# Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model

Sam Van den Vonder
Vrije Universiteit Brussel
Brussels, Belgium
svdvonde@vub.be

Joeri De Koster
Vrije Universiteit Brussel
Brussels, Belgium
jdekoste@vub.be

Florian Myter
Vrije Universiteit Brussel
Brussels, Belgium
fmyter@vub.be

Wolfgang De Meuter
Vrije Universiteit Brussel
Brussels, Belgium
wdmeuter@vub.be

## Abstract

In his famous paper entitled "Tackling the Awkward Squad", Peyton Jones studies how features that traditionally did not fit in the functional programming paradigm can be added to a functional language via careful language design (e.g. using monads), instead of allowing programmers to sprinkle around impure expressions and ad-hoc library calls, thereby turning the entire program into a non-functional program. Similarly, in this paper, we identify a number of code characteristics that do not map onto the reactive programming paradigm but that are present in many real life reactive programs. We propose a novel Actor-Reactor model that can serve as the basis for future language designs that allow a programmer to use the awkward squad without making the reactive parts of the program accidentally non-reactive.

***CCS Concepts*** • **Software and its engineering → Data flow languages**; **Multiparadigm languages**;

***Keywords*** functional reactive programming, actors, reactors, the actor-reactor model

**ACM Reference Format:**
Sam Van den Vonder, Joeri De Koster, Florian Myter, and Wolfgang De Meuter. 2017. Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model. In *Proceedings of 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'17).* ACM, New York, NY, USA, 7 pages.
https://doi.org/10.1145/3141858.3141863

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*REBLS'17, October 23, 2017, Vancouver, Canada*
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5515-5/17/10…$15.00
https://doi.org/10.1145/3141858.3141863

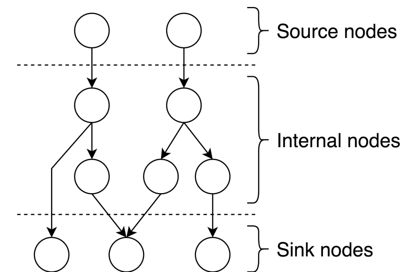


**Figure 1.** Structure of a reactive DAG

## 1 Introduction

Reactive programming is a highly declarative way to write event-driven programs. It is based around expressions (signals) that automatically recompute whenever the value of a sub-expression changes. As an example, consider the reactive program `c = a + b`. Any changes to the variables `a` or `b` automatically give rise to a recomputation of `c` as well. This is typically done by compiling the reactive program into a directed acyclic graph (DAG), as exemplified in figure 1. The free variables of the program (in our case `a` and `b`) correspond to sources of the DAG, and the expressions that have no depending expressions correspond to sinks of the DAG.

In the past decade we have seen a lot of reactive languages and reactive libraries written in mainstream languages. Their designs focus on the concepts for programming the **internal part** of a reactive system. Speaking in terms of the DAG, they focus on language features and ever more powerful higher-order abstractions that allow programmers to express those DAGs as easily and as declaratively as possible.

In the rest of the paper we will talk about reactive programs in terms of their DAG, which features three kinds of nodes. *Source* nodes correspond to the "input" signals of the reactive program. They are typically provided by some *external world. Internal* nodes are composed signals that constitute the actual semantics of the reactive program. *Sink* nodes correspond to the "output" signals of the reactive program. They are typically connected to the GUI, either implicitly (as the result of a `print` in the REPL in FrTime [2]) or explicitly

(e.g. by wiring signals to the DOM using `insertValueE` in FlapJax [5]).

In recent years reactive programming has gained a lot of popularity in mainstream software development, especially in the domain of web development. However, one cannot help but observe that reactive programming is only used for certain parts of the application: The reactive program is embedded in an application still written in ordinary imperative languages such as JavaScript or Scala. This **external part** of the reactive program typically consists of stateful program logic (e.g. the GUI rendering algorithm) and offline "computations". This desired coexistence means that both the input and output signals of the reactive program should interact in a predictable manner with the parts of the program written in the host language.

Mechanisms that support a clean embedding of reactive programs in the imperative parts of the program are poorly investigated in the literature. As we will explain, this is especially the case if those imperative parts correspond to "long lasting" computations, "very stateful" computations, and (distributed) multi-threaded computations. In analogy with Peyton Jones [6] we call this **the awkward squad for reactive programming**, since they are all aspects of a reactive software system that "do not neatly fit into the reactive paradigm", and that require an escape hatch to code written in another (mostly imperative) programming paradigm. This paper focuses on the programming language features that correspond to the "external world" of a reactive system. We present an experimental model in which a program is defined as a set of *actors* and *reactors*. The reactors are ordinary reactive programs, i.e. collections of expressions that depend on each other but which have free variables that correspond to the sources of the associated DAG. Actors are language entities with their own thread of execution. They can independently produce values (thereby producing the input for reactors) and they can act upon their internal state e.g. for I/O (thereby processing the output produced by reactors).

The main contributions of this paper are: (1) The definition of the "awkward squad for reactive programming", a set of non-reactive (imperative) application concerns that are currently difficult to express in reactive programming languages, and (2) the definition of the Actor-Reactor Model that cleanly separates these reactive and non-reactive concerns, thus allowing them to co-exist within the same application.

## 2 Problem Statement

In this section we analyse the problems that may occur when embedding a reactive program in a mainstream language, or when allowing the nodes of a reactive program to be programmed with the full power of a Turing complete (imperative) language.

### 2.1 Long Lasting Computations

Suppose that one of the DAG's nodes reacts to incoming web-based data by running some long lasting machine learning algorithm. Such long lasting computations are expressed using iteration constructs or by means of recursive procedures. They can be problematic to combine with reactive programming since they tend to "hijack" the execution thread, thereby temporarily stopping the reactive program. Long lasting computations can also emerge in more subtle ways. When one of the internal nodes iteratively computes the length of an incoming string, this will result in long lasting computations whenever long strings enter the DAG.

The idea of reactive programming is to automatically (re)execute the expressions that depend on source signals "every time" an external agent changes their value. But what do we mean exactly by "every time"? Consider a reactive program that reacts to an incoming third-party distributed data stream. To ensure that the program does not drop too many incoming values (or risk a buffer overflow) the reactive program should update itself "as often as possible". To *guarantee* this we necessarily have to assume that the execution time of the reactive program does not depend on the incoming values. In other words, the execution time of the reactive program has to be $O(1)$. This may sound controversial but this property is truly at the heart of *reactivity*: A program that is not in $O(1)$ will be "busy" for a period of time the duration of which depends on input provided by the external agent. During this offline period it is no longer reacting to external agents, and is therefore not *reactive* [1]. Since most languages and frameworks do not impose reactive programs to be in $O(1)$, we will call reactive programs whose nodes do satisfy this property **strongly reactive**.

Combining the need for long lasting computations (e.g. a machine learning algorithm) with strong reactivity will require a combination of at least two execution threads. One thread is the traditional reactive program that takes care of updating the DAG at regular times. The other thread runs the long lasting computation for as many incoming values as possible. The strategy used by the latter to cope with the input overflow is an orthogonal decision as long as it does not block or hijack the reactive thread (thereby making the reactive program "weakly reactive").

The language design problem that needs to be solved is how to allow the coexistence of multiple threads in a reactive program. Some threads correspond to the update algorithm of the strongly reactive parts of the program, and other threads take care of long lasting computations (which are typically stateful – see next section). The semantics of the interaction and coordination of all those threads should be simple and well-defined.

[1] With our definition of reactive we blur the lines between *reactivity* as a highly declarative means to write event-driven programs, and the expectation that those programs are highly responsive to change, such as user input.

### 2.2 Effectful Statements

Effectful statements such as assignments and output statements are extremely tricky to understand and debug in the internal nodes of a reactive DAG. That is because the update order of the nodes is not part of the semantics of the reactive program (apart from the fact that it typically guarantees glitch freedom), so the order in which the effects are executed is no longer under the programmer's control. Two nodes of the DAG that have an effect on the same *external* resource (e.g. a shared variable, a log for debugging, or some hardware actuator) will typically result in effects whose order depends on the implementation of the DAG update algorithm, and are thus very elusive for the reactive programmer. The problem gets worse when the reactive update algorithm is internally parallelised to improve the throughput of the reactive program, since accessing the shared resource will now also lead to race conditions.

The language design problem that needs to be be solved is how to allow the coexistence of effectful computations that react to values arriving at a certain internal node of the DAG without accidentally or non-deterministically affecting the behaviour of effectful computations that reside in other internal nodes of the DAG. A simple locking scheme with mutexes that regulate the order of the effects in the DAG's internal nodes is not a good solution since locking will result in a weakly reactive program. Hence, the effectful computations have to be evacuated outside the DAG and should be responsible for their own interaction and mutual coordination, while the DAG's update thread is free to work independently.

### 2.3 Embedding the DAG in the External World

When embedding a reactive program in a multi-threaded imperative program, multiple threads can modify the source nodes of the DAG (either sequentially or in parallel). This means that the reactive DAG itself becomes a stateful resource that is shared between various external threads. The result is a very subtle combination of parallel DAG update cycles that may affect each other in case the necessary bookkeeping is not implemented to keep updates from different threads separated. This implies either a mutex to synchronise all external threads accessing the DAG, or using an asynchronous message passing strategy and buffering all contributions to the DAG from one thread until contributions of the other threads are completely processed by the DAG. The issue then is that the reactive program receives messages from threads that produce values at different paces, or at paces that exceed the capacity of the reactive program (leading to buffer overflows).

Currently the source nodes of a DAG are typically changed via two different mechanisms:

1. All languages and frameworks feature a number of built-in reactive variables (such as `seconds` in FrTime [2]). These are free variables in the reactive program that are automagically updated by a computational process external to the reactive program.
2. Many languages embed a reactive program into an imperative program whose role it is to provide the inputs of the DAG. For example, in REScala, `Vars` are signals that are explicitly updated via assignments by the imperative part of the program [7]. In Flapjax, `$B` and `$E` construct new input signals that are updated via specific events in a web page [5].

The language design problem that needs to be solved is how to cleanly combine the external thread (or threads) that "fill the sources" of the DAG with the thread that is responsible for the reactive program. We have to find expressive declarative means that allow a programmer to "couple" the external threads to the reactive thread, to coordinate the threads, and to express what needs to be done when external threads are producing values at a much higher pace than what can be handled by the reactive thread.

## 3 The Actor-Reactor Model

The general idea of the actor-reactor model is to fully embrace the existence of the different "worlds" that constitute the "inside" and "outside" of reactive programs, to give these worlds their own thread and effect-set, and to design simple and well-defined operators to link them together. We have done this by allowing a programmer to (a) evacuate stateful and/or long lasting computations into a number of actors that manage their own state, thread, and input-output, (b) express multiple DAGs (called reactors) that each correspond to a reactive program with its own update thread, and (c) defining a preliminary set of operators that allow actors and reactors to "listen" to each other and "emit" values that can be accepted and processed by listening actors and reactors.

### 3.1 Running Example

We have built a prototype implementation of the Actor-Reactor model in Racket. We introduce our language constructs by building an instance of the CheerLights service. CheerLights is an Internet of Things project that synchronises a global network of multicoloured lights controlled by Twitter [8]. The project parses names of colours from tweets sent to the project's account, and publishes the corresponding RGB value to an API. For example, the tweet "*I love purple! #CheerLights*" is transformed into `RGB(0.5, 0, 0.5)`. Hobbyists use this API to illuminate their IoT devices. In the following sections we gradually build our application using two actors (Listing 1 and 2) and one reactor (Listing 3). The actors are responsible for (1) fetching real-time data from Twitter, and (2) displaying the results of the CheerLights service in a GUI. The reactor implements the CheerLights service itself and transforms names of colours to RGB values.

The actors and reactors of the application are instantiated and wired together in Listing 4.

### 3.2 Actors

The Actor model is a concurrency model based on asynchronous message passing and encapsulation of state. Actors are typically defined in terms of a *behaviour* and an *inbox* [4]. The behaviour of an actor defines its internal state and the messages the actor can process (i.e. its interface) [2]. The inbox of an actor is a queue that buffers messages before they are processed one by one. An actor can be *spawned* by instantiating a behaviour and creating an inbox. Sending a message to an actor is equivalent to storing the message in the inbox of the receiving actor.

We use actors to abstract over the "external world" of a reactive program. Since they run in their own thread they may perform long lasting computations (see Section 2.1), and since they encapsulate their own state they may perform effectful statements (see Section 2.2). To integrate with reactors they can publish new signals ex nihilo, and conversely they can integrate with existing signals. In other words they may "put values into" source nodes and act as sink nodes of a reactive program (see Section 2.3).

#### 3.2.1 Actors as Sources

In the context of our running example an actor can be used to fetch real-time data from Twitter and publish this info as new signal. This involves opening a socket connection to the Twitter API and (synchronously) reading data from the socket. The behaviour of this actor, implemented in Listing 1, defines one local state variable and two messages. On line 3 we declare the local state `current-stream` to be able to store a reference to the current twitter stream. Line 4 defines the `open-stream` message with one parameter `keyword`. Processing such a message amounts to: (line 6) opening a socket connection to start receiving relevant tweets, and (line 7) sending a `read-tweet` message to itself ("~>" is the asynchronous send operator). The `read-tweet` message on line 9 reads one tweet from the socket (line 10) and publishes the data as a signal. The way in which data is published may seem strange at first. First we split the tweet into a list of individual characters (line 11), and then we emit them over a signal that quickly changes with each character (lines 12-13) followed by a newline at the end of the tweet (line 14). Finally we send a recursive message to read the next tweet (line 15). The key aspect of this actor is the `SPIT` primitive that is used to emit values over a signal owned by the current actor. The first argument to `SPIT` is a label that identifies the

```
1  (define TwitterBehaviour
2   (ACTOR
3    (LOCAL_STATE current-stream)
4    (MESSAGE (open-stream keyword)
5     (set! current-stream
6      (open-twitter-connection keyword))
7     (~> SELF read-tweet))
8
9    (MESSAGE (read-tweet)
10    (define tweet-text (read-tweet current-stream))
11    (define chars (string-split tweet-text ""))
12    (for-each (lambda (char) (SPIT 'tweet char))
13              chars)
14    (SPIT 'tweet "\n")
15    (~> SELF read-tweet))))
```

**Listing 1.** Creating a signal from an actor

```
1  (define ClientBehaviour
2   (ACTOR
3    (IN change-colour)
4  ;; GUI code has been omitted ;;
5    (MESSAGE (change-colour rgb)
6     (~> SELF set-background rgb)))
```

**Listing 2.** Parameterizing actors with signals

signal (multiple signals can be published), and the second argument is the value that is emitted. Modifying the signal consecutively for each character in a tweet will help us in Section 3.3 to build a reactor that is strongly reactive. For now we do not show how to link this signal to another actor or reactor (we leave this for Section 3.4). To conclude, we now have an actor that, when receiving an `open-stream` message, publishes a new signal of real-time tweets.

#### 3.2.2 Actors as Sinks

Besides emitting, actors are also necessary to process the data produced by reactive programs (as sinks), since this computation must necessarily be separated from the reactive program itself (due to the reasons given in Section 2). For this they need to be able to integrate with signals. Assume a signal of RGB values exists in the application (we build this in the following section). In Listing 2 we implement the behaviour of a client actor that displays the value of an input signal as the background of a GUI. Via the `IN` primitive (line 3) the behaviour declaratively specifies that the actor acts as the sink for one signal that maps to `change-colour`. Furthermore, the behaviour defines a `change-colour` message (line 5) that modifies the GUI. The name similarity is no coincidence. When a signal to which an actor is "listening" updates its current value, this update is enqueued as a regular actor message. The name of the message is determined by the `IN` primitive. In Section 3.4 we will show how a concrete signal is wired to the actor, such that whenever the current colour changes, the actor receives a new `change-colour` message in its inbox and *acts* on this by modifying the user interface.

### 3.3 Reactors

Reactors are used to represent a functional reactive program, and are defined by a behaviour and an inbox. The behaviour

[2] There may be some confusion due to overlapping terminology between actor-based programming and reactive programming. In this paper we use the term *behaviour* to denote constructs similar to a behaviour in actor-based programming, which applies to both actors and reactors. We exclusively use the term *signal* to denote the type of time-varying value that is called a behaviour in reactive programming languages.

```
1 (define CheerLightsBehaviour
2  (REACTOR
3   (IN tweet-char)
4
5   (TICK colour (foldp word-append "" tweet-char
        is-colour?))
6   (TICK colour-rgb (colour->rgb colour))
7
8   (OUT colour-rgb)))
```

**Listing 3.** Implementation of a reactor

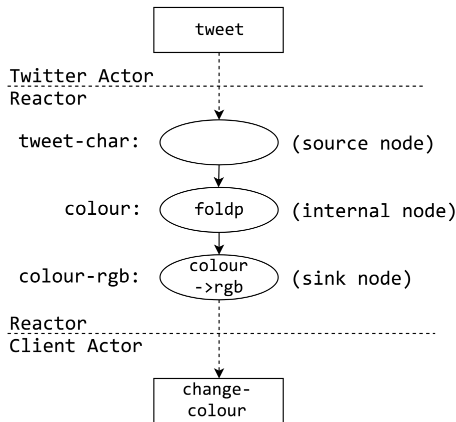


**Figure 2.** Internal DAG of the CheerLights reactor

defines a set of abstract inputs (sources) and a DAG that translates the inputs to outputs. Spawning a reactor instantiates the behaviour and creates an inbox (a queue). A reactor runs in new process that, when given a set of concrete signals, uses these signals as the sources of the reactive program. Every update to a source is buffered in the inbox until it percolates through the dependency graph one by one. We can use a reactor to implement the CheerLights service, which is an inherently reactive process that transforms a signal of tweets into a signal of RGB values.

### 3.3.1 Basic Structure

Listing 3 defines the behaviour of a reactor, whose internal DAG is represented in Figure 2. The basic structure is as follows:

1. The `IN` primitive declares the set of abstract source nodes of the internal DAG. In this case there is one input named `tweet-char` (line 3) which corresponds to the source node of Figure 2. In Section 3.4 we show how the source node is wired to the tweets published by the Twitter actor.
2. The `TICK` primitive is used to name and create new signals that represent intermediary computations, i.e. to create the internal node depicted in Figure 2. In our example we define a signal that aggregates characters into words (line 5) and converts the word to an RGB value (line 6). The `colour-rgb` node is not considered an internal node because no other signals depend on it.
3. The `OUT` primitive specifies the set of signals that are published by the reactor, for instance the `colour-rgb` signal (line 8), which corresponds to the sink node of Figure 2. In Section 3.4 we show how this sink node is wired to the client actor (containing a GUI).

The internal DAG is constructed by implicitly lifting functions. For example, `colour->rgb` on line 6 (that works on strings) is automatically lifted to work on a signal of strings. The propagation of values through the DAG uses a technique similar to Elm [3] where the Global Event Dispatcher roughly corresponds to the runtime and inbox of a reactor, and the source nodes of an Elm program are the source nodes of the reactor. By using this propagation algorithm we ensure glitch freedom within a reactor.

### 3.3.2 Strong Reactivity

The Twitter actor of Listing 1 publishes tweets as a signal of characters. By requiring a signal of characters as the source we can guarantee that the reactor is strongly reactive. The idea is that we then aggregate characters into words, and test those against a list of known colours. An alternative approach would be to publish the complete tweet text from the Twitter actor and to extract a colour in our reactor. However, this involves splitting the input string into words, thus making the computation of the reactor completely dependent on the size of the input. A very large input would thus cause the reactor to temporarily "stop reacting" due to factors outside of its control (whatever data it is given). In other words, if our reactor had used multiple sources, it would temporarily completely ignore the values of the other sources. By building our reactor as strongly reactive we avoid this problem entirely. To achieve this we define a `foldp` primitive with the following type signature:
$foldp :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow Signal\ a \rightarrow (b \rightarrow Bool) \rightarrow Signal\ b$
Our definition of `foldp` is similar to that of other reactive languages such as Elm [3]. The first arguments are an aggregation function, an initial accumulator, and a signal that supplies the values that are aggregated. Unlike a classical `foldp` we add a predicate to test the value of the accumulator after every iteration. The difference with a classical `foldp` is that the resulting signal is updated *only* when the predicate evaluates to `true`. The accumulator resets after every update.

In our reactor (line 5) we use `word-append` to aggregate characters into words (it returns an empty string when the character is a space or newline). Thus the `colour` signal will update only when the aggregated word succeeds the `is-colour?` test. The result is a reactor of which the computational complexity is not dependent on the size of the input, and instead it only depends on the efficiency of the implementations of internal operations such as `is-colour?`.

## 3.4 Composing Actors and Reactors

The final thing that remains is spawning and composing the actors and reactors of our application, i.e. linking the sources

```
1 (define twitter (SPAWN-ACTOR TwitterBehaviour))
2 (define tweet   (SIGNAL twitter 'tweet))
3 (~> twitter open-stream "#CheerLights")
4
5 (define cheerlights
       (SPAWN-REACTOR CheerLightsBehaviour tweet))
6 (define colour (SIGNAL cheerlights 'colour-rgb))
7
8 (define client (SPAWN-ACTOR ClientBehaviour))
9 (FOLLOW client colour)
```

**Listing 4.** Wiring the application together

and sinks of actors and reactors together. Here we declare that the Twitter actor is responsible for populating the source node of the CheerLights reactor, and that the data produced by the reactor should be processed by the client actor for displaying (which is an effectful operation). The composition of these components is shown in Listing 4. First the Twitter actor is spawned (line 1), and a reference to its published signal is constructed (line 2). Once the actor is initialised the signal is populated with data (line 3). Note that at this point no data has left the actor. When spawning the reactor of Listing 3 it is necessary to provide the signals that produce data for its source nodes. Therefore, it is spawned (line 5) with the `tweet` signal as input. Data now starts flowing from the actor through the reactor, resulting in the `colour` signal (line 6). Finally the client of Listing 2 is spawned on line 8 to act as a sink of the application. It is coupled to the reactor via the `FOLLOW` operator that links the output of the reactor to the input of the actor. Notice that dependencies for actors are dynamic and may thus change at runtime.

### 3.5 A Philosophical Perspective

Our model can be regarded as a union of "active" and "reactive" concepts. One can wonder whether it is possible to unify actors and reactors in one single language concept. We argue that this is probably not the case:

**Actors** are used to express code "that does something" *regardless* of what is happening in the outside world. As soon as an actor receives and processes a message it can perform computations that change the actor's state, go into infinite loops, and emit side effects, completely ignoring what is happening elsewhere. The process that decides "what happens next" is the instruction queue of the running method.

**Reactors** are used to express code "that does something" *as soon as* something happens in the outside world. Whenever it receives a value it will process that value instantaneously so that it is immediately ready to process the next incoming value (i.e. in $O(1)$). The process that decides "what happens next" is the inbox that queues all input of the source nodes.

We believe that there is no single concept that can reconcile both ideas at the same time.

### 3.6 The Actor-Reactor Model's Footprint

Existing reactive languages have faced the issue of integrating effectful statements in their language, and in some aspects their solutions already partially adopted certain aspects of the Actor-Reactor model.

### 3.7 FrTime

FrTime is a functional reactive language built in Racket [2]. One of the goals of the language is making maximal use of the programming environment. It achieves this by exposing two threads of control to the programmer. The first is the thread responsible for running the reactive program, and the second is a thread that runs a REPL that interfaces with the reactive program. The REPL can be used to modify some special signals (*cells*) which trigger a re-evaluation of the reactive program, and conversely they can be used to read the values of signals and continuously display their updated value. The interaction between the REPL thread and the reactive thread is similar to an actor and a reactor: The REPL is part of the external world and communicates with the reactive thread via asynchronous message passing. The reactive thread buffers messages from the external world and propagates them through the dependency graph one by one.

### 3.8 Elm

Elm is a functional reactive language for building client-side web applications [3]. Some of its primitives (e.g. `SyncGet` for web requests) block the reactive program until they complete. One of the important features of Elm is support for *asynchronous* reactive programming. It achieves this via an `async` primitive that ensures that long running computations need not block the reactive program. This primitive is effectively used to spawn invisible threads in the background to perform an *action* asynchronously. When the action is run to completion, the reactive program reacts to its return value. Furthermore there are other invisible threads such as the DOM whose thread is managed by the browser. Here the invisible threads correspond to actors since they perform effectful long lasting *actions* (fetching an image, updating the DOM, ...) and the main program corresponds to a reactor.

### 3.9 REScala

REScala is a reactive programming language based on Scala that unifies the concepts of functional reactive programming with object-oriented programming [7]. In other words it embeds a DAG within an object-oriented program. The source signals of the DAG are created as `Var` fields in objects that can be changed via an assignment operator. Assigning a new value to a signal is a synchronous operation that dictates a traversal of the DAG to propagate the changed value. Developers can make use of multiple Scala threads to change sources, but the assigning thread requires a lock on the DAG before the assignment can be executed. Other threads in

the program are thus blocked from making assignments to source signals as long as a propagation cycle is still ongoing. In our terminology the "external world" of the reactive program (i.e. the "world of objects") is similar to an actor, since it supplies the sources with values. Using multiple threads then corresponds to using multiple actors.

## 4 Limitations and Future Work

One of the next steps of our research is to formalise our model, e.g. via operational semantics. Via this formalisation we will investigate whether we can enforce the property of strong reactivity in reactors.

Since actors and reactors are isolated processes that run at their own pace, we have to tackle the *rate mismatch problem*, more often known as *backpressure*. Backpressure is an issue because actors and reactors may produce data at a much higher rate than a receiver can process. Eventually the inbox of the receiver will be completely full of old values, resulting in a lot of time spent on older (possible irrelevant) computations. Currently we assume that the inboxes of actors and reactors are unbounded, which is an assumption often made in actor languages. However, since reactivity and responsiveness are an important aspect of our system (c.f. strong reactivity), we want to offer bounded mailboxes that give the developer some real-time guarantees, accompanied with a backpressure algorithm that is configurable to suit the application's requirements.

Another limitation is that we currently do not support distribution. While we consciously chose abstractions that lend themselves to distribution, there are many issues such as partial failures and service discovery that remain unsolved in reactive programming. We can draw inspiration from AmbientTalk/R [1] that implements a mechanism for service discovery in distributed reactive programs, but does not tackle the issue of partial failures.

## 5 Conclusion

In this paper we motivate that reactive programming alone is not enough to write real-world applications. In Section 2 we define *the awkward squad for reactive programming*, a set of problems that "do not neatly fit into the reactive paradigm". We argue that the paradigm is incompatible with performing long lasting computations and effectful statements, and that it is difficult to embed reactive programs in a (distributed) multi-threaded external world. The key insight is that we believe there is no single abstraction to solve all problems, and instead we should cleanly separate the reactive part of an application from the semantically non-reactive parts.

In Section 3 we propose a model of actors and reactors where actors represent the non-reactive parts of an application, and reactors represent a typical reactive program. We solve the problem of long lasting computations (Section 2.1) by running actors and reactors in a separate thread, and clearly defining the semantics of their interaction. The issue of effectful statements (Section 2.2) is tackled by evacuating all effectful code from a reactive program into one or more actors. Finally we solve the issue of embedding a DAG in the external world (Section 2.3) by clearly defining what the external world entails: Actors that may emit values which can be linked to the source signals of a reactor, and vice-versa actors that can act as sinks for signals of a reactor.

## Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work is partially funded by Flanders Innovation & Entrepreneurship (VLAIO) with the FLAMENCO project under grant No.: 150044, and the Brussels Institute for Research and Innovation (Innoviris) with the Doctiris programme under grant No. 15-doct-07.

## References

[1] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. 2010. Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Objects, Models, Components, Patterns, TOOLS'10, Málaga, Spain (Lecture Notes in Computer Science)*, Vol. 6141. Springer, 41–60.

[2] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'06, Vienna, Austria (Lecture Notes in Computer Science)*, Vol. 3924. Springer, 294–308.

[3] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Programming Language Design and Implementation, PLDI'13, Seattle, WA, USA*. ACM, 411–422.

[4] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands*. ACM, 31–40.

[5] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *bject-Oriented Programming, Systems, Languages, and Applications, OOPSLA'09, Orlando, FL, USA*. ACM, 1–20.

[6] Simon Peyton Jones. 2001. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering Theories of Software Construction* 180 (Jan. 2001), 47–96. IOS Press.

[7] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: bridging between object-oriented and functional style in reactive applications. In *Modularity, MODULARITY'14, Lugano, Switzerland*. ACM, 25–36.

[8] Hans Scharler. 2017. CheerLights IoT. (August 2017). https://web.archive.org/web/20170811121942/http://cheerlights.com/ Referenced 11 aug 2017.