Declarative Deadlines in Functional-Reactive Programming

Kevin Baldor
University of Texas at San Antonio and
Southwest Research Institute, San Antonio, USA
kevin.baldor@acm.org

Jianwei Niu
University of Texas at San Antonio
jianwei.niu@utsa.edu

ABSTRACT

Functional Reactive Programming (FRP) enables a developer to describe *declaratively* a program's response to external events. FRP has many implementations that share many common features, but lack a common declarative mechanism for describing time constraints, such as deadlines or rate throttling. We present a set of Metric Temporal Logic (MTL) primitives for expressing real-time relationships between events in a reactive system. Providing these primitives in the core of an FRP framework allows a developer to define new time-based constraints without resorting to timers external to the FRP code. We describe the implementation of the basic MTL operators in a modified version of the Sodium FRP framework and demonstrate its use for an implementation of the *debounce*[4] primitive provided by the Java implementation of ReactiveX (RxJava).

Categories and Subject Descriptors

D.3.3 [**Software**]: Language Constructs and Features; F.4.1 [**Mathematical Logic**]: Logic and constraint programming

General Terms

FRP, MTL

1. INTRODUCTION

Functional Reactive Programming (FRP) provides a mechanism for declaratively describing a program's state as a function of its inputs. Push-based FRP implementations only compute updates in response to the arrival of input events. Expressing real-time relationships involving timeouts either requires that the reactive framework natively support expressions that involve real time or that the programmer schedule an asynchronous event outside of the FRP framework.

For example, the Java implementation of ReactiveX (RxJava) provides a number of so-called *backpressure* primitives that allow a programmer to limit the rate of input event arrival. Such primatives are useful for optimizing the performance of an application by, for example, waiting until the user stops typing for a certain amount of time before performing an expensive operation like parsing for syntax highlighting. But there are other time-based constraints that make sense for reactive applications; one might wish to guarantee that a "heartbeat" message will be generated periodically whenever a connection is active. Normally, a programmer might achieve this by launching a thread to produce a heartbeat input and then respond to that. This will work, but strikes us as contrary to the spirit of a declarative style of reactive programming – particularly if the thread is started and stopped from within the reactive code in response to input events.

Metric Temporal Logic (MTL) is a well-established logic in the runtime-verification community that extends Linear Temporal Logic (LTL) operators with real-time constraints. Here, we present the past-only subset of the dense-time, continuous-semantics, MTL as a candidate set of primitives for expressing temporal relationships between events with an explicit notion of real time for use in FRP. With it, we could use the syntax introduced in section 3 to address the heartbeat example with the expression

$$\neg\text{connection\_open} \vee \diamondminus_{[0,1)}\text{heartbeat\_message}.$$

This reads "Either the connection is not open or there must have been a heartbeat message within the last one time unit". A program designed to react to that statement ever becoming *false* by sending a heartbeat message would guarantee that the heartbeat message will be sent once per time unit whenever the connection is open.

We implemented, with some caveats, the MTL semantics with a minimal modification to the Sodium FRP framework which we describe in section 2. The Sodium project is an attempt to create a common core of FRP primitives across several languages, so we felt that it represents a reasonable base implementation to explore the introduction of real-time constraints. The features that we depended upon are presented in section 2.1. Section 3 describes the LTL and the MTL extension that our implementation supports. The actual implementation of the MTL operators are provided in section 4 and its use to implement the *debounce* operator of RxJava is presented in section 5.

2. SODIUM FRP

FRP implementations may be *push*, *pull*, or a combination of the two[7]. The original conception was pull-based, meaning that each reactive expression was evaluated by re-evaluating its inputs. This allowed inputs to be dense-time, continuous-valued, functions of time, but was potentially inefficient because it could re-evaluate expressions for which the inputs had not changed. Push-based implementations construct a reactive expression as a network of listeners comprising its sub-expressions. Each sub-expression fires only when its input has changed. This can be more efficient, but introduces a concern about the order of the evaluation of sub-expressions termed *glitch freedom.*

For this investigation, we selected the Sodium[5] FRP implementation. It is a push-based FRP framework that seeks to provide a consistent, minimal, set of FRP primitives across multiple languages. We selected it for this investigation because of its guarantee of *glitch-freedom* and its notion of *transaction* that supports effectively simultaneous input.

Glitch-freedom guarantees that, for a set of simultaneous input values, only one externally-visible output change will take place – even in the presence of diamond dependencies as described in [6]. Briefly, diamond dependencies arise when the results of one input travel over two parallel paths, both of which arrive at the same final node. If all operations take place concurrently, the final node may produce two outputs; one when the first path completes and a second when the second completes. The value that it will produce in that intermediate state depends on which of the two paths completes first. Glitch freedom guarantees that the intermediate value will not be produced. This allows us to write complex temporal logic expressions and interpret its output without waiting for the resolution of *eventual* consistency.

The `Transaction` is the locking and scheduling mechanism that Sodium uses to ensure glitch freedom. One advantage of Sodium's implementation is that a developer can use `Transaction.run(...)` to describe a set of inputs that arrive simultaneously. In this way, it is analogous to a database transaction in that it can be used to guarantee consistency of the program's state.

A note about nomenclature: For the benefit of readers coming from a object-oriented programming background, the publishers of [3] requested that Sodium use terminology that differs from the common usage in the FRP literature. In particular: *behaviors* are called *cells* and *events* are called *streams.* The term *cell* suggests the cell of a spreadsheet in that, for a cell, *c*, defined as the sum of two other cells, c_1 and c_2, any change to one of the input cells results in an immediate change to the value in *c*. The term *stream* describes an ephemeral input. It has value only at the time of the `Transaction` during which it arrives.

2.1 Sodium primitives

The implementation below depends on the following Sodium primitives

Cell.lift(f: B,A→C, a: Cell<A>, b: Cell<B>) → Cell<C>

This returns a *cell* such that whenever either a or b changes, its value updates to $f(a, b)$.

(s: Stream<A>).snapshot(a: Cell<B>, f: A,B→C)
→Stream<C>

This returns a *stream* such that whenever an event s' arrives from s, it emits an event with the value $f(s', a)$.

(Stream<A> s).hold(initialValue: A)→Cell<A>

Returns a *cell* originally equal to *initialValue* and then whenever an event s' arrives from s, it takes on the value s'.

(Stream<A> s).map(f: A→B)→Stream<B>

Returns a *stream* that whenever an event s' arrives from s, it emits an event with the value $f(s')$.

(Cell<A> c).map(f: A→B)→Cell<B>

Returns a *cell* that always has the value $f(c)$.

(Cell<A> c).updates()→Stream<A>

Returns a *stream* of each new value taken on by *c*. Its use is somewhat controversial; it disallows the idea of continuous-valued, continuously-variable Cells, which were a significant feature of the original implementation of FRP [8]. But it is convenient and sufficient for modeling MTL values, as they are discreet-valued.

3. TEMPORAL LOGICS

This section provides a brief overview of the past-time linear and metric temporal logics. As its name suggests, the past-time LTL provides a set of operators for describing when boolean expressions have been true in the past. As such, they are the only LTL operators appropriate for reactive programming. Strictly speaking, LTL describes only the order of sequence of truth values of boolean expressions; MTL adds real time constraints. We use only the simplest extension, providing a time interval – relative to the current time – over which the operator applies.

3.1 LTL

Expressions of the past-time version of LTL are of the form

$$\phi ::= p|q|\neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \mathcal{S} \phi \mid \diamondminus \phi \mid \boxminus \phi$$

where p and q represent a Boolean-valued expression – an input, for our purposes in FRP. The familiar logical operations negation ($\neg$), conjunction($\wedge$), and disjunction ($\vee$) operate as they do in standard propositional logic. The *since* operator $p\mathcal{S}q$ yields an expression that is *true* whenever q is *true* or p has been true since q became *false.* The *once* operator, $\diamondminus p$, is true whenever p is true or has been true at any point in the past. And the *historically* operator, $\boxminus p$, is true whenever p is *true* and has been *true* at all points since time zero. For our application, *time zero* would generally be the start time of the application.

An example LTL timeline is shown in figure 1. Each time instant is labeled with a set of true LTL statements to illustrate the meaning of the *once*, *historically*, and *since* operators.

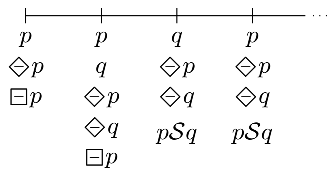


Figure 1: *True* expressions on an LTL timeline

Despite the word *temporal* in the name, the semantics of LTL do not have an explicit notion of time and are instead defined over a sequence of ordered discrete events and are undefined between those events.

3.2 MTL

Metric Temporal Logic extends the operators of LTL with the introduction of time intervals over which the operand expression must evaluate to *true*. Such intervals may be added to the *since*, *historically*, and *once* operators, but as we argued in [1] it is sufficient to monitor standard LTL operators and only intoduce metric temporal logic through the *once* operator $\diamondminus_I p$. Metric versions of *historically* and *since* can be synthesized as

$$\begin{aligned} \boxminus_I p &= \neg\diamondminus_I \neg p \\ p\mathcal{S}_I q &= \diamondminus_I q \wedge p\mathcal{S}q \end{aligned}$$

For the instantaneous case considered in this paper, the interval, I, must always be of the form $(0, \tau)$, $(0, \tau]$, $[0, \tau)$, or $[0, \tau]$ meaning that p must be *true* at at least one point within τ time units with closed or open intervals at either extreme.

3.3 Point-wise vs. continuous semantics

The original semantics of MTL[9] were the so-called point-wise semantics. That is, a fairly straightforward extension of the semantics of LTL. Expressions are only defined at discrete points with the addition of a timestamp on each point for evaluating the intervals.

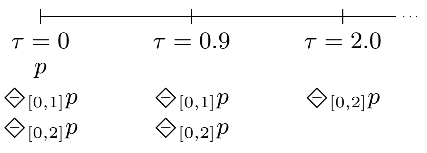


Figure 2: Pointwise semantics of MTL

We advocate for the use of dense-time boolean signals with continuous semantics that define both the input and output of MTL expressions in terms of them. In short, we use boolean signals that define a truth value at every time instant. This is a less-common, but not unprecedented, semantics in the model-checking and runtime-monitoring communities both of which benefit from the simplified algorithms of pointwise semantics or even discrete time. For FRP, we feel that it is a natural fit as it has historically [8] considered *behaviors* that vary over continuous time and composition is easier to reason about. For example, as Basin et. al. note in [2] under the point-wise semantics

$$\diamondminus_{[0,1]}\diamondminus_{[0,1]}p \neq \diamondminus_{[0,2]}p$$

Whether these statements are identical is a function of the sampling period. For a system that evaluates all expressions on a one-time-unit interval, they are always equivalent; for systems that only respond to external input, they will often not be equivalent.

Considering the example from figure 2 and extended in figure 3, where p is *true* at times 0.0, $\diamondminus_{[0,1]}p$ will be *true* at times 0.0, 0.9, and false at 2.0 because it is *true* whenever p is *true* and the gap between 0.9 and 2.0 is greater than one. Consequently, $\diamondminus_{[0,1]}\diamondminus_{[0,1]}p$ will be *true* at times 0.0, 0.9 and false at time 2.0, but $\diamondminus_{[0,2]}p$ will be *true* for time 0.0, 0.9, and 2.0.

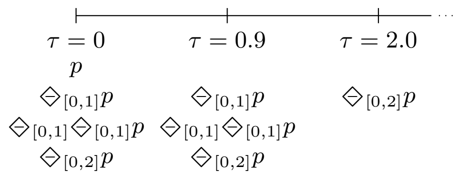
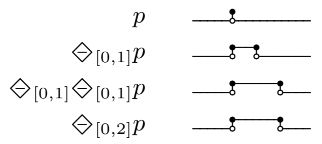


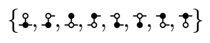
Figure 3: Unintuitive composition in pointwise semantics of MTL

Under the dense-time continuous semantics, the above example corresponds to p being *true* only at time 0.0 and false at all other times. $\diamondminus_{[0,1]}p$ will be *true* on the interval [0,1] and $\diamondminus_{[0,1]}\diamondminus_{[0,1]}p$ will be *true* on the interval [0, 2] which is identical to the interval over which $\diamondminus_{[0,2]}p$ is *true*.



3.4 Representing dense-time Boolean signals

Since a boolean signal can only take on two values, *true* or *false*, the uncountably infinite number of points over which its value is defined can be represented by a sequence of discrete transitions. Each transition isn't quite as simple as *true* or *false* because the dense time representation allows for a transition for which the value was false immediately before the time of the transition, *true* at the time of the transition, and then false immediately after the transition. In fact, this is our representation of an event; a statement that is *true* at only one instant. In our ealier work [1], we represented these transitions with eight transitions representing all combinations of truth immediately before, at, and immediately after the transition time.



Using that representation required that we enforce certain rules about adjacent transitions – namely that the outgoing truth value from the earlier transition must match the ingoing truth value of the later transition. It also led to awkward treatment of the initial transitions at time zero. In this paper we do away with both issues by representing transitions only in terms of the truth at the time of the transition and immediately afterward. This gives us four transition types

$$\{⊥, ↑, ↓, ⊤\}.$$

A boolean signal, s, can be represented as series of timestamped transitions $\{(\delta_0, \tau_0), (\delta_1, \tau_1), \dots\}$ with strictly increasing timestamps. The truth of a boolean signal, s, at time τ is given by

$$\mathrm{truth}(s,\tau) = \begin{cases} \delta \in \{↓, ⊤\} & (\delta,\tau) \in s \\ \delta \in \{↑, ⊤\} & (\_,\tau) \notin s \text{ with} \\ & \max \tau' \text{ s.t. } (\delta,\tau') \in s \wedge \tau' < \tau \end{cases}$$

This simply means that if there is a transition at time τ, then the signal is *true* only if the transition is one of the two that are *true* 'now', i.e. ↓ or ⊤. If not, then the truth of the signal is given by the transition immediately preceding time τ and in this case it is true only if it is one of the two transitions that are true immediately after the time of the transition, ↑ or ⊤.

4. APPROACH

We produced the FRP implementation of MTL operators in a modified Java-only subset of Sodium called Lithium[1] We elected to branch because we had anticipated more substantial changes to the core of Sodium – and more changes will be required to achieve a fully-precise implementation. But as of this writing, the only core change required was the introduction of a timestamp to the `Transaction` to support the correct scheduling of future events when modeling $\Diamond_I p$.

4.1 Timer Implementation

The $\Diamond_I p$ operator requires some kind of timer mechanism for producing the down transitions after the interval has expired with no change in input. We implemented that as timer-expired input and a single thread drawing timer-expiration events in the order of occurrence from a priority queue sorted by time. The queue supports the commands `addFutureEvent` and `cancelFutureEvent` for which the time of the future event is specified relative to the timestamp of the current `Transaction`.

We have added in a way that minimally impacts the core of the Sodium FRP implementation, but scheduling such an event is the sort of side-effect that should not be caused by any stage in a reactive framework. We know that it is sufficient with the current implementation of Sodium, but if its execution model were modified – even without a change in its semantics, we might no longer be able to rely on the prosecution of such external events.

Furthermore, there is no guarantee that the timeout thread will wake at precisely the correct time. This could allow input that arrived after the timer-expiration timestamp to incorrectly be processed before the timer expiration – potentially canceling a down transition that should have occurred had the timer-expiration event been processed first. This is a reasonably rare circumstance that we have been able to experiment with the use of MTL operators with this imperfect implementation, but a more correct implementation would require a more significant impact on the core FRP engine.

[1]the name was chosen to suggest a less-reactive framework since so much language support had been removed.

4.2 Transition values

As we described above, each update consists of one of four transition values $\delta \in \{⊥, ↑, ↓, ⊤\}$. At the instant of the transition, the value is considered *true* for $\delta \in \{↓, ⊤\}$ and *false* otherwise. Absent current transition, the signal for which the last transition was $\delta \in \{↑, ⊤\}$ will be considered *true* and will be considered *false* otherwise. We accomplish this by introducing a transition class that is aware of the `Transaction` during which it was created. When queried for its value during any other `Transaction`, its transition time can be assumed to be past and it will return the outgoing truth value: ⊤ for *true* and ⊥ for false.

4.3 Simple logical operators

The binary representation makes implementation of the operators ¬, ∧, and ∨ straightforward. Each is simply a matter of *lift*ing the appropriate function. To conserve space and demonstrate their use, we implement $p \vee q$ as $\neg(\neg p \wedge \neg q)$:

```
Transition negation(Transition p) {
  switch(p) {
    ⊥: return ⊤;
    ↑: return ↓;
    ↓: return ↑;
    ⊤: return ⊥;
  }
}

Cell<Transition> not(Cell<Transition> p){
  return p.map(negation);
}

Transition conjunction(Transition p, Transition q) {
  switch(p,q) {
    ⊥,⊥: return ⊥;
    ⊥,↑: return ⊥;
    ⊥,↓: return ⊥;
    ⊥,⊤: return ⊥;
    ↑,⊥: return ⊥;
    ↑,↑: return ↑;
    ↑,↓: return ⊥;
    ↑,⊤: return ↑;
    ↓,⊥: return ⊥;
    ↓,↑: return ⊥;
    ↓,↓: return ↓;
    ↓,⊤: return ↓;
    ⊤,⊥: return ⊥;
    ⊤,↑: return ↑;
    ⊤,↓: return ↓;
    ⊤,⊤: return ⊤;
  }
}

Cell<Transition> and(Cell<Transition> p,
                     Cell<Transition> q) {
  return lift( conjunction , p, q);
}

Cell<Transition> or(Cell<Transition> p,
                    Cell<Transition> q) {
  return not(lift( conjunction , not(p), not(q));
}
```

4.4 Since

Implementing *since* is somewhat complicated because its value at the current time depends in part upon its previous value. Even so, it consists solely of logical operations on itself and its inputs. The lambda expression `since_handoff` is used to detect the circumstance when p becomes *true* just as q becomes false such that there is no time at which both are false. And the lambda expression `since_hold` handles cases for which the previous value of the *since* expression was *true*.

```
Transition since_handoff(Transition p, Transition q) {
  switch(p,q) {
    $\overset{\circ}{\underline{\bullet}}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\circ}{\underline{\bullet}}$,$\overset{\overline{\circ}}{\bullet}$: return $\overset{\overline{\circ}}{\bullet}$;
    $\overset{\circ}{\underline{\bullet}}$,$\overset{\bullet}{\underline{\circ}}$: return $\overset{\bullet}{\underline{\circ}}$;
    $\overset{\circ}{\underline{\bullet}}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\overline{\circ}}{\bullet}$: return $\overset{\overline{\circ}}{\bullet}$;
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\bullet}{\underline{\circ}}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\overline{\circ}}{\bullet}$: return $\overset{\overline{\circ}}{\bullet}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\bullet}{\underline{\circ}}$: return $\overset{\bullet}{\underline{\circ}}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\overline{\circ}}{\bullet}$: return $\overset{\overline{\circ}}{\bullet}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\bullet}{\underline{\circ}}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
  }
}
```

```
Transition since_hold(Transition p, Transition s) {
  switch(p,s) {
    $\overset{\circ}{\underline{\bullet}}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\circ}{\underline{\bullet}}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\bullet}{\underline{\circ}}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\circ}{\underline{\bullet}}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
  }
}
```

The actual implementation of the *since* primitive relies on the Sodium concept of a `CellLoop` this allows the programmer to specify a reactive value as a function of itself. As of the time of this writing, one can not use `lift` on `CellLoop`s, but `snapshot` is allowed.

```
Cell<Transition> since(Cell<Transition> p,
                              Cell<Transition> q){
  CellLoop<Transition> s = new CellLoop<>();

  Cell<Transition> handoff =
          Cell.lift(since_handoff, p, q);
  Cell<Transition> hold = p.updates()
                    .snapshot(s, since_hold).hold(q);

  s.loop(or(q, or(handoff,hold)));
  return s;
}
```

4.5 Once

The *once* operator depends on a timer expiration event and provides an input `timout` on which to receive them. The `StreamSink` data type indicates a stream to which a program can explicitly send events, but Sodium enforces the restriction, that `StreamSink.send` can not be called by a listener; it must only be called outside of the FRP engine. In this case, on an external timer thread. We keep track of its currently-active timeout handler so that we can cancel it whenever the input becomes *true* before the timer expiration.

We maintain two streams of events, `ups` and `downs`, that are used to control the value of the timeout handler. Whenever any input event occurs that leaves the input in the *true* state, the timer is cancelled and an empty `Optional<TimerEntry>` instance is produced to reside in the handler. Whenever any input event occurs that leaves the input in the *false* state, a `TimerEntry` is created to produce the transition to *false* if no input transition occurs before the timer expires.

Beyond the timer accuracy issue mentioned above, there is another source of imprecision in this implementation. Sodium does not have a mechanism for merging simultaneous event streams. As of this writing, only one of the events will be sent. This is unlikely to be a serious issue when dealing with external events because real-world, dense-time, simultaneity occurs with probability zero. However, formulae can be constructed to produce a sort of diamond dependency on a single input and simultaneous updates become a possibility, so a perfect implementation would handle this case properly as well.

```
Transition up_transition(Transition v, Transition o) {
  switch(p,s) {
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\overline{\circ}}{\bullet}$;
    $\overset{\overline{\circ}}{\bullet}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\bullet}{\underline{\circ}}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\circ}{\underline{\bullet}}$: return $\overset{\overline{\bullet}}{\circ}$;
    $\overset{\overline{\bullet}}{\circ}$,$\overset{\overline{\bullet}}{\circ}$: return $\overset{\overline{\bullet}}{\circ}$;
  }
}
```

```
Cell<Transition> once_cc(Cell<Transition> p,
                              final long delay_ms) {
  StreamSink<Transition> timeout
          = new StreamSink<>();
  Cell<Optional<TimerEntry>> handler
          = new Cell<>();
  CellLoop<Transition> o = new CellLoop<>();

  Stream<Optional<TimerEntry>> ups
          = p.updates().filter($v \to v \in \{\overset{\overline{\circ}}{\bullet}, \overset{\overline{\bullet}}{\circ}\}$)
            .map(v->new Optional<>());
  Stream<Optional<TimerEntry>> downs
          = p.updates().filter($v \to v \in \{\overset{\circ}{\underline{\bullet}}, \overset{\bullet}{\underline{\circ}}\}$)
            .map($v \to v \in \{\overset{\bullet}{\underline{\circ}}, \overset{\overline{\bullet}}{\circ}\}$)
            .map($d \to$
              new Optional<>(
                addFutureEvent(delay_ms,
                           $t \to timeout.send(d)$)));
  handler = ups.merge(downs).hold(new Optional<>());
  p.updates().filter(v → v ≠ ⫰)
    .snapshot(handler,(t, h) → h)
    .filter(Optional::isPresent)
    .listen(h → cancelFutureEvent(h.get()));

  o.loop(p.updates()
    .filter(v → v ≠ ⫰)
    .snapshot(o, up_transition)
    .merge(timeout).hold(p.sample()));
  return o;
}
```

5. IMPLEMENTING DEBOUNCE

The debounce primitive of RxJava is a mechanism for reducing the number of input events that must actually be processed by subsequent stages of the reactive network. It only passes on those events that are not followed by another event within the time interval *period_ms*.

We implement this by maintaining a cell `lastValue` that contains the last value that has arrived on the input. This value will be sent after the time period has expired without an intervening event.

We also maintain a boolean signal named `arrivals` that represents the truth of the statement"an input event is arriving now", as such is it false except for the occasional infinitesimal interval of truth – represented as a series of ⫰ events. The boolean signal `debouncing` defined as $\Diamond_{[0,1]}$`arrivals` that indicates that an input value is ready to be sent.

We filter the transitions from `debouncing` until we observe a ⫰ event – indicating that the time period has expired with no event at which point the `lastValue` is sent on the output stream.

```
Stream<Type> debounce(Stream<Type> input,
                                        long period_ms){
  Cell<Optional<Type>> lastValue
            = input.map(i→new Optional<>(i))
              .hold(new Optional<>());
  Cell<Transition> arrivals = input.map(i→ ⫰).hold(⫯);
  Cell<Transition> debouncing
            = MTL.once_cc(arrivals,period_ms);
  return debouncing.updates()
    .filter(v → v = ⫰).snapshot(lastValue,(e,v)→v.get());
}
```

6. DISCUSSION AND FUTURE WORK

Our current implementation required only minimal modification to the Sodium FRP framework upon which it was based, but there were two limitations on the precision of its emulation of the real-time *once* operator. First, it relies upon an external timer that uses the Java `Thread.sleep` method to wait for timeouts. The only guarantee is that it will sleep for at least the requested time (unless interrupted), but it could sleep for an arbitrary additional amount of time. This time is likely to be short, but could lead to incorrect output if the timeout thread is starved for resources. Second, it is possible to construct a formula for which two timeout events are simultaneous. We have not found a suitable modification to the Sodium framework to handle this case properly.

In the near term, we intend to modify Lithium's `Transaction` update algorithm aware of the priority queue of waiting events to guarantee that timer expiration is handled before any input event that arrives after it should have expired. We continue to explore options for addressing the issue of simultaneous timer expiration and input arrival.

In the meantime, neither of these cases prevents the implementation of reactive constructs such as the `debounce` operation and other similar operations found in other reactive frameworks, but we would prefer that if MTL is to provide fundamental reactive primitives for describing such behavior, then it should implement the precise continuous semantics of dense-time metric temporal logic.

We have found that implementing such operations reliably requires that the core of the reactive framework or language be aware of the presence of timeouts and the need for a consistent timestamp for each event. Our experience suggests that, in general, there is no safe mechanism for adding real-time constraints to a reactive framework that does not support them natively. This is less of a concern for reactive frameworks that do not make guarantees of *glitch-freedom*, but for true FRP, it is important that the semantics of any supported operation be preserved regardless of the underlying implementation or runtime environment.

7. CONCLUSION

We propose the use of the MTL as a time-sensitive set of primitives from which the time-aware primitives of other reactive frameworks can be implemented. We claim that the dense-time, continuous semantics are the best fit within FRP because of their similarity to the FRP concept of *behavior*. We showed that they are nevertheless amenable to implementation within a push-based FRP framework due to the discrete range (binary) of values that Boolean *behaviors* may take on requiring only discrete updates. We then demonstrated the implementation of the RxJava *debounce* operator from the primitives of Sodium and the metric *once* operator.

We found that a precise implementation of the MTL semantics requires that the reactive framework be modified to be aware of scheduled future events, but do not believe that scheduling arbitrary future events is consistent with the declarative FRP style. As such, we argue that support for real-time constraints must be provided as a built-in feature of any FRP system that intends to support applications that depend on real-time constraints. We believe that the dense-time, continuous-semantics, of metric temporal logic MTL provide a reasonable set of operators for describing those constraints: It is based on a well established logic, accepts a reasonable efficient implementation, and maintains a representation of the truth of a formula at all points in time that is a natural fit for traditional FRP.

8. ACKNOWLEDGMENTS

Jianwei Niu is supported in part by NSF award CNS-0964710 and National Security Agency.

9. REFERENCES

[1] K. Baldor and J. Niu. Monitoring dense-time, continuous-semantics, metric temporal logic. In S. Qadeer and S. Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin Heidelberg, 2013.

[2] D. Basin, F. Klaedtke, and E. Zălinescu. Algorithms for monitoring real-time properties. In *Proceedings of the 2nd International Conference on Runtime Verification (RV)*, volume 7186 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, 2012.

[3] S. Blackheath. *Functional Reactive Programming [Early Access]*. Manning Publications Co., 6 edition, 7 2015.

[4] S. Blackheath. Rxjava backpressure. `https://github.com/ReactiveX/RxJava/wiki/Backpressure`, 2015.

[5] S. Blackheath. Sodium. `https://github.com/SodiumFRP/sodium`, 2015.

[6] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed rescala: An update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[7] C. Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[8] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[9] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 1990. 10.1007/BF01995674.