# Distributed REScala: An Update Algorithm for Distributed Reactive Programming

Joscha Drechsler, Guido Salvaneschi
Technische Universität Darmstadt, Germany
< $lastname$ >@cs.tu-darmstadt.de

Ragnar Mogk
Technische Universität Darmstadt, Germany
ragnar.mogk@stud.tu-darmstadt.de

Mira Mezini
Technische Universität Darmstadt, Germany; Lancaster University, UK
mezini@cs.tu-darmstadt.de

## Abstract

Reactive programming improves the design of reactive applications by relocating the logic for managing dependencies between dependent values away from the application logic to the language implementation. Many distributed applications are reactive. Yet, existing change propagation algorithms are not suitable in a distributed setting.

We propose Distributed REScala, a reactive language with a change propagation algorithm that works without centralized knowledge about the topology of the dependency structure among reactive values and avoids unnecessary propagation of changes, while retaining safety guarantees (*glitch freedom*). Distributed REScala enables distributed reactive programming, bringing the benefits of reactive programming to distributed applications. We demonstrate the enabled design improvements by a case study. We also empirically evaluate the performance of our algorithm in comparison to other algorithms in a simulated distributed setting.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***Keywords*** Reactive Programming, Scala, Distributed Programming

## 1. Introduction

Reactive applications actively update their state based on incomplete input that keeps growing over time. Applications with a user interface that continuously adapt their state to user's inputs or applications with a network interface that continuously process incoming network packets fall into this category. Historically, reactivity has been achieved via callbacks and inversion of control [14], commonly implemented using the observer pattern to facilitate modular composition. While successful in decoupling and thus making components reusable, the pattern has several major downsides [22]. It requires a lot of boilerplate code; callback interfaces and registries are re-implemented over and over; client code is bloated with declarations that wrap business logic into callback instances and bloated further by surrounding those with statements to register and unregister with these registries. Another issue is that notifications from multiple callbacks often arrive in unpredictable order. This makes it hard to avoid inconsistencies in accordingly updating local state, leading to bugs and bad user experiences.

Reactive Programming [3] (RP for short) simplifies the modular implementation and improves code quality of reactive applications. Languages in this class provide *reactive values*, abstractions for values that change over time. These values can be composed and derived in a declarative way. Their dependencies are tracked: Changes to any value automatically cause the recalculation of all derived values. Code quality improves since explicit encoding of the observer pattern and callback registries and maintenance of their state is not needed. Further, dependencies between values are assessed as a whole when deciding on update orders, so that inconsistencies due to wrong ordering of state updates do not occur, regardless of module borders.

The dominating category of software today is distributed applications. This family includes several types of reactive applications, including Web applications, monitoring systems, customer analytics, etc. In such applications, reactions to updates in state and events often have to happen over multiple hosts. Remote callbacks in the form of remote observer patterns or callbacks over publish-subscribe systems are typically used to implement push notifications in such applications. They therefore suffer from the same drawbacks that callbacks and the observer pattern cause in the local setting. Distributed applications can clearly benefit from RP. How-


*OOPSLA '14*, October 19 - 21 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660240

ever, existing implementations of RP either target specific kinds of distributed applications (e.g., only client-side [22]), do not provide safe value propagation [19], or adopt a synchronization and communication schema that is not acceptable for interactions between remote hosts [3].

We propose Distributed REScala, which implements SID-UP (Source IDentifier Update Propagation), an algorithm for propagating changes in a network of dependent reactive values that is suitable in a distributed setting. It renounces properties that are undesirable in a distributed setting, such as global centralized knowledge about the topology of the dependency structure among reactive values and unnecessary communication and synchronization between changes in completely independent parts of this structure, while retaining safety guarantees (*glitch freedom* [22]). To the best of our knowledge, such a solution has not been proposed before. The proposed algorithm thus enables distributed reactive programming (DRP for short), a powerful paradigm to design distributed applications.

In summary, we makes the following contributions:

- We characterize the design space of existing algorithms for change propagation in reactive programming, motivating the need for new algorithms that better suit the requirements of distributed applications.
- We present SID-UP, an algorithm for reactive programming in distributed applications, thus enabling DRP.
- We analyze and compare the complexity of different change propagation algorithms, including SID-UP.
- We discuss a small-scale case study to indicate design improvements enabled by DRP and its performance cost compared to designs based on distributed observer infrastructures.
- We empirically evaluate the efficiency of update algorithms in a distributed setting, and show that SID-UP outperforms existing algorithms.

While the abstract idea of DRP was presented in a vision paper [23], the SID-UP algorithm, the comprehensive discussion of the problems with the state of the art, and the evaluation, are new contributions of this paper. The implementation of SID-UP in a prototypical reactive language, the case study, and all evaluation artifacts are available online[1].

## 2. Background and Motivation

In this section, we introduce the case study used throughout the paper for illustration and evaluation purposes. We introduce key concepts of RP and motivate our work.

Our *case study* is ProfitReact, a software system that supports a manufacturing company. It consists of four modules. Clients place orders on an incoming server. The purchases

[1] http://www.stg.tu-darmstadt.de/research/

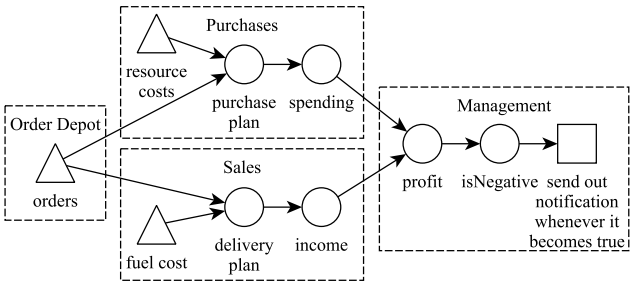


**Figure 1.** Reactive network graph of the case study.

department has a module that calculates a plan for acquiring the resources needed to produce the ordered goods. The sales department equivalently maintains a plan for delivering the produced goods. Both plans are updated as the order list changes. Finally, a management module combines the projected spending and the projected income, derived from the purchase and delivery plans respectively, into the projected profit. It defines an invariant that this profit must never be negative: Whenever this is violated, a notification is sent out to a responsible manager. To keep individual department's operations independent the four modules should run on separate machines, thus making the application distributed.

### 2.1 Reactive Architectures

A reactive architecture is well-suited for ProfitReact: There is a small number of inputs and a lot of derived state that has to be updated whenever some of the inputs change. In the following, we briefly introduce the key concepts of a reactive architecture and illustrate them by the case study.

Values in a reactive architecture are organized in a dependency graph (DG): Nodes therein represent reactive values and are connected via dependency relations. Figure 1 shows the graph of the case study (without UI components). Dashed boxes represent individual hosts. The set of nodes in the DG is denoted by $N$. Some nodes can be modified imperatively through user code. In the example, these are the list of orders, the fuel and the resource costs. We denote the set input nodes as $I \subseteq N$ and visualize them as triangles. Most nodes' values are the result of a user-defined computation using values from other nodes as input, i.e. they *depend* on other nodes. The formula that calculates the estimated income from the delivery plan is an example – it is associated with the node "income" that depends on the state of the "delivery plan" value. We refer to the set of dependent nodes as $D \subseteq N$ with $D \cap I = \varnothing$ and visualize them as circles.

We denote incoming dependencies of a node $d \in D$ as $\overrightarrow{dep}_d$ (arrow points towards node's name). In application code, these correspond to input values to the computation of $d$. In a reactive framework, for any $n \in N$, outgoing dependencies $\overleftarrow{dep}_n$ (arrow points away from node's name) are maintained automatically. Outgoing dependencies point

in the direction in which information flows and are used to push updates through the reactive network.

Derived values in the reactive network *react* to changes in their input values. When the system returns to a resting state, every derived value is up to date with respect to all its computation's input values. Every change to the reactive network happens in a so-called *update turn*. An update turn begins with the *admission phase*, where some input nodes $C \subseteq I$ are imperatively changed by client code. Subsequently, the *propagation phase* starts, during which all dependent nodes are updated to reflect the changes of the input values. Only nodes reachable along a path of outgoing dependencies from any changed source can change during a turn; if a node is not on such a path, no input value and thus also not its output value can change. We refer to the set of all nodes in this transitively reachable closure as $CC$ with $C \subseteq CC \subseteq N$ and $\forall n \in CC, \forall d \in \overleftarrow{dep_n} : d \in CC$. An update turn ends after all values are up to date again, returning the system to a resting state waiting for further inputs.

Every $n \in N$ holds a *steady value* $v_n$ at all times and a *pulse value* $p_n$ that only exists when an update turn that affected $n$ has not ended yet. Setting $p_n$ results in notifying all outgoing dependencies of $n$ ("pulsing"). A dependent node that receives such a notification must be *reevaluated* as one of its input values changed. This can result in the dependent node itself sending out a pulse, which causes updates to ripple through $CC$ during the propagation phase.

An important safety property that reactive networks need to provide is *glitch freedom*. A *glitch* can be defined as any computation on any node $d \in D$ triggered during an update turn while $d$'s input values are in an inconsistent state, i.e. some of them have been updated while others still will be updated later within the turn at hand. For an example of a glitch, consider the following scenario from our case study. A client places a big order, an input change propagated to purchases and sales modules. If the projected spending is updated first and the profit is recalculated without waiting for the updated income projection, a temporary, erroneous, negative value may be produced, triggering a false alarm.

### 2.2 Reactive Programming

A reactive architecture can be implemented in different ways. To implement ProfitReact in an object-oriented language one would probably encode explicit updates to regular variables through the observer pattern [14]. Reactive programming languages [3], on the other hand, have built-in support for reactive architectures. They provide abstractions for reactive values – nodes in the reactive network – used to model reactive applications declaratively; the updates of reactive values are managed automatically and in a glitch-free way. To ensure glitch freedom, a RP language or library employs a *propagation algorithm* that ensures that every node is updated only after all of its incoming dependencies that *will* change during the current update turn *have been* changed.

```
1 val projectedSpending : Signal[Int] =
    SignalRegistry.lookUp("purchases—projectedSpending")
2 val projectedIncome   : Signal[Int] =
    SignalRegistry.lookUp("sales—projectedIncome")
3 val projectedProfit   : Signal[Int] =
    subtract(projectedIncome, projectedSpending)
4 val profitIsNegative  : Signal[Boolean] =
    projectedProfit.map { _ < 0 }
5 profitIsNegative.observe{ isNegative: Boolean =>
6    if (isNegative) {
7       sendNotificationToManager()
8    }
9 }
```

**Figure 2.** Reactive programming code snippet.

```
1  var lastSpending: Int = 0
2  val spendingObserver = RemoteObserver[Int] { v: Int =>
3    lastSpending = v
4    recalculate()
5  }
6  val purchDept = RemoteObjectRegistry.lookUp("purchasesDept")
7  purchDept.addProjSpendingObserver(spendingObserver)
8
9  var lastIncome: Int = 0
10 val incomeObserver = RemoteObserver[Int] { v: Int =>
11   lastIncome = v
12   recalculate()
13 }
14 val salesDept = RemoteObjectRegistry.lookUp("salesDept")
15 salesDept.addProjIncomeObserver(incomeObserver)
16
17 var difference: Int = 0
18 var wasNegative: Boolean = false
19 def recalculate() = {
20   profit = lastIncome − lastSpending
21   val nowNegative = difference < 0
22   if(!wasNegative && nowNegative) {
23         sendNotificationToManager()
24   }
25   wasNegative = nowNegative
26 }
```

**Figure 3.** Code snippet using the observer pattern.

Figure 2 displays a code snippet implementing the reactive core functionality of the management module of our case study using our RP framework. The module first looks up the projected spending and income (line 1 and 2) in form of a remotely accessed *signal* (sometimes also called a *behavior*), the abstraction for sustained values that change over time. It composes them to calculate the projected profit (line 3). This profit signal is transformed to a boolean signal via a less-than-zero comparison (line 4), which is observed by user code (line 5) that notifies the manager (line 7) whenever the signal's value changes to `true` (line 6). As shown in Figure 2, in a DRP framework, developers should be able to share locally defined reactive values, such as the orders list, remotely, reflecting events occurring at the local reactive on other hosts. This could be done for instance by publishing them to a public registry, as is an established practice with remote objects. Remote hosts could look up these remote reactive values and should be able to compose them with other local or remote reactive values transparently.

Figure 3 implements the same functionality with the observer pattern, requiring far more boilerplate code. For instance, remotely connecting with, locally reflecting the state of, and reacting to changes of the projected spending is

spread over lines 1 to 7 in Figure 3. The same functionality is implemented in the single line 1 in Figure 2. With RP, the code is reduced to its relevant core. It is not polluted with the creation of observer instances or callback registry instantiations and interactions. Further, the implementation in Figure 3 is more error prone as it uses mutable variables to coordinate notifications (all **var** statements). In fact, the code is vulnerable to glitches due to the race condition between both observer notifications. Ensuring glitch freedom would require to break the modularity of the purchases and sales modules. Currently, their implementations do not distinguish between a notification caused by an update of the list of placed orders and one by an update of fuel or resource costs. In the former case, management has to wait for two notifications. In the second case, there won't be be a second notification. Exposing this information would require changes to these modules not related to their core responsibilities. RP instead provides glitch freedom out of the box.

Glitch freedom is as critical in the distributed as in a local setting. The four dependency edges of the diamond-shaped graph of the four departments responsible for the race condition that causes glitches in ProfitReact are routed over unrelated network connections. For distributed glitch freedom, edges over network connections thus have to be respected the same way as local edges. A single propagation algorithm with a holistic view must be used, as connecting individually glitch-free networks on each host by observer notifications would not result in an overall glitch free system.

There are several examples of distributed applications which require consistency guarantees that are lower than glitch freedom, for instance eventual consistency. As, though, there are applications, such as ProfitReact, that do require glitch freedom, we argue that an implementation for distributed reactive programming should be capable of providing this level of consistency. Exploring possible tradeoffs between performance and consistency, for example disabling glitch freedom in cases where the performance cost is too high, is left to future work.

### 2.3 Dynamic Dependencies

Dependency graphs of reactive applications may be dynamic in that nodes' incoming edges can change during update turns: New edges can be added and existing ones can disappear or be replaced. This supports important features such as conditionally accessed input values and higher-order reactives[2].

***Conditionally Accessed Input Values.*** The simplest example of dynamic dependencies comes from computations that access some of their inputs conditionally. An example is a signal `z` defined as `if c then x else y`, where `c`, `x`, and `y` are signals. At any point in time, the value of `z` depends on the current value of only two of its sub-signals: `c` and *only one of* `x` or `y`. Thus, the dependency on either `x` or `y` can be removed from the DG, as depicted in Figure 4: The dependency of `z` on `c` is static, but the incoming edges from `x` and `y` change dynamically, whenever `c` changes.

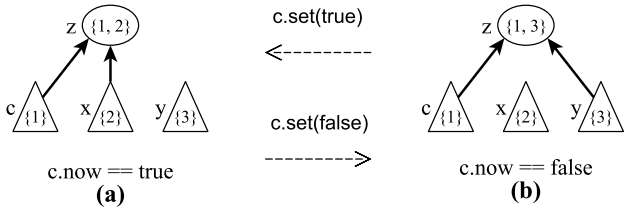


**Figure 4.** Dynamic dependencies caused by conditionals.

Propagating changes of `y` to `z` would cause unnecessary reevaluations of `z` as long as `c` is `true`. Conditionally accessed input values allow temporarily removal of not needed dependency edges at the cost of a topology change. This is a well-known technique to avoid unnecessary re-executions of potentially expensive computations [8, 22]. For instance, in the example, when `c` changes much less frequently than `x` and `y`, such a trade might prove valuable.

***Higher-Order Reactives*** (HOR's) are reactive values that refer to other reactive values. For illustration, consider Figure 5. It depicts the relation between the state of a GUI and of the underlying DG of a minimal reactive application example. The GUI displays a list of `Person` instances (marked 1), from which the user can select one (shaded gray), which is reflected in the node `selection` of type `Signal[Person]` (marked 3). On the left-hand side, the selection is `personA` (marked 2a), on the right-hand side `personC` (marked 2b). Each `Person` has a property `name` – a mutable string of type `Signal[String]`, modified from a different source each. The higher-order signal `selectedNameSignal` of type `Signal[Signal[String]]` (marked 4) is obtained by mapping the `selection` signal to its value's `name`. We refer to the higher-order signal as the *outer* reactive and the `name` signal it holds as its current value as the *inner* reactive.

The signal `selectedPersonName` is obtained by *flattening* the HOR `selectedNameSignal`. Flattening converts a signal of a signal of a value into a regular signal of a value, thereby hiding that the current value in fact depends on the current state of multiple nested reactive values. Nodes that perform flattening depend on the value of both the outer and inner signal of a HOR and thus entail dynamic incoming dependencies. `selectedPersonName` depends statically on the outer signal `selectedNameSignal`, but dynamically on the inner signal `name`. This inner dependency switches between `name` signals of `Persons`, whenever the outer signal's value changes as the user selects a different `Person`. In Figure 5 this is depicted by the bold dependency

[2] We do not address dynamic dependency discovery as employed by most other frameworks' `Signal{ ... }` [21, 24] or `Rx{ ... }` [26] syntax, but this exhibits the same phenomena in terms of DG changes and can thus be handled and supported identically.

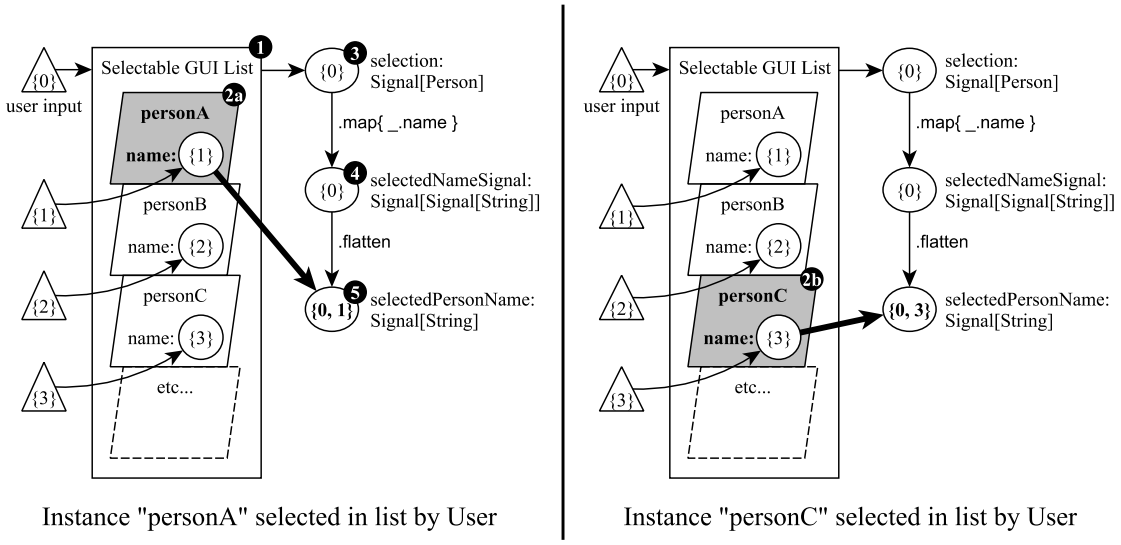


**Figure 5.** Dynamic incoming dependencies caused by changes to higher-order reactives.

edge switching between the `name` signal of `personA` on the left-hand side and that of `personC` on the right-hand side.

While desirable but optional in the local setting, we consider HORs a must-have in the distributed setting: Controlled and purposefully adding and removing edges allows one to attach and detach subsections at the "front" of the DG, i.e. inserting sections with new inputs, instead of only extending the graph by additional dependent nodes at the "back" end. In a distributed setting, where hosts with individual subgraphs can join or leave the application, e.g., a client with its own user inputs joining or leaving a multiplayer game, a safe way to connect these subgraphs into the application is required. HORs are the de-facto mechanism that solves this problem.

### 2.4 Change Propagation in a Distributed Setting

The discussion so far suggests that RP would be as useful for distributed applications as it is in a local setting. Hence, providing a framework for DRP is a desirable goal to achieve. Every propagation algorithm can be extended to work on distributed reactive networks. Yet, the distributed setting presents changed and additional requirements.

The first major difference is that remote method calls are a lot slower and more expensive than local calls. They are affected by network latency and all information required by a remote invocation must be marshalled back and forth, which requires additional computational resources. Second, global, unbounded one-to-many communication, where one host has to be able to reach all other hosts, is impossible or at least extremely difficult compared to many-to-one communication. The former requires to maintain a lot of address bindings and lists of participating hosts, whereas for the the latter only one single host has to be bound to a known reachable address. This is demonstrated by the popularity of client/server models and by client/server models being used even in almost all distributed end-user applications for dynamic host discovery to bootstrap and facilitate other communication topologies.

Next, we discuss shortcomings of existing propagation algorithms based on the requirements just outlined.

***Topological Sorting with Priority Queue.*** The most widely adopted glitch-free update propagation algorithm [8, 19, 21, 22] separates the DG into *layers*. All input nodes belong to layer 0 and all dependent nodes belong to the layer one above their "highest layer" incoming dependency. During the update turn, the layer number is used to populate a priority queue holding all nodes that must still be reevaluated. Initially, it contains all source nodes that were modified during the admission phase. Nodes are dequeued individually in order and reevaluated. If a node pulses, all nodes that depend on it are added to the queue. The turn completes once the queue is empty. To run this algorithm in a distributed setting, the priority queue need to be managed by a central coordinator, which would need remote control over all nodes in the reactive network on every host, yielding unbounded one-to-many communication. All node reevaluation would be executed via sequential individual synchronous remote invocations, crushing all potential for parallel computations with severe impact on performance.

***Parallel Propagator in Scala.Rx.*** In Scala.Rx [26], the propagation algorithm is wrapped into a `Propagator` interface, which models a strategy pattern that allows one to exchange the propagation algorithm. The default algorithm is the topological sorting with priority queue from above. But, Scala.Rx also offers a *parallel* propagator, which dequeues and reevaluates all nodes of the lowest layer in the

priority queue concurrently. As by construction, nodes on the same layer can not depend on each other and all nodes on the lowest layer are also ready for reevaluation without producing glitches. Converting this algorithm into a distributed version can be done in the same way as with the basic topological sorting algorithm. The major improvement is that the layer-wise concurrent execution of several of the previously synchronous remote calls reduces the length of the chain of calls. However, the algorithm still relies on unbounded one-to-many communication and still enforces a lot of unnecessary sequentiality: Nodes on different layers that do not have a path of dependencies between them should not need to wait for each other.

***Decentralized Flooding in ELM.*** In ELM [9], a central coordinator broadcasts every update turn's start to all input nodes. After the admission phase, all changed input nodes send out a "change" pulse and all unchanged input nodes send out a *no change* pulse. During the propagation phase nodes act purely based on received pulse notifications without any coordinating entity, updating as soon as they received a pulse from each incoming dependency for the current turn. Dependent nodes only update when at least one of their incoming dependencies sent a *changed* pulse; if they complete processing a turn where they would not pulse, they send out a *no change* pulse instead. Hence, every node, including completely unaffected ones, is involved in every update turn. Devising a distributed version would require the coordinator for the admission phase to be converted to a centralized entity. The coordinator is inactive during the propagation phase, meaning no unnecessary sequentiality is enforced. But, as the admission phase requires a broadcast to all sources from the coordinator, ELM also relies on unbounded one-to-many communication. Further, processing every update turn at every node implies a lot of wasted computational resources, as messages have to be processed in parts of the system that are completely unrelated to the changed input data.

Unlike other reactive systems, which propagate a single change at a time, ELM supports *pipelining* – multiple sequential turns can propagate through the dependency graph at the same time, in the form of sequential wave fronts. This feature is especially desirable in the distributed setting, where remote communication strongly increases update turn duration. Unfortunately, ELM's pipelining is inherently incompatible with dynamic dependencies in the topology graph. To allow different nodes to process different turns at the same time, every edge in ELM buffers every pulse message sent over it until its end node reads it while processing the respective turn. If new edges are created during an update turn, their start node may already be several turns ahead of their end node. In such a case, the pulse value required by the end node to complete its turn is no longer available at the start node and not stored in the buffer, because the edge did not exist when the value was sent out. Thus the end node is missing that value and gets stuck, unable to complete the turn; this blocks this turn and all successive turns from further progress, thus breaking the entire application. Without dynamic dependencies, ELM and its implementation of pipelining is unable to support higher-order reactives and hence not feasible for use in any distributed application with a potentially changing set of connected hosts.

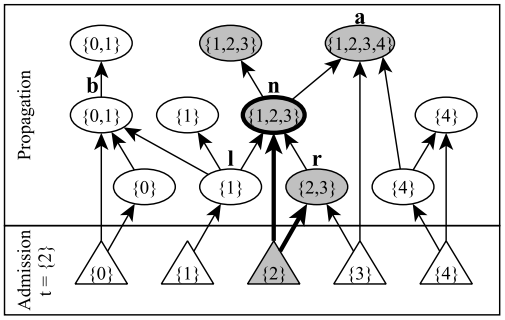


**Figure 6.** A reactive network with source identifier sets.

## 3. The SID-UP Propagation Algorithm

The design of SID-UP was driven by two goals: (a) Reduce communication outside of the regular pulse messages to a minimum, and (b) avoid unbounded one-to-many communication completely. The novelty of SID-UP consists of the combination of the following properties:

- Support for distributed reactive programming with remote reactives
- Glitch-freedom both in local and distributed settings
- No unbounded one-to-many communication
- No centralized coordinator during propagation phases
- Support for fully-fledged reactive programming, including dynamic dependency features – like higher-order reactives for dynamic network topologies
- Exploitation of concurrency potential (respecting glitch-freedom) for node re-evaluations
- Limited amount of remote communication – less than existing propagation algorithms

To achieve these properties, we implement all dependent nodes as individually acting threads with their actions governed solely by the pulses they receive. Each node stores information about the source nodes it is transitively connected to. Each pulse carries information about the sources changed during the admission phase of the update turn. Only nodes transitively reachable from changed sources and the edges between them are involved in transmitting and waiting for pulses. The remainder of this section describes in detail the inner workings of SID-UP.

### 3.1 The Basic Algorithm

***Topology of the Dependency Graph.*** Every source node $i \in I$ is tagged with a globally unique identifier $id_i$. Every node $n \in N$ in the DG holds a subset of identifiers $s_n \subseteq \{id_i : \forall i \in I\}$, representing sources from which $n$ is reachable along some path in the DG. Obviously, $\forall i \in I : s_i = \{id_i\}$. For a dependent node, $d \in D$, $s_d = \bigcup_{n \in \overrightarrow{dep}_d} s_n$; $s_d$ is initially calculated when $d$ is newly created and added to the DG. This provides in each turn $\forall n \in N, \forall i \in C : id_i \in s_n \leftrightarrow n \in CC$ (recall: $CC$ is the closure transitively reachable from changed sources $C$).

For illustration, consider the DG in Figure 6. The incoming dependencies of the node labelled $a$ are tagged with sets (from left to right) $\{1,2,3\}$, $\{3\}$ and $\{4\}$; thus $s_a = \{1,2,3,4\}$. This correlates with $a$ being transitively reachable from the source nodes 1, 2, 3, and 4. Node $b$ has incoming dependencies tagged with (from left to right) $\{0\}$, $\{0\}$ and $\{1\}$ and thus $s_b = \{0,1\}$.

***Admission Phase.*** The admitting thread collects the identifiers of all sources it changes in $t := \{id_i : \forall i \in C\}$, which is attached to pulses sent during the turn. Figure 6 shows the DG right after the admission phase. Only source node 2 was changed, thus $t = \{2\}$. Since only changed sources send out pulses, these are "change" pulses. Bold arrows in the figure represent pulses that have been sent out, but not yet processed. Nodes reachable from $t$ are shaded grey. They must be reevaluated and need to process and send pulses during the update turn.

Update turns in SID-UP must happen mutually exclusive: Concurrent updates may cause glitches by exposing partial results to each other whenever they affect the same node in the DG. If this can not be guaranteed by the nature of the application, a centralized coordinator is needed during the admission phase. Every thread that wants to admit a change must first contact this coordinator to acquire exclusive access to execute an update turn. After the turn completes, the exclusive access is relinquished to allow the next update turn to start. These are the only two messages outside of regular pulses in SID-UP and they require many-to-one communication only. Supporting concurrent update turns, thus getting rid of the centralized coordination, is ongoing work.

***Propagation Phase.*** Each $n \in N$ stores information about its "pulsing" state in the current propagation phase as a pulse value $p_n \in \{pending, unchanged, changed\}$; the initial value for each update turn is $pending$. After reevaluation, the value is $changed$ or $unchanged$ until the turn ends. In both cases, the node sends a pulse notification to all outgoing dependencies.

When a node $d$ receives a pulse and $p_d = pending$, it iterates over its incoming dependencies $dd \in \overrightarrow{dep}_d$. For each $dd$, it (a) checks whether $dd$ has pulsed ($p_{dd} \neq pending$) and (b) calculates the set of changed sources which $dd$ is reachable from as $x_d d := t \cap s_{dd}$. Observe, that $x_d d = \varnothing \leftrightarrow dd \notin CC$. If $\exists dd$ with $p_{dd} = pending \wedge x_d d \neq \varnothing$, $dd$ still has to pulse. Hence, $d$ resumes to *wait for further incoming pulses*. Otherwise, $d$ initiates the reevaluation procedure.

For illustration, consider the node $n$ in the center of Figure 6. On receiving the pulse via the bold incoming edge, it queries its dependencies left to right: For the left-most dependency $l$, $p_l = pending$, and $x_l = t \cap s_l = \{2\} \cap \{1\} = \varnothing$, meaning $l \notin CC$ and thus not involved in the update turn, i.e. $n$ does not need to wait for it. The next dependency is the source node with identifier 2, whose pulse value is $changed$, meaning $n$ does not need to wait for it either. For the dependency on the right, $r$, $p_r = pending$, as no pulse has been sent via this edge yet, and $x_r = t \cap s_r = \{2\} \cap \{2,3\} = \{2\} \neq \varnothing$. Thus, $r$ *is* involved in the update turn and $n$ must not reevaluate yet, but wait for the missing pulse from $r$ to avoid a glitch.

Once a node $d \in D$ receives its last pending pulse, it begins the reevaluation procedure. If $\exists dd \in \overrightarrow{dep}_d$, $p_{dd} = changed$, $d$ must be reevaluated (at least one input to the computation associated with $d$ changed). This can either (a) change $d$'s state, resulting in $p_d := changed$, or (b) leave $d$'s value unchanged, resulting in $p_d := unchanged$. In both cases, a pulse is sent out immediately after. Once a node pulsed, it will not reevaluate again until the turn completed. If no dependency changed, i.e. $\forall dd : (p_{dd} = pending \wedge x = \varnothing) \vee (p_{dd} = unchanged)$, reevaluation is not needed and $p_d$ is set to $unchanged$ directly. In all cases, the set $t$ received with all incoming pulses is forwarded with every pulse emitted by $d$.

To avoid storing the values of a node $n$ redundantly on every node $d \in \overleftarrow{dep}_n$, locally $d$ accesses the values $p_n$ and $v_n$ in pull mode. Remotely, however, this would imply communicating back and forth over network connections for sending the pulse and then retrieving the values. To avoid this and minimize the amount of remote communicatoins, dependency edges on remote reactives are bridged by *mirror nodes*. If $d$ is on a different host than $n$, a mirror node $m_n$ is created on $d$'s host (and every other remote host where $n$ is used) and $d$ actually depends on $m_n$. A special dependency on $n$'s host pulls the values from $n$ whenever it pulses and sends them to $m_n$ in a single message. Remote communication requires FIFO, exactly-once delivery, as provided e.g., by TCP. Upon reception, $m_n$ then pulses and offers the values to $d$ locally via regular pull-based access. In summary, values are stored once per host and accessed pull-based locally, but duplicated push-based between hosts. In addition to avoiding local value duplication and minimizing remote messages, this enables reasoning about the interaction between nodes without concern of remote edges, as any values are always provided by local nodes.

We assume no node or link failures, as these would make glitch freedom impossible. Handling failures is left for future work. Communication in SID-UP, except for the single coordinator message per turn (if needed), can be asynchronous.

Yet, to determine the moment at which an update turn completes, eventual reply messages in the style of Dijkstra and Scholten's *Diffusing Computations* [12] have to be awaited.

### 3.2 Handling of Dynamic Dependencies

SID-UP supports dynamic dependencies and as a consequence can handle dynamic network topology changes, such as connecting or disconnecting hosts, through higher-order reactives. The challenge created by dynamic dependencies lies in the source identifier sets that each node stores. These sets hold information about the topology of the graph leading up to a node, which becomes invalid when that topology changes. This occurs first on the immediate nodes whose incoming edges change and transitively affects their dependent nodes, as for those one of the identifier sets from one of their dependencies changed. When a set becomes invalid, it has to be updated to restore correct behavior and to provide glitch-freedom for future turns.

To transitively update identifier sets, SID-UP reuses the change propagation mechanism of the normal node values. Like normal change propagations, $s_n$ can change only for $n \in CC$. Just like $v_n$ and $p_n$, $s_n$ is the result of a calculation that aggregates the values of the $n$'s direct incoming dependencies. As with regular value changes, each node keeps track of the changes to its source identifier sets through a second flag that is set to either *changed* or *unchanged*. Just like the node's steady and pulse value are recalculated whenever any dependency pulsed, the node's source identifier set is recalculated when any dependency's source identifier set changed. If for all dependencies the set did not change or the node was not involved in the turn, the set update is skipped and recorded as *unchanged* immediately. Each node's single pulse for the turn is sent out only after both value and set recalculation (whichever were required on the given node) finished.

### 3.3 Correctness

We assume that the user-defined computations associated with each node terminate and consider correctness to be the following: Every update turn (a) is glitch-free, and (b) updates all nodes that need to be updated and terminates.

**Proof of (a).** By construction a node evaluates at most once during a turn. Hence, proving glitch freedom only requires to prove that no node evaluates before all its incoming dependencies have completed their updates. We show this by induction over the update propagation.

The *induction hypothesis* is that all $dd \in \overrightarrow{dep}_d \cap CC$ (predecessors transitively reachable from changed sources) have updated glitch-freely. The *induction base* considers all changed source nodes in the turn. They fulfill this by construction, sending out a single, final pulse at the end of the admission phase, when no more changes can be admitted for the turn. The *induction step* considers the update propagation over a dependent node $d \in D$. Before reevaluating, $d$ awaits the hypothesis to be fulfilled via pulses from all $dd$. Any predecessor $dd \in \overrightarrow{dep}_d \setminus CC$ cannot change during the turn. Thus, waiting only for pulses from $dd \in \overrightarrow{dep}_d \cap CC$ suffices to guarantee that all $dd \in \overrightarrow{dep}_d$ are in a consistent state before $d$ reevaluates. Hence, $d$'s update is glitch-free, sustaining the hypothesis.

The step is valid for dynamic nodes too. The set of incoming dependencies that such nodes must wait for may change during the update. But, whenever that happens, they simply re-perform the same waiting checks on these new dependencies. Once a dynamic node determines that all its incoming dependencies are in a completed state, the induction hypothesis implies that these nodes will not change again. Since the set of dependencies can only change based on updates from incoming dependencies, it is thus final as well. This guarantees the update executed at this state to be glitch free. Hence, the induction hypothesis is sustained for dynamic nodes, too.

**Proof of (b).** First, we show that a propagation turn cannot get stuck: If there are still dependent nodes that need to reevaluate, at least one of them is ready to do so. We show this by contradiction. Assume there is a node $d$ waiting indefinitely for a pulse from some $dd \in \overrightarrow{dep}_d$. By construction, the set intersection checks guarantee that $dd \in CC$. As sources in $CC$ always pulsed, $dd$ must be a dependent node. For it the same logic applies iteratively, creating a waiting path. As the DG is finite and acyclic, no such path of infinite length can exist, contradicting the assumption.

The property holds for dynamic dependencies too. When their dependencies change, dynamic nodes first register as dependent on the new dependencies and then query them for their pulse state. Since nodes set their pulse value before sending out the pulse notification, this order guarantees seeing either the set pulse value or receiving the pulse notification or both, but makes it impossible to miss both. Thus there is no way to miss any new dependency's update completion, thereby guaranteeing that the dynamic nodes do not get stuck indefinitely, either.

By construction, every node always sends every pulse to all its outgoing dependencies. Given that nodes cannot get stuck waiting, an update turn is guaranteed to reach all nodes in $CC$, which is a super-set of all nodes that must be updated during the turn (cf. Section 2.1). As all nodes' updates are glitch-free and every node does indeed update if it receives a pulse and at least one of its dependencies has *changed*, it is guaranteed that all nodes that must be updated will be updated. Further, once every node in $CC$ has finished updating, the propagation phase and thus the turn terminates.

## 4. Comparison to Existing Approaches

We compare SID-UP to existing algorithms by an example scenario and by complexity analysis. The comparisons are not meant to be rigorous, but to give intuitions of why SID-UP outperforms other solutions, as shown in Section 5.

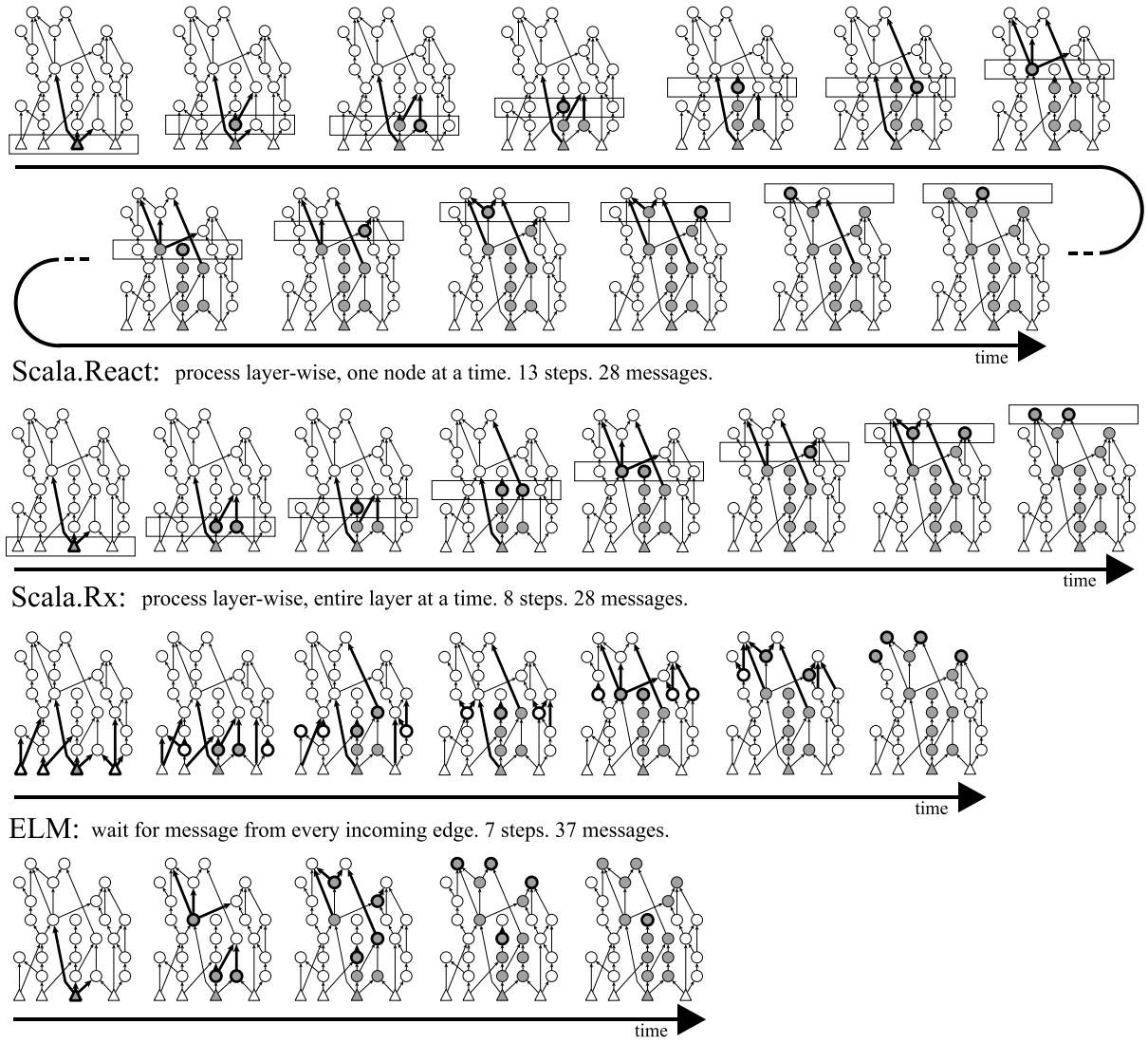


**Figure 7.** Visual comparison of update propagation with different algorithms.

### 4.1 Example-Based Comparison

To highlight the advantages of SID-UP, Figure 7 shows how an update propagates through a DG with different approaches. Bold edges represent pulse notifications pending processing by their receiving node. Bold-outlined nodes have just processed their incoming notifications and pulsed. Nodes shaded gray have reevaluated.

Scala.React and Scala.Rx (1st and 2nd timeline) proceed in topological order. The processing layer is highlighted by the overlapping rectangle. Scala.React is single-threaded: Only a single node in the rectangle is updating (shown in bold) at any point in time. In Scala.Rx all nodes on the same layer are updated concurrently, reducing the number of steps the algorithm requires in trade for some synchronization overhead after each level. In Scala.React and Scala.Rx bold dependency edges and bold-outlined nodes correspond to messages that have to be transmitted: The former are transmitted between the nodes at each end of the edge, the latter between the node and the centralized priority queue.

As outlined at the end of Section 2.4, ELM in its original form is not suitable for the distributed setting, because its pipelining feature renders HORs impossible, and is actually incomparable to the other algorithms. Yet, for completeness we include it in the comparison, but make sure that its differences to the other algorithms show no effect by looking at a single turn without topology changes in the graph. ELM (3rd line) does not have a priority queue that restricts reevaluations to a single active layer. As can be seen from the figure, nodes update in different layers of the graph. As nodes reevaluate after each incoming edge is bold without authorization from a coordinator, only bold dependency edges correspond to messages. Nodes with a bold outline do

not require a message to be transmitted, reducing the amount of remote communication. The price to pay is that *all* nodes must process *every* turn, propagating *no change* pulses in unaffected areas of the DG.

Similar to ELM, SID-UP (fourth line) requires no coordinator during propagation. But, SID-UP significantly reduces the number of messages: There are far less bold dependency edges than in ELM's timeline. With the set intersections, only nodes reachable from changed sources must become active during a turn. The reduced number of messages does not necessarily indicate a faster termination of the update turn: While ELM for every received message only decrements a counter, SID-UP computes set intersections and sometimes unions, increasing the time per step.

### 4.2 ELM$^s$

From the intuitive comparison above, except that it affects more nodes, ELM's propagation model behaves just like other algorithms, which are compatible with dynamic dependencies. It seems like it should be possible to devise an algorithm that combines the propagation model with support for dynamic dependencies. Indeed, to enable a more complete evaluation, we devised such an algorithm, called ELM$^s$, which we give here an overview of.

ELM$^s$ is a mixture of SID-UP and ELM: Starting from SID-UP's implementation, we added ELM's global update propagation by collecting all sources into a centralized coordinator which initiates update turns globally. Further, we stripped the source identifier sets from all nodes, replacing the set intersection check with the constant `true`, as every node always processes every update turn in ELM. The resulting algorithm behaves just like ELM when propagating a single update turn, such as in the intuitive comparison above, requiring unbounded one-to-many communication from the coordinator during the admission phase and executing the propagation phase without coordinator involvement or set operations. But it trades off pipelining against support for dynamic dependencies: Successive update turns are executed sequentially instead of pipelined (hence the $^s$ suffix for the name), which allows nodes to retain their pulse value until the end of each turn. This way, dynamically added dependency edges will never miss a pulse value from their start node already having progressed to future turns. This makes dynamic dependencies safe to use, thus rendering the algorithm feasible for use in dynamic network topologies by supporting HORs and comparable to the others.

### 4.3 Complexity Comparison

SID-UP is designed to reduce the number of remote messages and to enable computations on different hosts to proceed asynchronously as much as possible. Our analysis unfolds along these two dimensions. For simplicity, we assume a fixed topology of the dependency graph.

***Turn Evaluation Time.*** Initially we assume that all nodes require the same processing time $const$ to reevaluate. In Scala.React, all nodes are evaluated sequentially, so $T_{Scala.React} \sim u \cdot const$. In Scala.Rx, nodes on the same level are evaluated in parallel. Hence, the total time depends on the topology. The worst case is when there is one node per level, resulting in the execution time being equal to that of Scala.React, $T_{Scala.Rx,W} \sim u \cdot const$. In the best case, all nodes are on the same level and can be reevaluated in parallel, hence $T_{Scala.Rx,B} \sim const$.

Under the assumption that nodes have equal computation time, SID-UP and ELM$^s$ exhibit the same performance as Scala.Rx: $C_{SID-UP,W} = C_{ELM^s,W} \sim u \cdot const$ and $C_{SID-UP,B} = C_{ELM^s,B} \sim const$. If nodes have different computation times $t(n_i)$, SID-UP and ELM$^s$ can perform strictly better than Scala.Rx.

This is the case for dependency graphs with parallel branches; for simplicity we assume binary branches. Figure 8 shows an example with two branches $A = a_1 ... a_n$ and $B = b_1 ... b_m$, where $a_i$ and $b_i$ are sequences of nodes. In Scala.Rx, nodes are processed level by level: Nodes $a_1$ and $b_1$ are processed in parallel, then $a_2$ and $b_2$, etc. The time required to process each level is bound by the slowest node of the level, in the example $max(t(a_i), t(b_i))$. In summary, the time required to process all nodes in the branches is $T_{Scala.Rx,branch} = \sum_{i=1}^{max(n,m)} max(t(a_i), t(b_i))$ where, for simplicity, we assume $t(a_{i>n}) = 0$ and $t(b_{i>m}) = 0$ to account for the case $n \neq m$.

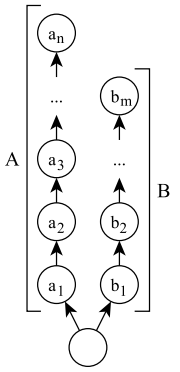

**Figure 8.** Two branches in a DG.

In SID-UP and ELM$^s$, evaluation does not progress level by level. Consider the case where $a_1$ and $b_1$ in Figure 8 start being evaluated in parallel. When $a_1$ completes, $a_2$ starts being evaluated even if $b_1$ did not complete yet. Hence, the time required to evaluate a branch is the sum of the time of each node in it, e.g., for B, $\sum_{i=1}^{m} t(b_i)$. Overall, we have $T_{SID-UP,branch} = T_{ELM^s,branch} = max(\sum_{i=1}^{n} t(a_i), \sum_{i=1}^{m} t(b_i))$. It is easy to see that $T_{SID-UP,branch} = T_{EML^s,branch} \leq T_{Scala.Rx,branch}$. SID-UP and ELM$^s$ are mostly equal with respect to user computations, except for specifically constructed cases. They thus usually only differ due to differences in their computational overhead per node.

***Number of Messages.*** Let $c = |CC|$ be the number of nodes transitively reachable from any change source through outgoing dependency edges, and $u$ the number of nodes actually updated (i.e. $p_n = changed$). Because only nodes in $CC$ can change, it follows that $u \leq c \leq n$, where $n = |N|$. In a complete graph there are $e = n^2$ edges.

In Scala.React, propagation reaches only nodes actually updated: Outgoing dependencies of a node are enqueued in the priority queue only if that node's value changed. Each such node receives a message from the coordinator to initiate its reevaluation. Hence, the number of messages for a propagation turn in Scala.React can be estimated as $M_{Scala.React} \sim (u^2 + u)$, i.e. pulse messages over all outgoing edges from updated nodes plus messages sent by the coordinator. Since the communication scheme is identical, the same analysis applies to Scala.Rx. SID-UP has no coordinator during propagation and transmits messages only over edges between nodes in $CC$, resulting in $M_{SID-UP} \sim c^2$. ELM$^s$ has no coordinator during propagation either, but sends a message over each edge, thus $M_{ELM^s} \sim n^2$.

In practice, three phenomena occur. (a) Dependency graphs are usually sparse, i.e. $e \to n$. Thus, $M_{Scala.React} \sim u+u = 2u$, $M_{SID-UP} \sim c$ and $M_{ELM^s} \sim n$. (b) Propagation phases that leave many nodes in $CC$ unchanged are rare, meaning $u \to c$. (c) Propagation phases that affect the entire graph are rare, too, so $c \ll n$. Thus, usually, $M_{Scala.React} \lessapprox M_{SID-UP}$ as $u \sim c$ and $M_{Scala.React} \leq M_{ELM^s}$ as well as $M_{SID-UP} \leq M_{ELM^s}$, since $u \sim c \ll n$.

## 5. Evaluation

We empirically evaluate SID-UP to answer two questions: (a) Does DRP based on SID-UP yield improved design/code quality at reasonable performance cost? (b) Is SID-UP more efficient than existing algorithms in a distributed setting?

### 5.1 Case Study

For the evaluation, we separate the case study application from Section 2 into two domains: One encompasses local computations and user interface, the other encompasses the remote value propagation and observation. Reactive programming can be used for either of these domains. Its benefits on (local) reactive computations have already been empirically demonstrated [24]. We thus purposefully do *not* compare the application's entire code base: To avoid these known benefits affecting the measurements of code metrics for the parts relevant to distributed reactive programming, we excluded any user interface and bootstrap code from the experiment. All measurements in the following therefore only compare the part of the application responsible for remotely propagating value updates.

We organized the evaluation as follows: We compared the case study with a variant for the remote value propagation based on the observer pattern and one based on reactive programming. Since we implemented both the remote observers as well as remote reactives on top of Java RMI, this demonstrates the impact caused purely by the different programming models, independently of what kind of marshalling technique is used for remote calls. As explained in Section 2.1, the application requires glitch freedom to work correctly. While for reactive programming, this is provided automatically, in the observer-based variant we had to implement glitch freedom as part of the application logic. There is a multitude of options for implementing this, e.g., by fixed-time waiting as in clocked hardware, or by registering management as an observer on the order depot to always be notified last, after both purchases and sales have been notified, thereby relying on invocation order. We chose to simply attach a boolean flag in the notifications from sales and purchases that informs management, whether it must wait for the respective other department. While this is a clear violation of modularity (purchases encodes knowledge about the data dependencies in sales and vice versa) and far less flexible than reactive programming, it is the most efficient solution in terms of additional lines of code and minimal performance cost.

| Metric | Observer | Unsafe Observer | Reactive |
|---|---|---|---|
| LOC | 95 | 82 | 60 |
| Callbacks | 3 | 3 | 0 |
| Vars | 8 | 7 | 0 |
| Vals | 3 | 3 | 10 |

**Table 1.** Metrics extracted from the case study

***Code Metrics.*** Table 1 shows software code metrics, which are indicators for the quality of the source code, from comparing an implementation based on distributed reactive programming (column *Reactive*) with an implementation based on remote observers with manually implemented glitch freedom (column *Observer*). We also compare against an unsafe remote observer variant, which is the same as the regular observer variant, but without the manual glitch freedom implementation (column *Unsafe Observer*).

As can be seen from the table, distributed reactive programming achieves the best metrics, indicating it provides the best code quality: There are fewer lines of code (line *LOC*) due to removal of callback wrapping code (line *Callbacks*) and of the manual glitch freedom implementation. As a result, the code is free of bloat and only models the actually desired application functionality. Further, we were able to remove all manually maintained mutable variables (line *Vars*) from the code and rewrite the application entirely using constant variable declarations (line *Vals*) referencing reactive abstractions that encapsulate this mutability. This indicates that implementations based on distributed reactive values are also more robust than those built using remote callbacks, as bugs from accidental and erroneous manual state mutations are made impossible. Further, it provides glitch freedom without requiring violations of modularity in any department. A qualitative analysis of a small piece of this code base was provided for the snippet in Figure 2 (reactive programming) and the snippet in Figure 3 (unsafe observer) in Section 2.

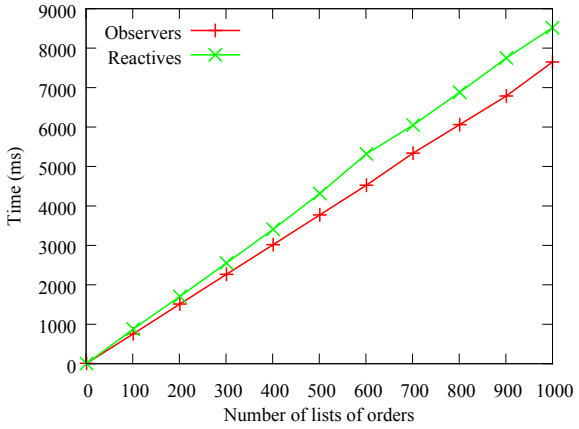


**Figure 9.** Performance of the case study.

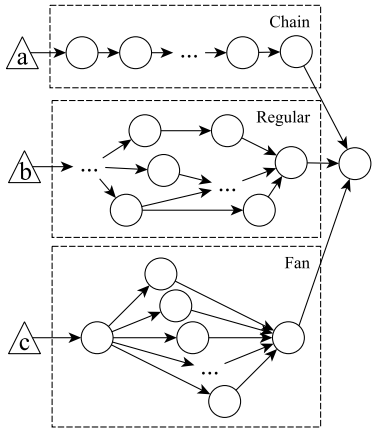


**Figure 10.** Graph used in the benchmarks.

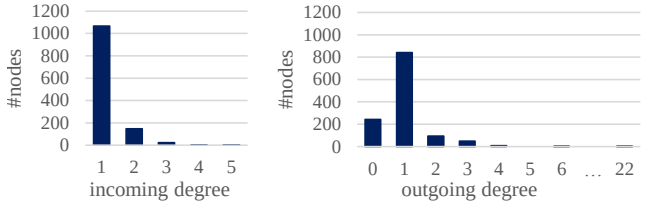


**Figure 11.** Distribution of dependency degrees.

The comparison against the unsafe observer implementation shows that the above advantages still apply to software that does not require glitch freedom. Removing the manual glitch freedom implementation lessens the gap in terms of code metrics. Yet, column *unsafe observer* still exhibits worse metrics than *reactive*.

***Performance Cost.*** For the performance comparison, we inspected again only the part of the application concerning remote value propagation. Both the application responsiveness for the user as well as the amount of data throughput depend mainly on the time it takes for every input value change to be propagated through the entire application. To measure this time, we implemented a loop that publishes a new list of orders at the order depot whenever the previous update had taken effect in management (cf. Figure 1).

Figure 9 shows the amount of time required to perform an increasing number of subsequent update turns. Each update turn sets a new list of ten orders in the order depot. Unsurprisingly, the implementation based on remote observers is faster in completing pushing updates through the application. This is easily explained, as the generality of reactive programming comes at a cost. In our case, this cost consists of the operations performed with all the source identifier sets on each node. Though, as the graph shows, the processing time of reactive programming supported by SID-UP still exhibits similar complexity and only increases required time by a factor of approximately 10% in this case.

### 5.2 Benchmarks

To empirically compare SID-UP with existing algorithms, we implemented a benchmark in form of a reactive network through which we propagate an update. The algorithms being compared are expected to perform differently on different topologies due to various approaches for parallelism and message transmissions. Hence, we assembled a graph from modules implementing various topologies. Figure 10 shows a schema of the graph we used. The "chain" module implements a linear chain of reactive nodes that does not allow parallelism. The "regular" module implements a graph with some nodes connected with varying degrees of fan-in and fan-out dependency connections that allows some parallelism. To be realistic, these degrees are chosen according to statistic distributions, which we measured by instrumenting 20 local reactive applications we developed in previous case studies; Figure 11 shows the distribution of the number of nodes for each value of incoming degree (left) and outgoing degree (right). The "fan" module implements a topology where one node fans out into a lot of immediate successors, all of which can be reevaluated concurrently. Each module contains 25 nodes and both "regular" and "fan" contain a few nodes whose values do not change during the update turn, i.e. although a dependency changes, they update to an *unchanged* pulse value or equivalently do not add their outgoing dependencies to the priority queue. Updates can be initiated inside each module separately through individual source nodes, although for the duration benchmarks we always update all sources to affect the entire graph. Finally, a dependent node at the end unifies the updates from all modules to detect the update turn completion.

Optimizations of the dependency graph's topology may improve the performance of reactive programs. Since typically each node in the graph introduces a certain amount of overhead, optimizing the topology by reducing the number of nodes leads to better performance. The most prominent technique here is Lowering [6]. We performed our analyses without applying such techniques, as they are orthog-

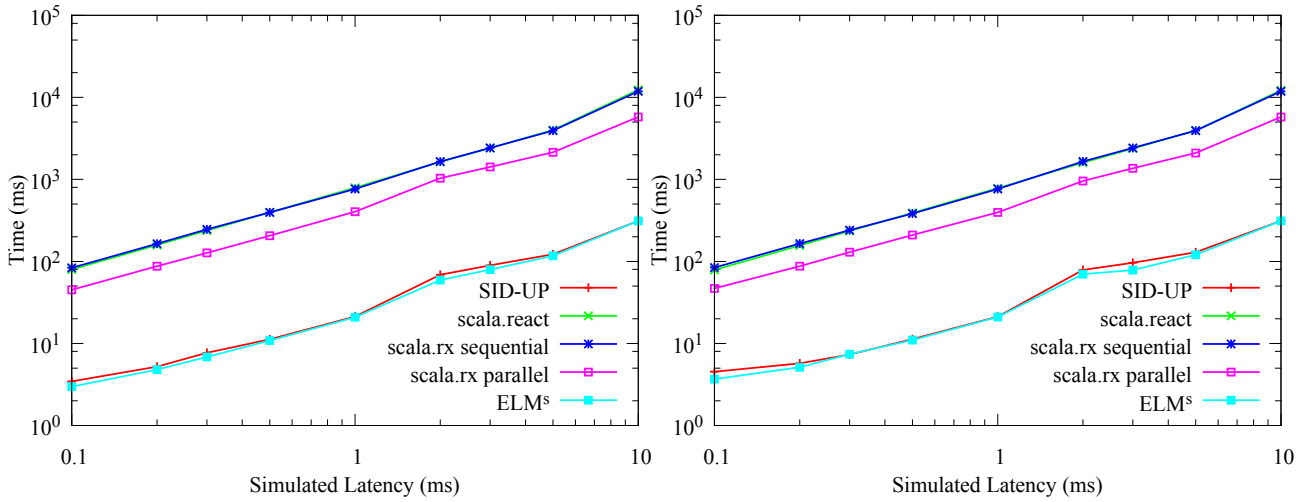


**Figure 12.** Update turn completion duration over increasing latency with few and many sources.

onal to the propagation algorithm in use. Each algorithm simply traverses and interacts with the topology graph, optimized to not. Whether or not the graph has had some nodes collapsed into fewer nodes does not change these interactions. Vice versa, topology optimizations do not require any involvement from the run-time propagation algorithm and can be computed statically. That said, we expect such optimizations to benefit SID-UP the most of all included algorithms because SID-UP with its comparatively very complex set operations has the biggest overhead per node. Evaluating the effectiveness of different combinations of topology optimizations and propagation algorithms would be an interesting study, but extends beyond the scope of this paper.

The experiments compare five propagation algorithms[3]: Scala.React's and Scala.Rx's implementation of sequential topological sorting with priority queue, Scala.Rx's implementation of the parallel propagation strategy and our reference implementation of SID-UP. We also include our SID-UP ELM hybrid algorithm ELM$^s$ from Section 4.2 to measure the overhead caused by the set operations used in SID-UP: As we showed in Section 4.3, SID-UP and ELM$^s$ exhibit the same turn durations in terms of user computations and only differ in execution time only by their individual computational overhead per node.

We "distribute" the graph in Figure 10 by pretending that each module runs on an individual host; the source and sink nodes as well as the coordinator (if needed) run on another separate host. As no framework except SID-UP actually supports distribution, we simulate network latency by injecting waiting times wherever a method call would have to be send over the network, i.e. whenever a node sends a pulse notification to a node on another host, or whenever the coordinator sends a command to a node on a different host than its own. This may seem like a disadvantageous comparison for algorithms dedicated to distributed graphs, since only very few edges are actually remote connections. But, the main disadvantage of the topological sorting-based algorithms stems from remote communication with the priority queue, which happens outside of the dependency edges between nodes. Therefore, these few remote edges suffice to show the full effect.

The left graph in Figure 12 shows for each algorithm the time it takes to complete an update turn on the graph in Figure 10 when simulating various amounts of delay on the network edges. SID-UP and ELM$^s$ clearly outperform the other algorithms, especially considering the double logarithmic scaling. All curves are approximately linear, meaning that all algorithms scale linearly with increasing network latency. But, because of double logarithmic scaling, the differences between these curves indicate different factors of scaling with latency, with SID-UP and ELM$^s$ achieving the lowest factor.

This is in line with previous findings that mark-and-sweep algorithms generally outperform those based on topological sorting [27]. We did not include a straight-forward mark-and-sweep algorithm in the comparison, as executing both phases would immediately imply duplicating all remote message delays. But, both SID-UP and ELM$^s$ can be seen as special cases of mark-and-sweep algorithms, where the sweep phase is done implicitly: In SID-UP, all nodes $n \in CC$ are implicitly marked through the set intersection test. In ELM$^s$, simply every node is implicitly considered marked for every turn.

A comparison of the curves of SID-UP and ELM$^s$ enables to estimate the overhead caused by the more complex set operations in SID-UP. As there is barely a difference, this comparison indicates that this overhead is negligible. But, the

[3] All tests were performed with Scala 2.10.3, Sun Java HotSpot Client VM 1.7 update 10, Windows 7 64 bit and an Intel Core i5-3320M with 8 GB of RAM. To run the benchmarks, we fixed a bug of Scala.Rx's[26] garbage collection support present in commit e4f4070cac cloned on 11/26/2013, which caused linearly increasing execution time.

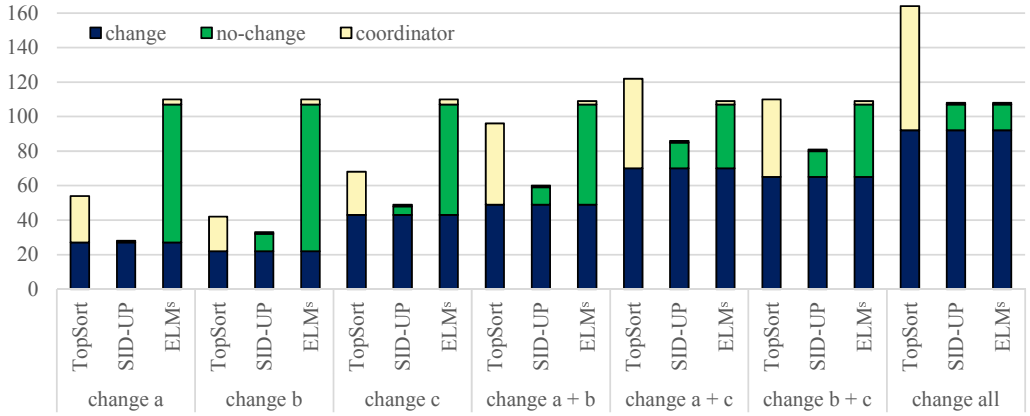


**Figure 13.** Number of messages for propagating through different sub-graphs from Figure 10.

overhead is not constant: The larger the sets involved in these operations become, the more the overhead for SID-UP will grow. To evaluate this in detail, we repeat the same experiment with around 150 additional sources spread evenly over all nodes in the graph. With about two input nodes for every dependent node, this gives an overestimation of the impact they will have on SID-UP in normal use cases, as usually dependent nodes make up the majority of a dependency graph. Further, we do not update those sources to keep their impact on the other algorithms minimal. The right graph in Figure 12 shows the results: SID-UP and ELM$^s$ perform slightly worse, while there is essentially no impact on the other algorithms. This is to be expected as for SID-UP the identifier sets grow and ELM$^s$'s global processing has to send additional messages over all the new sources. However, SID-UP and ELM$^s$ still clearly outperform the other algorithms and still do not differ much from each other. Hence, we conclude that even in the presence of many large identifier sets SID-UP's overhead remains negligible.

To empirically validate the superiority of SID-UP over ELM$^s$ from the message count analysis in Section 4.3, we counted the number of message sends by each algorithm when updating different sets of sources from the graph in Figure 10. Through this message count the overall usage of computational resources to process an update turn can be estimated. This usage is not fully measured in the timing benchmarks, because it is processed in parallel to the slower path through the graph along which value changes are actually computed. Yet, it blocks other workloads and requires more energy. Figure 13 shows the results: For all algorithms the number of change pulses is identical in each case. But, the algorithms differ in their additional messages: Scala.React and both versions of Scala.Rx require additional coordinator messages for changed nodes. SID-UP requires no coordinator messages, but instead propagates a few pulse messages from *unchanged* nodes. These two sets of messages are generally incomparable because the number of additional coordinator or respectively *unchanged* pulse messages are dependent on the actual value changes from the user computations that occur during the update turn. ELM$^s$, however, performs clearly worse: Because of its globally processed update turns, it requires the same number of messages for every update. This is the number that SID-UP uses only in the worst-case, when all sources a, b and c and successively the entire graph are updated – a use case that essentially never occurs in practice as user code usually acts localized and only admits new values to very few select sources. In all of these common cases, SID-UP uses strictly less messages to complete the update turn, meaning it consumes less computational resources overall. With the set operation overhead being negligible and the overall resource consumption being lower, SID-UP thus outperforms ELM$^s$. Further, ELM$^s$ relies on unbounded one-to-many communication, visible here in the form of one coordinator message per source. This puts additional requirements on the infrastructure that must be available to run ELM$^s$, making it more difficult to use.

Summarizing, we conclude that SID-UP provides the best trade-off between update turn completion time and computational resource usage. Furthermore, it is the only algorithm that does not rely on unbounded one-to-many communication, and thus is the easiest to execute in any given distributed environment.

## 6. Related Work

We discuss extentions of reactive programming towards parallelism and distribution and summarize state of the art in reactive programming and related fields.

### 6.1 Parallel and Distributed Reactive Programming.

Scala.Rx [26] implements a propagation algorithm that supports a limited amount of parallelism during update propagation. ELM [9] enables more parallelism and introduces concepts of pipelined and asynchronous update turns. Both of their propagation algorithms, however, have major downsides when used in distributed settings. We discussed these

downfalls in Section 2.4 and showed a comparison against SID-UP in Sections 4 and 5.2.

Flapjax [22] implements Reactive Programming in JavaScript to support reactive Web applications. However, Flapjax addresses only client side code in the client-server model. The server side of the application can potentially be implemented in a reactive language, but the reactive system on the client and on the server are not aware of each other. As a result there is not guarantee of glitch freedom spanning the whole application.

AmbientTalk/R [19] provides Reactive Programming for mobile peer-to-peer networks. It does, however, strongly focus on mobile clients and unreliability in communication stemming from network topology changes as clients move around and join or leave the network. In such an environment, the cost of maintaining glitch freedom is infeasibly high. Thus, like Flapjax, AmbientTalk/R does not provide any concept for glitch freedom involving multiple hosts.

### 6.2 Technologies for Reactive Applications

*Reactive Languages* track the dependencies among reactive values and automatically perform the necessary updates. We already described Scala.React (Scala) and Flapjax (JavaScript). FrTime [8] is a reactive language implemented on top of Scheme. In contrast to Scala.React, which is based on the DSL support offered by the Scala language, and Flapjax, which is provided as a library or a dedicated JavaScript preprocessor, FrTime leverages macro expansion to lift traditional values to reactives. REScala [24] is an embedding of reactive programming in Scala and focuses on the integration of reactive abstractions into object-oriented applications [25]. Demetrescu et al. [11] describe a runtime environment which natively supports reactive memory to implement dataflow constraints.

*Functional Reactive Programming* has been originally proposed by Conal Elliott in the strictly functional language Haskell [13]. Recent work in functional reactive programming focuses on enforcing good properties of reactive applications. Krishnaswami *et al.* [18] use linear types to guarantee bounded-space execution of reactive programs. Krishnaswami provides an FRP implementation for programs that are provably free of spacetime leaks [17]. Jeffrey [16] developed a type system for FRP that provides liveness guarantees of reactive programs.

*Self-Adjusting Computation* is about deriving incremental algorithms from batch ones [2]. In self-adjusting computation, programs are statically analyzed to detect the dependencies among values and derive a dependency graph similar to the one used in reactive programming. This technique, originally developed for the functional setting, has been recently applied to imperative programs [1]. An interesting line of research involves the application of these concepts to streams in the context of Big Data. Incoop [5] is an incremental MapReduce [10] framework. When the input dataset changes, Incoop performs a fine-grained update of the output previously computed. The whole incrementalization process is transparent to the user that interacts with a traditional MapReduce interface.

*Synchronous Dataflow Languages* model reactive applications as reactive networks where a signal is propagated synchronously. Examples of such languages include Lustre [7] Signal [15], and Esterel [4]. Synchronous dataflow languages focus on bounded memory and time executions which are fundamental for real-time and critical systems such as microcontrollers.

*Declarative Networking* approaches enable the concise specification of network protocols and services that are compiled to dataflow programs. Network Datalog [20] is such an approach: It provides incremental updates to the algorithm result as a response to network changes. There are, however, important differences between Network Datalog and (Distributed) Reactive Programming. While the former focuses on the specific domain of network protocols and requires programming in Prolog-style rules and queries, the latter focusses on general purpose applicability and ease of use by adopting and extending more mainstream programming languages.

## 7. Conclusion and Future Work

In this paper, we presented distributed reactive programming in Distributed REScala. DRP can improve the design of distributed, observer-based applications as indicated by the small-scale case study presented in this paper. We showed that existing algorithms for (local) reactive programming are not suitable for distribution. The algorithm of Distributed REScala, named SID-UP, reduces the amount of messages sent over the network and features a high degree of parallelism, while still preserving glitch freedom. We demonstrated empirically that SID-UP outperforms existing solutions in terms of the trade-off between update turn completion time and computational resource usage.

There are several areas for future work. We plan to extend our approach to support concurrent update turns and to handle failures. We will also conduct more case studies using distribute reactive programming and apply distributed reactive programming to refactor existing observer-based applications.

## Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research under grants No. 16BY1206E and No. 01IC12S01V and by the European Research Council, grant No. 321217.

## References

[1] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of pro-*

*gramming languages*, pages 309–322, New York, NY, USA, 2008. ACM.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, Nov. 2006.

[3] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52:1–52:34, Aug. 2013.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of computer programming*, 19(2):87 – 152, 1992.

[5] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[6] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–80, New York, NY, USA, 2007. ACM.

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[8] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, pages 294–308. Springer, 2006.

[9] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 411–422. ACM, 2013.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[11] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 407–426, New York, NY, USA, 2011. ACM.

[12] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[13] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, New York, NY, USA, 1997. ACM.

[14] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I: Conference Contributions - Volume I*, pages 31–44, London, UK, 1991. Springer-Verlag.

[15] T. Gautier, P. Le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Functional programming languages and computer architecture*, pages 257–277, London, UK, 1987. Springer-Verlag.

[16] A. Jeffrey. Functional reactive programming with liveness guarantees. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 233–244, New York, NY, USA, 2013. ACM.

[17] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 221–232, New York, NY, USA, 2013. ACM.

[18] N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 45–58, New York, NY, USA, 2012. ACM.

[19] A. Lombide Carreton, S. Mostinckx, T. Cutsem, and W. Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In J. Vitek, editor, *TOOLS*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, 2010.

[20] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[21] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, EPFL, 2012.

[22] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 1–20, New York, NY, USA, 2009. ACM.

[23] G. Salvaneschi, J. Drechsler, and M. Mezini. Towards distributed reactive programming. In R. Nicola and C. Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 226–235. Springer Berlin Heidelberg, 2013.

[24] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th annual international conference on Aspect-oriented software development*, New York, NY, USA, 2014. ACM.

[25] G. Salvaneschi and M. Mezini. Towards reactive programming for object-oriented applications. In S. Chiba, . Tanter, E. Bodden, S. Maoz, and J. Kienzle, editors, *Transactions on Aspect-Oriented Software Development XI*, volume 8400 of *Lecture Notes in Computer Science*, pages 227–261. Springer Berlin Heidelberg, 2014.

[26] Scala.rx Web site. https://github.com/lihaoyi/scala.rx.

[27] B. T. Vander Zanden, R. Halterman, B. A. Myers, R. McDaniel, R. Miller, P. Szekely, D. A. Giuse, and D. Kosbie. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):776–796, 2001.