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

**Sam Van den Vonder**
Software Languages Lab, Vrije Universiteit Brussel, Belgium

**Thierry Renaux**
Software Languages Lab, Vrije Universiteit Brussel, Belgium

**Bjarno Oeyen**
Software Languages Lab, Vrije Universiteit Brussel, Belgium

**Joeri De Koster**
Software Languages Lab, Vrije Universiteit Brussel, Belgium

**Wolfgang De Meuter**
Software Languages Lab, Vrije Universiteit Brussel, Belgium

## Abstract

Reactive programming is a programming paradigm whereby programs are internally represented by a dependency graph, which is used to automatically (re)compute parts of a program whenever its input changes. In practice reactive programming can only be used for some parts of an application: a reactive program is usually embedded in an application that is still written in ordinary imperative languages such as JavaScript or Scala. In this paper we investigate this embedding and we distill "the awkward squad for reactive programming" as 3 concerns that are essential for real-world software development, but that do not fit within reactive programming. They are related to long lasting computations, side-effects, and the coordination between imperative and reactive code. To solve these issues we design a new programming model called the Actor-Reactor Model in which programs are split up in a number of actors and reactors. Actors and reactors enforce a strict separation of imperative and reactive code, and they can be composed via a number of composition operators that make use of data streams. We demonstrate the model via our own implementation in a language called Stella.

**2012 ACM Subject Classification** Software and its engineering → Data flow languages; Software and its engineering → Multiparadigm languages

**Keywords and phrases** functional reactive programming, reactive programming, reactive streams, actors, reactors

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2020.19

**Supplementary Material** ECOOP 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.2.7` and `https://doi.org/10.5281/zenodo.3862954`.

**Funding** *Sam Van den Vonder*: Research Foundation – Flanders (FWO) grant No. 1S95318N
*Thierry Renaux*: Flanders Innovation & Entrepreneurship (VLAIO) "Cybersecurity Initiative Flanders" program
*Bjarno Oeyen*: Research Foundation – Flanders (FWO) grant No. 1S93820N

## 1 Introduction

Reactive programming is a programming paradigm that revolves around automatically changing the state of a program based on perpetually incoming values. Historically, it was conceived as a solution to the problems of *inversion of control* and *callback hell* which occur when programming event-driven programs [3]. Reactive programming languages such as FrTime [12], Flapjax [34], Elm [15] and REScala [46] propose increasingly rich abstraction





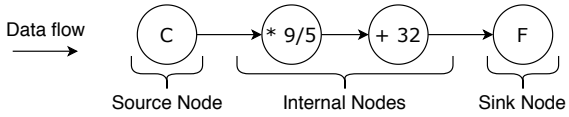



**Figure 1** Compiled structure of a reactive program to a DAG. Data flow is usually depicted from top to bottom, but depicted here from left to right.

mechanisms to represent and compose so-called "time-varying values" or *signals*, which are values that can change over time. The "Hello World!" of reactive programming converts measurements of a Celsius thermometer to Fahrenheit. Given a signal `C` that represents a changing temperature in Celsius, the expression `F = (C * 9/5) + 32` declares a new signal `F` that represents the temperature in Fahrenheit. Changes to the value of `C` automatically give rise to a recomputation of `F`. This is typically realised by compiling the reactive program into a directed acyclic graph (DAG), as exemplified in Figure 1.

There are many incarnations of reactive languages and libraries built with various mainstream languages. In academia, reactive languages include FrTime [12] (Racket), Scala.React [33] (Scala), REScala [46] (Scala) and Flapjax [34] (JavaScript). In industry, large companies such as Facebook develop and maintain frameworks such as ReactJS [28] and React Native [21] for reactive Graphical User Interfaces (GUIs), ReactiveX is a specification for reactive streams implemented in over 18 languages [40] some of which are developed or maintained by companies such as Microsoft and Netflix, and an implementation of the "Reactive Streams" specification has been included in Java since version 9 [29, 13]. Results from an empirical study on program comprehension suggest favourable results for reactive programming when compared to the Observer pattern in object-oriented programming [47].

In the rest of the paper we will abstract over the details of particular reactive programming languages and talk about them in terms of their role in the canonical DAG model for reactive programming. *Source nodes* correspond to the "input" signals of the reactive program. Their values are typically provided by code that is external to the reactive program. *Internal nodes* are composed signals that constitute the reactive program. *Sink nodes* correspond to the "output" signals of the reactive program that constitute the program output.

Reactive programming languages and frameworks focus on the design and concepts of the *internal part* of a reactive program. Explained in terms of the DAG, they focus on language features and abstractions whereby programmers can express DAGs as easily and as declaratively as possible. Reactive programs also have 2 other parts: an *input part* provides input to the reactive program by modifying its source nodes, and another (possibly different) *output part* processes the output of the reactive program. For example, in the temperature converter of Figure 1, the source `C` may be connected to an input field in the GUI, and the result `F` may be displayed in the same GUI. In another application, `C` may be connected to a distributed web-based data stream whereby the input of the reactive program is produced by an entirely different machine (e.g. a weather station).

One cannot help but observe that reactive programming is only used to implement the internal part of a reactive program, and that it is usually embedded in an application that is still written in ordinary imperative languages such as JavaScript or Scala. The main problem tackled in this paper is that the parts of existing reactive programming languages and frameworks that are responsible for input and output are ill-defined: They use ad-hoc mechanisms that can violate the invariants of reactive programs. This is especially problematic for long lasting computations that block the reactive program [7], and computations with

**Listing 1** REScala reactive program that matches a regular expression with an input string.

```
1 val userInputSignal = Var("") // initial signal value is ""
2 val matches = Signal { "(A+)*B".r.findFirstMatchIn(userInputSignal()) }
```

side-effects (e.g. modifying or rendering a GUI) [19, 33]. Mechanisms to embed reactive code between the imperative input and output parts of applications are poorly investigated in literature. To this end, we have 2 main contributions.

1. In analogy with the "awkward squad" for functional programming which are a set of application concerns that are essential for real-world software development, but that do not fit within the purely functional programming paradigm [37], in Section 2 we identify the *"awkward squad for reactive programming"*. They are (1) long lasting computations, (2) embedding imperative code in reactive code, and (3) embedding reactive code in imperative code.
2. Just like functional programming solves the problem by evacuating the awkward squad to a different location (i.e. monadic) of the program, we propose a programming model that solves these issues. First, in Section 3 we define a class-based object-oriented data model that will guarantee that reactive programs cannot execute imperative code and long lasting computations. Second, in Section 4 we introduce the *Actor-Reactor Model* whereby the imperative input & output parts of reactive programs must be modelled as *actors*, and the internal parts of reactive programs are modelled as *reactors*. Together, actors and reactors enforce that imperative and reactive code remains separated, but can still co-exist within the same application. To clearly demonstrate the concepts of this model we have designed and implemented an experimental language called Stella.

## 2 Identifying the Awkward Squad for Reactive Programming

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

### 2.1 Long Lasting Computations

Responsiveness is 1 of the 4 key properties of reactive systems outlined in the so-called Reactive Manifesto [7], [31, Chapter 1]: "*Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service.*" [7] However, reactive languages and frameworks often impose little to no restrictions on the types of expressions that can be used within a reactive program. It is often easy to write code that accidentally makes the program no longer reactive. For example, consider the reactive program in Listing 1 (written in REScala) that checks whether user-provided input strings match a regular expression. Line 1 defines a new source node called `userInputSignal`, and Line 2 defines an internal node called `matches` that is derived from the source node. Whenever the value of `userInputSignal` changes, the value of `matches` reflects whether the new string matches the regular expression `(A+)*B` (e.g. `AB` and `AAAB`, but not `AAA`). This program has a worst-case complexity of $\mathcal{O}(2^n)$ with `n` the size of the input string. Matching the string `AAAAA` fails after 112 steps, and matching 50 `A`'s fails only after ~3 quadrillion steps [44]. This program clearly cannot be called reactive in the aforementioned sense. While this specific example may be a contrived case of catastrophic backtracking in

regular expressions, a developer can easily and accidentally introduce computations into reactive programs that (occasionally) have unintended consequences for their execution time. We call this the **Reactive Thread Hijacking Problem**, because long lasting computations can "hijack" the thread of execution of a reactive program, thereby stopping the reactive program from being able to react to new input.

> The language design problem that needs to be solved is how to ensure that a "reliable upper bound" can be imposed on long lasting computations within reactive programs.

The Reactive Manifesto is intentionally vague about how to achieve this. We identified 3 levels of reactivity that provide different program termination guarantees.

### 2.1.1 Weak Reactivity

The set of programs that we call *weakly reactive* are those for which there are no guarantees with respect to how long they will take to execute. This is usually because reactive programs can be programmed with the full power of a Turing-complete language. Programs written in reactive languages such as FrTime [12], Flapjax [34], Elm [15], REScala [46] and AmbientTalk/R [9], or frameworks such as ReactJS [28], ReactiveX [40] and Akka Streams [45, Chapter 13] all fall into this category.

### 2.1.2 Eventual Reactivity

The set of programs that we call *eventually reactive* are those for which it can be proven that the program will eventually terminate. There are both static and dynamic approaches to enforce program termination, each with varying levels of restrictions that they impose on the underlying program. Static enforcement includes work such as total functional programming [51], primitive recursion [8, Chapter 3], and model checkers such as TERMINATOR [11]. Techniques that dynamically enforce program termination can make use of both static information and run-time values, such as size-change termination [35].

### 2.1.3 Strong Reactivity

The set of programs that we call *strongly reactive* are those for which the execution time of the reactive program does not depend on the size of its input. In other words, the asymptotic worst-case complexity of the reactive program is guaranteed to be in $\mathcal{O}(1)$. Reactive programs that are strongly reactive will never be unexpectedly slow because of certain types of input. We consider this to be the strongest form of reactivity, but the trade-off is that the types of programs that can be written is severely restricted.

Synchronous programming languages such as Esterel [5], Lustre [25], Céu [48] and Signal [23] apply reactive programming to real-time systems [4]. They rely on the assumption of the *synchronous hypothesis*, where the output of the program is conceptually synchronous with its input and "instantaneous" [17]. In contrast with strong reactivity this is not a constraint on program complexity or execution time, but on the correctness of event processing whereby 1 event must be completely processed by the reactive program before the next event is considered. Constant-time (non-reactive) programming languages such as FaCT [10] and Jasmin [2] are concerned with bounding execution times of certain instruction to prevent leaking secret information (encryption keys) based on execution time. Constant-time programming is not a restriction of program complexity, but a restriction on the execution time of specific instructions, which may not vary in function of sensitive run-time

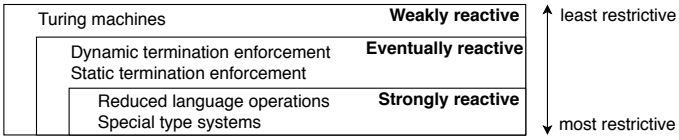



**Figure 2** Termination guarantees for different levels of reactivity.

**Listing 2** REScala reactive program with side-effects.

```
1 val counter = Var(0) // initial source value is 0
2 Signal { print("A" + counter() + " ") }
3 Signal { print("B" + counter() + " ") }
```

information. ActiveSheets [54] is a reactive language where programs consist of Microsoft Excel spreadsheets. While many spreadsheet operations are strongly reactive (e.g. arithmetic), there are exceptions such as `SEARCH` and `REPLACE` for searching and replacing substrings. Finally, the RT-FRP language [56] is a statically typed reactive programming language where the time and space (in memory) cost for each execution step is statically bound. To the best of our knowledge this language is strongly reactive.

### 2.1.4 Summary

Expressions in existing reactive languages and frameworks can accidentally or unintentionally cause long lasting computations that block the reactive program, thus turning the program no longer reactive. We identified 3 different levels of reactivity that are summarised in Figure 2, together with the possible techniques on how to enforce them. Stricter enforcement of program termination will result in reactive programs that have stronger guarantees with respect to the processing of their input, i.e. a more reliable upper bound can be placed on their execution time. However, the trade-off is that the types of programs that can be written is also reduced with stricter enforcement. It is up to developers to choose a reactive language or framework that can enforce a particular level of reactivity that is appropriate for the application at hand, i.e. eventually reactive or strongly reactive.

## 2.2 Embedding Imperative Code in Reactive Code

Effectful computations are extremely tricky to understand and debug when they are embedded within the nodes of a dependency graph [19, 33]. Consider the reactive program (written in REScala) in Listing 2. Line 1 defines a new source node called `counter`, and Lines 2 and 3 define two internal nodes that print either `A` or `B` to the console, followed by the value of the counter. When this program is executed, the initial value of `counter` is propagated through the program, and `"A0 B0"` is printed to the console in the order of evaluation (from top to bottom). However, when the value of `counter` is updated to 1, approximately 50% of the time the program output is reversed and `"B1 A1"` is printed. We call this problem the **Reactive Update Order Leak**, because effectful computations leak information about the update order of subexpressions within reactive programs, and their correct execution may even rely on a specific order.

The update order of the DAG is usually not part of the semantics of a reactive program. Reactive programming languages such as FrTime [12], Flapjax [34] and REScala [46] prevent *glitches* (temporary inconsistencies in the program [12]) by specifying that updates should be executed in a topological order of the dependency graph. Some implementations parallelise

the execution of certain regions of the DAG, such as the conceptual propagation model of Elm [15] and a parallel version of the REScala update algorithm [18]. Streaming frameworks such as ReactiveX [40] and Akka Streams [45, Chapter 13] do not feature such an algorithm, and instead only specify that parent nodes should be updated before their child nodes.

All of the aforementioned technologies allow multiple valid update orders to be used for a given program. This is good for language implementers, because it gives them a lot of freedom to tweak and optimise (e.g. parallelise) how values propagate through the reactive program. However, for application developers this means that the concrete update order can vary across different implementations or versions of the same language or framework. The order can even change at run-time, which is the case in our experiments with the code snippet of Listing 2, where the execution of the program yields nondeterministic results.

The root of the problem are unconstrained side-effects in interactive applications. Not only can side-effects cause bugs that are difficult to find and reproduce because of an unlucky ordering in some propagations through the DAG, but they are also very difficult to coordinate and have a detrimental effect on behavioural composition [20]. Recognising these issues, most reactive programming languages and frameworks already forbid side-effects within the DAG of a reactive program, either through language design or via programmer guidelines (e.g. REScala guidelines [41, 1.5.3]). However, as we will discuss in Section 2.3, they are rarely successful in banishing side-effects completely.

> 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 dependency graph without accidentally or nondeterministically affecting the behaviour of effectful computations that reside in other nodes of the dependency graph.

## 2.3 Embedding Reactive Code in Imperative Code

Programs written in existing reactive programming languages and frameworks are not subject to the Reactive Update Order Leak (Section 2.2) when their code is *purely* functional. However, we observe that this requirement is not met in practice, because parts of real-world programs often depend on long lasting computations and effectful computations. For instance, input can come from a GUI or be streamed in from another computer over a network connection, and output may be used to modify a GUI or to push a notification to a user. A complete solution must therefore be able to bridge what we call the **Reactive/Imperative Impedance Mismatch**, in analogy with the Object-Relational Impedance Mismatch for object-oriented programming.

Because the embedding of reactive code within imperative code is unavoidable in real-world reactive applications, existing reactive languages and frameworks all tackle the Reactive/Imperative Impedance Mismatch to some degree. We argue that their solutions either have limited applicability, or are ad hoc solutions with unclear semantics. What follows is a non-exhaustive list of mechanisms that we could identify in related work.

**Domain specific features** are special language features tailored to a specific domain, usually involving GUIs. For instance, Flapjax [34], Elm [15], and ReactJS [28] provide a DSL for building GUIs whereby source nodes are automatically created and updated by GUI components, and the GUI automatically integrates with sink nodes. Such languages typically also feature built-in domain specific signals with narrow applicability, such as Elm's `Mouse.position` signal [15].

**Special forms** are built-in language constructs with special evaluation rules. Reactive languages often offer special forms to provide operators that cannot be implemented from within the language itself. For example, Elm [15] offers a built-in `syncGet` operation to execute synchronous HTTP requests. To prevent this computation from blocking the reactive program, Elm also offers an `async` special form to execute operations in the background.

**Metaprogramming** involves mechanisms to manually construct or modify parts of the reactive program. For example, FrTime and REScala offer built-in primitives to create and (destructively) modify source nodes of the reactive program. The semantics of manual assignments to source nodes are often unclear, especially when multiple threads of execution are involved. Streaming frameworks such as ReactiveX [40] and Akka Streams [45, Chapter 13] offer built-in operators to define new streams ex nihilo, reducing their dependence on special forms. Still, if there is no built-in support for a specific data source, a programmer has to open up the implementation of operators to carefully craft a new one.

**Hidden concurrency** involves mechanisms whereby multiple threads of execution are used, but where the concurrent nature of the operations is hidden from the programmer. For example, in FrTime a dedicated thread manages the execution of the reactive program behind the scenes, while the Racket Read-Eval-Print Loop continues running on the main thread. From the REPL, a Scheme program can "send" input values to the reactive thread. Conversely, the output of FrTime programs can be reactively displayed in the output of the REPL. However, multi-threading and concurrency-control are not part of the FrTime programming model.

**Periodic polling** enables embedding by periodically updating the source values of a reactive program from some computational process that is external to the reactive program. The quintessential example of this approach is the `seconds` behaviour as found in FrTime [12]. The `seconds` behaviour is a free variable in the reactive program that is automagically updated to the current Unix time. Typically, dependents of the `seconds` behaviour use the timestamp only to determine whether they should imperatively poll an application-specific data source. Such code lets the reactive runtime schedule imperative code, giving rise to Reactive Update Order Leaks.

> The language design problem that needs to be solved is how to design a reactive programming language that supports the features of the awkward squad, but without introducing the Reactive Thread Hijacking Problem and the Reactive Update Order Leak.

In other words, it is important that imperative code and reactive code can coexist within the same application in such a way that the imperative code cannot accidentally violate the invariants of the reactive code. The mechanisms employed by existing languages and frameworks are often highly specialised, have unclear semantics with regards to their interactions with the reactive program, and they do not solve these problems.

## 2.4 Solution: General Idea

The idea is to embrace that reactive applications always consist of both imperative and reactive parts. Both are desired, and they complement each other [19]. We give each their own thread and effect-set, and design simple composition operators to link them together.

**Listing 3** A "Hello World!" program in Stella.

```
(actor Main
  (def-constructor (start)
    (println "Hello World!")))
```

The result is called the Actor-Reactor Model, which can be summarised as follows.

- *Actors* are used to represent the imperative input and output parts of reactive programs. They imperatively manage their own state and input-output, and are solely responsible for executing long lasting computations and effectful computations.
- *Reactors* are used to encapsulate reactive programs, each with their own DAG and update thread. By construction it is impossible for reactors to perform long lasting computations and side-effects.
- Coordination of actors and reactors happens via the *data streams* which they consume and produce. We define a set of composition operators whereby the streams defined by actors can be linked to the source nodes of reactors and vice-versa how actors can act on changes to the sink nodes of reactors.

We have implemented the Actor-Reactor Model in an experimental language called Stella, which can be roughly divided into two levels. First, an object-oriented base language is used to restrict reactors from producing side-effects and performing long lasting computations (Section 3). Second, actors and reactors are used to strictly separate the imperative and reactive parts of programs (Section 4). We have written a prototypical Continuation-Passing Style interpreter for Stella in TypeScript (in the style of [22, Chapter 5]).

## 3 Base Language: OOP with Effect and Termination Guarantees

In this section we will focus on the object-oriented base language of Stella. Throughout all code examples we will consistently syntax highlight special forms (expressions with special evaluation rules) in blue. Strings are surrounded by double quotes and are highlighted in green. Symbols start with a single quote and are highlighted in red.

Stella is a dynamically typed language where all run-time values are objects. Its native objects are booleans, numbers, strings, symbols and the value `#undefined`, which are instances of the classes `Boolean`, `Number`, `String`, `Symbol` and `Undefined` respectively. A program in Stella consists of 3 sets of top-level definitions: a set of classes, a set of *actor behaviours* ("the class of an actor"), and a set of *reactor behaviours* ("the class of a reactor").

### 3.1 Basic Expressions and "Hello World!"

Each Stella program must contain an actor behaviour called `Main`. This actor behaviour must have a constructor named `start`. To start a Stella program, the Stella runtime spawns an instance of the `Main` actor behaviour and invokes the `start` constructor. Consider the *"Hello World!"* program in Listing 3 that defines such a behaviour. The body of its `start` constructor contains a single `println` statement in operator prefix notation (Polish Notation). This can be seen as synchronously sending a `println` message with no arguments to the `"Hello World!"` object of class `String`. Similarly, `(+ 1 2)` can be seen as sending a `+` message to the object representation of the number 1 with one argument which is the object corresponding to number 2. In the rest of this paper we will use the terminology of "calling" or "invoking" a method rather than sending synchronous messages (cf. SmallTalk [24]).

**Listing 4** Examples of basic expressions in Stella

```
1 (def hello "Hey")
2 (set! hello "... from the other side")
3 (if (equal? hello "Hey") (println "yes") (println "no")) // console prints "no"
```

Expressions that may be used in the body of constructors and methods follow S-expression syntax [1, Chapter 1] and are shown in Listing 4. Local variables are introduced via the `def` special form (similar to `define` in Scheme), assignments use the `set!` special form, conditionals use the `if` and `cond` special forms. Two methods to test for equality are implemented by the superclass of all classes (`Object`): `equal?` tests for object equality, `eq?` for object reference equality. All values are `#true` except for `#false` and `#undefined`.

## 3.2 Abstract Data Types

Code in Stella can be either imperative or reactive. Both kinds of code can operate on the same data, but the allowed sets of operations differ. Whereas imperative code may use the full power of a Turing-complete language, reactive code is restricted to operations that are guaranteed to terminate and that are free from side-effects. To this end, abstract data types in Stella are represented by classes which may define local fields, constructors, methods, as well as *routines*. Routines are special kinds of methods whose expressive power is a strict subset of that of methods. Routines have the following properties.

1. Routines have no side-effects.
2. Routines always terminate.
3. Routines can only invoke other routines.

To explain how we can enforce those properties, consider the `Pair` class defined in Listing 5 which can be used to represent linked lists. Local fields of the class are declared on Line 2. Line 4 defines a constructor called `initialize-with` with 2 formal parameters called `initial-car` and `initial-cdr` that will initialize the fields of a new pair. Lines 8–9 define two routines called `first` and `second` with no arguments which are "getters" for the `car` and `cdr` field, and correspondingly, Lines 10 and 11 define two methods `set-first!` and `set-second!` which are setters for these fields. Line 13 defines a routine called `length` that will compute the length of a linked list by calling `length` on the `cdr` field as long as it is also a pair[1]. We can enforce the properties of routines as follows.

Routines that contain expressions with side-effects are rejected by the interpreter. In our case they are `set!` and a couple of other special forms[2]. Together with property #3, which is upheld using a run-time check, this ensures that routines will never have side-effects. This must be checked at run-time because Stella is a dynamic language. A run-time error occurs when a routine calls a method, for example `println` which performs IO.

Our current implementation of Stella uses *size-change termination* (SCT) for higher-order programs [35] to ensure at run-time that routines terminate. In a nutshell, this form of SCT dynamically constructs a *size-change graph* based on the argument values of a routine call

[1] Note that the `type-of` invocation in Listing 5 Line 15 returns a symbol that represents the name of the class. This is because classes are not reified as objects in our language (such as in SmallTalk [24, Chapter 5]). They cannot be referenced directly *except* via the `new` special form to create an instance.

[2] The special forms forbidden in reactors and routines are `set!`, `spawn-actor`, `spawn-reactor`, `send`, `emit`, `monitor`, and `react-to`.

**Listing 5** The Pair class which has 2 kinds of operations, methods and routines.

```
(class Pair
  (def-fields car cdr)

  (def-constructor (initialize-with initial-car initial-cdr)
    (set! car initial-car)
    (set! cdr initial-cdr))

  (def-routine (first) car)
  (def-routine (second) cdr)
  (def-method (set-first! new-car) (set! car new-car))
  (def-method (set-second! new-cdr) (set! cdr new-cdr))

  (def-routine (length)
    (cond ((eq? cdr #undefined) 1)
          ((eq? (type-of cdr) 'Pair) (+ 1 (length cdr)))
          (else 2))))
```

**Listing 6** Creating a circular data structure with `Pair` of Listing 5.

```
(def p1 (new Pair 'initialize-with 1 2))
(set-second! p1 p1)
(length p1) // successfully rejected via a run-time error
```

every time a routine is called. Before entering a new routine call, the *size-change termination principle* is used to compare the argument values to those of earlier calls to that routine (of the same class) higher on the call stack. If the new argument list is not decreasing in size, a run-time exception is thrown. Note that the algorithm assumes a well-founded partial order on values, for example, for numbers this is defined as $|x| < |y|$. In this case a recursive routine such as `factorial` is permitted as long as recursion stops when the value of its argument has decreased to 0.

As an example, consider the excerpt of Stella code in Listing 6 that uses the `Pair` class of Listing 5. Here, a `Pair` is made to point to itself, creating a circular linked list. A subsequent call to `length` on the `Pair` gives rise to a recursive call to `length` on the same class. SCT rejects this call, since the argument values have not decreased since previous invocations.

While [35] report that the overhead of their SCT analysis can be as high as a factor 100 (e.g. for merge-sort), it is negligible for applications that perform a lot of work in between recursive calls (they give a factorial function as an example). In any case, overhead remains constant with respect to the size of the input of the program. Crucially, it works for high-level programming languages such as Stella, and – unlike other approaches such as specialized type systems – SCT requires no programmer assistance. Stella is not coupled to any specific SCT algorithm, or even to SCT itself. What *is* part of the semantics of Stella, is that long-lasting computations are illegal, and that termination of routines must be enforced by the Stella runtime.

## 4 The Actor-Reactor Model

With the base language defined, we can now describe how Stella tackles the problem described in Section 2.3. We sketched the general idea in Section 2.4. The Actor-Reactor Model separates programs into *actors* and *reactors*. Actors handle the parts of a program that involve long-lasting computations or side-effects, whereas reactors handle the parts that are inherently reactive, or that are more easily expressed using reactive programming.

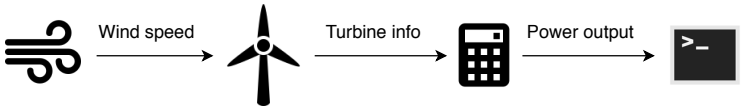



**Figure 3** Diagram of data flow in a wind turbine simulator consisting of a wind, a turbine, a turbine power calculator, and a console to print the result.

Coordination between actors and reactors is achieved using *data streams* that are produced by actors and reactors. Stella defines a number of composition operators to statically and dynamically manage how data can flow to and from actors and reactors. In the following sections we first introduce actors and data streams. In Section 4.3 we introduce reactors and how to compose them with actors.

## 4.1 Running Example: Wind Turbine Simulator

Consider a wind turbine simulator that calculates the real-time power output of a wind turbine. A simple simulator can be defined using the 4 components depicted in Figure 3. From left to right: a blowing wind, a wind turbine, a power calculator, and a console to print the output. Simulating wind can involve complex wind patterns and how they affect a turbine, or it can be a simple process that generates random values corresponding to the current wind speed. Even the simple version is challenging because it involves a process "a wind" that is continuously running and always changing. It also requires reactive processes: A blowing wind impacts the rotation of the wind turbine, which then impacts the power output. Important to note is that eliminating long lasting computations within reactive programs by itself is not enough to prevent them from blocking reactive programs. It is equally important that reactive processes cannot be accidentally blocked by long lasting computations in other parts of the program, for example the simulation of wind patterns.

We model wind as an actor because it is inherently an active process that is always running and changing independently. The console is also an actor because it performs IO. We opt to model a turbine and the power calculator as reactors: a turbine *reacts* to a wind, and a power calculator *reacts* to changes in the turbine.

## 4.2 Actors and Data Streams

Actors are typically defined in terms of a *behaviour* and a *mailbox* [30]. The behaviour of an actor describes its internal state and the messages that can be processed. Messages that are sent to an actor are inserted into its mailbox (e.g. a FIFO queue), and the actor continuously dequeues messages from its mailbox to process them one-by-one. Actors in Stella are based on the Active Objects model [57, 30, 16], where actor behaviours are defined similar to classes in object-oriented programming. In addition to receiving and processing messages, actors can be used to implement zero or more data streams to which they can *emit* (publish) values. The following sections explain the different parts of actor behaviours using the wind from our running example. We explain how actor behaviours are defined, how data streams can be implemented using actors, and how actors can monitor data streams.

### 4.2.1 Actor Behaviours

Listing 7 defines the `Wind` actor behaviour. An actor behaviour has a number of local fields, in this case 1 field called `rng` (Line 3). A constructor called `init` is defined on Line 5, which initializes the `rng` field with a new random number generator (an object). A method called

**Listing 7** The `Wind` actor behaviour to implement a stream that represents wind speed.

```
1  (actor Wind
2    (def-stream speed 1)
3    (def-fields rng)
4
5    (def-constructor (init) (set! rng (new Random)))
6
7    (def-method (blow)
8      (emit speed (integer-between rng 0 30))
9      (sleep 10000)
10     (send #self 'blow)))
```

`blow` without arguments is defined on Line 7, whose implementation we will explain later when we define data streams. A `Wind` actor is thus capable of processing `blow` messages that are inserted into its mailbox, which amounts to invoking the corresponding `blow` method.

The special form `spawn-actor` is used to spawn new actors, in this case to create new winds. The following expression spawns an instance of `Wind`, which could be used to represent the mistral wind. A reference is returned to the new actor, which is an object of type `ActorReference`. To initialize the actor, `spawn-actor` takes the name of a constructor (as a symbol) and any arguments it requires. Constructors are special kinds of methods that may only be invoked once, and only as the very first message an actor processes. The act of spawning an actor via a constructor is semantically equivalent to spawning an actor and inserting a message in its empty mailbox that will initialize the actor.

```
1  (def mistral (spawn-actor Wind 'init))
```

Actors communicate via asynchronous message passing via the `send` special form that inserts a new message in the mailbox of the designated actor. The following expression sends a `blow` message to `mistral`, which is expected to be an actor reference. While in this case `blow` does not expect any arguments, they would be provided after the `'blow` symbol. The message payload between actors (and reactors) is always passed by (deep) copy, and actors can send messages to themselves by sending them to `#self`, which represents a reference to the current actor.

```
1  (send mistral 'blow)
```

### 4.2.2 Declaring Data Streams

Every actor can export data streams. The behaviour of an actor determines which data streams an actor implements via `def-stream` as seen on Line 2 of Listing 7. This expression takes two arguments: the name of the stream and its *arity*. The name is used to uniquely identify a particular stream that belongs to a given actor, and the arity of a stream specifies the number of elements that must be emitted to the stream in one "emit step". In our example a `Wind` actor exports a single stream called `speed` with arity 1.

Stream arity is used to emit new values that are intrinsically connected, and must therefore always change simultaneously. For example, an actor might export a stream called `location` of arity 2 that represents coordinates in the form of `latitude` and `longitude`. Since they describe the real-time location of some real-world moving object, latitude and longitude should always change simultaneously. Otherwise, a consumer of the stream might first update the entire application (e.g. some world map) with a new value for latitude, and only after some time with the corresponding value for longitude. After the first update the

**Listing 8** Monitoring data streams with actors

```
1 (actor Main
2   (def-constructor (start)
3     (def sirocco (spawn-actor Wind 'init))
4     (monitor sirocco.speed 'print-wind))
5
6   (def-method (print-wind wind-speed)
7     (println "the new wind speed is: " wind-speed)))
```

application would be in an inconsistent state, similar to a *glitch* in reactive programming [12]. An alternative approach is to emit `location` as a single object. However, we will use stream arity to facilitate the composition of actors and reactors in Section 4.3.

### 4.2.3 Publishing to Data Streams

Actors can emit values to their own data streams by using the `emit` special form. Emitting a value amounts to sending the new value to all subscribers of the stream. Consider the `blow` method in Listing 7. Whenever a `Wind` actor processes a `blow` message, on Line 8 the actor will `emit` a new value to the `speed` stream, which in this case is a random number between 0 and 30 representing the wind speed in meters/second. Now, a special type of message (a publication) is added to the mailbox of subscribers, which are other actors or reactors. Because the `speed` stream is defined with arity 1, `emit` only requires 1 argument. The actor then sleeps for 10000ms (Line 9), and afterwards sends itself a new `blow` message to emit another value (Line 10).

### 4.2.4 Qualifying and Monitoring Data Streams

Two mechanisms are required to create a subscription on a data stream: *qualification* and *monitoring*. Both are explained using the `Main` behaviour in Listing 8 which can be seen as the console from our running example, but instead of monitoring and printing power output, it monitors and prints the wind speed. When the `start` constructor is executed, Line 3 first spawns an instance of the `Wind` behaviour, and Line 4 exemplifies both qualification and monitoring.

Qualification is the act of designating a reference to a particular stream exported by a particular actor (or reactor). On Line 4, the expression `sirocco.speed` evaluates to an object of class `Stream` that represents a reference to the `speed` stream exported by the `sirocco` actor. Data will only start flowing once a consumer (an actor or reactor) subscribes to the stream, in which case data flows directly from the producer to the consumer.

Actors subscribe to data streams by monitoring them. This is exemplified by Line 4, where the `sirocco.speed` stream is monitored for changes. Whenever this stream emits a new value, a `'print-wind` message will enter the mailbox of the actor, which is processed by the corresponding `print-wind` method on Line 6. This method requires exactly 1 formal parameter because the `speed` stream has an arity of 1. We will see in Section 4.3 that reactors do not explicitly monitor data streams like actors, and instead they will automatically react to values that are emitted to data streams.

## 4.3 Reactors

Reactive languages rely on 2 fundamental mechanisms. First, at compile-time, the program text is compiled to a DAG that consists of source nodes that represent the input of the reactive program, sink nodes represent the output of the reactive program, and internal nodes represent all computations that occur between the sources and sinks. Second, at run-time, a *reactive engine* is responsible for propagating new values through the DAG, such that the calculation that makes up the output remains consistent with the values of the input. The reactive engine is smart enough to prevent glitches [12].

In the following sections we gradually explain Stella's reactors by implementing a simple reactive wind turbine and power calculator. Stella features 2 different ways to statically compose *reactor behaviours*: via *point-wise* and *point-free* composition, named after function composition in Haskell [32, Chapter 5]. In Section 4.3.6 we explain the run-time semantics of reactors and how they manage dependencies on data streams.

### 4.3.1 Definitions

A reactor consists of 3 layers.

**Layer 1: Reactor Behaviour.** A *reactor behaviour* describes the static properties of a reactive program, represented by a DAG that is constructed at "DAG compile-time" (a pre-processing step of our interpreter). The DAG is constructed from the program text, for example the reactor behaviour of Listing 9 which we will explain in Section 4.3.2. It describes the source nodes, sink nodes, and internal nodes of the reactive program, and all of the dependencies between them. We may refer to an actor or reactor behaviour as simply "behaviour" if it is clear from context to which one we are referring.

**Layer 2: Reactor Deployment.** Reactive languages usually store the run-time information of a reactive program (e.g. node values and local state) directly in the nodes of the DAG. However, in our case reactor behaviours can be composed and reused by multiple reactors, and every use of a DAG can be in a different state depending on the values that were propagated. Therefore, a *reactor deployment* represents a specific instance of a reactor behaviour. In other words, a reactor deployment stores all run-time information pertinent to a specific instance of a DAG that is used by a specific reactor.

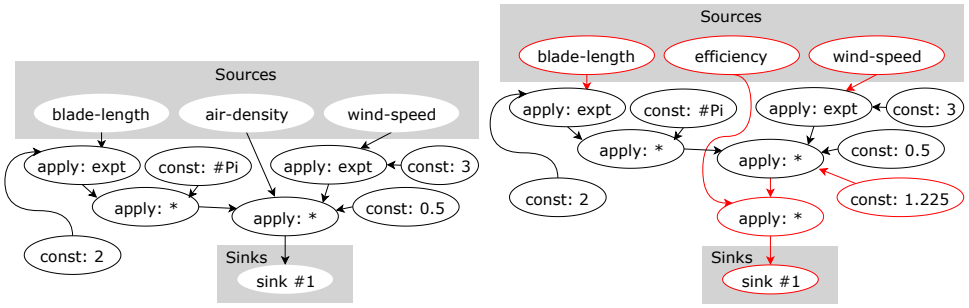
**Layer 3: Reactor.** A *reactor* is process with a mailbox and a *vat* (collection) of reactor deployments. It is the driving force behind a reactive program: a reactor continuously dequeues values from its mailbox and propagates them through the destined deployment via a built-in reactive engine. Similar to how actors are spawned from actor behaviours, a reactor is spawned from a reactor behaviour (that represents a DAG). At spawn-time, the reactor creates an initial deployment for this DAG, which we call the *root deployment*. A reactor has exactly 1 output stream of arity `n`, where `n` corresponds to the number of sinks of the root deployment. Every time the root deployment updates, its sink values are emitted on the output stream of the reactor. Our definition of reactors intentionally covers deployments that give rise to other deployments within the same vat, but we will not discuss those features in this paper. Reactors will therefore always contain exactly 1 deployment (namely the root deployment).

### 4.3.2 Basic Reactor Behaviours

A reactor behaviour is the textual representation of the DAG of a reactive program: it has a name, at least 1 source, at least 1 sink, and any number of internal nodes that describe the computations between the sources and sinks. An example of a computation is the theoretical

**Listing 9** The `WindPower` behaviour calculates the maximum theoretical power output of a turbine.

```
(reactor (WindPower blade-length air-density wind-speed)
  (def swept-area (* #Pi (expt blade-length 2)))
  (out (* 0.5 swept-area air-density (expt wind-speed 3))))
```




**(a)** `WindPower` **(b)** `PowerOutput`

**Figure 4** Side-by-side comparison for the DAGs of `WindPower` and `PowerOutput` of Listing 9 and 10. Nodes and dependencies introduced by `PowerOutput` are highlighted in red.

power output of a wind turbine (in Watt) which is based on the area swept by its blades, the air density surrounding the turbine, and the velocity of the wind [43]. It can be calculated as follows:

$$Power\ (W) = 0.5 \times Swept\ Area\ (m^2) \times Air\ Density\ (kg/m^3) \times Velocity^3\ (m/s)$$

This formula is implemented in Listing 9 in a reactor behaviour called `WindPower` with 3 source nodes and 1 sink node. Its 3 sources are called `blade-length`, `air-density`, and `wind-speed`. There is one local variable called `swept-area`, and one sink node that is immediately linked to the result of a multiplication. The DAG of this behaviour is depicted in Figure 4a. Every invocation of a routine is depicted by an "apply" node, and all expressions without dependencies are wrapped in a "const" node that will be computed when the DAG is compiled. Since the run-time values that are propagated through the DAG are regular objects, reactors can only invoke routines on those objects, and the invocation of regular methods will result in a run-time error.

### 4.3.3 Point-wise Graph Composition

While the `WindPower` behaviour computes the theoretical power output of a turbine, a more accurate calculation takes into account turbine efficiency (typically between 10–30% [42]). To this end, Listing 10 defines a behaviour called `PowerOutput` that shows how reactor behaviours can be composed in a point-wise manner. It has 3 sources called `blade-length`, `efficiency` and `wind-speed`. They correspond to the 3 pieces of information that a wind

**Listing 10** Point-wise composition of reactor behaviours.

```
(reactor (PowerOutput blade-length efficiency wind-speed)
  (def wind-power (tick WindPower blade-length 1.225 wind-speed))
  (out (* efficiency wind-power)))
```

**Listing 11** The `Turbine` behaviour implements a simple wind turbine.

```
1 (reactor (Turbine blade-length efficiency wind)
2   (out blade-length efficiency wind.speed))
```

turbine is expected to produce to be able to calculate its power output. Line 2 performs a point-wise composition of reactor behaviours via the `tick` special form. This can be thought of as a function application but for reactor behaviours, of which the result is bound to the `wind-power` variable. Line 3 then scales the theoretical output by the efficiency of the turbine.

A `tick` operation is resolved at compile-time as the inlining of a DAG. First, the source nodes of the composed behaviour (`WindPower`) are connected to the corresponding arguments of `tick`. Second, the sink node of the composed behaviour is connected back into the composer, in this case to all nodes in `PowerOutput` that use the `wind-power` variable (multiple sinks would be defined via a `def-values` special form). The resulting graph is depicted in Figure 4b, where the nodes and dependencies introduced by `PowerOutput` are highlighted in red. For brevity, in the `tick` expression we assume a default value of 1.225kg/m$^3$ (the air density at 15°C at sea level).

While it is not part of our goals, the structure of the DAG can be optimised when it is compiled. In this case the source and sink nodes of the inlined `WindPower` behaviour were automatically eliminated because they are redundant, and because (by definition) sources and sinks can only exist at the boundaries of a DAG. For example, the constant value 1.225 provided by the composer in Listing 10 is represented by a constant node in Figure 4b (depicted in the bottom right). This node is directly connected to the apply node because the original `air-density` source node (which it replaces) was eliminated.

### 4.3.4 Behaviour Stream Composition

From the `PowerOutput` behaviour we know that a wind turbine should provide its blade length, efficiency, and the current wind speed affecting it. The simplest possible implementation of a turbine that we can think of is defined in Listing 11. This reactor behaviour has 3 sources: `blade-length` is a number usually between 20 and 80 (meters), `efficiency` is a number between 0.1 and 0.3 (10-30%), and `wind` should be a reference to an actor. The output of `Turbine` is a stream of arity 3 that echoes the blade length and efficiency, and the qualification "`wind.speed`" will echo the contents of the `speed` stream.

A qualification in the body of a reactor behaviour represents a dependency to a stream. However, handling such dependencies can be quite tricky since there are potentially two kinds of changing values. First, the exporting actor of a stream may change, for instance when a new actor reference is propagated for the `wind` source node (e.g. `sirocco` or `mistral`). Second, the `speed` stream is continuously emitting new values. A reader may recognise this as a *higher-order* stream [34]. The source node is conceptually a stream of values, and every value is an actor (or reactor) that exports streams.

Inspired by the compilation of the *async* expression in Elm [15], every qualification is compiled to 2 graph nodes. First, an internal node manages the dependency on the referenced stream. Second, an *implicit source node* is responsible for processing the values emitted by the stream. The resulting graph for `Turbine` is depicted in Figure 5. When the source node `wind` changes to a different actor (e.g. another wind) then this new value is propagated to the special qualification node. This node unsubscribes from its current stream (if present)

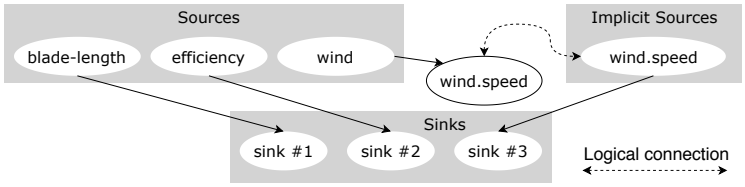



**Figure 5** DAG of the `Turbine` behaviour.

**Listing 12** Point-free composition of reactor behaviours.

```
(reactor TurbinePowerOutput (ror PowerOutput Turbine))
```

and subscribes to the newly referenced `speed` stream. The value of the implicit source is immediately changed to the most recent value emitted by the newly referenced stream. From that point onwards, whenever publications of the new stream enter the mailbox of the reactor, the reactor will process them by changing the value of the corresponding implicit source node.

### 4.3.5 Point-free Graph Composition

There are two ways to implement a wind turbine that is linked to a power calculator. Either a new reactor is spawned for each of them and their input/output streams are subsequently linked together, or the 2 reactor behaviours are composed and spawned as a single reactor. Both approaches are valid, and which one is more desirable depends on the application. We take the second approach by composing the 2 behaviours via *point-free* graph composition.

In Haskell, new functions can be defined point-free via a function composition operator. The composition `f ∘ g` ("`f` after `g`") is a function that first applies `g` to its argument, then `f` to the value returned by `g`; `(f . g) x = f(g(x))`. Similarly, the `ror` operator composes reactor behaviours: $\texttt{r}_1 \circ \texttt{r}_2$ constructs a new behaviour where data is first propagated through $\texttt{r}_2$ and then through $\texttt{r}_1$. The DAG of this behaviour is the composition of $\texttt{r}_1$ and $\texttt{r}_2$, where the sinks of $\texttt{r}_2$ are connected to the sources of $\texttt{r}_1$.

As an example, Listing 12 defines a new behaviour called `TurbinePowerOutput` that combines the behaviours of `Turbine` and `PowerOutput`. We designed those behaviours such that they can be easily composed, i.e. the sinks of `Turbine` directly match the sources of `PowerOutput`. If the behaviours would not directly fit together, intermediate behaviours can take care of reordering sources and sinks, or transforming data.

The `ror` operator is capable of connecting *multiple* "input" behaviours to 1 "output" behaviour. The following expression defines a new behaviour `R` that is the composition of an output behaviour $R_{out}$ with input behaviours $R_1$ to $R_n$.

```
(reactor R (ror R_out R1 R2 ... Rn))
```

Behaviour `R` can be compiled as long as the number of sinks of all input behaviours matches the number of sources in $R_{out}$. If this is the case, they will be connected in order from left to right to construct the behaviour `R`. The sources of `R` will be the same as the sources of the inputs, ordered from left to right.

$$sources(R) := sources(R_1) + sources(R_2) + \ldots + sources(R_n)$$

The sinks of `R` are the same as the sinks of $R_{out}$.

$$sinks(R) := sinks(R_{out})$$

Additional point-free graph composition operators with different semantics are conceivable, such as the `parallel` and `parallel*` operators in [36].

### 4.3.6 Run-time Semantics of Reactors: Spawning and Linking

We now complete the running example of Section 4.1 by showing how the program is started, and how actors and reactors are linked together. We will focus on the composition of actors and reactors rather than the internal semantics of reactors. Internally, it suffices to know that every reactor has a reactive engine that is responsible for propagating values from sources to sinks. Similar to Flapjax [34] and REScala [46], we based our propagation algorithm on that of FrTime [12], but without the complexity of a dynamic dependency graph. It ensures that only the parts of a DAG that are affected by a change will be recomputed, and that computations produce no glitches when multiple nodes change simultaneously.

Listing 13 implements the `Main` program for the running example. Its purpose is to spawn an actor representing a wind, spawn a reactor representing a wind turbine and its power output calculation, and to print this power output to the console. This functionality is implemented by the `start` constructor. Most of the expressions have been discussed previously. Line 3 spawns an instance of the `Wind` actor behaviour and Line 4 sends it a `blow` message. The wind will now start periodically emitting values.

The `spawn-reactor` expression on Line 5 spawns a new reactor that is now waiting for data on its sources. Remember that a reactor is defined as a vat of reactor deployments, and in this case `TurbinePowerOutput` will be the root deployment, as well as the only deployment for this reactor throughout its lifetime.

Reactors are linked to actors (or other reactors) via the `react-to` special form that will change the values of source nodes. If the new value of a source node is an object of type `Stream`, instead of changing the value of the source node, the reactor will automatically create a subscription on the stream (possibly replacing an existing subscription). Then, whenever new values are emitted by this stream, they enter the mailbox of the reactor which will process them by modifying the value of the corresponding source node. In Listing 13 the sources of `turbine` are changed on Line 6 to 80, 0.3, and `sirocco`. In the context of our application, this means that this is a reactor that represents a turbine with a blade length of 80 meters, an efficiency of 30%, which is influenced by the `sirocco` wind. To print the output of the turbine the console, on Line 7 the main actor monitors the turbine for changes. Reactors export exactly 1 output stream called `out`.

Reactors are aware of stream arity, and the `react-to` composition operator can be used to react to streams with an arity greater than 1. In this case, the reactor requires exactly as many source nodes as the arity of the input streams. For example, a reactor that is made to react (via `react-to`) to a stream of arity 2 will require exactly 2 source nodes. Whenever the input stream emits new values they will enter the mailbox as a single publication, but (in this case) the value of the 2 source nodes of the reactor will change simultaneously. The `react-to` composition operator ensures that there is a one-to-one mapping between its arguments and the source nodes of the reactor.

**Listing 13** The `Main` program for the wind turbine simulator of Section 4.1.

```
(actor Main
  (def-constructor (start)
    (def sirocco (spawn-actor Wind 'init))
    (send sirocco 'blow)
    (def turbine (spawn-reactor TurbinePowerOutput))
    (react-to turbine 80 0.3 sirocco)
    (monitor turbine.out 'print))

  (def-method (print watt)
    (println "turbine produced: " (round (/ watt 1000000)) " MW")))
```

## 5 Evaluating the Awkward Squad for Reactive Programming

We introduced the awkward squad for reactive programming as a set of issues that are essential for real-world software development, but that do not fit within reactive programming. In this section we investigate the extent to which these issues can be present in existing reactive languages and frameworks. We find that it is indeed the case that existing languages and frameworks expose operations or mechanisms to developers that fall within one or all issues of the awkward squad. In many cases these operations and mechanisms will be unavoidable for developers, either because they are an inherent part of the language or framework, or because otherwise it would be impossible to write certain applications, e.g. applications with a GUI. In this paper we do not investigate the extent to which possibly issue-causing operations are used in real applications, and if they are present, which bugs they can possibly cause in those specific applications.

Table 1 lists a number of reactive languages and frameworks that we consider to be representative for the state-of-the-art. They are FrTime [12] (Racket), Flapjax [34] (JavaScript), REScala [46] (Scala), ReactJS [28] (JavaScript), Akka Streams [45, Chapter 13] (Scala), and RxJS [49] (JavaScript). We used them to implement our running example of Section 4.1[3]. Below the double horizontal line we list 3 other reactive languages which we did not use to implement the application (for technical reasons) but which we classified according to their respective papers. They will be discussed in Section 5.8. Based on our findings we categorised these languages and frameworks as follows.

**Reactive Thread Hijacking Problem (RTHP).** Does the language or framework solve the Reactive Thread Hijacking Problem? In other words, does the language or framework prevent infinite computations from blocking the reactive program? We make no distinction between eventual reactivity and strong reactivity. If not, we discuss the features that can be used to block the reactive program.

**Reactive Update Order Leak (RUOL).** Does the language or framework solve the Reactive Update Order Leak? In other words, does the language or framework disallow side-effects in internal nodes of the DAG? If not, we will highlight the features where side-effects can be executed inside the DAG.

**Reactive/Imperative Impedance Mismatch (RIIM).** In Section 2.3 we listed a number of different mechanisms used by reactive languages to be able to embed reactive code within imperative code. We discuss, to the best of our knowledge, which of the listed mechanisms are used.

[3] All code from this paper and the implementations used to guide our evaluation of the different languages and frameworks are available in an artefact. To download this artefact, see the supplementary material on the first page of this paper.

**Table 1** Categorisation of reactive languages and frameworks.

| | RTHP | RUOL | RIIM |
|---|---|---|---|
| **FrTime** | × | × | Periodic polling, hidden concurrency, domain specific features, metaprogramming |
| **Flapjax** | × | × | Metaprogramming, domain specific features |
| **REScala** | × | × | Metaprogramming |
| **ReactJS** | × | × | Metaprogramming, domain specific features |
| **Akka Streams** | × | × | ✓ |
| **RxJS** | × | × | Metaprogramming |
| **Stella** | ✓ | ✓ | ✓ |
| **Elm** | × | × | Domain specific features, special forms |
| **ActiveSheets** | ✓ | ✓ | ✓ |
| **Coherence** | × | ✓ | ✓ |

## 5.1 FrTime

FrTime is a functional reactive programming language built in Racket that can be interacted with via the Racket Read-Eval-Print Loop [12].

**RTHP.** The execution time of expressions in FrTime is unrestricted. While there exists a limited set of built-in functions (e.g. arithmetic) that always terminate, the built-in `lift-strict` primitive is used to integrate any (possibly infinitely looping) Racket function with the DAG. The implementation of FrTime exposes multiple threads of execution to developers: one thread is responsible for updating the reactive program, and another thread is responsible for the REPL. To implement an infinite loop that represents a wind, we blocked one thread of execution to continuously modify a source node of the dependency graph.

**RUOL.** Any Racket function may be used in internal nodes of the DAG via the aforementioned `lift-strict` function. Additionally, FrTime offers abstractions for "event streams" that can be mapped and filtered (via `map-e` and `filter-e` respectively) using using regular Racket functions.

**RIIM.** We have identified 4 mechanisms that are used to embed reactive code within imperative code. Firstly, there are 2 built-in signals called `seconds` and `milliseconds` that are updated automagically by the runtime, which can be used for periodic polling. Secondly, multiple threads of execution are exposed to developers. While the Racket REPL thread can be used to send and receive values to and from the reactive program, the semantics of their interaction is not part of the language definition. Thirdly, FrTime has domain specific features in the form of a wrapper around the Racket GUI toolkit [26], which automatically integrates with the DAG. Lastly, source nodes of the DAG can be manually defined via `cells` and `event-receivers`, and they can be assigned to via `set-cell!` and `send-event` respectively. The semantics of modifying a source node is unspecified, especially when assignments to the same sources occur in multiple threads.

## 5.2 Flapjax

Flapjax is a reactive programming language based on JavaScript [34].

**RTHP.** Nodes in the dependency graph consist of arbitrary JavaScript functions, which are unrestricted in their execution time. To create an infinite loop that implements a wind without blocking the browser, we implemented a non-blocking loop by wrapping

JavaScript's `setTimeout` to asynchronously schedule an event in the JavaScript event-loop. Some built-in operations can unintentionally block the reactive program. For example, the `evalForeignScriptValE` operation is applied to a reactive value that contains a URL. It retrieves and evaluates the (foreign) JavaScript file on the URL, and publishes its return value as a new reactive value [50].

**RUOL.** Flapjax programs consist of arbitrary JavaScript expressions that may involve arbitrary side-effects inside the DAG. Some built-in operations execute side-effects in the DAG, such as the GUI modification operators `insertDomB` and `insertDomE` to insert a reactive value in the browser DOM, operations such as `getWebServiceObjectE` to perform XMLHttpRequests, or the aforementioned `evalForeignScriptValE` to evaluate an arbitrary JavaScript file.

**RIIM.** To embed reactive code within imperative JavaScript code, Flapjax has 2 mechanisms. Source nodes of the DAG can be manually created from the GUI via operations such as `extractValueB` and `extractValueE` which are updated automatically by the runtime, and ex nihilo via `receiverE` (to create a new event stream) and `sendEvent` (to modify an event stream). Flapjax also offers special features to construct GUI elements that automatically integrate with reactive values.

## 5.3 REScala

REScala is a reactive programming library in Scala that unifies the concepts of functional reactive programming with object-oriented programming [46].

**RTHP.** REScala imposes no restrictions on the execution time of expressions inside the DAG of a reactive program. Since it is built as a library, regular Scala functions are used to perform computations on reactive values. To model a wind from our running example without blocking the reactive program, we manually constructed a new Scala thread with an infinite loop that non-reactively modifies a source node of the reactive program.

**RUOL.** Since REScala is conceived as a library, the Scala functions used in nodes of the DAG may perform arbitrary side-effects. As a design guideline, the REScala documentation explicitly mentions that functions inside the DAG must be pure [41, 1.5.3].

**RIIM.** To embed reactive code within imperative code, REScala offers special features to manually create new source nodes of the DAG (`Vars`) and to modify them via assignment. Conversely, callbacks can be installed on sink nodes of the DAG to act on their changes.

## 5.4 ReactJS

ReactJS is a JavaScript reactive GUI framework developed by Facebook [28], which is used to develop reactive web applications and mobile applications [21].

**RTHP.** The types of expressions that constitute a ReactJS program are regular JavaScript expressions, and are unrestricted in their computation time. To implement a wind without blocking the browser, we used JavaScript's `setInterval` that calls a function on a repeated interval.

**RUOL.** There are no restrictions on side-effects inside a ReactJS dependency graph. The documentation mentions that reactive components should be pure only with respect to their "props" object, which is a framework-provided object that is used to create dependencies between reactive components [27].

**RIIM.** The state of a reactive component is manually modified via a special `setState` method that triggers a new propagation cycle. ReactJS offers domain specific features (via *JSX templates*) to construct user interfaces that automatically display the values of sink nodes of the dependency graph.

## 5.5 Akka Streams

Akka Streams is a streaming library in Scala based on the Akka actor library [45, Chapter 13]. We had some difficulties reproducing the semantics of our running example in Akka Streams because we found it to be very difficult to create dependency graphs that are not linear (multiple sources or sinks) and which have similar update semantics.

**RTHP.** There are no restrictions on performing long lasting computations inside the DAG, e.g. via the stream operator `map` to apply a regular Scala function to a stream, or `zipWith` to combine 2 or more streams via a regular Scala function.

**RUOL.** Many built-in streaming operators are designed to be without side-effects. While we could not find explicit programmer guidelines discussing side-effects in the Akka Streams documentation, we believe that stream operators are intended to be pure, including those that accept arbitrary Scala functions (e.g. `map` or `zipWith`). We found at least some interest by users of the Akka project on GitHub for stream operators that execute side-effects. In one instance a user requested a novel operator to execute side-effects without performing a value transformation, noting that the only way to achieve the desired effect was via the `map` operator (which, in the experience of the user, lead to code duplication) [52]. In response to this issue a new `wireTap` stream operator was added to Akka version 2.5.13. A new variant of this operator (with different semantics) is currently being requested by a different user [53]. This anecdotal evidence is at least an indication that functional purity is not always upheld by the users of Akka. Some operators with side-effects are built-in, for example, the `log` operator logs the elements flowing through a stream, and the `ask` operator sends an asynchronous message to an actor. Sink nodes also perform side-effects to act on the elements of a data stream, e.g. to print results to the console via a `forEach` operator.

**RIIM.** As far as we know, Akka Streams has a clean separation between the code that is responsible for supplying values to the reactive program (which are actors) and the code that is responsible for the reactive program itself (which are actors that run data streams). Note that this is not always the case, since there exist many operators to create new source nodes ex nihilo whereby the reactive program (an actor running data streams) is itself responsible for collecting/retrieving its input data. For example, the `FiloIO.fromPath` operator that creates a source node to read the contents of a file.

## 5.6 RxJS

RxJS [49] is a streaming library for JavaScript based on the ReactiveX specification [40], which has currently been implemented in 18 languages.

**RTHP.** There are no restrictions on the execution time of the expressions that constitute a stream, and long lasting computations will block the entire program. Implementations of ReactiveX in other languages may support *Schedulers* which are designed to introduce multithreaded processing to streams, but this can cause other issues, especially when used in combination with side-effects.

**RUOL.** Many built-in streaming operators are designed to be without side-effects, and the documentation of RxJS describes operators in general as "pure functions" [39]. We believe that operators that rely on arbitrary JavaScript functions (e.g. `map` or `zipWith`) are intended to be functionally pure as well (but which cannot be enforced). In some cases side-effects are unavoidable, for example when defining a type of source node called a `Subject`, which is manually updated to start the propagation of new values. Frameworks in the family of ReactiveX (such as RxJS) also offer a whole range of "`Do`" operators that are specifically designed to execute side-effects within a reactive program without performing a value transformation [38].

**RIIM.** Programmers in RxJS can manually create and update new source nodes (a `Subject`) of the DAG, and they can manually create new streaming operators using metaprogramming to read from unsupported data sources. Callbacks are registered on sink nodes of the program to imperatively act upon their changes (e.g. by modifying the GUI, or printing to the console).

## 5.7 Stella

**RTHP.** Stella solves the Reactive Thread Hijacking Problem by eliminating infinite computations from reactive programs. Whether our implementation also solves the problem of responsiveness in general depends on the expectations of the application developer. We believe there is no one-size-fits-all solution to ensure that an application remains "reactive" or "responsive", since their meaning is likely to change depending on the application requirements or domain. The Actor-Reactor Model facilitates restricting certain parts of the application (the reactive parts) to provide extra guarantees with respect to responsiveness or computational complexity without introducing the problems that we identify in this paper. The design choice that we made in Stella is to enforce eventual reactivity via size-change termination. Thus, Stella ensures that reactors must *eventually* terminate, and that the execution thread of a reactor is not accidentally hijacked by computations in other parts of the program (other actors or reactors). In different application domains with stronger memory or timing requirements (e.g. safety systems, robotics, ...) it would be conceivable to further restrict reactors, for example by imposing strong reactivity and bounded-size mailboxes.

**RUOL.** Stella solves the Reactive Update Order Leak by ensuring that effectful computations cannot be part of the dependency graph of a reactive program. This is realised by routines in the object-oriented base language. However, using methods and routines may be a burden for programmers, since they must make an effort to correctly program a piece of functionality as a regular method or as a routine. When using built-in classes or libraries they must also know whether functionality is offered as a method or as a routine.

**RIIM.** Stella solves the Reactive/Imperative Impedance Mismatch. It ensures that the embedding of reactive code within imperative code does not introduce the Reactive Thread Hijacking Problem and the Reactive Update Order Leak, and that the semantics of embedded reactive code are clear. There are 2 composition operators available to actors: `react-to` and `monitor`. To imperatively change the source nodes of a reactive program, actors must use `react-to` with clearly defined semantics: a message is sent to the reactor that, when processed, changes its source nodes. To imperatively act on the changes of reactive programs, actors must use `monitor`: whenever the monitored stream produces a new result, a new message is enqueued in the mailbox of the actor. Reactors have no imperative operators to "send" or "receive" values, or to manually react to the changes of values (e.g. via callbacks). In the true spirit of reactive programming, they can only declaratively express dependencies on data sources via their source nodes and qualifications.

## 5.8 Additional Mentions

There are some reactive languages and frameworks that require a special mention. When possible we list them in Table 1 below the double horizontal line.

Elm [15] is a reactive programming language that compiles to JavaScript. We were unable to build our running example in Elm because its current distribution is no longer reactive [14]. Expressions inside the DAG in Elm may perform infinite computations that block the reactive program. An interesting observation is that Elm is presented as a purely functional reactive programming language, but its paper describes a `syncGet` operation to execute a web request (e.g. to fetch an image from a URL), which is clearly a side-effect. The reason why this operation (among others) is built-in is because it is *necessary* for building web applications, but introducing this operation in the reactive language also causes the problems of the awkward squad. This is exactly the essence of the awkward squad.

ActiveSheets [54] is a reactive language based on Microsoft Excel where the DAG consists of regular spreadsheet operations. ActiveSheets adds features to Excel to automatically insert values into cells based on external data streams, and, like a regular spreadsheet, updates to cells automatically propagate throughout the program. The core language of ActiveSheets is formalised and used to prove that, for any given update of a cell, computation time and memory usage are bound. Furthermore, as far as we know there are no spreadsheet operations that have side-effects on other cells, so there can be no side-effects in internal nodes of the DAG. However, ActiveSheets is not a general purpose programming language. Conceptually an ActiveSheets program can be represented by 1 reactor that implements the spreadsheet logic.

In the Coherence language, code is divided in *derivations* and *reactions* [19]. Derivation is used to automatically compute the program output by deriving values from input via purely functional computations. Side-effects are isolated to different parts of the code, namely reactions, that are responsible for imperatively keeping the application state consistent with derived values. This stems from the insight that derivation and reaction need each other, but that coordinating side-effects in an event-based application is extremely difficult. A reactive program constructed via derivations is guaranteed to be free of side-effects, and is thus not subject to the Reactive Update Order Leak. The Coherent Reaction programming model offers no mechanisms to solve the Reactive Thread Hijacking Problem.

HipHop is a synchronous reactive programming language inspired by Esterel with an implementation in Scheme [6] and JavaScript [55]. HipHop is not classified in Table 1 because the programming style and evaluation model of synchronous reactive programming languages makes them difficult to compare with the approaches in Table 1. However, there are interesting parallels between some aspects of the HipHop language and the Actor-Reactor Model. HipHop is embedded as a DSL within the Hop language, and Hop code interfaces with HipHop code via *reactive machines* that conceptually fulfil the same role as reactors. Hop code sends input events to a reactive machine and manually triggers a propagation cycle. Output events produced by the HipHop machine can be observed by Hop code via event handlers. Interestingly, one of the core language statements called `atom&` is used within HipHop to execute Hop code, which may contain side-effects and recursive functions. While side-effects in synchronous reactive programs are arguably not subject to the issues discussed in Section 2.2, the authors make a note that the "*execution time [of atom statements] should be kept negligible in practice*" [6, 3.4].

## 6 Conclusion

To conclude this paper we reflect on our 2 largest contributions, namely identifying the awkward squad for reactive programming and the Actor-Reactor Model. We believe that there is no panacea to write both imperative and reactive programs within a single unified language that exposes the same concepts and operations in both types of programs. Instead, we believe that imperative and reactive programs are fundamentally different, and that they should be programmed whilst guaranteeing their own invariants. To this end the Actor-Reactor Model serves as a new mental model to classify and design reactive systems.

As a secondary contribution, our definition of reactors may prove to be valuable to the field of reactive programming in two ways. First, we define distinct terminology for the different stages of a reactive program: a reactor behaviour represents a dependency graph that is constructed from code, a deployment is a specific instance of a graph that keeps track of all run-time information and state, and a reactor contains the reactive engine that propagates values through one or more deployments. Conceptually, the reactive programs in many existing reactive programming languages are analogous to 1 reactor with 1 deployment. By using our definitions, we open the door for modularity and composition of reactive programs both statically (e.g. via point-wise and point-free composition operators) and dynamically by composing data streams. Second, we conjecture that reactors together with the mechanism of qualification is an alternative, but equally powerful, way to construct higher-order reactive programs. Reactors may provide more insights or clarities with respect to the run-time semantics and resource usage of higher order reactive programs.

The Actor-Reactor Model may inspire designers of reactive languages, framework developers, and researchers to be strict about what can and cannot be programmed within a reactive program, and it may help them to define precisely the rules by which reactive programs interact with their environment. Additionally, the Actor-Reactor Model may be especially useful in application domains where certain parts of a program must be reactive, for example, with specific memory or timing constraints (e.g. robotics and safety systems). Operations that might not fit this model, but which are necessary for program development, are evacuated into actors which are complementary to the reactive program, but which do not violate the invariants of the reactive programming model.

## References


1 Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

2 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017. doi:10.1145/3133956.3134078.

3 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52:1–52:34, 2013. doi:10.1145/2501654.2501666.

4 Gérard Berry. Real time programming: Special purpose or general purpose languages. In Gerhard Ritter, editor, *Information Processing 89, Proceedings of the IFIP 11th World Computer Congress, San Francisco, USA, August 28 - September 1, 1989.*, pages 11–17. North-Holland/IFIP, 1989.

5 Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. doi:10.1016/0167-6423(92)90005-V.

6 Gérard Berry and Manuel Serrano. Hop and hiphop: Multitier web orchestration. In Raja Natarajan, editor, *Distributed Computing and Internet Technology - 10th International Conference, ICDCIT 2014, Bhubaneswar, India, February 6-9, 2014. Proceedings*, volume 8337 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2014. doi:10.1007/978-3-319-04483-5_1.

7 Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The reactive manifesto. https://web.archive.org/web/20191210084324/https://www.reactivemanifesto.org/. Accessed: 2019-12-10.

8 Walter S Brainerd and Lawrence H Landweber. *Theory of computation*. John Wiley & Sons, Inc., 1974. URL: https://archive.org/details/theoryofcomputat00brai.

9 Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2010. doi:10.1007/978-3-642-13953-6_3.

10 Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pages 69–76. IEEE Computer Society, 2017. doi:10.1109/SecDev.2017.24.

11 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 415–426. ACM, 2006. doi:10.1145/1133981.1134029.

12 Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *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 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2006. doi:10.1007/11693024_20.

13 Oracle Corporation. JEP 266: More concurrency updates. https://web.archive.org/web/20191009093608/https://openjdk.java.net/jeps/266. Accessed: 2019-10-09.

14 Evan Czaplicki. A farewell to frp. https://web.archive.org/web/20191208051242/https://elm-lang.org/news/farewell-to-frp. Accessed: 2019-12-30.

15 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 411–422. ACM, 2013. doi:10.1145/2491956.2462161.

16 Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Computing Surveys*, 50(5):76:1–76:39, 2017. doi:10.1145/3122848.

17 Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. The synchronous hypothesis and synchronous languages. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. doi:10.1201/9781420038163.ch8.

18 Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-safe reactive programming. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):107:1–107:30, 2018. doi:10.1145/3276477.

19 Jonathan Edwards. Coherent reaction. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 925–932. ACM, 2009. doi:10.1145/1639950.1640058.

20 Jonathan Edwards. Coherent reaction. Technical Report MIT-CSAIL-TR-2009-024, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, Cambridge, 02139 Massachusetts, USA, June 2009. URL: http://web.archive.org/web/20181103183154/http://dspace.mit.edu/bitstream/handle/1721.1/45563/MIT-CSAIL-TR-2009-024.pdf?sequence=1.

21 Bonnie Eisenman. *Learning React Native: Building Native Mobile Apps with JavaScript*. O'Reilly Media, Inc., 2 edition, 2017.

22 Daniel P. Friedman and Mitchell Wand. *Essentials of programming languages (3. ed.)*. MIT Press, 2008.

23 Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987. doi:10.1007/3-540-18317-5_15.

24 Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

25 Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, 1992. doi:10.1109/32.159839.

26 Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2006. doi:10.1007/11737414_18.

27 Facebook Inc. Components and props. https://web.archive.org/web/20191126131226/http://reactjs.org/docs/components-and-props.html. Accessed: 2019-11-26.

28 Facebook Inc. React: A javascript library for building user interfaces. https://web.archive.org/web/20191009084855/http://reactjs.org/. Accessed: 2019-10-09.

29 Reactive Streams Initiative. Reactive streams. https://web.archive.org/web/20191009093755/https://www.reactive-streams.org/. Accessed: 2019-10-09.

30 Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In Sylvan Clebsch, Travis Desell, Philipp Haller, and Alessandro Ricci, editors, *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, pages 31–40. ACM, 2016. doi:10.1145/3001886.3001890.

31 Roland Kuhn, Brian Hanafee, and Jamie Allen. *Reactive Design Patterns*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.

32 Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

33 Ingo Maier and Martin Odersky. Deprecating the observer pattern with scala.react. Technical Report EPFL-REPORT-176887, École Polytechnique Fédérale de Lausanne, EPFL IC IINFCOM LAMP, Station 14, 1015 Lausanne, 2012. URL: http://web.archive.org/web/20200522141109/https://infoscience.epfl.ch/record/176887.

34 Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 1–20. ACM, 2009. doi:10.1145/1640089.1640091.

35 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, pages 845–859. ACM, 2019. doi:10.1145/3314221.3314643.

36 Bjarno Oeyen, Humberto Rodríguez-Avila, Sam Van den Vonder, and Wolfgang De Meuter. Composable higher-order reactors as the basis for a live reactive programming environment. In Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, Lukasz Ziarek, and Francisco Sant'Anna, editors, *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2018, Boston, MA, USA, November 4, 2018*, pages 51–60. ACM, 2018. doi:10.1145/3281278.3281284.

37 Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering Theories of Software Construction*, 180:47–96, January 2001. IOS Press.

38 ReactiveX. Do. https://web.archive.org/web/20200415125833/http://reactivex.io/documentation/operators/do.html. Accessed: 2020-04-15.

39 ReactiveX. Introduction. https://web.archive.org/web/20200415132837/http://reactivex.io/rxjs/manual/overview.html. Accessed: 2020-04-15.

40 ReactiveX. ReactiveX: An api for asynchronous programming with observable streams. https://web.archive.org/web/20191009085652/http://reactivex.io/. Accessed: 2019-10-09.

41 REScala. REScala manual. https://web.archive.org/web/20191126124033/http://www.rescala-lang.com/manual/. Accessed: 2019-11-26.

42 REUK. Betz limit. https://web.archive.org/web/20191025112828/http://www.reuk.co.uk/wordpress/wind/calculation-of-wind-power/. Accessed: 2019-10-25.

43 REUK. Calculation of wind power. https://web.archive.org/web/20191025113431/http://www.reuk.co.uk/wordpress/wind/betz-limit/. Accessed: 2019-10-25.

44 RexEgg. The explosive quantifier trap. https://web.archive.org/web/20191223155226/https://www.rexegg.com/regex-explosive-quantifiers.html. Accessed: 2019-12-23.

45 Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in action.* Manning Publications Co., 1 edition, 2016.

46 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: bridging between object-oriented and functional style in reactive applications. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 25–36. ACM, 2014. doi:10.1145/2577080.2577083.

47 Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017. doi:10.1109/TSE.2017.2655524.

48 Francisco Sant'Anna, Roberto Ierusalimschy, Noemi de La Rocque Rodriguez, Silvana Rossetto, and Adriano Branco. The design and implementation of the synchronous language CÉU. *ACM Trans. Embedded Comput. Syst.*, 16(4):98:1–98:26, 2017. doi:10.1145/3035544.

49 RxJS Team. RxJS: Reactive extensions library for javascript. https://web.archive.org/web/20191125123104/https://rxjs.dev/. Accessed: 2019-11-25.

50 The Flapjax Team. Flapjax framework api documentation. https://web.archive.org/web/20191128081915/https://www.flapjax-lang.org/docs/. Accessed: 2019-11-28.

51 D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004. doi:10.3217/jucs-010-07-0751.

**52** GitHub user "htimur". Akka Streams: Utility function for side effects #23512. https://web.archive.org/web/20200415164156/https://github.com/akka/akka/issues/23512. Accessed: 2020-04-15.

**53** GitHub user "otto-dev". Request: Overloaded version of .alsoTo that takes a function #28524. https://web.archive.org/web/20200415154306/https://github.com/akka/akka/issues/28524. Accessed: 2020-04-15.

**54** Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2014. doi:10.1007/978-3-662-44202-9_15.

**55** Colin Vidal, Gérard Berry, and Manuel Serrano. Hiphop.js: a language to orchestrate web applications. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 2193–2195. ACM, 2018. doi:10.1145/3167132.3167440.

**56** Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, pages 146–156. ACM, 2001. doi:10.1145/507635.507654.

**57** Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In Norman K. Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings.*, pages 258–268. ACM, 1986. doi:10.1145/28697.28722.