Integrating complex event processing and transactional dataflow

Integration von komplexer Ereignisverarbeitung und transaktionellen Datenflusssystemen
Bachelor thesis by Vincent Stollenwerk
Date of submission: October 12, 2021

1. Review: Prof. Dr. Mira Mezini
2. Review: Ragnar Mogk
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Software Technology Group

Integrating complex event processing and transactional dataflow
Integration von komplexer Ereignisverarbeitung und transaktionellen Datenflusssystemen

Bachelor thesis by Vincent Stollenwerk

1. Review: Prof. Dr. Mira Mezini
2. Review: Ragnar Mogk

Date of submission: October 12, 2021

Darmstadt

Abstract

Modern applications are often driven by user interactions and user triggered events. Traditional event handling logic using imperative code and the observer pattern can quickly become complex and error-prone. Reactive programming frameworks try to solve this by simplifying specification of event handling and dataflow. Although modern reactive frameworks can improve the event handling experience a lot, they still lack expressiveness for more complex events. When handling such event, a fallback to imperative code and observers is often needed. In this thesis we extend the ReScala framework with so-called reactors to provide a simple tool for complex event processing in a reactive context. Reactors embed imperative code into a reactive object, which changes state depending of series and combinations of events.

Contents

1 Introduction

A large part of today's software is concerned with processing event and data streams. A web application, for example, has to handle mouse clicks and keyboard inputs. Or a monitoring tool has to continuously check performance metrics for anomalies.

Using imperative code (e.g., the observer pattern) to handle such events is complex and error prone. According to a 2006 presentation, 1/3 of the code in Adobe's desktop applications was dedicated to event handling. Additionally, this code was the source of 1/2 of all bugs reported during the product life cycle [7].

Since then, lots of research studied the reactive programming (RP) paradigm. A recent study shows, programs following the reactive programming paradigm result in significantly more comprehensible code than traditional object oriented code using the observer pattern [9]. Additionally, more and more reactive frameworks are developed and released. Notably FlapJax [5] and REScala [6, 8] in academia and frameworks like Facebook's React.js in web development.

While these reactive approaches already greatly improve the code for handling tasks like updating a GUI element after the user clicked a button, they mostly still can not provide useful abstractions for handling more complex tasks. Especially doing tasks that need to combine multiple events is hard. For example a smart gardening system, which always has to water the plants until a humidity sensor is satisfied, when the sun has been shining for more than 2 hours after the plants have been watered, or after it has rained. Two aspects of this example can not be directly expressed with current reactive languages:

1. Resetting the sun shine timer would require manual state management through observing the rain event and checking whether manual watering is finished.
2. Starting and stopping the watering process would require manual state management and observation of multiple values (sunshine timer and humidity sensor).

Although using manual state management would work, the resulting code lacks the improved comprehensibility and maintainability of reactive code.

Generally, manual state management and imperative code can always be used to solve problems that cannot be expressed with reactive abstractions. To bridge manual state management and reactive code, often times observers are needed. REScala, for example, has an API to attach observers to event streams. When implementing manual state management using observers, the resulting code often violates important software engineering principles, because the observer pattern promotes side-effects, restricts composability and hinders separation of concerns [4].

There are specialized complex event processing engines like Apache Flink, Apache Ignite, or Apache Storm, which are used for event detection and event correlation in streams. Therefore, they are often used to combine multiple events or data streams and detect patterns on them. Although these engines could be used to handle complex events, which can not be expressed easily using RP, these engines are mostly specialized for high-throughput stream processing and huge amounts of data. To allow for this, they usually run as separate processes. Because of the resulting overhead, running them mostly makes sense for bigger workloads.

Scala.React proposes the concept of *reactors* [4]. They provide a reactive operator that can manage state on its own. Therefore, they are an alternative to imperative, manual state management using observers. Reactors are self-contained entities that can handle combinations of multiple different events using imperative descriptions. Although they already allow for simpler expression of complex events, in their proposed version they still have side effects and are not fully self-contained.

In this thesis, we transfer the concept of reactors from Scala.React to REScala and implement our own version. We extend the functionality of Scala.React and add values to our reactors. Having values eliminates the need for side-effects like modifying external values, or call methods outside of the scope of our reactors. Consequently, our reactors are also members of the reactive graph. Therefore, programmers can treat and compose reactor values like signals, which leads to simpler code and improves composability.

First, we demonstrate the functionality and benefits with an introductory example. Then we discuss how the reactors are implemented into the REScala library. Finally, we evaluate the performance of our reactor implementation and compare the performance of different reactor features. We show that our implementation can handle orders of magnitude of six to seven thousand operations per millisecond on a modern midrange processor and

therefore, believe that it is a suitable solution to simplify complex event handling using REScala.

2 Motivation and Background

In this chapter, we explain what the reactive programming paradigm is and how it can improve event handling code. Then we show how it can struggle with complex events and implement a keylogger to compare different programming paradigms and ways to implement complex event processing. These implementations are:

1. A traditional implementation using the observer pattern
2. A reactive version using so called reactors from Scala.React
3. A reactive version using our version of reactors

We discuss the advantages and disadvantages of the different implementations and show how our solution improves upon the existing solutions.

2.1 Problem Statement

In today's world, event-based, reactive applications are a huge part of software development. For example, user interfaces have to change according to user input, sensor systems have to react to new measurements, and microservices have to handle each others events. The common way of handling such event handling logic using imperative code and the observer pattern often results in code that is error prone and difficult to understand. Therefore, a multitude of technologies implementing the reactive programming paradigm rose in popularity.

While reactive programming simplifies writing event handling code a lot, detection and combination of more complex event logic can still be hard. In this regard, current reactive frameworks like REScala lack expressiveness and therefore, implementations often have to fall back to the observer pattern.

This thesis aims to improve handling of such complex events by extending the REScala framework with a concept we call reactors.

2.2 Complex Event Handling by Example

When speaking about complex events we mean events that themselves are a combination of other events. A typical example is detection and handling of a drag-and-drop operation. It starts with the mouse being pressed, then follows updates of the cursor location, until it finishes with the release of the mouse. Each of these steps are separate events that form a new event in combination. Problems like these can be found in applications with user interactions, smart home management systems, vehicle monitoring, and more [2].

In the following sections we show the typical complications when implementing complex event processing using a keylogger as an example. A keylogger is suitable for showing them because it has to generate a data stream by combining several events. In our example it will be activated by a start and a stop event and while it is active it will combine key events of a keyboard into a list of keystrokes.

2.2.1 Traditional approach

Listing 1 shows how such an application could be implemented in a non-reactive way, using the observer pattern. Line 1 defines a list that is used to hold the pressed keys. Lines 2–5 define an observer, which listens for key presses and appends them to the previously defined list. Notably, this observer is only saved in a variable and not attached yet. Lines 7–10 add an observer to the “start” button, which empties the list for the key presses and attaches the key press observer to the keyboard. Lines 12–14 add an observer to the “stop” button, which removes the key press observer from the keyboard and calls some function to process the logged key presses.

Although the shown code works, it has a few characteristics that make it hard to understand and error prone:

1. The `keyObserver` modifies the `keyPresses` variable, which is outside of the observers’ scope. Therefore, the observer has **side-effects**.

```
var keyPresses = new ListBuffer[String]()
val keyObserver = { (e: KeyEvent) =>
  val keyText = KeyEvent.getKeyText(e.getKeyCode())
  keyPresses += keyText
}

startButton.addActionListener { e =>
  keyPresses.clear()
  keyboard.addActionListener(keyObserver)
}

stopButton.addActionListener { e =>
  keyboard.removeActionListener(keyObserver)
  processKeypresses(keyPresses)
}
```

Listing 1: Observer pattern

2. The `keyObserver`'s **life-cycle has to be actively managed**. It is attached and removed from within the other button observers. This can quickly become hard to oversee and complex.
3. **The control flow is inverted**. Although the events we track have a logical order (start button click, key presses, stop button click), the behavior is described per event, which obscures the order that we are tracking.

This can be simplified by using reactive programming.

2.2.2 Reactive Approach

As alluded in the introduction, the reactive programming paradigm is a way to simplify data flow and event handling logic. In reactive programming relationships and dependencies between values are described in a declarative, time-independent way. If input values change, the reactive framework automatically updates the values of its dependents according to the declared relationships. In REScala, these changing values are called *signals*. Notably, the programmer does not have to explicitly and imperatively handle every update in code.

REScala has two main elements: *signals* and *events*. Signals carry values like traditional variables, but they can be updated automatically by dependencies and propagate changes

```
val keyPresses: Signal[List[String]] = Var(List())

val startObserver = startButton.observe({ _ =>
  keyPresses = keyEvent.fold(List()) { (oldPresses, newKey) =>
    oldPresses :+ newKey
  }
})

val stopObserver = stopButton.observe({ _ =>
  keyPresses = Var(List())
})

processKeypresses(keyPresses)
```

Listing 2: Keylogger using reactive programming in REScala

to other signals that depend on them. Events can be considered a special form of signals. In their default state they hold no value, but they can be fired. If they are fired, they are considered activated and hold a value. They lose their value as soon as their activation has propagated the reactive graph. REScala also provides an API to add observers to events. They can be used to bridge the gap between object oriented and reactive programming.

Listing 2 shows how the key logger could be improved by using REScala and reactive programming. Notably, when a new `keyEvent` is triggered, REScala automatically propagates the change and updates `keyPresses` accordingly. This change is then also automatically propagated to `processKeypresses`. Reactive programs therefore, describe a graph in their declaration, where the nodes are signals and the connections are the relationships between them.

By declaring the `keyPresses` variable as a signal and using `fold()` to combine events into a signal value, the reactive implementation can improve upon the traditional approach using the observer pattern.

Although, key-presses can now be constructed without **side-effects** by folding the event into a signal, two problems still remain unresolved:

1. The **life-cycle** of starting and stopping the logging using `fold` still has to be **actively managed**. It is still attached by observers observing the start and the stop buttons.
2. The **control flow is still inverted**. The tracking logic still has to be done per event and split across different observers. Therefore, the code is not in the logical order of events.

```
1  Reactor.loop { self =>
2    val keyPresses = new List[String]()
3    self await startButton
4    self.loopUntil(stopButton) {
5      val keyEvent = self await keyEvent
6      val keyText = KeyEvent.getKeyText(keyEvent.getKeyCode())
7      keyPresses = keyPresses := keyText
8    }
9    processKeypresses(keyPresses)
10 }
```

Listing 3: Scala.React

Scala.React iterates upon this by proposing the concept of *reactors*.

2.2.3 Scala.React reactors

Reactors provide an environment to describe and handle sequences of events without inversion of control and less boilerplate code.

Listing 3 shows how the key logger example could be simplified and would be implemented using Scala.React's reactors. Line 1 creates a new `Reactor`. There are two factory methods for Scala.React reactors: `Reactor.once` and `Reactor.loop`. `Reactor.once` executes the body once, while the body of `Reactor.loop` is executed repeatedly. Line 2 initializes a list of key presses. Notably, the list is now kept inside the scope of the reactor. Line 3 uses the `await` function to wait until the start button is pressed. `await` can be used to pause the execution of the reactor until an event is fired. Line 4 then starts a loop, which is stopped when the stop button is pressed. Again, `loop` is a function provided by Scala.React reactors which repeats the code inside until an event is fired. Inside this loop the reactor waits for a key press (`keyEvent`) and adds it to the `keyPresses` list. When the `stopButton` is pressed and the loop execution is stopped, a function to further process the key presses is called in line 9.

This implementation solves two of the three problems of our traditional approach:

1. It eliminates the observers. Therefore, **no observer life-cycles have to be actively managed**.
2. The **control flow is not inverted** anymore. The code to handle the events is in the same order as their occurrence.

```
1  val keyLogger = Reactor.loop(new ListBuffer[String]()) {
2    S.modify(keyPresses => keyPresses.clear())
3      .next(startButton) {
4      S.until(stopButton, {
5        S.loop {
6          S.next(keyEvent) { e =>
7            val keyText = KeyEvent.getKeyText(e.getKeyCode())
8            S.modify(keyPresses => keyPresses += keyText)
9    }}}}}
10 }
11
12 processKeypresses(keyLogger)
```

Listing 4: Key logger in REScala

Still, the need to call `processKeypresses` and pass the inner `keyPresses` variable could be considered a **side-effect**.

2.2.4 Our Proposal – Reactors in REScala

We see reactors as a solution for simplifying complex event processing in reactive programming. As a result, we propose our own version of reactors, based on and extending the reactors from Scala.React.

In our implementation reactors are a *reactive* (an entity in the reactive graph) on their own, like signals or events. Therefore, they have their own value and can be incorporated into the dependency graph. This eliminates the need for side-effects that the implementation proposed in Scala.React had.

Listing 4 shows a possible implementation of the key logger, using our version of reactors. Notably, line 1 initializes a new looping `Reactor` with an empty list. Line 3 then clears the list. Line 4 specifies to wait for the `startButton` to be pressed. Line 6 uses the `until` action, which gets passed an interrupt event and a body, stopping the execution of the body when the interrupt event is fired. Lines 7–13 define this body. In the body, line 8 defines a loop which waits for the next `keyEvent` in line 10 and adds it to the list of pressed keys, which is the value of the reactor itself in lines 11–13. Our syntax will be discussed in detail in Section 2.3.1.

Our implementation shares the positive characteristics of the Scala.React reactors:

1. It eliminates the need for observers and thereby **removes the need for active observer life-cycle management**.
2. The **control flow is not inverted**. It is written in the same order as the events occur.

But additionally, it also **removes the need for side-effects** because it is a reactive itself.

2.3 Reactors in REScala

In REScala, reactors are a new reactive that calculates its value by running through a defined sequence of actions. This sequence of actions can be seen as the reactor's code. Some of these actions can wait for events or branch conditionally. Therefore, reactors can house complex calculation rules or state machines.

Because REScala reactors are reactives on their own, they are part of the reactive graph. Like signals, they always have a value and they can depend on other reactives. Their values automatically update when the values of their dependencies change.

2.3.1 Syntax and Usage

Reactors can be initialized using two factory methods. One of those creates a reactor that runs through its code only once. The other one creates a reactor that loops through its code indefinitely.

```
Reactor.once(<initial value>){ <body> }

Reactor.loop(<initial value>){ <body> }
```

The reactor body contains the reactors' actions, which have to be encoded in form of "reactor stages". The actions of a stage are always executed at a single point in time. This means, if the value of the reactor is changed multiple times in one stage, semantically only the last change exists.

There are six actions. To better explain these actions we use a reactor, that counts button presses as an example. The counter always starts at 0. Then it increments itself by 1 every time a button is pressed. When the counter reaches 10, it should automatically set itself to 100 and stop incrementing. A reset button can reset the counter. An implementation of this counter can be found in Listing 5.

set Sets the reactor's value to a new value.
Semantics: Changes the value to the specified value. This happens instantly when it is executed.
Example: The *set* action can be used to initialize and reset the counter to 0, as well as to set it to 100, when the counter reaches 10.

modify Accepts a function which takes the reactor's current value and sets the reactor's value to the function's return value.
Semantics: Evaluates the function and passes the current reactor value along. Sets the value of the reactor to the function's result. This happens instantly when it is executed.
Example: The *modify* action can be used to increment the counter's value.

next Accepts an event and a new reactor stage. Executes the new stage when the event fires.
Semantics: Checks if the event is currently firing and swaps the current stage with the new stage if this is the case. If the event is not firing, *next* does nothing and is reevaluated in the next update turn again. A reactor can only progress though an *next* action per update turn. The *next* action returns the value of the event if the event has one. This value can be accessed in the scope of its body.
Example: The *next* action can be used to wait for key-presses before incrementing the value.

read Accepts a function which takes the reactor's current value and returns a new stage. The new stage is executed immediately. An example use case would be to return different stages depending on the current reactor value.
Semantics: Evaluates the function and passes the current reactor value along. Set's the reactor to the resulting stage. This happens instantly when it is executed.
Example: When the increment button is pressed, *read* can be used to determine if 10 was reached and conditionally return stages, which either increment the reactor or set it to 100.

loop Accepts a new reactor stage. The given stage will be executed in a loop.
Semantics: Repeats the actions in the loop body indefinitely. If the loop body does not contain waiting actions, it can lead to infinite loops.
Example: The *loop* action can be used to repeat the incrementing (waiting for a button press and setting the new value) every time the button is pressed.

until Accepts an event and two new reactor stages. A "body" stage and a "interrupt" stage. The body stage is then executed until the event fires. When the event fires,

the execution of the body stage will be stopped and the interrupt stage is started.
Semantics: Executes the body instantly. If the event is firing it switches to the interrupt stage and continues execution instantly.
Example: The *until* action can be used to implement the reset functionality. When the reset button is pressed, until stops the execution of its regular body and continues with the reset.

Syntactically, stages are constructed by calling chained actions behind the `S` object, which matches the methods of the `Stage` class to simplify the resulting syntax. Actions are implemented as methods of this `Stage` class and every action method returns the `Stage` object to allow for chaining.

```
val incrementEvent = Evt[Unit]()
val resetEvent = Evt[Unit]()
val reactor = Reactor.loop(0) {
  S.until(resetEvent,
    body = {
      S.loop {
        S.next(incrementEvent) {
          S.read { currentValue =>
            if (currentValue >= 9) {
              S.modify(value => value + 1)
            } else {
              S.set(100)
            }
          }
        }
      }
    },
    interrupt = {
      S.set(0)
    }
  )
}
```

Listing 5: Reactor counter example

3 Implementation

Reactors are implemented as REScala reactives. They therefore have a state from which their current value can be derived. In REScala all reactives are managed by the scheduler. Consequently, our reactor implementation has to implement the `Derived` trait, which itself extends the `ReSource` trait.

These traits come with two design constraints:

1. All of the reactor's state needs to be held in a single `state` variable.
2. All changes in state have to happen in the `reevaluate()` method through result tickets that are managed by the scheduler. It is called if any of the dependencies of the reactor have changed and it can happen that it is called multiple times. Therefore, `reevaluate()` must be free of side effects.

3.1 Reactor Stages

As described in Section 2.3.1, the execution of reactors is divided into reactor stages. All actions of a stage are guaranteed to be executed at once. Semantically they are executed in the same transaction of the reactive framework.

Stages are represented by a `Stage` class, which holds a list of reactor actions and has a method to add each action. The `Stage` class is also exposed to users of REScala and thereby part of the REScala language.

The implementation of the actions and their methods is explained in Section 3.5. In general, calls to the action methods (like `set()`, `modify()`, or `loop()`) always add a `ReactorAction` object to the `actions` list. These `ReactorAction` objects hold the information necessary to execute the action. Especially, they can also contain stages with actions that lead to other stages, like the `next()` action.

```
case class Stage[T](actions: List[ReactorAction[T]] = Nil) {
  def set(...): Stage[T] = ...
  def modify(...): Stage[T] = ...
  def loop(...): Stage[T] = ...
  ...
}
```

Listing 6: Definition of the Stage class

```
case class ReactorState[T](
  currentValue: T,
  currentStage: Stage[T]
)
```

Listing 7: Definition of the ReactorStage class

3.2 Encoding the Reactor's State

The state of the reactor has to be encoded in a single variable. To achieve this, we store the state in form of a `ReactorState` object.

The `ReactorState` class has two attributes and a type parameter `T`. Throughout the implementation `T` is the type of the reactor's value. The `currentValue` attribute stores the reactor's current, user facing value. The `currentStage` attribute stores the stage that needs to be processed next. Because a stage has a list of actions, knowing the current stage also means knowing the next actions to execute.

Using this class, the reactor's state can be modified by changing its `currentValue` or by changing its `currentStage`.

3.3 Initialization

Reactor objects can be initialized with two factory methods: `Reactor.once()` and `Reactor.loop()`. In REScala reactives have to be created by the scheduler because the scheduler has to track all reactives and their dependencies to manage the automatic and reactive propagation of changes.

```
private[rescala] def create[V, T <: Derived](
  incoming: Set[ReSource],
  initv: V,
  inite: Boolean
)(instantiateReactive: State[V] => T): T = ...
```

Listing 8: CreationTicket.create() method

```
private def createReactor[T](
  initialValue: T,
  initialStage: Stage[T]
): Reactor[T] = {
  CreationTicket.fromScheduler(scheduler)
    .create(
      Set(),
      new ReactorState[T](initialValue, initialStage),
      inite = true
  ) { createdState: State[ReactorState[T]] =>
    new Reactor[T](createdState)
  }
}
```

Listing 9: createReactor() method

The creation of reactives through the scheduler can be done with `CreationTicket`s that can be retrieved from the scheduler. Specifically, the reactives can be created with the `create()` method of a `CreationTicket`.

The `create()` method has two parameter lists. The first list takes three parameters: `incoming`, `initv`, and `inite`. The `incoming` parameter is a `Set` of incoming dependencies of the reactive. The `initv` parameter is the reactive's initial state and the `inite` parameter is a boolean that specifies whether the reactive has to be reevaluated after the initialization. Reevaluation is discussed in detail in Section 3.4.

The second parameter list has a single parameter, the function `instantiateReactive`, which has to instantiate the reactive from a `State` object.

Listing 9 shows how we use the `CreationTicket.create()` method to initialize our reactor.

The `createReactor()` method shown in Listing 9 can create a reactor from an initial value and an initial stage. The `initialValue` is the value that is used before any value

is set from within the reactor body and the `initialStage` specifies the behavior of the reactor.

In line 5 the method retrieves a `CreationTicket` from the REScala scheduler. This ticket is then used in line 6 to create the reactor, using the `CreationTicket.create()` method described in Listing 8. In this `create()` method we specify that we have no known incoming dependencies by passing the empty `Set` in line 7. In line 8, we pass the initial state of the reactor in form of a `ReactorState` object, as explained in Listing 7. Line 9 specifies that the reactor needs to be reevaluated directly after the initialization. This is needed to directly execute the reactor's body. Reevaluation is discussed in detail in Section 3.4. For the `instantiateReactive` function we only call the constructor of the `Reactor` class with the `createdState` variable.

The `Reactor.once()` factory method now only calls this method to create the reactor:

```
1  def once[T](
2    initialValue: T,
3  )(initialStage: Stage[T]): Reactor[T] = {
4    createReactor(initialValue, initialStage)
5  }
```

Listing 10: Reactor.once() factory method

The `Reactor.loop()` factory method does the same with an added step:

```
1  def loop[T](
2    initialValue: T,
3  )(initialStage: Stage[T]): Reactor[T] = {
4    val loopingStage = initialStage.copy(List(
5      ReactorAction.LoopAction(initialStage, initialStage)
6    ))
7    createReactor(initialValue, initialStage)
8  }
```

Listing 11: Reactor.loop() factory method

As shown in Listing 11, the initial stage gets modified by wrapping it with another stage that only has the *loop* action. Therefore, using `Reactor.loop()` internally is the same as using `Reactor.once()` and wrapping the body with a *loop* action.

3.4 Reevaluation

3.4.1 REScala Internals

In REScala reactives are updated through reevaluation. Every reactive needs to implement the `reevaluate()` method:

```
1 protected[rescala] def reevaluate(input: ReIn): Rout
```

Every time a reactive is changed from outside the reactive graph, an update turn is started by the scheduler. During this update turn, the scheduler is guaranteed to call the `reevaluate()` method of a reactive, if its dependencies have changed. Although it is guaranteed that the `reevaluate()` method is called at least once after all known dependencies are updated, the scheduler is allowed to call it multiple times. Therefore, the reevaluate method has to be side-effect free.

The `reevaluate()` method shown above has an `input` parameter of type `ReIn` and returns an object of type `Rout`. These types are type aliases for `ReevTicket[Value]` (reevaluation ticket) and `Result[Value]`.

```
1 final type ReIn = ReevTicket[Value]
2 final type Rout = Result[Value]
```

Listing 12: Type aliases ReIn and Rout

Similar to the `CreationTicket` used during Section 3.3 Initialization, the `ReevTicket` provides an interface to interact with the scheduler. For our implementation of reactors, the methods shown in Listing 13 are especially relevant.

In this listing, type `V` is the type of the reactive's state. The method `trackDependencies()` of the `ReevTicket` (line 3) allows it to track dynamic dependencies. For this method, the `initial` parameter can be used to specify already known dependencies. If it is not called, only previously known dependencies can be accessed from within the ticket. These static dependencies can, for example, be specified during the initialization (see Section 3.3).

The variable `before` (line 4) contains the state of the reactive before the update turn started, while the method `withValue` (line 5) can be used to create a `ReevTicket` with a new state, which is passed as the `v` parameter.

```
1  abstract class ReevTicket[V](...)
2       extends DynamicTicket(...) with Result[V] {
3    final def trackDependencies(initial: Set[ReSource]): ReevTicket[V]
4    final def before: V
5    final def withValue(v: V): ReevTicket[V]
6    ...
7  }
8
9  abstract class DynamicTicket(...) extends StaticTicket(...) {
10   final def depend[A](reactive: Interp[A]): A
11   ...
12 }
```

Listing 13: ReevTicket and DynamicTicket classes

Because `ReevTicket` extends `DynamicTicket`, it inherits the `depend()` method of `DynamicTicket`. This `depend()` method (line 10) can be used to signal a dependency and access the value of it. Therefore, a reactive to depend on has to be passed to the method. The method then returns the current value of the reactive. Notably, the returned value is the value that the reactive has during the update turn, not the value that it had before.

Notably, because `ReevTicket` has the trait `Result`, it can also be returned as result value `Rout`.

3.4.2 Reevaluation of reactors

With this internal knowledge of REScala, the following part describes the implementation of the `reevaluate()` method in detail.

First, in line 2, we call `trackDependencies()` with an empty set to specify that we do not know any static dependencies, but also track dynamic dependencies. Although reactors do not have static dependencies, reactors can depend on user-specified events that also change from stage to stage through the `next` and `until` actions.

Line 3 initializes a `progressedNextAction` variable, which is set to `true` when the reactor progresses a stage because a `next` action is triggered. It is used to implement waiting between consecutive `next` actions by the `processActions` method, whose definition is hinted at by line 5.

```
1  override protected[rescala] def reevaluate(input: ReIn): Rout = {
2    input.trackDependencies(Set())
3    var progressedNextAction = false
4
5    def processActions[A](...) = ...
6
7    val resState = processActions(input.before)
8
9    input.withValue(resState)
10 }
```

Listing 14: reevaluate() method of the Reactor object

The `processActions()` method is a recursive method that iterates through the actions of the current stage, changes the reactor's state accordingly, and returns the resulting new state of the reactor. It is a key part of the reevaluation process and called in line 7.

Finally, `input.withValue(resState)` is used to create a `Result`, which has the resulting state `resState` as value. The REScala scheduler uses this to update the `state` of the reactor at the end of the update turn.

The processActions method

The `processActions` method is defined in the scope of `reevaluate` and therefore has access to the `input` and `progressedNextAction` variables (see Listing 14).

It recursively iterates through the list of actions of the current stage and returns a new `ReactorState` object that is modified according to the changes made by the action.

The `processActions()` method shown in Listing 15 consists of action-specific method definitions and a `match` statement. The `match` statement calls the appropriate method, depending on the type of the first action in the list of actions of the current stage. If the action can be chained, the corresponding method also does a recursive call to `processActions()` with the remaining actions. The action-specific methods are explained in detail in Section 3.5.

```scala
def processActions[A](
  currentState: ReactorState[T]
): ReactorState[T] = {
  def setAction(...): ReactorState[T] = ...
  def nextAction(...): ReactorState[T] = ...
  ...

  currentState.currentStage.actions match {
    case Nil => currentState
    case ReactorAction.SetAction(v) :: tail =>
      setAction(v, tail)
    case ReactorAction.ModifyAction(modifier) :: tail =>
      modifyAction(modifier, currentState.currentValue, tail)
    case ReactorAction.NextAction(event, handler) :: _ =>
      nextAction(event, handler)
    ...
  }
}
```

Listing 15: processActions() method

3.5 Reactor Actions

This section explains the implementation of reactor actions in general and the specifics of implementing our six reactor actions: *set, modify, next, read, loop, and until*.

Because the whole state has to encoded in the `ReactorState` object, the object must be able to track the progress of the reactor through its actions. As described in Section 3.2, the state mainly consists of the reactor's current value and its current stage. In our implementation, the actions are saved with the stage. Therefore, the `Stage` class has the member `actions`, which is a list of reactor actions (see Listing 6). Since actions are only executed once, there is no need to store executed actions. Consequently, we can track the progress of actions by removing executed actions from the `Stage.actions` list.

In this list the reactor actions are represented by objects. There is one case class per action, holding the relevant data to exercise the action. These actions all share the trait `ReactorAction`:

```scala
sealed trait ReactorAction[T]
```

```
1  // Reactor.scala
2  case class Stage[T](actions: List[ReactorAction[T]] == Nil) {
3    private def addAction(newValue: ReactorAction[T]): Stage[T] = {
4      copy(actions = actions :+ newValue)
5    }
6
7    def set(...): Stage[T] = {
8      addAction(ReactorAction.SetAction(...))
9    }
10
11   def modify(...): Stage[T] = ...
12   ...
13 }
14
15 // Demo.scala
16 val reactor = Reactor.once(0) {
17   S.set(42)
18 }
```

Listing 16: Specifying reactor actions through the user-facing API

When a user adds an action to a stage, the method creates a new instance of the appropriate reactor action class and adds it to the `actions` list of the stage.

The example in Listing 16 shows how the `set()` method, which is used to add the *set* action to the reactor stage, adds a `SetAction` object to the `actions` list of the stage. The `set` method in line 7 only consists of a call to `addAction()`, which is defined in line 3 and creates and returns a copy of the current stage with the provided action appended to the actions list.

This list is then processed during reevaluation, as described in Section 3.4.

3.5.1 Action specific implementations

Set action

The *set* action is used to set the value of the reactor to a new value. Therefore, the `Stage.set()` method has the new value as a parameter. The `SetAction` object, which is appended to the actions list of the `Stage` object, stores this value as a class member.

```
1  case class SetAction[T](res: T) extends ReactorAction[T]
2
3  def set(newValue: T): Stage[T] = {
4    addAction(ReactorAction.SetAction(newValue))
5  }
```

Listing 17: Implementation of the *set* action

```
1  def setAction(
2    value: T,
3    remainingActions: List[ReactorAction[T]]
4  ): ReactorState[T] = {
5    processActions(currentState.copy(
6      currentValue = value,
7      currentStage = Stage(remainingActions)))
8  }
```

Listing 18: *Set* action reevaluation method

Listing 17 shows the class definition of `SetAction` in line 1. Notably, it only has one member variable, the new value. Lines 3–5 show the definition of the `Stage.set()` method. It uses the new value specified in the reactor body and adds a new `SetAction` object with this value to the actions list of the stage.

The `setAction()` method in `reevaluate()` (see Chapter 3.4), that is called by `processActions` (see Listing 15), has two parameters: `value`, the new value of the reactor, and `remainingActions`, the list of actions without the set action itself.

The resulting state is generated and returned by copying the current state in line 5, replacing the `currentValue` variable with the new value specified by the *set* action in line 6, and replacing `currentStage` with a new stage, that only has the actions that remain after the *set* action in line 7.

Modify action

The *modify* action can change the value of the reactor in dependence of the current value. Therefore, `Stage.modify()` accepts a function that takes the current value as input and returns a new value.

```
1  case class ModifyAction[T](modifier: T => T) extends ReactorAction[T]
```

```
def modifyAction(
  modifier: T => T,
  currentValue: T,
  tail: List[ReactorAction[T]]
): ReactorState[T] = {
  val modifiedValue = modifier(currentValue)
  setAction(modifiedValue, tail)
}
```

Listing 19: *Modify* action reevaluation method

The implementation of the *modify* action is very similar to the implementation of the *set* action. As the class declaration above shows, the `ModifyAction` class only stores the `modifier` function instead of the resulting value.

The reevaluation method `modifyAction()` shown in Listing 19 has three parameters. The function, which is used to get the new value `modifier`, the current value of the reactor `currentValue` and the list of actions that remain without the *modify* action `tail`.

When the `modifyAction()` method is called, it first calls the `modifier` function with the current value of the reactor in line 6 and saves its result. The resulting value is then passed on to the `setAction()` method in line 7, to change the value of the reactor to the new, modified value.

Read action

The *read* action can be used to change the reactor stage depending on the current value. It is therefore passed a function that takes the current value as input and returns a new stage. Again, this function is stored in the `ReadAction` object, as shown in Listing 20.

```
case class ReadAction[T](stageBuilder: T => Stage[T])
    extends ReactorAction[T]
```

Listing 20: Implementation of the *read* action

The reevaluation method `readAction()` also shares the same structure as the reevaluation method `modifyAction()` (see Listing 19).

```
1  def readAction(
2    builder: T => Stage[T],
3    currentValue: T
4  ): ReactorState[T] = {
5    val nextStage = builder(currentValue)
6    processActions(currentState.copy(currentStage = nextStage))
7  }
```

Listing 21: *Read* action reevaluation method

```
1  case class NextAction[T, E](
2    event: Event[E],
3    handler: E => Stage[T]
4  ) extends ReactorAction[T]
```

Listing 22: Implementation of the *next* action

The reevaluation method `readAction()`, as shown in Listing 21, has two parameters: The function that builds the new stage `builder` and the current value of the reactor. Notably, it does not have a `tail` parameter like the *set* and *modify* actions. That is because it results in a new stage and therefore does not execute actions that follow the *read* action in its current stage.

When it is called, it first calls the `builder` function with the current value as parameter in line 5. It then copies the current state and replaces its stage with the stage returned by the builder. This new state is then recursively passed to `processActions` to continue processing the new stage in line 6.

Next action

The *next* action can be used to pause reactor execution until an event is fired. When the specified event is fired, the reactor continues with a new stage. Therefore, the `Stage.next()` method has two parameters: The event to wait for and a function that takes the event value as input and returns a new stage. This is reflected in the `NextAction` class shown in Listing 22.

The class has two member variables: The event that the action is waiting for and a function that creates the new stage when the event is fired, called `handler`.

```
1  def nextAction[E](
2    event: Event[E],
3    handler: E => Stage[T]
4  ): ReactorState[T] = {
5    if (progressedNextAction) {
6      return currentState
7    }
8
9    val eventValue = input.depend(event)
10   eventValue match {
11     case None => currentState
12     case Some(value) =>
13       progressedNextAction = true
14       val stages = handler(value)
15       processActions(currentState.copy(currentStage = stages))
16   }
17 }
```

Listing 23: *Next* action reevaluation method

Listing 23 shows the reevaluation method for the *next* action. As hinted at in Section 3.4.2, because the `nextAction()` method is defined in the scope of the `reevaluate()` method, it has access to the `input` variable of reevaluate.

The `nextAction()` method has two parameters: The event that the action is waiting for and the function that builds the new stage `handler`. When called, the method first checks whether the reactor already progressed a stage during the current update turn by checking the `progressedNextAction` variable in line 5. If the reactor already progressed a stage, the current state is returned. Without this check, the reactor would progress through multiple `next` stages that listen for the same event, even though the event is only fired once.

If this is not the case, the method retrieves the current value of the event in line 9. Even though the value of events is always `None` during normal use, when an event is fired, it has the value that it is fired with during the update turn. Therefore, we can check whether the event is fired during the current update turn by checking if the event has a value other than `None`. If the value of the event is `None`, the `nextAction` returns the current state and does nothing, as shown in line 11.

If the event has a value and is thereby firing in the current update turn, the method will first set the `progressedNextAction` variable to `true` in line 13. It then calls the stage builder function passed by the user with the value that the event is fired with in line 14.

```
case class LoopAction[T](
  currentStage: Stage[T],
  initialStage: Stage[T]
) extends ReactorAction[T]

// Stage class
def loop(body: => Stage[T]): Stage[T] = {
  addAction(ReactorAction.LoopAction(body, body))
}
```

Listing 24: Implementation of the *loop* action

In the end, the method copies the current state, replaces the current stage with the new stage returned by the handler and calls the `processActions()` method recursively with this resulting state, as shown in line 15.

Loop action

The *loop* action repeats the actions specified in its body in a loop. Therefore, the `Stage.loop()` method is passed a new reactor stage. Internally, this is stored in a `LoopAction` object.

The `LoopAction` class has two members: The `currentStage` and the `initialStage`. It is therefore very similar to the reactor itself and capable of its own state management (see Section 3.2). The looping is then implemented by never removing the stage of the loop action and the loop action itself from the reactor state. When the loop action is evaluated, it manages the execution of the appropriate stage and action and when all stages and actions inside the loop are executed, the loop stage can restore the initial state of the loop stage from the `initialStage` variable and start over.

The reevaluation method of the *loop* action shown in Listing 25 has the current internal state of the loop and its initial state as parameters. First, the method starts execution of the loop's inner stage by calling `processActions` with a copy of the current state, where the stage is swapped with the internal stage of the loop, in line 5. Notably, this returns the resulting state, but the resulting state is only stored in a variable `resultState` and not applied to the reactor.

Next, the method checks if the resulting state has any actions remaining in line 7. If it does not have remaining actions, it starts another loop iteration by resetting the resulting

```
def loopAction(
  loopStage: Stage[T],
  initialStage: Stage[T]
): ReactorState[T] = {
  val resultState = processActions(
    currentState.copy(currentStage = loopStage))
  if (resultState.currentStage.actions.isEmpty) {
    return processActions(
      resultState.copy(currentStage = initialStage))
  }

  resultState.copy(currentStage =
    Stage(List(ReactorAction.LoopAction(
      resultState.currentStage, initialStage))))
}
```

Listing 25: *Loop* action reevaluation method

```
case class UntilAction[T, E](
  event: Event[E],
  body: Stage[T],
  interrupt: E => Stage[T]
) extends ReactorAction[T]
```

Listing 26: Implementation of the *until* action

state to the loop's initial stage and calling `processActions()` on it in line 8.

Otherwise, if the resulting state has remaining actions, in line 12 the method returns the resulting state wrapped into a stage which only contains the reactor action. This ensures that the reactor not only continues execution at the right action, but also that the stage will loop when the remaining actions are processed.

Until action

The *until* action executes one stage until an event is fired. When the event is fired, it continues execution with another stage.

Listing 26 shows the class that represents the *until* action. It has three members: The `event`, which triggers the switch between the two stages, the stage `body`, that is executed

```
1  def untilAction[E](
2    event: Event[E],
3    body: Stage[T],
4    interrupt: E => Stage[T]
5  ): ReactorState[T] = {
6    val eventValue = input.depend(event)
7    eventValue match {
8      case None =>
9        val resultState = processActions(
10         currentState.copy(currentState = body)
11       )
12       resultState.copy(
13         currentStage = Stage(List(
14           ReactorAction.UntilAction(
15             event,
16             resultStage.currentStage,
17             interrupt
18       ))))
19     case Some(value) =>
20       val stages = interrupt(value)
21       processActions(currentState.copy(currentStage = stages))
22   }
23 }
```

Listing 27: *Until* action reevaluation method

first, and a function `interrupt`, building the stage from the event value to continue execution after the event is fired.

The implementation of the reevaluation method for the *until* action combines the techniques from *next* and *loop*. As shown in line 6 of Listing 27, it first retrieves the current value of the event.

If the event is not firing in the current update turn, the event value is `None` and execution continues in line 8. In this case, execution is continued by calling `processAction()` on the body in line 9. Similar to the implementation of the *loop* action (see Listing 25), the resulting state is wrapped into a stage that only contains an `UntilAction` to hand over state management to the *until* action.

Otherwise, if the event is firing during the current update turn, the event has a value and therefore execution continues in line 19. In this case the interrupt handler `interrupt()` is called with the value of the event to generate the new stage in line 20. Because the until action is now irrelevant, it can now be excluded from the reactor's state. Execution is

continued by calling `processActions()` on the current state with the new stage built by the interrupt handler in line 21.

3.6 Conclusion

Reactors are implemented as REScala reactives. Therefore, they need to follow the design constraints the REScala framework dictates for reactives. Particularly, the reactor's state has to be held in a single `state` variable, initialization of reactors has to be done through the REScala scheduler, and updates and changes in value must be made in a side-effect free reevaluation method.

We have described the way we encode the state of reactors and took an in-depth look into the implementation of the different reactor actions.

4 Performance Evaluation

To get a sense for the performance of the reactor implementation, we have benchmarked the different actions of the reactor. This chapter explains our test setup and presents the resulting performance metrics.

4.1 Test Setup

All measurements were taken on an AMD Ryzen 3600 system running Manjaro Linux 21.1.4 with kernel version 5.10.68–1–MANJARO. The project was built against Scala 2.13.6 and running on the OpenJDK 64-Bit Server VM version 16.0.2.

To address the non-determinism of JVM-based programs, that is introduced by the garbage collector and just-in-time compilation and optimization in the virtual machine, we used the Java Microbenchmark Harness (JMH) and ran every test on 20 JVM invocations with 3 warmup and 5 measurement iterations. Therefore, the resulting numbers show the steady state performance of our implementation [1].

4.2 Benchmarking Framework

To gain a general sense of performance, we wanted to evaluate each action on its own. Because reactor actions are not usable without reactors and, consequently, REScala, their stand-alone performance is not easily measurable. Therefore, we have developed a unified benchmarking framework for our reactor actions:

Every benchmark shares the structure shown in Listing 28. We define an event `trigger` and a looping reactor. The looping reactor uses *next* to wait for the `trigger` event to

```
1  var trigger = Evt[Unit]()
2  var reactor = Reactor.loop(0) {
3    S.next(trigger) {
4      // Action to Benchmark
5    }
6  }
7
8  @Benchmark
9  def run(): Unit = trigger.fire()
```

Listing 28: Action benchmarking framework

```
1  var trigger = Evt[Unit]()
2  var reactor = Reactor.loop(0) {
3    S.next(trigger) {
4      S.set(42)
5    }
6  }
```

Listing 29: *Set* action benchmark

fire. When the trigger event fires, the reactor continues with the action that is being benchmarked, loops, and waits for the next *next* action.

Because of the way transactions are implemented in the REScala framework, the call to `trigger.fire()` in line 9 returns only after the change is propagated and the transaction is complete.

We use this property to benchmark the action by measuring the time a call to `trigger.fire()` takes to return. Every iteration is 1 second long and during this time we count how often `trigger.fire()` can be called. Consequently, the results can not be compared with other benchmarks that do not use this framework. However, it allows for comparability between our internal measurements.

4.3 Benchmark Implementations

We have benchmarked the actions *next*, *until*, *set*, *read*, and *modify*. In these benchmarks *set*, *read*, and *modify* are implemented similarly, using the framework described in Section 4.2. Listing 29 shows how this looks like for the *set* action.

```
var trigger = Evt[Unit]()
var reactor = Reactor.loop(0) {
  S.next(trigger) {
    S.end
  }
}
```

Listing 30: *Next* action benchmark

```
var trigger = Evt[Unit]()
var reactor = Reactor.loop(0) {
  S.until(
    trigger,
    body = {
      S.end
    }: Stage[Int],
    interruptHandler = {
      S.end
    }: Stage[Int]
  )
}
```

Listing 31: *Until* action benchmark

The *next* and *until* actions are implemented differently, because they need to be triggered themselves. Therefore, we use the empty framework as the benchmark for the *next* action. Consequently, the measurements for the *next* action have to be viewed differently than those of the other actions. Listing 30 shows how this is implemented.

Similar to the *next* action, the *until* action itself must also be triggered. Therefore, we have also implemented such a baseline benchmark for the *until* action. Consequently, the results of the *until* and the *next* benchmarks can be compared to each other.

Listing 31 shows the implementation of the *until* benchmark. It waits for the trigger and does nothing, when the trigger is fired.

Benchmark	Count	Score	Error	Units
Baseline: Event propagation	100	4333,463	± 234,028	ops/ms
Baseline: Next	100	7399,575	± 389,012	ops/ms
Baseline: Until	100	6651,810	± 429,490	ops/ms
Modify	100	7057,409	± 424,780	ops/ms
Read	100	7180,084	± 426,778	ops/ms
Set	100	7231,222	± 453,137	ops/ms

Table 4.1: Benchmark results with 20 forks and 5 iterations

4.4 Results

Table 4.1 shows the results of our benchmarks. The table includes the sample size of our measurements (count), a score that shows how many times the `trigger` event could be fired per millisecond (score), the standard error (error), and the unit of measurements (units).

Generally, the standard error is relatively large. This is due to large performance differences caused by non-deterministic optimizations between JVM invocations. This can be verified by looking at the results in Table 4.2, which shows measurements taken with only one fork per benchmark. For these measurements, the standard error is significantly smaller, but the resulting averages are less comparable because the resulting values only represent the specific JVM invocation and are not representative for general invocations.

As expected, the score of all actions is in the same magnitude, but the (baseline) *next* action is a little bit faster than the other actions that are a combination of the *next* action and the corresponding other action. The *read* and *set* methods perform mostly similarly while the *modify* action is a little bit slower. When comparing the *next* action with the *until* action, the former is a little bit faster. One reason could be, that during execution of the *until* action more data has to be copied than during execution of the *next* action.

Overall, the benchmarks, resulting in thousands of operations per millisecond across all actions, show that our reactor implementation can be used efficiently and without major slowdowns.

Benchmark	Count	Score	Error	Units
Baseline: Event propagation	5	4927,544 ±	89,035	ops/ms
Baseline: Next	5	5244,224 ±	8,304	ops/ms
Baseline: Until	5	6298,195 ±	118,029	ops/ms
Modify	5	6247,942 ±	56,632	ops/ms
Read	5	7798,267 ±	302,851	ops/ms
Set	5	8566,005 ±	221,415	ops/ms

Table 4.2: Benchmark results with 1 forks and 5 iterations

4.5 Conclusion

In this chapter, we have benchmarked the different reactor actions using JMH. We have introduced a reactor benchmarking framework to allow for comparable performance measurements. Using this framework, we have compared the performance of the actions relative to each other and showed that reactors offer similar performance to other REScala operations and therefore are suitable to use, when REScala in general is applicable.

5 Conclusion and Future Work

Writing event handling logic in an imperative way and using the observer pattern is hard and error prone. Several studies have shown that using the reactive programming paradigm for event-driven applications can vastly improve code comprehensibility and reduce code complexity [9, 3].

While reactive frameworks make it simple to specify dataflow and write event handling logic for simple events, detecting and responding to complex events can still be hard. Scala.React partially solves this with their reactors [4]. In this thesis, we have introduced reactors to the REScala language, improved their usefulness by giving them a value, thereby integrating them into the reactive graph, and extended their functionality with new operations (actions).

Finally, we have benchmarked the performance of our reactor implementation and shown that they can be used without introducing substantial performance overhead.

In the future, detection and handling of time-dependent events using reactors could be examined. Also, traditionally complex event processing is often associated with stream processing and detecting patterns and events in stream data. Reactors need to specifically know the beginning of an event and can not do sliding-window-like scans on data streams. Additionally, a case study implementing a real-world application using reactors could be interesting.

Bibliography

[1] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically Rigorous Java Performance Evaluation”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA ’07. New York, NY, USA: Association for Computing Machinery, Oct. 21, 2007, pp. 57–76. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297033. URL: https://doi.org/10.1145/1297027.1297033 (visited on 10/06/2021).

[2] Ulrich Hedtstück. “Einführung mit typischen Anwendungen”. In: *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*. Ed. by Ulrich Hedtstück. Berlin, Heidelberg: Springer, 2020, pp. 1–15. ISBN: 978-3-662-61576-8. DOI: 10.1007/978-3-662-61576-8_1. URL: https://doi.org/10.1007/978-3-662-61576-8_1 (visited on 07/30/2021).

[3] Gustaf Holst and Alexander Gillberg. *The Impact of Reactive Programming on Code Complexity and Readability: A Case Study*. 2020. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:miun:diva-39510 (visited on 10/08/2021).

[4] Ingo Maier and Martin Odersky. *Deprecating the Observer Pattern with Scala.React*. Infoscience. 2012. URL: https://infoscience.epfl.ch/record/176887 (visited on 07/31/2021).

[5] Leo A. Meyerovich et al. “Flapjax: A Programming Language for Ajax Applications”. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. New York, NY, USA: Association for Computing Machinery, Oct. 25, 2009, pp. 1–20. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640091. URL: https://doi.org/10.1145/1640089.1640091 (visited on 07/27/2021).

[6] Ragnar Mogk. “A Programming Paradigm for Reliable Applications in a Decentralized Setting”. In: (), p. 220.

[7] Sean Parent. “A Possible Future of Software Development”. 2006. URL: https://stlab.cc/legacy/papers-and-presentations.html (visited on 04/10/2021).

[8] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. “REScala: Bridging between Object-Oriented and Functional Style in Reactive Applications”. In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY ’14. New York, NY, USA: Association for Computing Machinery, Apr. 22, 2014, pp. 25–36. ISBN: 978-1-4503-2772-5. DOI: 10.1145/2577080.2577083. URL: https://doi.org/10.1145/2577080.2577083 (visited on 07/06/2021).

[9] Guido Salvaneschi et al. “On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study”. In: *IEEE Transactions on Software Engineering* 43.12 (Dec. 2017), pp. 1125–1143. ISSN: 1939-3520. DOI: 10.1109/TSE.2017.2655524.