日本ソフトウェア科学会第 39 回大会 (2022 年度) 講演論文集

A Complete Glitch-Free Propagation Algorithm for Distributed Functional Reactive Programming

Ju Guiyuan　Sosuke Moriguchi　Takuo Watanabe

Functional reactive programming (FRP) provides a good abstraction for developing reactive programs. Because many distributed applications are reactive, FRP should be beneficial for distributed systems. However, undesirable phenomena called glitches can happen due to the asynchronous nature of distributed systems. A glitch is a temporal inconsistency that can occur in the value propagation in FRP. Many propagation algorithms have been proposed to solve the problem, each with its advantages and disadvantages. This paper presents a new value propagation algorithm for a distributed functional reactive programming language. This algorithm provides a uniform method to guarantee single-source and complete glitch freedom. It performs well without using distributed locking by introducing pulse nodes that effectively act as a global clock. Furthermore, it provides fault tolerance for pulse nodes. We evaluate the performance of the algorithm empirically and compare it with other algorithms in a simulated distributed setting.

1 Introduction

A *reactive system* is a computational system that continues interaction with its environment. It constantly updates its state and produces the output based on the changing inputs. Many software systems are reactive: *i.e.*, GUIs react to user inputs, and IoT networks react to sensor inputs. *Reactive Programming* (RP) is a programming paradigm that supports building reactive systems based on time-varying values and their change-propagation [1]. It provides programmers with a better abstraction to program reactive systems. For example, it solves the inverted control flow problem that is common in the observer pattern [5]. *Functional Reactive Programming* (FRP) [4] is a variant of RP, it provides a declarative way to program reactive systems.

There are many existing implementations of (F)RP, such as Elm [2] and React [8] in the field of Web GUI. However, these implementations mainly target non-distributed systems. While nowadays distributed systems have become increasingly important, it is beneficial to introduce FRP into the programming of distributed systems. Unlike non-distributed systems, it is more challenging to realize *distributed FRP* (DFRP) safely and efficiently. The notion of *glitches* is one of the problems that make DFRP difficult. A glitch is a temporal inconsistency that can occur in the value propagation in FRP. While it is easy to make non-distributed FRP systems glitch-free, providing such property in distributed settings takes more consideration. Although there have been many approaches to resolve glitches in distributed (F)RP systems [10] [3] [6] [7], each has its limitations. This paper introduces a new value propagation algorithm for a distributed functional reactive programming language called Distributed XFRP. It reduces average message latency and produces more valid updates than existing algorithms.

The rest of the paper is organized as follows. The next section has an introduction to the concept of Distributed FRP (DFRP), followed by the classification of glitches and glitch-freedom in DFRP. Section 3 presents our propagation algorithm called PSNK, and Section 4 briefly explains existing algorithms for comparison purposes. Section 5 de-

This is an unrefereed paper. Copyrights belong to the Author(s).

居桂園, 森口草介, 渡部卓雄, Department of Computer Science, School of Computing, Tokyo Institute of Technology, 東京工業大学 情報理工学院 情報工学系.

```
module FanController
in  tmp: Float,      % temperature sensor
    hmd: Float,      % humidity sensor
out fan: Bool        % fan switch

% threshold
const th = 75

% discomfort index
node di = 0.81 * tmp + 0.01 * hmd *
          (0.99 * tmp - 14.3) + 46.3

% fan status
node fan = di >= th
```

Code 1 Distributed XFRP Source Code

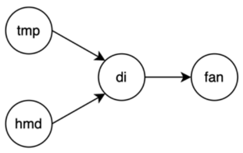


Fig. 1 Graphical Representation

scribes the evaluation of our algorithm based on several simulations, and Section 6 discusses their results. Section 7 examines related work, and Section 8 concludes the paper with future directions.

2 Distributed FRP

2.1 Language Distributed XFRP

An FRP program can be described as a dependency graph, where nodes represent time-varying values, and arcs represent data dependencies. This model can naturally be combined with the Actor model to expand it into distributed systems [11]. However, the properties of distributed systems give birth to many implementation difficulties, notably the glitch problem. Here we take a language called Distributed XFRP [10] as an example of DFRP. Distributed XFRP is a purely functional reactive programming language for building distributed applications, it is based on another FRP language called Emfrp [9] designed for non-distributed embedded systems. The implementation of Distributed XFRP avoids a certain type of glitch called single-source glitch.

Code 1 is an example from the paper [10], which describes a fan controller module that depends on a temperature sensor and a humidity sensor. The fan status is controlled by the discomfort index which is computed from the current value of the temperature and humidity sensor. If the discomfort index is higher than the threshold, the fan is turned on, otherwise turned off. Fig. 1 shows the dependency of the time-varying values, an arrow from A to B means A is in the definition of B. Each node in this dependency graph can be distributed into different locations, forming a distributed system. The actor model is utilized as the underlying model of each node, enabling nodes to communicate by sending and receiving messages.

This example code is a simple case of DFRP, when dependency becomes more complex, the so-called glitch will happen.

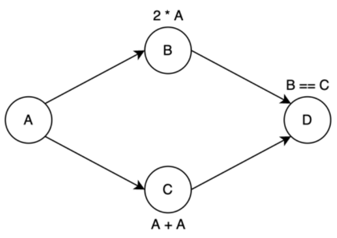


Fig. 2 Single-Source Glitch

2.2 Glitches and Glitch-Freedom

Glitches are temporal inconsistencies that occur during the propagation of changes. According to [6], glitch freedom is classified into two types: single-source glitch freedom and complete glitch freedom, we call a violation of them single-source glitch and concurrent glitch respectively. Glitches may happen in both FRP and DFRP, but due to the unreliable connections and concurrent updates in distributed systems, it is difficult to implement a glitch-free system efficiently in a distributed setting. In the coming sections, we are going to explain what is a single-source glitch and concurrent glitch, along with common solutions in both non-distributed and distributed settings.

2.2.1 Single-Source Glitch Freedom

Fig. 2 shows a graph susceptible to single-source glitch. Node `A` represents a source node, source nodes are the sources of updates, for example, it may be a sensor device that produces messages pe-

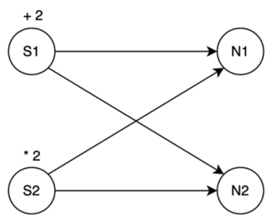


Fig. 3 Concurrent Glitch

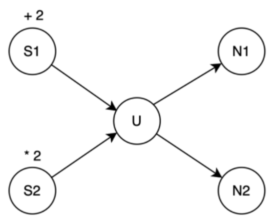


Fig. 4 Source Unification

riodically and send them to subsequent nodes, here source `A` produce integers, node `B` computes the result of `2 * A`, and node `C` computes the result of `A + A`, since they should always be equal to each other, the value of node `D` should always be `true`. However, depending on the update order, such invariant might be violated. For example, source `A` sent a value 2, if node `D` observe the result from node `B` earlier than node `C`(because of the different network latency), then the value of node `B` is 4, while node `C` is 0 (suppose the initial value of `A`, `B` and `C` is 0), thus the value of node `D` becomes `false` temporarily, this is called single-source glitch.

For non-distributed FRP(single thread), single-source glitch can be easily solved by doing topological sorting on the dependency graph, so that whenever updating a node, the values of dependencies are already the newest version. But for DFRP, this method isn't applicable because of its inefficiency. Distributed XFRP took another way to solve this problem, it adds *version* to each message sent by sources, a version includes the identifier of a source, and a counter for this message, by comparing the versions of received messages in node `D`, node `D` can wait until all the matching messages arrived and then compute a new update, through this way, single-source glitch freedom is achieved in Distributed XFRP.

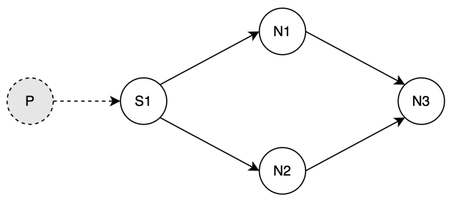
2.2.2 Complete Glitch Freedom

While a single-source glitch is about a single source, a concurrent glitch is about concurrent updates from multiple sources. Consider a graph like Fig. 3, source `S1` send a command `+ 2` to node `N1` and `N2`, which plus 2 to the current values; source `S2` send a command `* 2` to node `N1` and `N2`, which multiply current values with 2. Depending on the receiving order of messages observed on node `N1` and `N2`, the resulting sequence of updates may be different from each other. For example, `N1` observed `+ 2` before `* 2`, so its updates are 2 and then 4 (suppose the initial value of `N1` and `N2` are 0), but `N2` observed `* 2` before `+ 2`, its updates are 0 and then 2, although both `N1` and `N2` depend on the same set of sources, their updates are different from each other, this is called concurrent glitch.

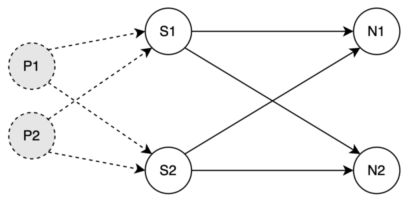
Non-distributed FRP can solve it easily by topological sorting combined with an iterative execution model. Iterative execution means when an update occurs, it will be processed by all the subsequent nodes, during this processing, other updates are blocked. Suppose `S1` generate the update `+ 2` slightly earlier than `S2`, then both `N1` and `N2` receive the command `+ 2`, now the values are both 2, and then the command from `S2` are sent to `N1` and `N2`, both of the values become 4.

For DFRP, it's far more difficult, we don't have a central mechanism to control the execution order of each node, and we cannot block other sources when an update of one source is being processing in the system without degrading the performance. Distributed XFRP doesn't guarantee the property of complete glitch freedom, but it provides a possible solution to it — Source Unification.

In Fig. 4, an extra node `U` is added to unify updates from `S1` and `S2`, after unification, the order of updates observed from `N1` and `N2` is always the same, hence no concurrent glitch. It is an effective solution, but it suffers from the single point of failure problem: if the unification node stops working for some reason, the whole system will also stop functioning. If we add multiple unification nodes, the synchronization between these unification nodes is still a problem. Because of these drawbacks, we propose a new propagation algorithm in the next section.



(a) Single-Source Glitch Freedom in PSNK



(b) Complete Glitch Freedom in PSNK

Fig. 5 PSNK

3 Proposed Algorithm

This section presents our propagation algorithm called PSNK. The key idea is to introduce *pulse nodes* that initiate the synchronized propagation from source nodes to sink nodes.

Similar to source unification, in PSNK, we also add extra nodes to the existing dependency graph, but differently, we place them before source nodes. As shown in 5(a) and 5(b), there are one or more nodes placed before sources, we call these nodes pulse nodes. The number of pulse nodes isn't restricted, we can place as many pulse nodes as we want, as long as each of them is connected to every source node.

The algorithm has two parts, one for source nodes, and another one for other nodes except for source and pulse nodes. Both of them require messages to arrive in first-in-first-out order and each message is delivered exactly once.

Pulse nodes send pulses periodically to every source, a pulse doesn't contain any information, no pulse identity or counter. In Algorithm 1 for source nodes, there is a counter that counts the currently received pulse, when the value of the source is updated (e.g. the underlying sensor generates a new value), and if this is the first update concerning current pulse counter, the source sends this update attached with the current count of the pulse.

Algorithm 1 For Source Nodes

```
Require: Messages arrive in order and delivered exactly once.
  for all $n \in$ Source do
    $p_{count} := 0$                    //The count of pulse
    $u :=$ null               //Storing the value of updates
    $p_{changed}, u_{changed} :=$ false
    loop
      $(r_t, r_v) :=$ receive()        //$r_t$: type of message
                                       //$r_v$: value
      if $r_t =$ pulse then
        $p_{count} \leftarrow p_{count} + 1$
        $p_{changed} \leftarrow$ true
      end if
      if $r_t =$ update then
        $u \leftarrow r_v$
        $u_{changed} \leftarrow$ true
      end if
      if $p_{changed}$ and $u_{changed}$ then
        send$(n, u, p_{count})$
        $p_{changed}, u_{changed} \leftarrow$ false
      end if
    end loop
  end for
```

In Algorithm 2 for other nodes, each node holds a queue for each incoming connection, whenever a new message arrives, it will be inserted into the corresponding queue, and the algorithm starts a loop, it will check whether all queues are not empty, if all are not empty, the messages with the smallest pulse count will be dequeued, the dequeued messages are combined with the `Last` buffer, which holds the dequeued messages from the last loop, if the messages in the `Last` buffer are enough to compute a new update (the number of messages is the same as the incoming connections, which means all precedent nodes have a value), a new update will be computed. This loop will continue until one or more queues become empty.

We give a simplified explanation based on the example of Fig. 3. Consider now we add one or more pulse nodes into this graph, suppose the command of `+ 2` in `S1` appears when the pulse counter is 0, and the command of `* 2` in `S2` appears when the pulse counter is 1, according to the algorithm for `N1` and `N2`, no matter when these messages arrive, both `N1` and `N2` will first dequeue the message containing the command `+ 2` and then `* 2` into `Last` buffer, assuming initially there are two `do nothing` commands already exist in the `Last`

Algorithm 2 For Other Nodes

```
Require: Messages arrive in order and delivered exactly once.
  for all $n \in \text{Node} \cup \text{Sink}$ do
    $I_n := \{i \mid i \to n\}$
    $\text{Last}_\text{n} := \emptyset$                    //key: node
                              //value: pulse → update
    $\text{Buffer}_\text{n} := \emptyset$                  //key: node
                        //value: a queue of pulse → update
    loop
      $(r_i, r_e, r_p) = \text{receive}()$           //$r_i$: sender id
                      //$r_e$: update value, $r_p$: pulse value
      $\text{enqueue}(\text{Buffer}_\text{n}[r_i], r_p \to r_e)$
      while for all $i \in I_n$, $\text{Buffer}_\text{n}[i]$ is not empty do
        $\text{pulse}_\text{min} := +\infty$
        for all $i \in I_n$ do
          $p \mathrel{+}= \text{key}(\text{peek}(\text{Buffer}_\text{n}[i]))$
          if $p < \text{pulse}_\text{min}$ then
            $\text{pulse}_\text{min} \leftarrow p$
          end if
        end for
        for all $i \in I_n$ do
          if $\text{key}(\text{peek}(\text{Buffer}_\text{n}[i])) = \text{pulse}_\text{min}$ then
            $Last_n \mathrel{+}= \{i \to \text{dequeue}(\text{Buffer}_\text{n}[i])\}$
          end if
        end for
        if $|\text{Last}_\text{n}| = |I_n|$ then
          $e := \text{compute}(\text{Last}_\text{n})$
          $send(n, e, \text{pulse}_\text{min})$
        end if
      end while
    end loop
  end for
```

buffer, the result will be computed first with `+ 2` and then `* 2`, thus both `N1` and `N2` generate new updates first 2 and then 4.

The pulse nodes essentially act as a global clock. For a single-source glitch, the "clock" information is similar to the version in Distributed XFRP; For a concurrent glitch, every node can coordinate messages consistently.

PSNK doesn't have the same single point of failure problem as in source unification, because it allows to use more than one pulse node, and even if some of the pulse nodes crash, the only effect is that each source node receives fewer pulses, but the received amount of pulses are still the same.

The limitation of PSNK is when the average frequency of pulse nodes is lower than source nodes, some updates from source nodes may be dropped because the algorithm in source nodes only sends the first update for each pulse.

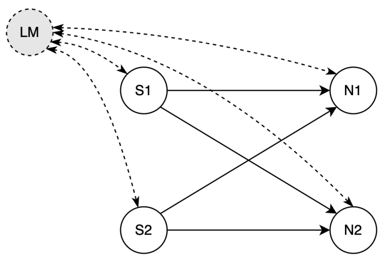


Fig. 6 DREAM

4 Other Algorithms

We introduce two existing algorithms called DREAM [6] and QPROP [7] briefly before the experiments in Section 5 which compare PSNK with them, both of them provide the property of single-source and complete glitch freedom, but each has its drawbacks.

Because both DREAM and QPROP use a similar technique to solve single-source glitches, we only discuss concurrent glitches here.

4.1 DREAM

As shown in Fig. 6, DREAM introduces a coordinator node (lock manager) to manage the lock, every time a source wants to send an update, it needs to acquire a lock from the lock manager, if there is no lock, the update will be sent immediately, otherwise, it needs to wait until the release of all locks.

Every pair of sources in DREAM has a conflict set, which includes all the conflicting nodes (nodes that are susceptible to concurrent glitches) if they send messages concurrently. Every time a source acquired a lock, all the nodes in the conflict set will be locked until each of them received an update from this source, then the lock will be released, and another source can send its update now.

Even although DREAM only locks the smallest set of nodes, the lock still blocks many concurrent updates and harms the overall performance.

4.2 QPROP

Every node in QPROP needs to do a check whenever it receives a new message to avoid glitches. As shown in Fig. 7, assume there is a node `N3` that connect to both `N1` and `N2`. $U_{N1}(val_{S1}, val_{S2})$ de-

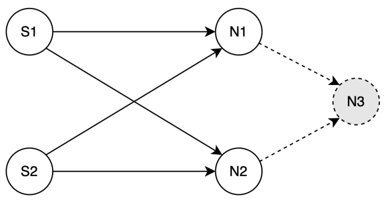


Fig. 7 QPROP

notes the updating of N1 using a value val_{S1} propagated by S1 and a value val_{S2} propagated by S2, $\text{Time}_{S1}(v)$ denotes the S1's clock time (similar to version in Distributed XFRP) tagged to value v. The updating rule is described as below:

$$U_{N3}(val_{N1}, val_{N2}) \iff \text{Time}_{S1}(val_{N1}) == \text{Time}_{S1}(val_{N2}) \land \text{Time}_{S2}(val_{N1}) == \text{Time}_{S2}(val_{N2})$$

Only if this constraint is satisfied, the update $U_{N3}(val_{N1}, val_{N2})$ can be computed and sent.

QPROP doesn't use any lock, the sources can update concurrently, but due to the latency of network connections, a live lock can occur [7], which means almost all of the messages are dropped because they don't satisfy the constraint.

5 Experiments

To evaluate the performance of our proposed algorithm, we did several experiments by simulation. In all simulations, if not specified, the default number of pulses is 2, the default number of sources is 4, the default number of nodes is 100, the default depth of the dependency graph is 10, and each source generates one update every 4 milliseconds and send 50 messages in each simulation, each pulse send one pulse message every 2 milliseconds, and the default network latency is randomly chosen from a Gaussian distribution whose mean is 30 milliseconds and deviation is 10 milliseconds for each connection. The computation in each node takes zero time.

5.1 Valid Update from Sources

The first experiment is only for our algorithm PSNK, because it is possible to drop updates from sources, we need to measure the percentage of the sent updates.

There are 2 lines in Fig. 8, the green line means there is only one pulse node, and the red line represents there are 2 pulse nodes. When the number of pulse nodes are the same, *source sent/updated* increases along with the $f(pulse)/f(source)$ ($f(x)$ means the frequency of x). It means when the frequency of pulses gets higher, fewer messages get dropped. And when $f(pulse)$ equals to $f(source)$, most of messages (over 90 percent) was successfully sent.

The red line climbs faster than the green line, which means with 2 pulse nodes, compared to only one pulse node with the same pulse frequency, fewer updates were dropped.

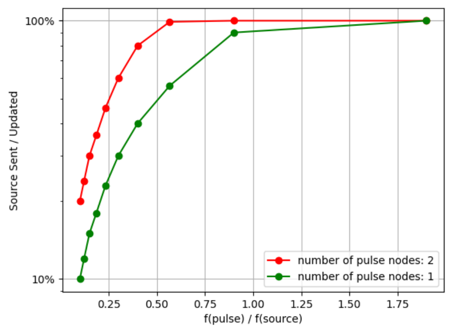


Fig. 8 The Percentage of Valid Update from Sources

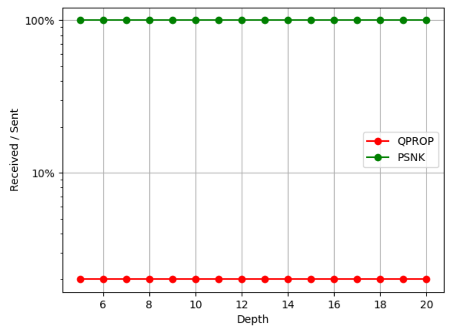


Fig. 9 The Percentage of Valid Update Received in Sink Nodes

5.2 Valid Update Received in Sink Nodes

This experiment evaluates the percentage of received messages in relation to the sent updates from sources in sink nodes(sink nodes are the nodes

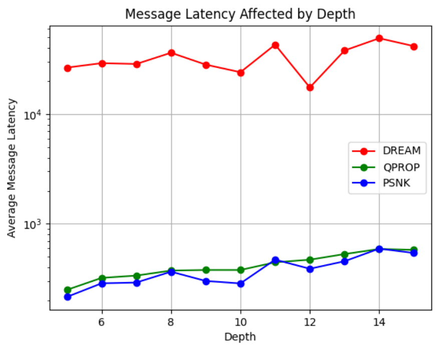


Fig. 10 Average Message Latency Affected by Depth

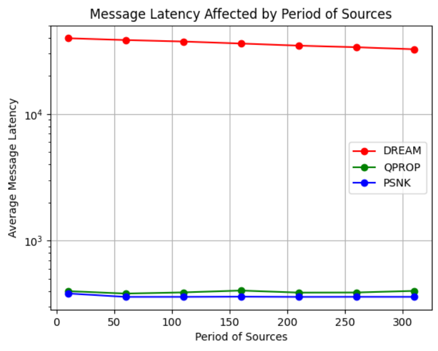


Fig. 11 Average Message Latency affected by Period of Sources

without outgoing connections). We only evaluate QPROP and PSNK, because only they have the chance to drop messages. Message dropping in QPROP is possible to happen in every node, while in PSNK it is only possible in source nodes, and compared to QPROP, PSNK can increase the message frequency of pulse nodes to reduce the message dropping. Since we've already evaluated the valid update from sources for PSNK in the previous section, we use a high enough pulse frequency here to make sure almost no update is dropped in PSNK.

The x-axis in Fig. 9 means the depth of the dependency graph, we can see the percentage of valid updates in nodes of QPROP is around 2%, while that of PSNK is almost 100%,

5.3 Average Message Latency Affected by Depth

From here we start to evaluate all three algorithms. Average message latency means the average latency of every message sent by sources until they arrive sinks in milliseconds. And the depth means the depth of the dependency graph.

From Fig. 10, the latency of DREAM is much higher than both PSNK and QPROP, while PSNK and QPROP are close to each other.

5.4 Average Message Latency Affected by Period of Sources

In this experiment, we evaluated the average latency affected by the period of sources. In Fig. 11, the period of sources ranges from 10 to 310 milliseconds. As we can see, the result is similar, DREAM is worse than QPROP and PSNK, while PSNK and QPROP are close to each other.

6 Discussion

From the experiments, we can see that DREAM performs worse than QPROP and PSNK, this is reasonable, because it uses a central lock manager, and concurrent updates will be blocked if there is a conflicting source sending its update, which decreases the performance of DREAM.

For QPROP and PSNK, the average message latency is similar to each other, however, compared to PSNK with a high enough pulse frequency, QPROP has a much higher possibility to drop messages.

The obvious limitation of our proposed algorithm is that when the frequency of pulse nodes is lower than that of source nodes, updates from sources may be dropped, and actually no matter how high the frequency of pulse nodes is, the possibility of dropping messages always exists. As a result, the current implementation of PSNK is not suitable for distributed systems where every update from sources is crucial.

Furthermore, every pulse node needs to be connected to every source node, the number of connections will grow rapidly when there are more and more pulse and source nodes.

7 Related Work

There are many algorithms, libraries, and languages proposed for reactive programming, either distributed or not. We summarize some of them in this section.

Elm [2] is an FRP language for client-side programming in browsers. It is not designed for distributed systems, but its propagation algorithm can be extended to distributed settings. The extended version ELM^s [3] triggers *no change* messages in all sources whenever one source sends an update, the number of messages is much higher than in other algorithms.

The SID-UP propagation algorithm in a reactive language called Distributed REScala [3] is a distributed algorithm that prevents both single-source and concurrent glitches, but its evaluation model is iterative, which means no other sources can send an update when there is an existing source that is sending an update.

The source unification method from Distributed XFRP [10] can ensure the property of concurrent glitches, but it is hard to add multiple unification nodes, which introduces the single point of failure problem.

DREAM [6] use a lock manager to avoid concurrent glitch, compared with SID-UP, it only prevents conflict sources rather than all sources from updating, thus having a better performance than SID-UP [3], but the existence of a central lock still hurt the performance.

QPROP [7] doesn't have any lock, which allows concurrent updates of all sources, however, the difficulty of satisfying the constraint prevents it from producing more valid updates.

8 Conclusion and Future Direction

We proposed a new propagation algorithm called PSNK that is meant to improve the Distributed XFRP, by using pulse nodes, this algorithm guarantees the property of the single-source and complete glitch freedom. We did some experiments to evaluate its performance compared to other algorithms, and conclude that PSNK produces more valid updates and has a lower latency.

There is some work left to do in the future, notably to find a way to prevent message dropping and to reduce the connections from pulse nodes to source nodes.

Acknowledgments This work is supported in part by JSPS KAKENHI Grant Number 21K11822 and 22K11967.

References

[1] Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S., and De Meuter, W.: A Survey on Reactive Programming, *ACM Computing Surveys*, Vol. 45, No. 4(2013), pp. 52:1–52:34.

[2] Czaplicki, E.: Elm: Concurrent FRP for Functional GUI, Master's thesis, School of Engineering and Applied Sciences, Harvard University, Mar. 2012.

[3] Drechsler, J., Salvaneschi, G., Mogk, R., and Mezini, M.: Distributed REScala: An Update Algorithm for Distributed Reactive Programming, *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2014)*, ACM, ACM, 2014, pp. 361–376.

[4] Elliott, C. and Hudak, P.: Functional Reactive Animation, *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, ACM, 1997, pp. 263–273.

[5] Maier, I., Rompf, T., and Odersky, M.: Deprecating the Observer Pattern, Technical Report EPFL-REPORT-148043, EPFL, 2010.

[6] Margara, A. and Salvaneschi, G.: On the Semantics of Distributed Reactive Programming: The Cost of Consistency, *IEEE Transactions on Software Engineering*, Vol. 44, No. 7(2018), pp. 689–711.

[7] Myter, F., Scholliers, C., and De Meuter, W.: Distributed Reactive Programming for Reactive Distributed Systems, *The Art, Science, and Engineering of Programming*, Vol. 3, No. 3(2019), pp. 5:1–5:52.

[8] React: React: A JavaScript library for building user interfaces, https://reactjs.org, Last accessed: Aug. 2022.

[9] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity*, ACM, Mar. 2016, pp. 36–44.

[10] Shibanai, K. and Watanabe, T.: Distributed Functional Reactive Programming on Actor-Based Runtime, *8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018)*, ACM, Nov 2018, pp. 13–22.

[11] Watanabe, T. and Sawada, K.: Towards an Integration of the Actor Model in an FRP Language for Small-Scale Embedded Systems, *6th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2016)*, Oct. 2016.