head of a stream and the tail, can determine whether a stream is accepted, rejected or information on the following tail is needed, after several iteration I chose to use the polymorphic and algebraic type `Acceptor a` in listing 19.

```haskell
data Acceptor a = Accept
                | Reject
                | NextA (a → Acceptor a)

```

Figure 19: Acceptor datatype

the three value constructors has the following semantics are; `Accept` represents an acceptor that has accepted a stream; `Reject` represents an acceptor that has rejected a stream or; a `NextA` which represents an acceptor whose rejection depend on the tail, `NextA` can be thought of as a continuation. Whose first element will eventually be passed to a function of type `(a -> Acceptor a)`. To see how acceptors work consider the following problem. Suppose we wish to tests whether $\pi \vDash p \wedge \mathbf{X}q$ p: The head is odd; and q: The head is even. A truth table for and formula, with some example inputs would

Stream elements	p	q	$\mathbf{X}q$	$p \wedge \mathbf{X}q$
$2, 2, 3, \ldots$	F	T	T	F
$1, 2, 3, \ldots$	T	F	T	T
$1, 3, 8, \ldots$	T	F	F	F

Table 1: Truthtable of LTL-formular satisfaction

If we wish model the formula $p \wedge \mathbf{X}q$ with an acceptor the atoms and relevant formulas in are defined in listing 20.

```
check :: (a → Bool) → (a → Acceptor a)
check predicate x = if predicate x then Accept else Reject
-- these have type Acceptor Int
p = NextA $ check odd
q = NextA $ check even
pAndXQ = NextA $ λx1 → if odd x1 then q else Reject

-- the following are all true
accept pAndXQ (strExtend [1,2,3])
not (accept pAndXQ (strExtend [2,2,3]))
not (accept pAndXQ (strExtend [1,3,8]))
not (accept pAndXQ (strExtend [2,3,8]))

evalAcceptor :: Acceptor a → Bool
evalAcceptor Accept = True
evalAcceptor Reject = False
evalAcceptor (NextA _) = True

```

Figure 20: LTL as Acceptor

The acceptor is also equipped with the function:
`accept' :: Int -> Acceptor a -> Str a -> Acceptor a`.
This function advances the acceptor whiles reading value from the tip of the stream. It stops after a number of reads specified by the third parameter. `accept'` will produce a new acceptor, whose subtype can be mapped to a boolean with the evaluation function evalAcceptor, the mapping is so that `Accept` and `Reject` are intuitively to mapped `True` and `False` respectively. Not so intuitively is it, that evalAcceptor NextA returns True, again the prime argument is the spirit of testing, like for `evalLTL`. However the separation of accept' and the evaluation to boolean, makes it is easy to write a custom evaluation function, which maps a continuation to false instead.

while declaring acceptors for each is possible, the aim is to use the expressiveness and conciseness of LTL this is exactly what is done in listing 21

```haskell
mkAcceptor :: TPred a → Acceptor a
mkAcceptor formulae = case formulae of
    Tautology       → Accept
    Contradiction   → Reject
    Atom headPred   → NextA (λh → if headPred h then Accept else Reject)
    And phi psi     → mkAcceptor phi ‘andA‘ mkAcceptor psi
    Imminently phi  → NextA (λ_ → mkAcceptor phi)
    Until phi psi   → mkAcceptor $ psi ‘Or‘ Imminently (phi ‘Until‘ psi)
    -- Or, Not, Always, Implies, After and Eventually are excluded for brevity.

andA :: Acceptor a → Acceptor a → Acceptor a
andA Reject _               = Reject
andA _ Reject               = Reject
andA Accept aNextA          = aNextA
andA aNextA Accept          = aNextA
andA (NextA f1) (NextA f2)  = NextA (λx → f1 x ‘andA‘ f2 x)

```

Figure 21: Function to create an acceptor from an LTL-expresssion

There are some important things to note: Imminently is not a recursive call, since it immediately returns a NextA and because all temporal predicates relies on Imminently this means that the checksleft is not needed, Imminently serves as the base case for all temporal evaluations. With this implementation the following property

$$\forall e.\forall s.(\texttt{evalLTL e s = accept (mkAcceptor e) s})$$

which I used for testing the correctness of QuickRat.

3.4 Stream Generation

A simple way to make random stream is by creating a `Str a` where each is value is generated by `Gen a`. Listing 22 does exactly that by recursively setting the head of the stream to be an arbitrary value of the type `a`, where the absence of a stopping condition demonstrates Haskell's laziness.

```haskell
instance (Arbitrary a) ⇒ Arbitrary (Str a) where
arbitrary = do
    x ← arbitrary :: Gen a
    xs ← arbitrary :: Gen (Str a)
    return $ x ::: delay xs

```

Figure 22: Arbitrary's instance definition of Str a

22 can only produce streams of independent values. But such streams, does not usually model the problem domain well, and furthermore the probability of getting a stream that satisfies a LTL formula is negligible. Therefore I use a transducer instead, from the following specification.

- It should be capable of generating values of a specified predicate
- It should be capable of generating a new transducer which can generate the next value
- An indication of failure, if it fails in generating the above.

The last case, occurs if a given formula is a contradiction or if the atomic predicate is never satisfied by the generator, the latter will elaborated in the discussion. Through a handful of iterations, I found that the in listing 23 type alias would suffice.

```haskell
newtype Transducer a = NextT (Gen (Maybe (a, Transducer a)))

arbitraryTransducer :: forall a. (Arbitrary a) => Transducer a
arbitraryTransducer = NextT $ do
    element <- (arbitrary :: Gen a)
    return (Just (element, arbitraryTransducer))

instance (Arbitrary a) => Arbitrary (Transducer a) where
    arbitrary = return arbitraryTransducer

trans :: Transducer a -> Gen (Str a)
trans (NextT aGen) = do
    aMaybe <- aGen
    let (value, aTransducer) = fromJust aMaybe -- Will error if no element available
    rest <- trans aTransducer
    return $ value ::: delay rest

transducerOfStr :: Str a -> Transducer a
transducerOfStr (h ::: t) = NextT $ return $ Just (h, transducerOfStr (adv t))


```

Figure 23: Transducer, function for generating a transducer for arbitrary elements and trans which will produce a stream generator for use with QuickCheck

To appreciate how this works, consider how `trans (arbitraryTransducer::Int)` would run, arbitraryTransducer is lazily evaluated, so only in the bind of trans aMaybe will actually be generated, if aMaybe is a Just the value will be the head of the stream, and the tail will made by the succeeding and in this case identical transducer. Using this I rewrote the instance of arbitrary as it is given in listing 23, without changing the semantics. The listing also showcases a transducer that generates elements equal to those of an incoming stream.

3.4.1 Generating from LTL

The general strategy for generating is shown from the bottom part of 13 where `mkTransducer` takes an LTL-formula and simply returns an transducer built under the restriction imposed by the formula, with the acceptor as guideline for this. The 'built under the restriction' part is achieved by `restrictWith` in listing 24.

```haskell
ofAcceptor :: (Arbitrary a) ⇒ Acceptor a → Transducer a
ofAcceptor anAcceptor = arbitraryTransducer `restrictWith` anAcceptor

mkTransducer :: (Arbitrary a) ⇒ TPred a → Transducer a
mkTransducer = ofAcceptor . mkAcceptor

restrictWith :: forall a. Show a ⇒ Transducer a → Acceptor a → Transducer a
restrictWith _ Reject = rejectTransducer
restrictWith aTransducer Accept = aTransducer
restrictWith aTransducer (NextA someTest) = rwInner aTransducer someTest

rwInner :: forall a. Show a ⇒ Transducer a → (a → Acceptor a) → Transducer a
rwInner (NextT gen) passTest = NextT $ loop (1000::Int) where
    loop :: Int → Gen (Maybe (a, Transducer a))
    loop n = do
        sz ← getSize
        value ← if  sz < 3 then scale (+3) gen else gen
        case value of
            Nothing → return Nothing
            Just (genVal, xGen) →
                case passTest genVal of
                Accept → return $ Just (genVal, xGen)
                Reject → if n == 0 then return Nothing else loop (n - 1)
                NextA passTest' → return $ Just (genVal, rwInner xGen passTest')

```

Figure 24: Definition of mkTransducer

`restrictWith` works such that if the the given acceptor is an `Accept` no restriction occurs and the given transducer is the return value, this might be the acceptor is constructed from an formula which is equivalent to a tautology. Similarly if the acceptor is a reject a transducer that can only generate nothing is returned. If however the Acceptor contains a test on an element restriction can occur, which is done using the loop. The general strategy of the loop is to try generating new values from the generator in the given transducer, for each new test of a value generated it can either be accepted, rejected or be a continuation, which are handled ind the last case-expression. If the value is accepted it will return a value along with a generator for the next value. If it is rejected it will try to generate a new value for the same test or if it has already tried sufficiently, it will cause QuickCheck to fail the test. If the result is a continuation the value is also accepted, but the returned generator is a recursive call to generate the next.

An important caveat of `restrictWith` is that if the test on a value produced is too narrow the it can become unlikely that, `rwInner` will find a match within the hard coded 1000 tries, a way to mitigate this is to use with a guided transducer rather an arbitrary one, all of this is shown in listing 25.

```haskell
oddOddGenerator :: Gen (Str Int)
oddOddGenerator = trans $ mkTransducer $ Atom odd `And` Imminently (Atom odd)

guidedTransducer :: forall a. (Arbitrary a) => Gen a -> Transducer a
guidedTransducer aGen = NextT $ do
    element <- aGen
    return (Just (element, guidedTransducer))

threadDucer :: Transducer String
threadDucer = guidedTransducer $ oneof $ map return ["t1", "t7", "t33", "t42"]

willError = trans $  mkTransducer (Always (Atom (=="t42")))
willWork = trans $ threadDucer `restrictWith`
    mkAcceptor (Always (Atom (=="t42")))

```

Figure 25: Examples of use and issue of type multiplicities

There the first transducer will produce an error because it is unlikely that the default arbitrary of String will produce an exact match, if however I know that the only possibly values in the actual stream are those in the list, I can use one of QuickChecks combinators to build a generator, that only produces one of the values. Which is why the last expression works.

4 Discussion

4.1 Related Work

To discuss the features of QuickRat, I will present how previous work has treated the core tasks of generating and evaluating signals. The articles referred to are titled *Runtime verification and validation of functional reactive systems* (Perez & Nilsson, 2020) and *Quickstrom: Property-based acceptance testing with ltl specifications* (O'Connor & Wickström, 2022). The former is concerned with signals formulated with Arrowised `frp`, which we have already mentioned in the introduction and the latter generates signals to simulate UI events in web applications and evaluated in an LTL-dialect.

4.1.1 Evaluating signals

In the introduction I have already touched upon how Perez and Nilsson, 2020 evaluates signals, as QuickRat is heavily inspired by this, and the Syntax of the LTL is nearly identical. For actually evaluating a signal Perez and Nilsson, 2020 uses
`evalT :: TPred a -> SignalSampleStream a -> Bool`. QuickStrom takes a slightly by the expected behavior is defined in an LTL dialect, that adds a couple of additions to LTL; first `Next` has been split into required next, weak next and strong next; secondly always, eventually and until requires and extra integer, that specifies for how many steps it should check.

Suppose that a tester would like to express the that the GUI should flicker between dark and light, he would have to formulate this in Specstrom a language which resembles JavaScript, where it is required that she specifies the duration of *Globally* and the variant for next, for this case it would probably suffice to check always for 300 time-steps, and use the weak variant of next. The distinction between weak and strong next, makes sense in QuickStrom because of its four-valued, rather our two valued evaluation, that they adopted from RV-LTL(Bauer et al., 2011), this choice will will be elaborated further in this section.

4.1.2 Generating signals

Perez and Nilsson, 2020's solution is somewhat different from ours because the library is based on continuous streams and uses finite streams for testing as which is reflected in

listing 26. Therefore, testers must specify a distribution and range for time-sampling, as well as the length of the finite stream. Time sampling is out of our control as we use `Str`'s so the essential part is the first argument of type `(Int -> DTime -> Gen a)` the semantics for this function is to return a generator for an element in the stream given the sample count and time since last sample. Their solution is easy to comprehend, but passes majority of the work on implementing a suitable generator on to the tester, an example of which is shown.

```
generateStreamWith :: Arbitrary a
                   ⇒ (Int → DTime → Gen a)
                   → Distribution
                   → Range
                   → Length
                   → Gen (SignalSampleStream a)

oddOddStream :: Gen (SignalSampleStream Int)
oddOddStream = generateStreamWith
               (\n _ →  if n ⇒ 10 && n ≤ 15 then posGen else arbitrary)
               (DistNormal (0.033, 0.01))
               (Just 0.031, Just 0.035)
               (Just 20)
    where
    posGen = abs <$> arbitrary

```

Figure 26: Stream generation in Perez and Nilsson, 2020

Quickstrom (O'Connor & Wickström, 2022) solves a similar problems, where it generates streams of user interactions, which interactions are relevant is specified by the tester, this could be clicks on buttons a b, notionally a stream of user interaction can be represented as `Str [Bool]`, where length of the list is the number of interaction specified by the tester. It is important to note that Perez and Nilsson, 2020 generates these completely Random, although the mention some approaches in future work.

4.2 Core of QuickRat

At its core QuickRat contains a syntax for LTL and implements two state machines, whose relation depicted in 13. The basic principal is that an acceptor is created from an LTL-formula ϕ, which can be used to directly to assert a stream, or the acceptor can be used as a guide, to a transducer which will generate an arbitrary streams, which satisfies ϕ. I envision that QuickRat can be used with three different levels complexity, where each

higher levels have broader pratical usecases.

As the first level consider the properties formulated in Haskell using QuickCheck and QuickRat in listing 27

```
reactToUser :: Str (Bool,Bool) → Str Bool
reactToUser userInput = ...

prop_0And1Pressed_lightFlickers =
    ltlProperty
        reactToUser
        (Atom fst `And` Atom snd)
        $ Atom not `And` Imminently (Atom id)
            `Or` Atom id `And` Imminently (Atom not)

messager :: Str Int → Str (Maybe String)
messager _ = ...

prop_fivePositiveValues_emitMessageNext =
    ltlProperty
        messager
        (After 10 (Atom (> 0)) `Until` After 15 Tautology)
        $ After 15 (Atom (== Just "Jackpot"))

```

Figure 27: Comparative examples with QuickRat.

the examples show the utility function ltlProperty allows for easy testing of a function that which takes on stream to produce another, if for instance we would like test functions that take more arguments such as a `switch' :: Str a -> Str (Maybe (Str a)) -> Str a`, we can do so by doing partial evaluation such that the the function under test. As a second option we can generate streams with tuples of multiple elements, one element for each stream, for instance one might produce a `Gen (Str (a, Maybe (Str a)))` to test a switch.

The second level of usage is by supplying a custom made generator of type `Gen TypeOfElement` and supply it to `guidedTransducer` instead of `mkTransducer`, an example of which is given in listing 24, using guided transducers can narrow the search space for restriction, but should for the same reason be used with caution. One might also use `evalAcceptorStrict` which returns false, rather than true if the finalAcceptor is of subtype `NextA (a -> Acceptor a)`.

The third level of usage, is to construct acceptor manually and then parse as the second argument to restrict with, this can be used to specify streams that are not expressible

in a finite LTL-exression, examples of such streams that exhibit infinite behaviors are alternating elements or streams that always are increasing.

The three levels demonstrates, that QuickRat that *many* properties formulated in LTL can be tested using `ltlProperty` or by specifying transducers or acceptor to use with QuickChecks `forAll`

4.3 Limitations of QuickRat

As I emphasized earlier, many but not all properties can be tested with QuickRat. Properties that cannot expected to be tested correctly are either due to feasibility issues or/and liveness/safety issues, which I will discuss thoroughly now.

4.3.1 Expressiveness of LTL

Some streams cannot be uniquely satisfied by an LTL-formula. For instance the stream of digits in any irrational number such as $\pi = 3 \to 1 \to 4 \to \dots$ would require an infinite formula like `Atom (==3) ‘And‘ Imminently (Atom (==1)) ‘And‘ ...`, and such infinite formula cannot be expressed as a `TPred` on a computer.

4.3.2 Absence of bugs

As it is always the case with PBT as opposed to formal verification, it can be hard to know whether all cases has been described achieved. This is also the case for QuickRat, that like all testing can only show the presence of bugs, not their absence.

4.3.3 Feasibility

Consider the code presented in listing 25 how it can become infeasible, that is take more than 1000 tries to find a suitable candidate for a next element, in the number of tries it

takes to generate a single suitable entire stream can be modelled by

$$n_t = p^{-n}$$

n_t : average number of total tries

r : probability of getting an accepted value

n : size of the prefix to be evaluated

which is under the crude assumption that r is constant for all elements. I emphasize that the 1000 tries and the prefix size of 20 which I used in this prototype can be optimised and preferably be made adjustable depending on the input being generated, after all the test presented hear usually finish instantly. But as an example of how many tries it take consider the tests in listing 28

```
testTransducer expected = quickCheck $ forAll
    (trans $ mkTransducer $ Atom (expected ==))
    $ evalLTL $ Atom (expected ==))
...
testTransducer (True, False, True, False, True, False)
ghci> +++ OK, passed 100 tests.
testTransducer (True, False, True, False, True, False, True, False)
ghci> *** Failed! (after 55 tests) ...
testTransducer (True, False, True, False, True, False, True, False, True, False)
ghci> *** Failed! (after 2 tests) ...

```

Figure 28: Feasibility issues

where the transducer only generates 55 test cases, when the type multiplicity of the element is 2^8 because a suitable element is not found within the allowed 1000 tries. Generalizing this behavior over types such as strings and integers is hard because p does not only depend on the multiplicity of the type but also the probability distribution that the values are drawn from. To mitigate this problem one may make a guided transducer, allow for more tries or adjust the prefix size of 20. Generalizing this behavior over types such as strings and integers is hard because p does not only depend on the multiplicity of the type but also the probability distribution that the values are drawn from.

Another cause of high running times are that logical equivalencies in LTL that has not been used to reduce computations. For instance if given the formula `Not (Not phi)` `mkAcceptor` does not immediately return `phi`, but rather calls itself twice. Along the same the lines, but opposite, `Or` is defined using De Morgan's rather by implementing an acceptor for it, this means that for each `Or` four `TPred` as must be evaluated.

4.3.4 Liveness and safety

I have already hinted that QuickRat introduces a problem of liveness, this is easiest to consider the case where the user would like to test for liveness of a program that uses to competing the threads of liveness to hold we expect
`Always (Eventually (Atom (=="t1") ‘And‘ Always (Eventually (Atom (=="t2")` on a stream of thread names to be executes is to hold. And indeed it does, but because only a finite portion of the stream is checked accept will return true for all streams. Generally the issue is that even though we have not seen an argument to eventually become true, we cannot exclude that it will become so later.

Safety is a similar problem, with the general issue being that even though, we have not seen an argument become false we cannot guarantee that it will so, at some point. Again the scheduler is an example, and a formulation for such could be that even though, t1 has always become active at any point the checked stream, we do not now whether it will suddenly stop. For safety a failing test implies a that property does not hold, but not vice versa, for liveness the implication goes in the opposite direction.

To deal with the issues of liveness and safety QuickStrom used a four valued-evalution, and for each temporal operator a count limit must be included. For the sake simplicity, and to align with LTL I have not introduced such a limit. But the syntax of such could look like `\Always 20 (Eventually 3 (Atom (=="t1")))` formula expressing that in the next 20 time steps there should always be a "t1" within the three next steps, which can express some kind of finite liveness. The four values in QuickStrom that replaces satisfied and not satisfied in LTL are: *definitively false*, which can be obtained from a violated safety property; *presumptively false*, is the result of an unfilled liveness property, *presumptively true* is the result of an unfilled safety property and *definitively true* is the result of an fullfilled liveness property.(Bauer et al., 2011) All of these four can actually be mapped on to failing/passing test of evalAcceptor / or evalAcceptorStrict, as long an formula matches only safety or liveness properties.

4.4 Possible Future Work

As QuickRat is a functioning prototype rather than a highly polished, usability focus library, this section will focus on the functioning part rather than the latter, even though the importance of usability is significant.

As noted in the feasibility section, experiments on the running time and could be

made to make some proper default values for number of tries on each new element, the prefix size checked, as well giving the ability for the tester adjust these.

Along these lines the size parameter could also be taken more account, as its semantics matches well with adjusting the `checkCount`.

At the moment values are picked disregarding previous ones, if not so it could be possible for restrictWith to an informed search. At the moment if the 1000 tries are used, the test will fail, a potential improvement could be if it was marked as 'GaveUp', which is supported by quickCheck.

Another possible improvement would be to implement a reduction of all LTL-formula such that the get reduced to easiest computable formula, a big advantage of this is that contradications can be evaluated in one step.

QuickRat does not implement shrinking, but this could be useful, to see if 'simpler' version of streams will fail, for instance shrinkings a stream generated from an expression, could stream with shorter important prefixes or streams that satisfies simpler formulas which are follows from implication of the original one.

For testing the correctness of QuickRat an instance of Arbitrary could be defined on `TPred a` so `mkAcceptor` could test with the property that
`evalLTL expr == accept mkAcceptor`.

5 Conclusion

I have presented QuickRat, a tool that allow for Property Based Testing expressed with Linear Temporal Logic on streams in the Functional Reactive Programming language Rattus. Like other libraries has done for Arrowised Functional Reactive Programming QuickRat implements an formula syntax as a type in Haskell and semantics for whether a stream satisfies an expression, while making decisions on how to best deal with issues of safety and liveness, that are inevitable .

The prime novelty of QuickRat, however is for evaluating streams, but generating streams that fulfill such formulas, by allowing as shown in listing 29

```
ltlProperty :: (Str a → Str b) → TPred a → TPred b → Property
ltlProperty fUnderTest inputSpec outputSpec = ...
ghci> quickCheck (ltlProperty ... )
+++ OK, passed 100 tests

```

Figure 29: Sum up of ltlProperty

It does so by running using two state machines, one which with guess elements arbitrarily and then filter according to an acceptor, which is constructed from LTL-formula. Generating streams are not feasible for all formulas, but still I believe that QuickRat can make testing of functions that act on streams better. I hope that this thesis will inspire others or at least myself to create more reliable applications.

Acknowledgements

I would like to thank my supervisor Patrick Bahr for his focused attention throughout this project, Emil and Gregers fruitful sparring and lastly a very little thank for each and everyone who paid taxes the last 30 years, I promise to do the same for the next my 30 years.

References

Bahr, P. (2022). Modal FRP for all: Functional reactive programming without space leaks in haskell. *Journal of Functional Programming*, *32*. https://doi.org/10.1017/s0956796822000132

Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, *20*(4). https://doi.org/10.1145/2000799.2000800

Claessen, K., & Hughes, J. (2000). QuickCheck. *ACM SIGPLAN Notices*, *35*(9), 268–279. https://doi.org/10.1145/357766.351266

Elliott, C., & Hudak, P. (1997). Functional reactive animation. *International Conference on Functional Programming*. http://conal.net/papers/icfp97/

Hudak, P., Courtney, A., Nilsson, H., & Peterson, J. (2003). Arrows, robots, and functional reactive programming. In *Advanced functional programming* (pp. 159–187). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-44833-4_6

Krishnaswami, N. R. (2013). Higher-order functional reactive programming without spacetime leaks. *SIGPLAN Not.*, *48*(9), 221–232. https://doi.org/10.1145/2544174.2500588

O'Connor, L., & Wickström, O. (2022). Quickstrom: Property-based acceptance testing with ltl specifications. https://doi.org/10.5281/zenodo.6416483

Paterson, R. (2001). A new notation for arrows. *SIGPLAN Not.*, *36*(10), 229–240. https://doi.org/10.1145/507669.507664

Perez, I., & Nilsson, H. (2020). Runtime verification and validation of functional reactive systems. *Journal of Functional Programming*, *30*. https://doi.org/10.1017/s0956796820000210

Runciman, C., Naylor, M., & Lindblad, F. (2008). Smallcheck and lazy smallcheck. *Proceedings of the first ACM SIGPLAN symposium on Haskell*.