Distributed Functional Reactive Programming on Actor-Based Runtime

Kazuhiro Shibanai
Department of Computer Science
Tokyo Institute of Technology
Tokyo, Japan
shibanai@psg.c.titech.ac.jp

Takuo Watanabe
Department of Computer Science
Tokyo Institute of Technology
Tokyo, Japan
takuo@acm.org

Abstract

Reactive programming over a network is a challenging task because efficient elimination of temporary violations of data flow invariants, known as glitches, in a distributed setting is still an open issue. In this paper, we propose a method for constructing a distributed reactive programming system of which runtime guarantees the properties of single source glitch-freedom and the robustness against out-of-order messages. Based on the method, we developed a purely functional reactive programming language XFRP whose compiler produces Erlang code. Using some examples, we show that the proposed method is beneficial for constructing distributed reactive applications without suffering from inconsistencies.

CCS Concepts • Software and its engineering → Distributed programming languages; Functional languages; Data flow languages;

Keywords Distributed Functional Reactive Programming, Glitch Freedom, Synchronization, Actor-Based Runtime System, Erlang

ACM Reference Format:

1 Introduction

Reactive Programming (RP) is a programming paradigm where a system is described in terms of continuously changing time-varying values and propagation of change[3]. In recent years, RP has gained popularity in various fields: Web programming (React, Vue, and many similar web frameworks), mobile application development (React Native, RxJava, etc.), mobile IoT networks[4, 19], and so on. Functional Reactive Programming (FRP)[10] is a variant of RP where time-varying values and/or their relationships are described in purely functional expressions and thus a system is expressed in a declarative manner.

The change propagation among time-varying values can be classified as dataflow computation. From this viewpoint, reactive programming provides a high-level and declarative abstraction for describing concurrent systems. Thus, integrating (F)RP with existing concurrent computation models is interesting in both theoretical and practical aspects. One of the authors proposed an actor-based execution model of an FRP language for embedded systems, which can reduce the execution cost of a program written in the language by utilizing asynchronous messages in the change propagation[24]. Van den Vonder et al. introduced another direction of integration named Actor-Reactor model that can widen the expressiveness of a reactive programming language by using actors to describe long-lasting or stateful behaviors[22].

Although the idea of integrating (F)RP and the Actor model seems to be beneficial for describing distributed systems, there exist several obstacles to overcome. One of the primary problems is glitches. A glitch is a temporal inconsistency in time-varying values (described in Section 2.2).

Methods for avoiding glitches have been proposed in the existing literature[5, 9, 13, 16]. A straightforward solution is to use a topological sorting of time-varying values in dependency order. However, it is still an open problem to resolve glitches in distributed systems with failures such as node crashes, out-of-order messages, and lost/duplicate messages.

This paper provides a method for constructing distributed reactive systems with the following properties: single source glitch-freedom and the robustness against out-of-order delivery. Besides, we developed a compiler that translates from XFRP, which is a purely functional reactive programming language, into Erlang based on this method. We assume
that the message delivery mechanism adopted in this paper obeys exactly-once delivery semantics, which means that a message must arrive at its recipient and cannot be lost nor duplicated. Discussion about other delivery semantics (e.g., at-most-once delivery) is given in Section 9.1.

The main contributions of this paper are the following:

- We show that distributed systems can be easily constructed in a distributed FRP language with a practical example.
- We propose a novel method that lets distributed reactive systems free from glitches and the effect of out-of-order delivery.
- We show a problem in history-sensitive values (cf. Section 2.2) caused by single-source glitch freedom and give a possible solution for it.
- We propose a mechanism of retransmission for the solution of message losses in a reactive system and an efficient algorithm for the realignment of messages.

The rest of this paper is organized as follows. Section 2 discusses the motivations for our work. Section 3 explains the syntax and execution model of our language XFRP with a small example. Section 4 and 5 present the algorithm of XFRP. Section 6 discusses the problem of single source glitch-freedom and also provides a solution. Section 7 discusses the Erlang implementation of XFRP. A practical example application is provided in Section 8. Section 9 proposes keys to deal with message losses and the realignment of messages. Section 10 surveys related work for distributed RP and Section 11 concludes the paper.

2 Motivation

2.1 Distributed Reactive Programming and Actors

The goal of this work is to develop a distributed execution model of an FRP language and show that FRP is also beneficial for describing distributed systems. Towards this goal, we adopt the Actor model[2] as the runtime of the language. The reasons are: (1) actors and asynchronous messages are suitable to represent time-varying values and the change propagation[24], (2) actors can express such behaviors that (pure) FRP is inconvenient to deal with[22], and (3) there are already stable implementations of actor-based languages and frameworks (e.g., Erlang, Akka) for constructing actual distributed systems.

Figure 1 shows the conceptual model of distributed reactive systems described in our FRP language with the actor-based runtime. As the figure exhibits, a system is structured in four layers.

The base layer is composed of low-level components: physical machines, CPUs, TCP/IP networks, etc. The layer also includes concurrent execution abstractions provided by an operating system such as processes or threads.

The Actor layer controls the executions of actors and the message delivery. The examples of the tasks in this layer are the placement of actors, message routing, execution scheduling, and fault-tolerant mechanisms (e.g., supervisors or persistent actors). Though this layer seems to have numerous tasks, we can utilize the existing actor-based languages and frameworks.

The Reactive Framework layer maintains time-varying values and performs the change propagation among them. The layer also provides the mechanism for glitch-avoidance (see Section 2.2). The mechanisms in this layer are the main topics of this paper.

2.2 Glitch-Freedom

The avoidance of glitches is an important design consideration for reactive programming systems. Glitches are temporal inconsistencies that occur during the propagation of changes. For example, consider the following program written in the syntax of XFRP (see Section 3):

```
node x = a + a
node y = a * 2
node z = (x,y)
```

where a, x, y, z are time-varying values defined using keyword node. In a glitch-free system, all occurrences of the same time-varying value must have the same value at each time point. Thus, we should always recognize that the first and second elements of z are the same. The property does not hold in a system with glitches. For example, consider a situation that the value of a changes. There may be a moment that x and y observe different values for a, and thus the elements of z are different.

Margara and Salvaneschi[15] classified glitch-freedom into two types: single-source and complete. The former requirement is that the update of a source is propagated to the nodes that depend on it without glitches but other sources are not considered at the update. The latter takes the causal relation of all the sources into account in addition to the requirement of the former. Let us explain the difference by using a small example: node x = (a,b). The time-varying value x is a pair of two independent sources (time-varying

Figure 1. Layers of Reactive System on Actors
values) $a$ and $b$, where $a = 1$ and $b = 1$ initially. Let us consider that $a$ becomes 2, and then $b$ becomes 4. Single-source glitch-freedom allows that $x$ may change to (2,1) or (1,4) before it becomes (2,4) because the temporal order between the changes of independent sources is ignored. On the contrary, complete glitch-freedom requires that $x$ must change to (2,1) before (2,4) because the order between the inputs is preserved. In XFRP, basically single source glitch-freedom is assumed but it also uses the concept of complete glitch-freedom.

According to [3], distributed reactive programming systems have not achieved glitch-freedom yet because of network failures, delays, and lack of a global clock. This paper provides a novel method to tackle glitches in distributed systems.

## 3 XFRP

### 3.1 Basics

XFRP is a general-purpose purely functional reactive programming language developed as a successor of Emfrp[21], which is designed for small-scale embedded systems. In XFRP, a system (module) is composed of time-varying values classified in the following categories: sources (inputs), sinks (outputs), and nodes. Sources emit externally given values such as keyboard inputs, network packets from another computer, and measurements from a sensor device. Sinks are the destinations of propagation. They receive results and normally affect the outer world by displaying characters, changing the voltage output, etc. Nodes lie between sources and sinks and update their values by evaluating associated expressions every time sources change. The definition of an expression is given in a purely functional style, which means it has no side effect or mutable states. In summary, changes of sources propagate throughout nodes and finally reach sinks.

Figure 2a shows a simple fan controller application taken from [23]. The program has two sources $tmp$ and $hmd$ respectively representing the current temperature [°C] and humidity [%] measured by external sensors. The node $di$ calculates the temperature-humidity index (the degree of discomfort experienced by humans), and the truth value of $fan$ changes based on the value. If the value of $tmp$ changes, $di$ detects it immediately and recalculates the index, followed by the recomputation of $fan$.

The graph in Figure 2b represents the dependency on time-varying values (sources, sinks, and nodes). An arrow from $x$ to $y$ means that $x$ is in the definition of $y$. The structure of the graph corresponds to the flow of propagation as explained above. The graph must be directed acyclic, that is, has no cycles because changes will propagate infinitely if the graph has a cycle. However, a cycle is allowed if it has a dependency relation with $\text{@last}$, which is explained in the next section.

### 3.2 History-Sensitive Values

All expressions in XFRP are pure and the language provides no destructive assignments. This means that each node can hold only the current value of itself, i.e., there is no way to express states. However, this restriction makes it hard to express even simple calculations such as counting the number of inputs or summing up values.

To overcome the above restriction, XFRP provides an operator $\text{@last}$ as Emfrp[21]. For a node $n$, $n\text{@last}$ represents the previous value of $n$. For example, a node $\text{sum}$ that sums up a source $a$ can be written as

$$\text{node init}[0] = a + \text{sum} @ \text{last}$$

where $\text{init}[0]$ is a specifier of the initial value of the node. At the very beginning of the program execution, the value of $\text{sum} @ \text{last}$ is 0. In this example, $\text{sum} @ \text{last}$ is used within the definition of the node ($\text{sum}$) to refer to the previous value of the node itself. However, the operator $\text{@last}$ can be applied to arbitrary nodes in the program.

Although the syntax of XFRP does not allow expressions like $n @ \text{last} @ \text{last}$, the equivalent expression can be defined by introducing another node as:

$$\text{node n1 = n @ \text{last}}$$
$$\text{node n2 = n1 @ \text{last}}$$

where $n1$ is the previous value of $n$ and $n2$ is the next-to-last value of $n$.

The concept of $\text{@last}$ originates from a higher-order function known as $\text{foldp}$ existing in some FRP platforms, which takes a starting value and an accumulation function and calculates a new value on each change (See [6] for a detailed description). As discussed above, $\text{@last}$ can be applied to arbitrary nodes and thus is more general than $\text{foldp}$. We also
3.3 Syntax

Figure 3 shows the abstract syntax of XFRP. We let the metavariable \( i \) ranges over identifiers used unique in a program, \( n \) ranges over identifiers used as node names, \( p \) ranges over pattern expressions (e.g. \((a, b)\) or \(x::xs\)), and \( c \) ranges over constants. Constants include numbers, boolean values (True or False), unit\(^1\), constant names, and tuples or arrays of constants. Expressions are similar to those of ML, but notice that closures (functions) are absent because it cannot be used as time-varying values. Instead, named function declarations like that of C or Java are introduced. The node names (references to nodes) can only be used in node definitions because functions and constants should be pure and deterministic over time.

XFRP is a statically typed language with type inference. Thus, the type annotations in definitions can be omitted if these types are deducible. Unlike some FRP languages that have special (builtin) types for time-varying values such as Signal[T] (in REScala, for example), XFRP does not provide such dedicated types at least in the syntax as in Emfrp. In XFRP, we can use names of time-varying values as if they were normal (non-time-varying) values. In other words, we do not need to use explicit lift functions that map normal values to time-varying values. This design greatly simplifies the programming process in XFRP. The formal rules of the type system of XFRP are out of the scope of this paper, and it will not be discussed further.

3.4 Execution Model

This subsection describes the execution model of XFRP, which is based on the Actor model. In contrast to our previous work\cite{24} that introduced an actor-based runtime system for resource-constrained uniprocessor systems, we consider distributed execution in this work.

In XFRP’s runtime, each time-varying value is realized by one actor. These actors are initialized when and only when the program starts. The propagation graph is fixed during execution, although the network topology in the Actor model, in general, is dynamic. The dynamicity in this language remains future work.

When a time-varying value changes, a message with the changed value is sent to all the nodes (actors) that depend on the value and the receiver nodes recompute their expression and also send the results to the downstream (Push-based FRP). The way of communication between actors is asynchronous (no waiting for an acknowledgment) and has no requirement on the order of message arrival, which models delaying or flipping packets in a network. Thus, the order of sink values may differ from that of the sources. For instance, if the order of an input is \(1 \rightarrow 2 \rightarrow 3\), the output may be \(2 \rightarrow 3 \rightarrow 1\) or \(1 \rightarrow 3 \rightarrow 2\). For further discussion on the realignment of arrival messages, see Section 9.2.

In this paper, we do not deal with node failure (crashes) or message dropping, namely, nodes are persistent and messages eventually reach the destination in this model. Section 9.1 discusses this topic.

The notion of an iteration, i.e., an end-to-end synchronous evaluation cycle, which exists in some FRP languages such as Emfrp\cite{21}, does not in XFRP. In such an iterative model, inputs are processed one by one throughout the system, that is, a new input value does not propagate in the graph until the previous one reaches the sink. In contrast with such synchronous executions, in XFRP, nodes process input values asynchronously without waiting for messages to reach the destination. In a distributed setting, this execution model can make the latency of a system lower, compared to the iterative model.

3.5 Host Specifier

In XFRP, we can specify the hostname of a machine where nodes are to be placed. The format of hostnames is platform-dependent. In our compiler (Section 7), they are written
We introduce some dependency relations on time-varying values of sources, sinks, and nodes, which realizes glitch-freedom. A nodes to track the happened-before relation between changes of zero.

Each source has its own counter which is incremented when a new value is produced. A

\[
\text{(a@hostname) } x : \text{Int}, \ y : \text{Int}, \ \text{(b@hostname) } z : \text{Int}
\]

... \[\text{(a@hostname) } \text{node} \ r = x \ast y \ \text{(b@hostname) } \text{node} \ s = z
\]

Figure 4. Host Specifiers

like name@host in accordance with the specification of a distributed Erlang system[1].

Host specifiers are placed before the definitions of nodes and sources as in Figure 4. An unspecified node inherits the host of the previous definition. If any host is not specified for the nodes, a system works on a single local node.

4 Preliminaries

4.1 Dependency

We introduce some dependency relations on time-varying values. Let \( n_1 \) and \( n_2 \) be time-varying values (sources, sinks, or nodes). We use \( n_1 \rightarrow_c n_2 \) to express that the name of \( n_1 \) occurs in the definition (expression) of \( n_2 \). Similarly, \( n_1 \rightarrow_L n_2 \) means that the expression \( n_1@\text{last} \) occurs in the definition of \( n_2 \). For example, if \( x \) is defined as \( \text{node} \ x = y + z@\text{last} \), both \( y \rightarrow_c x \) and \( z \rightarrow_L x \) hold.

We also define the relation \( \rightarrow \) as the union of \( \rightarrow_c \) and \( \rightarrow_L \), i.e., \( n_1 \rightarrow n_2 \Leftrightarrow n_1 \rightarrow_c n_2 \lor n_1 \rightarrow_L n_2 \). The binary relations \( \rightarrow_c \), \( \rightarrow_L \), and \( \rightarrow \) are not transitive; e.g., \( n_1 \rightarrow n_2 \) and \( n_2 \rightarrow n_3 \) does not imply \( n_1 \rightarrow n_3 \). As usual, \( \rightarrow^* \) indicates the reflexive and transitive closure of \( \rightarrow \). The roots of a node are sources whose changes affect it. Let Source be the set of sources in a program. The set of roots of node \( n \) is defined as [Root(n)] = \{ \( i \in \text{Source} \mid i \rightarrow^* n \} \). Note that [Root(i)] = \{ \( i \} \) holds for any \( i \in \text{Source} \). This means that a source does not depend on other sources or nodes.

The dependency graph of an XFRP program is a directed graph whose vertices are the union of sources, sinks, and nodes, and edges are represented by \( \rightarrow \). By restricting the edges of the graph to \( \rightarrow_c \), we should obtain a DAG. In current XFRP, the graph is constructed at compile time and it does not change its structure at runtime. The dynamic evolution of the dependencies is left for future work.

4.2 Versions

Each source has its own counter which is incremented when a new value is produced. A version is a pair of the identifier of a source (root) and its counter, e.g., the version of a source named "pulse" that has produced values three times is represented as (pulse, 3). The number of a version starts from zero.

Versions are attached to messages propagated through nodes to track the happened-before relation between changes of sources, sinks, and nodes, which realizes glitch-freedom. A message has exactly one version that is not modified before it reaches a sink except that source unification (explained in Section 6) occurs.

4.3 Data Format

As described in Section 3.4, in the runtime of XFRP, each node (or source, sink) is represented as an actor. The state of such an actor (say \( n \)) is a triple (Buffer\(_n\), Last\(_n\), Deferred\(_n\)). Buffer retains the input values that have been received but have not yet been processed. It is an associative array that maps a version (see Section 4.2) to the fields of values. Last is the fields of values that was received last time. When a new value arrives, the values that are independent of the roots of the received value are used to compute. Deferred is a list of root identifiers and used in nodes with two or more roots to wait for the arrival of all the required inputs.

The format of a message is a triple of the identifier of a sender node, a current value of the node, and its version. For example, if a node \( n \) changes its value \( v \) by receiving a new input with a version \((i, 1)\), it sends a message \((n, v, (i, 1))\) to downstream.

5 Updating Algorithm

This section describes the change propagation algorithm used in XFRP runtime. Our algorithm, shown in Algorithm 1, is similar to the DREAM algorithm (the single-source glitch-freedom variant)[15] in the following points: "attaching versions to time-varying values", "the guarantee of glitch-freedom", "the message-passing system model", and so on. The main differences are the explicit handling of a (last) state and the capability for out-of-order messages.

The algorithm consists of two parts. The former is the initialization of each actor performed just after it is spawned. The latter is the behavior of an actor that continuously runs on each node. The procedure is divided into two parts: 1) matching a set of messages from the Buffer, calculating the change and propagating the result again and 2) waiting for a message and storing it into the Buffer. The reason why receiving comes after matching is to avoid deadlocks that may happen in a situation like Figure 5a. If \( y \) starts receiving at first in this program, it waits forever and no one in the system can proceed because \( x \) also waits for \( y \) but \( y \) sends nothing until a message arrives. Conversely, if \( y \) starts from a matching phase, the field \( x@\text{last} \) is matched in the initialized Buffer of \( y \) first, then it sends the result to \( x \) and also to \( y \) when it receives a signal from \( n \).

The behavior is written in a process-based Actor style like Erlang, which is one variation of the Actor model and uses "receive" as a blocking operation[12]. However, the program can be easily adapted for other models of Actors such as Akka’s event-based non-blocking receive.

Remember that "\{(k \map v)\}" is a notation of an element of an associative array whose key is \( k \) and value is \( v \) and
Algorithm 1 XFRP for Actors

Procedure 1: Initialization
1: for all \( n \in \text{Node}' \) do
2: for \( s \in \text{Source} \) do
3: if \( m_L \in \{(m \in \text{Node}') | m \rightarrow_L n \land s \rightarrow^* m\} \) do
4: \( \text{Buffer}_n[(s, 0)] += \{m_l \@ \text{Last} \mapsto \text{(init value of } m_L)\} \)
5: end for
6: end for
7: end for

Procedure 2: For each node (including sink node) \( n \),
1: \( O := \{o \in \text{Node}' | n \rightarrow o\} \)
2: loop
3: // Phase 1: Matching
4: for all \( \{(s, v) \mapsto M_{sv}\} \in \text{Buffer}_n \) do
5: \( M := \{i \in \text{Node}' \cup \text{Source} | i \rightarrow n \land s \rightarrow^* i\} \)
6: if \( M = \text{keys}(M_{sv}) \) then
7: \( R := \text{Last}_n \setminus M \)
8: if \( R \cup M = \{i \in \text{Node}' \cup \text{Source} | i \rightarrow n\} \) then
9: \( e := \text{compute}(R \cup M) \)
10: send \((n, e, d)\) to \( O \)
11: if \( \text{Deferred}_n \neq \emptyset \) then
12: for \( d \in \text{Deferred}_n \) do
13: send \((n, e, d)\) to \( O \)
14: end for
15: Deferred_n := \emptyset
16: end if
17: \( \text{Buffer}_n \leftarrow \text{Buffer}_n \setminus M_{sv} \)
18: \( \text{Last}_n \leftarrow \text{Last}_n \cup M_{sv} \)
19: else
20: \( \text{Buffer}_n \leftarrow \text{Buffer}_n \setminus M_{sv} \)
21: \( \text{Last}_n \leftarrow \text{Last}_n \cup M_{sv} \)
22: Deferred_n := Deferred_n \cup \{(s, v)\}
23: end if
24: end if
25: // Phase 2: Waiting for Receive
26: \( \{(r_i, r_e, (r_s, r_o)) := \text{receive}\} \)
27: if \( r_i \rightarrow_C n \) then
28: \( \text{Buffer}_n[(r_s, r_o)] += \{r_i \mapsto r_e\} \)
29: end if
30: if \( r_i \rightarrow_L n \) then
31: \( \text{Buffer}_n[(r_s, r_o + 1)] += \{r_i \@ \text{Last} \mapsto r_e\} \)
32: end if
33: end if
34: end loop

(Note: Node’ = Node \cup \text{Sink})

“Buffer\(_n[(v, i)] +\{k \mapsto v\}” has a meaning of appending a value \( v \) to a field \( k \) of a version \((v, i)\) in Buffer\(_n_. \)

The rest of this section focuses on these procedures and phases in detail.

5.1 Initialization
In the initialization, the initial values of nodes are set on the nodes that use them with \( \text{@Last} \) in their definitions. For example, the initialized Buffer\(_d\) in a program as described in Figure 5(b) is

\[
\{(in1, 0) \mapsto \{x@\text{Last} \mapsto x_{\text{init}}\},
\{in2, 0) \mapsto \{y@\text{Last} \mapsto y_{\text{init}}\}\},
\]

where \( x_{\text{init}} \) and \( y_{\text{init}} \) are the initial values of them.

5.2 Phase 1: Matching from Buffer
In Phase 1, an actor searches a set of input values that are enough to evaluate the expression from its Buffer and then sends the updated value to the nodes depending on it. Matching (Line 4) is performed over every entry of Buffer ordered by the colexicographic ordering of version pairs: \((n, v) < (n', v') \iff (v < v') \lor (v = v' \land n < n')\).

Three cases of conditions as below are possible for each version in Buffer:

1. The input values whose root is this version are insufficient to calculate the update,
2. The input values whose root is this version are sufficient but the rest of the values in this node are insufficient,
3. All the values are sufficient and ready to be calculated.

In the first case, it does nothing and (conceptually) waits for further messages to arrive. In the second case (Line 20-22), the process of evaluation and propagation is delayed until this case proceeds to the third one by adding the current version \((s, v)\) to the list Deferred (Line 22). At this time, the entry of this version is removed from Buffer (Line 21). In the last case (Line 7-18), an actor evaluates the expression by using the values of the field in Buffer \( M \) and the ones whose roots are independent of this version \( R \) (Line 9). After that, it sends the result to the depending nodes (Line 10). If there are delayed versions, it also sends the values with these versions (Line 11-16). The entry of this version is removed from Buffer and Last is updated at the end (Line 17-18).

5.3 Phase 2: Receiving Message
After a matching phase, an actor waits for messages. When a message comes, it is stored in the field corresponding to the version in the Buffer (Line 29, 32). A node value used with a \( \@ \text{Last} \) operator is placed on the next version of it. In Figure 5b, for example, if \( d \) receives \((x, 10, (in1, 3))\) from

\[
\text{Figure 5. Graph of Dependency with \@Last}
\]
the node \( x \), a map \( \{ x \mapsto 10 \} \) is added to the field with key \((\text{in}1, 4)\) in Buffer\( y \).

6 Problem of @last Operator

6.1 Inconsistency with @last

As the model adopts single-source glitch-freedom in values and node states, there is a problem of the nondeterminism in @last versions, which leads to inconsistencies.

Let us take an example (Figure 6). The node \( x \) merges two inputs \( a \) and \( b \), and \( y \) refers to the last value of \( x \) and also these sources. We assume that both \( a \) and \( b \) send their sequential counters (e.g., 0, 1, 2, …) and they have already emitted initial values 0. Then, Buffer\( y \) becomes

\[
\{(\text{in}1, 1) \mapsto \{x@last \mapsto (0,0)\}, (\text{in}2, 1) \mapsto \{x@last \mapsto (0,0)\}\}.
\]

In Buffer\( y \), a field \( x@last \) exists in two different versions. This type of branches of versions can cause an inconsistency. In fact, if another value 1 is sent from \( \text{in}1 \) and two nodes update the change, Buffer\( y \) becomes

\[
\{(\text{in}1, 2) \mapsto \{x@last \mapsto (1,0)\}, (\text{in}2, 1) \mapsto \{x@last \mapsto (0,0)\}\},
\]

and it is said that two different @last values of \( x \) exist in a node \( y \). The problem is caused because of the requirement of glitch-freedom per source but this counterintuitive state may lead to an unexpected behavior.

To guarantee single-source glitch-freedom with stateful nodes, we impose a limitation that the number of the roots of a node that is used with a @last expression is at most one. Formally, it is required to satisfy the following equation:

\[
\forall n \in \text{Node}. \left( \exists m \in \text{Node}. n \rightarrow_L m \right) \implies |\text{Root}(n)| \leq 1
\]

where \(|X|\) is a cardinal number of a set \( X \). The dependency graph of Figure 6 apparently violates this, while that of Figure 5b is valid because the root of \( x \) and \( y \) is unique.

6.2 Source Unification

The limitation of @last is considered so strong that it becomes difficult to implement practical applications. Then, a mechanism of merging multiple inputs into one single source is introduced in order to avoid the constraint, and we call it source unification.

Programmers specify which inputs to be unified explicitly in a code, like Line 5 in Figure 8. A unifying source node is placed just after the target inputs internally as illustrated in Figure 7. The node receives messages from the original inputs and forwards to the nodes which depend on them with replacing the versions of them. A unifying node has its own version counter that is attached to an outgoing message and it is incremented when a new message is sent. Also, it is expected to realign received messages in causal order before forwarding. For example, the inputs \( \text{in}1 \) and \( \text{in}2 \) emit values: \( v_1 \) with a version \((\text{in}1, 0)\), \( v_2 \) with \((\text{in}1, 1)\), \( v_3 \) with \((\text{in}2, 0)\), and \( v_4 \) with \((\text{in}2, 1)\), then even if the node unifies receives them in any order, the node sends from \( v_1 \) to \( v_2 \) in sequence with changing the versions of them into \((\text{unified}, 0)\), \((\text{unified}, 1)\), \((\text{unified}, 2)\), and \((\text{unified}, 3)\). Formally speaking, unification transforms partially ordered versions into linearly order, which enables a stateful node to determine “its last value.”

7 Implementation

Our compiler of XFRP\(^3\), which is implemented in OCaml, translates a module into a single-source Erlang code. It generates the behaviors (functions) of actors (processes) for each node and input, a procedure of initialization, and the user-defined functions that provide values to the sources (\text{in}/1\(^4\)) and receive values from the sinks (\text{out}/2). The function \text{in}/1 takes one argument of the hostname of sources and is defined for every host to give values to the source in it. Normally the function is infinite recursive and continuously emits input values. The function \text{out}/2 takes two arguments of the name of a sink and a value provided by it. The function is called every time the value of the sink changes. Examples of these functions are shown in Section 8.

The distribution mechanism places actors on hosts connected via TCP/IP networks. The name of a host is specified as explained in Section 3.5 and used in the first argument of spawn, which is a builtin function that creates a process at a specified host. When an actor is spawned, it waits for the other actors to be initiated before it starts the process of Algorithm 1.

8 Example Application

Figure 8 is an example code of a simple collaborative real-time editor. As shown in Figure 9, the system consists of two clients and one server. Each client has a physical keyboard as

\(^3\)https://github.com/45deg/distributed-xfrp

\(^4\)A function named \( f \) with arity \( N \) is often denoted as \( f/N \) in Erlang.
a source and the server has a sink that is a board displaying the editing text along with two cursors corresponding to the clients.

The source named keyN corresponds to the keyboard for the client N (N = 1, 2) and a stateful node posN represents the cursor position (row, column) of client N, which can be moved by arrow keys. The node buffer has a list of merged inputted characters with their positions.

Since the node buffer has a @last reference, the roots of the node (key1 and key2) should be unified (Line 5 in Figure 8) as discussed in Section 6.2. If they are not unified, the history of buffer will have two branches corresponding to both roots. This implies that as if there were two independent buffers per client. In this case, the board only displays one of the buffers at a time. When there is an input from the client corresponding to the other (hidden) buffer, the board immediately switches to it. However, this is not the expected behavior of the collaborative editor.

Figure 8. CollaborativeBoard code

```erlang
module CollaborativeBoard
in
  (client1@hostname) key1 : Char,
  (client2@hostname) key2 : Char,
  unify(key1, key2)
out
  board : (([Int, Int], [[(Int, Int), Char]]))
fun assoc_update(key, value, list) = ...% list #{ key => value }
  fun movebykey(pos, key) = ...
  fun updateBuf(pos, key, buffer) = ...
  fun update_buf(pos, key, buffer) =
    if (' ' <= key && key <= '~') % is printable?
      then assoc_update(pos, key, buffer)
      else buffer
  (client1@hostname)
  node init[[8,8]] pos1 = movebykey(pos1@last, key1)
  (client2@hostname)
  node init[[8,8]] pos2 = movebykey(pos2@last, key2)
  (server@hostname)
  node init[] buffer =
    let b1 = update_buf(pos1, key1, buffer@last) in
    let b2 = update_buf(pos2, key2, b1) in
    node board = ([pos1, pos2], buffer)
```

Figure 9. Graphical Representation of CollaborativeBoard

Here we focus on the expressions of nodes. Appeared in pos1 and pos2, which holds the positions, a function movebykey takes a position and a key code and returns a modified position if an arrow key is pressed. Otherwise, it returns the position without changes. assoc_update(key, value, list) is a utility function that returns the list of pairs list with replacing the second element of a pair whose first element is key with value. That is, assoc_update(k,v, [..., (k,v0), ...]) = [..., (k,v), ...]. By using this helper function, update_buf(pos, key, buffer) replaces a character in the position pos in buffer if the input key is a printable ASCII character. The buffer is updated by inputs from two and calculating with update_buf.

To provide inputs and display the result, we need to implement external I/O capabilities in Erlang. Figure 10 is a code of example I/O definitions. The input functions in/1 and in_loop/1 poll key inputs after sending a dummy value, which lets the inputs of buffer sufficient. Note that these functions are distributed and invoked on every client. The function out(board,V) is called when the value of board is changed. The first argument of an output function is a atom (a literal constant in Erlang) of the name of a sink, so programmers can handle with the value of each sink by pattern matching with the argument.

9 Discussion

9.1 Handling of Message Losses

The execution model proposed in this paper can accept the out-of-order message delivery but cannot handle the lost of messages. Even a single message drops, the computations in the system may stop, especially a @last operator is in the definitions.

One of the solutions for this issue is to introduce an automatic repeat-request mechanism. In this protocol, a sender waits for an acknowledgment from the receiver after it sends
9.2 Realignment

In some applications, it is expected that the output of the system should be ordered by happened-before relations. It is easy to modify the algorithm for this requirement. One solution is to make a status field which holds the latest versions where the update have been processed and to add another requirement to a matching phase: it accepts only a message whose version is the next of the latest version in the node. However, it can be achieved by the realignment of the sink nodes only instead of all the actors. A question is which is efficient with respect to latency and buffer usage. We conducted a simple experiment for that.

In this experiment, a system has one source, one sink, and 100 nodes between them. The intermediate nodes are randomly connected by current node references and have no cycles. In the network, the source sends 100 messages at the same time and all nodes postpone sending until 0-1000 milliseconds to simulate network delay. We measure latency (the time interval of the propagation from the source to the sink) and buffer size (the total number of pending versions in every Buffer.) for two methods: the realignment mechanism works (A) on each node and (B) only the sink node. The evaluation was conducted on a single machine (MacBook Pro Retina 13-inch Early 2015, 2.7 GHz Intel Core i5, 8GB RAM), using Mac OS X 10.13.6 and Erlang/OTP 20.

Figure 11 shows the result of them. The average latency of the (B) is about 1.54 seconds smaller than (A). In (A), the interval between the first and 100th message arrival time is 1.918 seconds, while it is 0.961 seconds in (B). In addition, the buffer size of (A) is nearly 1500 at the peak, while that of (B) is smaller than 1000.

The result shows that the realignment should be done in the sink nodes. However, it is possible that realignments in all nodes are efficient in some complex graphs. Hence, more intensive research is needed for this topic.

10 Related Work

A plenty of reactive programming platforms or languages have been proposed for many years [3]. We mainly focus on those that are aimed at distributed systems in this section.

DREAM [14, 15] is a distributed reactive middleware that provides elective consistency models: FIFO, causal, single-source glitch freedom, and complete glitch freedom, but they assume that all messages are delivered in a FIFO order.

REScala [20] is a functional reactive library implemented in Scala. SID-UP (Source Identifier Update Propagation) [8, 9] is an efficient propagation algorithm for distributed reactive programs in REScala and it supports complete glitch freedom while the execution model is iterative, as discussed in Section 3.4.

Recently, a new method for REScala is proposed [17] and it provides fault tolerance for distributed reactive programming with reasonable performance. Besides, Myter et al. [18] proposed another method for handling partial failures in distributed reactive systems. However, these methods focus on node crashes rather than on network inconsistencies.

Quarp [19] is a mechanism for distributed reactive programming formalized by operational semantics. It appends meta-information to messages between nodes capturing the context of the value. By giving the version of the value to the context, glitch freedom can be achieved. However, it does not have an operation equivalent to @last.

11 Conclusion

In this paper, we proposed a new distributed functional reactive programming language XFRP and an updating algorithm...
for the actor-based runtime of the language. The algorithm guarantees single-source glitch-freedom in a distributed system with the existence of out-of-order delivery of messages. An example shows that practical distributed reactive applications can be easily created in XFRP. The discussion about retransmission implies the possibility that the proposed algorithm can be extended to cope with other message delivery models.

There are several tasks left for future work. One is the improvements in language features. Since nodes in a system are static in the current XFRP specification, they cannot be duplicated or moved to another machine and have no mechanism for joining and leaving the network. Such features may extend the range of applicability of this language, e.g., decentralized multiparty services, massive-scale sensor networks, or real-time stream processing. Another issue is to make the algorithm capable of other fault models such as node crashes or message missing. To cope with these faults while keeping glitch-freedom is an important task for future research.

Acknowledgments

This work is supported in part by JSPS KAKENHI Grant No. 18K11236.

References