Logical Reactive Programming

Thierry Renaux, Lode Hoste, Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
{trenaux,lhoste,wdmeuter}@vub.ac.be

ABSTRACT

The reactive programming paradigm enables programmers to express interactive programs as a transformation from input streams to output streams. Multiple strands of reactive programming have been proposed in the past. We argue that none of the existing classes of reactive programming sufficiently caters to interactive programs in which patterns of discrete events determine how the program should react. We propose a novel class of reactive programming, called logical reactive programming, in which patterns of events can be declaratively specified using logical rules. We analyze the differences between logical reactive programming languages and traditional reactive programming languages, and initiate discussion on its limitations.

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

Keywords

Reactive programming, Logical reactive programming, Pattern matching

1. INTRODUCTION

In recent years, the paradigm of (functional) reactive programming became a popular means for modeling interactive programs. The different reactive programming languages and reactive frameworks varied somewhat [3]: some are embedded in pure functional languages and preserve referential transparency [6], others integrate into imperative, object oriented languages [9]. In some, reactivity is transparent, with reactive elements and 'dead' elements both as first-class entities in the language, whereas in others the reactive elements require special treatment [4]. Some languages allow arbitrary imperative control flow to affect reactive elements [6], others rigidly restrict even the use of reactive values in conditional branches [5].

All these approaches are considered "reactive", though, since they offer some degree of automatic recomputation whenever the input to the program changes. However, one other characteristic is shared by all these approaches, which is not in itself a defining trait of reactive programming: all these approaches react by means of applying a procedure to the changing inputs, which transforms the data into a new event stream. From there on out, an imperative backend takes care of updating displays, pushing packages through the network, etc. The reactive programs themselves, though, are restricted to merely transforming a set of streams using a procedure using operations such as `filter`, `map`, etc. We refer to this category of languages as "procedure oriented reactive programming languages", to contrast them with the class of languages this paper introduces.

A broad domain of interactive applications exists which are not cleanly expressible in procedure oriented reactive languages. In those applications, individual values in the event streams carry no meaning. Instead, meaning can be found by correlating multiple values from different events, potentially provided by multiple streams. Consider for instance gesture based interaction with a multitouch device. Events specifying where the display is currently being touched are useful for some interactions, but when detecting gestures, the meaning is present in the recent history of touch events; not in the value of the last touch event itself. Instead of reacting to individual events, reactions must be initiated in response to patterns of events.

The range of events that needs to be remembered differs from pattern to pattern. More importantly, the duration is expressed in term of real, "wall-clock" time. The time windows do not consist of a fixed number of events, but are overlapping windows containing all events occurring in some time interval relative to other events. Using procedure oriented reactive programming, programmers are again left to managing state manually, the onerous task reactive programming set out to free them from.

We propose a novel class of reactive programming called "logical reactive programming" which specializes in reactively detecting patterns in event streams. In logical reactive programming languages programmers declaratively specify the patterns of events to react to. The underlying bookkeeping and state tracking is transparently taken care of by the runtime, exposing only the high-level concepts of events and reactive patterns.

This paper introduces the primitive constructs of logical reactive programming (section 2). The ways in which these cooperate to form a full-fledged interactive program is demonstrated by a reactive gesture detection program (section 3). We use the Midas [12] language as an example of a logical RP language. We further discuss the differences and similarities with procedure oriented reactive programming languages (section 4), as well as with related work from the domain of Complex Event Processing (section 5).

2. LOGICAL REACTIVE PROGRAMMING

2.1 Detecting Patterns

The central pillars of logical reactive programming are pattern matching and unification. A logical reactive program specifies that it reacts to the streams S_1, S_2, S_3, etc. by declaring a pattern p:

$$ce_1 \wedge ce_2 \wedge ce_3 \wedge \ldots$$

in which each condition element ce_x takes the form

$$e :: S \text{ where } c.$$

A condition element identifies a stream S, and specifies that each concrete event instance from S should in turn be bound to e, but only if condition c is satisfied.

Synchronizing streams is achieved using conjunction. A conjunction

$$(e_1 :: S_1 \text{ where } c_1) \wedge (e_2 :: S_2 \text{ where } c_2)$$

causes a pairwise combination of all events on stream S_1 with all events on stream S_2. Any boolean predicate can serve as condition, but conditions can refer to events bound previously, such that condition c_1 can refer to the event bound to e_1, while condition c_2 can refer to both events bound to e_1 and e_2.

When a new event arrives on one of the streams, a reactive recomputation must be triggered. However, earlier events stay relevant: a new binding for e_1 has to be combined in a pairwise fashion with all previous bindings for e_2, not just with the latest binding. For each pair (e_1, e_2) for which both conditions c_1 and c_2 hold, a pairwise binding with all previous bindings for e_3 must be made, and so forth for the remaining events. This can be implemented efficiently using the Rete algorithm [7].

Conditions may refer to the temporal properties of events, and hence define temporal constraints between events. A set of bound events, with temporal constraints one to another, define a series of nested windows over events. The green markings in Figure 1 depicts how events on stream $S1$ project windows on stream $S2$ by specifying a fixed lower and upper bound on inter-event timing. Similarly, the red markings show the event windows defined by each event, as children of a green window.

2.2 Reacting to Patterns

In addition to patterns, logical reactive programs contain modifiers. Modifiers specify a way of reacting, and can be written intermingled with patterns. Such a combination of patterns and modifiers form a declarative rule. Within a rule, at least one pattern must occur before the first modifier, for otherwise the modifier should trigger without triggering event: rules are never invoked, they react automatically when the event streams they listen to change. Similarly, the last element in a reactive rule must be a modifier, for otherwise the matching of the last pattern is without observable effects[1].

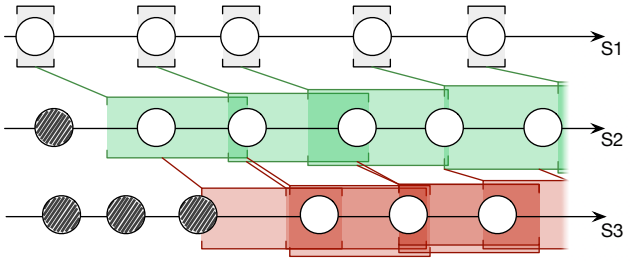

Figure 1: Time windows starting at an offset relative to occurrences of event on another stream

Four types of modifiers can be used inside a rule: `assert`, `retract`, `modify`, and `call`. An assertion modifier creates a new event using data from events bound higher up in the rule, and emits it as a stream. Assertions hence form the out-edges of a rule. Generated streams are implicitly merged with other streams of events with same 'type': if two rules assert events of type S, rules depending on stream S will receive events from both rules.

Retraction modifiers serve a dual purpose: they too create a new event using data from events bound higher up in the rule, but causes the removal of an earlier identical event from the stream[2]. `modify` modifiers combine the retraction of an event and the assertion of an event based on the retracted event.

Finally, `call` modifiers make it possible to invoke behavior through a foreign function interface.

2.3 Programming Abstractions

The basic abstraction for logical reactive computation are rules, as introduced above. The basic abstraction for data are events, which are sets of attribute-value pairs with a type tag. These type tags identify the event's template. Event templates are 'blueprints' for events, and are similar to class definitions in object-oriented languages. Most importantly, templates specify the attribute that an event will contain. An attribute `time` is always present in an event template, even if the developer does not specify it.

In addition, event templates may define `attempts` [13], which are predicates which bundle a set of condition elements, and only return a value when all their conditions are met. Developers can use attempts to abstract code whenever multiple rules share similar conditions. Examples are temporal conditions such as Allen's interval algebra [2] and spatial relations.

[1]Since rules are never invoked, there is also no "call site" to return to.

[2]Rules can be assigned priorities, such that rules with higher priority can suppress events before lower priority rules get a chance to react.

A common scenario in the domain of gesture programming is that two rules only differ in a few spatio-temporal values. In logical RP, code reuse and abstraction are obtained by moving shared conditions into an attempt. Furthermore, attempts provide parameterization and therefore enable customization.

Modules are used to group reusable functionality such as attempts and attribute definitions. They are similar to templates but cannot be instantiated. Modules can be included inside other modules and templates, and act similar to mixins in object-oriented languages such as Ruby. Templates can hence be built by composing multiple modules, automatically acquiring their attributes and attempts.

3. MOTIVATING EXAMPLE: GESTURES

Consider a reactive program underlying the user interface of an application. In addition to button-presses and accelerometer data, the user should be able to interacts with the application by means of multitouch gestures. Since touchscreens provide event-data specifying coordinates of touches, reactive programming can be leveraged to react to touches. As we will demonstrate, implementing even simple gestures such as a "swipe right" gesture requires state management and is problematic in procedure oriented reactive languages. First, though, we will show how this problem can be tackled using Midas [12], a logical reactive programming language.

3.1 Logical Reactive Solution

Listing 1 shows the definition of a rule `swipeRight` which reacts to a series of three `Touch2D` events which jointly form a swipe to the right. To keep the example simple, the reaction consists solely of the assertion of a `SwipeRight` event using the source events' properties. Line 2 specifies that each event from stream $Touch2D$ will be bound to $t1$. Similarly, line 3 binds each event from stream $Touch2D$ to $t2$, one at a time. The body of a rule forms an implicit conjunction, such that event bound to $t1$ will be paired in a Cartesian product with events bound to $t2$. Contrast this to procedure oriented reactive languages, in which such code would bind $t1$ and $t2$ to the same value in the same turn[3]. This ability to refer to events arbitrarily far apart in the stream provides logical RP with the expressive power we leverage in this example. While we will enforce strict temporal relations between the events, we will not require $t1$ to be any fixed number of *changes* removed from $t2$.

With the first two event instances bound, lines 4 and 5 specify temporal and spatial requirements[4]. Lines 6 through 8 bind two more event instances, and install similar tests. Lines 4, 7, and 9 jointly establish a firm temporal dependency between all event instances. In addition to specifying the runtime semantics of the reactive program, this dependency is sufficiently strict that it can be used to perform a reachability analysis on event data. This enables automatic memory management of event data by discarding old events

[3]Alternatively, there could be an intermediary stage where either $t1$ or $t2$ is updated to the last value, but the other still refers to the previous. However, that would normally be considered a glitch.

[4]The concrete pixel values – e.g., `277.px` – are automatically extracted from a dataset to optimize the recognition rates [8].

Listing 1: Swipe right gesture spotting rule in LRP

```
 1 rule swipeRight
 2   t1 = Touch2D
 3   t2 = Touch2D
 4   t1←beforeF t2
 5   t1←translated_nearF t2, 277.px, 5.px, 76.px
 6   t3 = Touch2D
 7   t2←beforeF t3
 8   t1←translated_nearF t3, 647.px, 10.px, 76.px
 9   t3←withinF t1, 100.ms, 1000.ms
10   no { b = Touch2D
11        b←afterF t1
12        b←beforeF t3
13        Math.abs(t1.y -b.y) > 184.px }
14   assert SwipeRight { x_lo ⇒t1.x, y_lo ⇒t1.y,
15                       x_hi ⇒t3.x, y_hi ⇒t3.y,
16                       time_lo ⇒t1.time,
17                       time_hi ⇒t3.time }
18 end
```

when their presence or absence is no longer externally discernible [10].

Before finally reacting, a bounding box must be checked on the gesture. If the swipe does not sufficiently closely resemble a horizontal stroke, the gesture must be invalidated. To this end, lines 10 through 13 required the absence of an invalidating event. No event instance b may be found on stream $Touch2D$ that occurs after the event bound to $t1$, before the event bound to $t4$, and whose `y`-value differs too much from $t1$'s. This is expressed by logical negation, which has negation-as-failure semantics.

3.2 Procedure Oriented Reactive Solution

Detection of the same "swipe right" gesture can be written in procedure oriented reactive languages, though we will argue the logical reactive version is both clearer and more readily optimized by a specialized runtime. Figure 2 demonstrates how the windows from Figure 1 can be thought of as a directed acyclic graph rooted in the events from the first stream. Figure 3 shows the pairwise matching needed to construct this graph. Using this idea, we did a best effort attempt to express the swipe right gesture pattern's logic in a procedure oriented RP solution. The gist of the approach is to make a Cartesian product between all streams, and filter out combinations which do not meet the constraints. This translates rather naturally to an interleaving of (`flat`)`map` and `filter`. A partial implementation in RxJava [1] can be found in Listing 2. The code 'marches off to the right', though that's merely an artifact of the implementation language. Functional languages tend to deal more elegantly with series of monadic *and-then* operations. Regardless, three real issues exist with this partial solution:

1. The explicit nesting of closures stands in contrast to the idea of reactive programming, and being freed from the "callback hell". While the logic to express is reaction logic, *procedure oriented reactive languages are not able to express this problem as a reactive problem.*
2. The program in Listing 2 *omits an important part of the functionality* from the Midas code in 1. The required absence of an event `b` which diverges too far

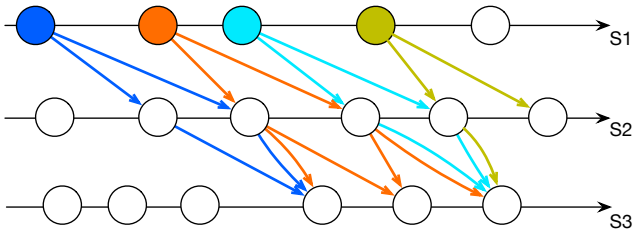

Figure 2: The directed acyclic graph implied by the windows in Figure 1

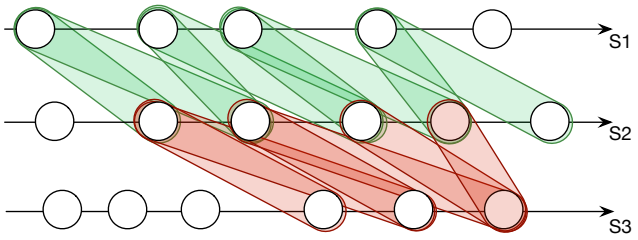

Figure 3: The pairs of events that need to be combined to form the match graph in Figure 2

from the horizontal line through `t1` is not checked in the RxJava program. The omission stems from our inability to express the time-sensitive absence of events in a clean manner. Time is monotonously non-decreasing. Hence, no event meeting the constraints on `b` can exist once `ts` has been processed up to 1000ms past `t1`. Conveying this to the runtime using existing procedure oriented RP constructs is a prohibitively error prone exercise. Yet without this information, the runtime cannot figure out when the absence of an event matching `b` can be guaranteed. Using system time as a proxy for the logical time up to which events have been processed leads to glitches.

3. The reachability analysis underlying the event memory reclamation in Marr et al. [10] works like an automatic garbage collector when applied to logical RP. It leverages the monotonicity of time, combined with information from the latest events of all other streams used in a rule to determines which events are no longer relevant. For instance, in Figure 1 the crossed-out events can be automatically detected and discarded. It is not clear how to implement this using the general purpose stream operators provided by procedure oriented RP languages, such as `map` and `filter`. These operators do not support the notion of event expiration.

   Unfortunately, porting the event expiration technique to the stream operators, impacts the operators' semantics. It requires that each (`flat`)`map` instantiates a distinct stream instance, and that such a stream instance should lose references to events based on the following criterion: the other stream instances that are mapped over in the same lexical block are guaranteed never to exposes a set of events anymore which passes all the `filter`s defined in that lexical block. Put another way, in Listing 3, the stream instance underlying `s2` should drop an event as soon as it is the case that

Listing 2: A RxJava program with similar behavior

```
1  Observable<Touch2D> ts = getTouch2DStream();
2  Observable<SwipeRight> swipeRights =
3    ts.flatMap(t1 ->
4      ts.filter(t2 -> t1.before(t2))
5        .filter(t2 ->
6          t1.translatedNear(t2, 277, 5, 76))
7        .flatMap(t2 ->
8          ts.filter(t3 -> t2.before(t3))
9            .filter(t3 ->
10             t1.translatedNear(t3, 647, 10, 76)))
11           .filter(t3 ->
12             t3.within(t1, 100, 1000))
13           .map(t3 -> new SwipeRight(t1.x, t1.y,
14                                     t3.x, t3.y,
15                                     t1.time,
16                                     t3.time)));
17 swipeRights.subscribe(handleSwipeRight);
```

Listing 3: A nesting of `filter`s and (`flat`)`map`s

```
1 s1.filter(e1 -> c1(e1))
2   .flatmap(e1 ->
3     s2.filter(e2 -> c2(e1, e2))
4       .flatmap(e2 ->
5         s3.filter(e3 -> c3(e1, e2, e3))
6           .map(e3 -> react(e1, e2, e3))
```

binding that event to `e2` is guaranteed never to trigger `react(e1, e2, e3)`. To this effect, the runtime needs to understand both the bounds on future values on the streams `s1` and `s3`, and the conditions `c1`, `c2`, and `c3`. Since temporal constraints are not handled separately from e.g., spatial constraints, the runtime cannot even zoom in on the constraints which indeed are restricted to monotonously non-decreasing streams.

A manual approach is possible, where stream instances are manually constructed as overlapping `window`s that are created for each new element on the stream, and closed only when the time window ends. However, this faces the same issues as dealing with negation. Programmers are required to reinvent and implement a logical RP runtime, or otherwise their *programs necessarily retain all event data indefinitely*. In addition to depleting the finite resource of working memory, the code also becomes increasingly slow as the nested `flatmap`s keep looping over all old events, even though none of these events will ever pass a `filter` on temporal constraints anymore. Partial solutions, such as limiting the amount of events stored for each source stream to some fixed timespan, fail when a stream is used in multiple locations in a reactive program, once for high-frequency/short duration reasoning and once for low-frequency/long duration reasoning: while the a logical RP runtime would have no trouble maintaining e.g. both a stream tracking one event per hour over multiple months, and a stream tracking thousands of events per second over a five second period, the naive solution would have to cater to the 'lowest common denominator' of thousands of events per second over a multi-month period.

4. DISCUSSION: PROCEDURE ORIENTED AND LOGICAL RP

Procedure oriented and logical reactive programming offer two distinct, yet complementary paradigms for reactively handling events. Logical RP fills a gap which procedure oriented languages fail to satisfy. In Section 3 we demonstrated three issues with reactively detecting patterns using procedure oriented RP languages: *1*) the return to *Callback Hell*, *2*) the inability to declaratively react to the absence of events, and *3*) the space leak caused by the reactive runtime's inability to detect when events become irrelevant. This section discusses more generally how both classes of reactive programming relate.

4.1 Lifting

Reactive programs are generally built using the same language constructs as non-reactive languages, although reactive semantics are given to some constructs. An interesting feature of reactive languages is hence how constructs are *lifted* to the reactive world. In some languages, lifting happens transparently, while in others procedures need to be manually lifted to be able to deal with reactive inputs [3]. In logical RP, operations on streams are implicit. Conditions are checked on specific, bound event instances, not by applying a `filter` on a stream. Similarly, values that go into newly created events are computed from bound instances of other events, not by manually (`flat`)`map`ping over a stream. We argue that the nature of logical RP is biassed towards transparent lifting. Logical RP languages allow arbitrary, non stream-aware operations to be applied to reactive values, without transforming the operations or explicitly calling stream operators. The runtime transparently ensures these operations are reapplied whenever the values change.

4.2 Synchronization between Streams

When a part of a reactive program depends on multiple streams, the program defines a synchronization of those streams. When pull-semantics [3] are used, this boils down to sampling all streams, yielding the latest values. For push-semantics [3], the appearance of new events triggers a computation, and the latest values that appeared on the other streams are reused. In stream operators, this could be expressed using `combineLatest`. This mode of synchronization handles differences in the rate at which events arrive on the different streams gracefully. To achieve other kinds of rate matching, different stream operators can be used. For instance, the synchronization model of the traditional dataflow execution model [11] can be mimicked using `zip` or `And/Then/When` [1]. As discussed above, logical RP can be approximated using `flatmap` and `filter`.

Changing the synchronization model inherently changes the dynamics of the reactive program. In procedure oriented reactive programming, each reactive operation takes in one or more reactive streams, and produces a new stream. For each change on the input streams, zero or one changes is generated on the output stream. This is not the case for logical RP. *A single new event may cause an arbitrary number of output values to be generated.*

Consider the following pattern:

$$e_1 : S_1 \wedge e_2 : S_2 \text{ where } e_1 \text{ withinF } e_2, -1\text{.s}, 1\text{.s}.$$

When an event e_{1a} arrives on stream S_1, and in the next half second events e_{2a}, e_{2b}, and e_{2c} arrive on stream S_2, three bindings are created: (e_{1a}, e_{2a}), (e_{1a}, e_{2b}), and (e_{1a}, e_{2c}). Up till here, the only difference with traditional RP behavior is that a first e_{1a} was not coupled with some initial value of stream S_2. However when a second event, e_{1b}, is emitted on stream S_1, the different dynamics of logical RP become clear. On arrival of e_{1b}, three additional bindings are generated: (e_{1b}, e_{2a}), (e_{1b}, e_{2b}), and (e_{1b}, e_{2c}). As mentioned before, a Cartesian product of all events is incrementally generated as events occur. The consequence is that the rate of events inside a logical reactive rule may be greater than the rates of events of the streams it reacts to. This change in the dynamics of routing updates in response to external changes entails that the notion of atomic 'turns' during which reactivity can be ignored does not work out for logical RP. The use of `flatmap` in our procedure oriented reactive gesture detection example originates from this need to merge the tree of updates fanning out from the conjuncts in the logical pattern.

4.3 Synchronizing with the same Stream

The example program in Section 3 was peculiar in more than just syntax and type of synchronization. The example used multiple events from the same stream. The events reacted to are not just driven by the same primitive signal (such as e.g. a `seconds` behavior [9]). No transformations are even applied to the streams (such as e.g. using `seconds` in combination with `(+ seconds 1)`). Instead, the `Touch2D` stream is combined directly with the same `Touch2D` stream. As a result, each `Touch2D` event is at some point bound to both `t1` and `t2`, only to have this pair discarded after the temporal constraint that `t1` should occur before `t2` is processed. The latter is however a choice made in application logic: it is in general perfectly legal for a logical RP program to bind the exact same event to two logical variables in the same pattern. Contrast this to procedure oriented RP programs, in which only this latter approach is possible. In procedure oriented RP, the fact that multiple snapshots of the same stream yield the same value at the same time is even considered a correctness requirement. Procedure oriented reactive languages lack clean, reactive abstractions for matching multiple discrete events which are arbitrarily far apart but originate from the same stream.

5. COMPLEX EVENT PROCESSING

An approach related to logical RP is Complex Event Processing. CEP's goal is handling complex events, i.e., events consisting out of multiple more primitive events. CEP is hence closely related to logical RP, since logical RP's central notion of patterns serves to define such complex events in terms of more primitive events. The differences between CEP and logical RP therefore merits discussion.

A primary difference is Logical RP's focus on event instances. Patterns in Logical RP bind discrete event instances, and are hence able to steer the reaction using properties from individual events. In contrast, CEP focusses on aggregation, exposing e.g. averages or minima over event streams, or counting the number of occurrences.

CEP programs are written as continuous queries. Reaction logic is absent or 'bolted on'. Logical RP, on the other hand, aims to feel more like traditional programming. The running example in this paper focussed on contrasting logical RP with its sibling classes of reactive programming, and hence does not demonstrate this property. However, logical RP's ability to interleave conditions and reactions significantly reduces friction in encoding meaningful reaction logic directly in the language. Logical RP languages offer programming abstractions for conditional computation (`attempt`s), and for grouping properties of data (`module`s). This facilitates building entire programs in logical RP, depending only on a runtime system for primitive input and output. We refer to Renaux et al. [12] for a discussion of the applicability of an LRP language for writing complete programs.

6. CONCLUSIONS

We have presented a new class of reactive programming languages called logical reactive programming. We argue that logical RP is more suited than more traditional, procedure oriented reactive programming languages at expressing patterns of events, where semantics stem from complicated correlations between events, spread out over longer periods.

We hope to initiate discussion on how best to combine logical and other classes of reactive programming, and how to embed them in existing languages in a similar way as how Reactive Extensions are integrated using LINQ.

References

[1] RxJava – Reactive Extensions for the JVM. `https://github.com/ReactiveX/RxJava`. Accessed: 2015-08-16.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983. ISSN 0001-0782. doi: 10.1145/182.358434. URL `http://doi.acm.org/10.1145/182.358434`.

[3] E. Bainomugisha, J. Vallejos, C. De Roover, A. L. Carreton, and W. De Meuter. Interruptible context-dependent executions: a fresh look at programming context-aware applications. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, pages 67–84, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1562-3. doi: 10.1145/2384592.2384600. URL `http://doi.acm.org/10.1145/2384592.2384600`.

[4] A. Courtney. Frappé: Functional reactive programming in java. In *Practical Aspects of Declarative Languages*, pages 29–44. Springer, 2001.

[5] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM, 2013.

[6] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[7] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982. ISSN 0004-3702.

[8] L. Hoste, B. De Rooms, and B. Signer. Declarative Gesture Spotting Using Inferred and Refined Control Points. In *Proceedings of ICPRAM 2013, 2nd International Conference on Pattern Recognition Applications and Methods*, Barcelona, Spain, Februari 2013.

[9] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Functional and Logic Programming*, pages 259–276. Springer, 2006.

[10] S. Marr, T. Renaux, L. Hoste, and W. De Meuter. Parallel Gesture Recognition with Soft Real-Time Guarantees. *Science of Computer Programming*, February 2014.

[11] R. S. Nikhil. Can dataflow subsume von neumann computing? *SIGARCH Comput. Archit. News*, 17(3):262–272, Apr. 1989. ISSN 0163-5964. doi: 10.1145/74926.74955. URL `http://doi.acm.org/10.1145/74926.74955`.

[12] T. Renaux, L. Hoste, C. Scholliers, and W. De Meuter. Software Engineering Principles in the Midas Gesture Specification Language. In *Proceedings of PRoMoTo 2014, 2nd International Workshop on Programming for Mobile and Touch, Portland*, Portland, Oregon, USA, October 2014.

[13] C. Scholliers, L. Hoste, B. Signer, and W. De Meuter. Midas: A Declarative Multi-Touch Interaction Framework. In *Proceedings of TEI 2011, 5th International Conference on Tangible, Embedded and Embodied Interaction*, Funchal, Portugal, Jan 2011.