Developing Coordinating Distributed Applications in a Pure FRP Language

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

ABSTRACT

This paper discusses how coordinating distributed systems can be expressed using a pure functional reactive programming (FRP) language designed for distributed applications. FRP provides functional abstractions for values changing over time, with which reactive behaviors can be naturally described. Since each computational node in a coordinating distributed system can be seen as a reactive component, FRP is well suited for describing the intra-node computation. Via a case study of a wireless sensor-actor network (WSAN), this paper shows that both inter-node coordination and intra-node computation can be uniformly described using the language.

CCS CONCEPTS

• **Software and its engineering** → *Distributed programming languages*; *Functional languages*; *Data flow languages.*

KEYWORDS

Functional Reactive Programming, Embedded Systems, Coordination, Wireless Sensor-Actor Netowrks

1 INTRODUCTION

Functional Reactive Programming (FRP) is a programming paradigm for developing reactive systems. In FRP, a system is described in terms of continuously changing *time-varying values* and propagation of changes [3]. Originally, FRP is introduced to describe interactive animations in the pure functional language Haskell [6]. Until now, the paradigm and its some non-functional variants gained popularity in various fields such as Web programming, mobile application development, mobile IoT networks, and embedded systems.

The change propagation among time-varying values can be viewed as dataflow computation. From this viewpoint, FRP provides a high-level and declarative abstraction for describing concurrent systems. Thus, integrating FRP with existing concurrent computation models is interesting in both theoretical and practical aspects. The author 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 [18]. 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 [16].

To develop distributed applications, we designed and implemented Distributed XFRP, a statically-typed, purely functional reactive programming language. The runtime system of the language is based on the Actor model [1] and change propagation among time-varying values is implemented as asynchronous message passing. In our previous paper [15], we proposed a new algorithm for change-propagation via asynchronous messages and showed that such a distributed runtime system can be used to implement pure FRP without suffering from the phenomenon called *glitches* (temporal inconsistencies in the change propagation).

The compiler[1] of the language translates a source program into an Erlang program. In contrast to our previous work [18] that introduced an actor-based runtime system for resource-constrained uniprocessor systems, the language actually supports distributed execution of pure FRP programs.

The contribution presented in this paper is that, via a case study, we emphasize that FRP is well suited for describing coordination. We present a wireless sensor-actor network (WSAN) application that controls the air-environment of a long corridor, and show that the behavior of the WSAN, which is a typical coordination task, can be easily written in Distributed XFRP. Moreover, we discuss that the language supports incremental development of such applications.

The rest of the paper is organized as follows. The next section introduces the non-distributed subset of our language to explain some basic notions of FRP. In Section 3, we briefly describe the execution model of Distributed XFRP. Section 4 presents a WSAN case study to emphasize that Distributed XFRP is beneficial for describing coordination. Section 5 surveys related work and Section 6 concludes the paper.

2 XFRP

XFRP is a general-purpose functional reactive programming language developed as a successor of Emfrp [14], which is designed for small-scale embedded systems. In Section 4, we develop a WSAN example using the distributed dialect of XFRP [15]. This section briefly describes a non-distributed subset of XFRP.

2.1 Basics

In XFRP, a system (*module*) is composed of the definitions of time-varying values and other components. Time-varying values are called *nodes* in the language. Nodes are classified in the following categories: *source* (input), *sink* (output), and *internal.* Source nodes (hereinafter referred to as sources) emit externally given values such as keyboard inputs, network packets from another computer, and measurements from a sensor device. Sink nodes (hereinafter referred to as sinks) are the destinations of propagation. They receive

[1] https://github.com/45deg/distributed-xfrp

Listing 1: Fan Controller in XFRP

```
1  module FanController  % module name
2  in  tmp : Float,      % temperature sensor
3      hmd : Float       % humidity sensor
4  out fan : Bool        % fan switch
5
6  % discomfort (temperature-humidity) index
7  node di =
8      0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
9
10 % fan status
11 node init[False] fan = di >= th
12
13 % threshold
14 node th = 75.0 + if fan@last then -0.5 else 0.5
```

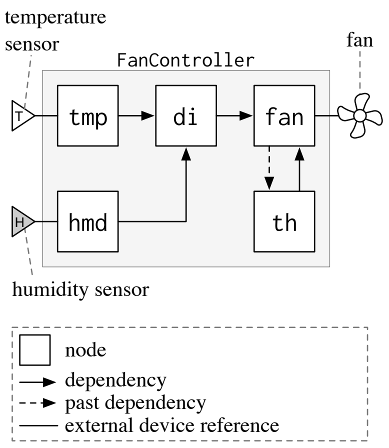

Figure 1: Graph Representation of `FanController`

results and normally affect the outer world by displaying characters, changing the voltage output, etc. Internal 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 functional style, which means it has no side effect or mutable states. In summary, changes of sources propagate throughout internal nodes and finally reach sinks.

Figure 1 shows a simple fan controller example from our previous paper [17]. The program has two sources `tmp` and `hmd` respectively representing the current temperature [°C] and humidity [%] measured by external sensors. The node `di` expresses the current discomfort index (the degree of discomfort experienced by a human). The value of `di` immediately reflects any changes in the sensor readings (`tmp` or `hmd`).

The operator `@last` allows access to the *previous value* (the value at the previous moment) of an arbitrary node. Using this operator, we can define history sensitive (or stateful) behaviors. In Listing 1, `@last` is used to realize a simple hysteresis control that protects the fan motor from frequent switching. The definition of `th` (line 14 in Listing 1) contains the subexpression `fan@last` that refers to the value of `fan` at the previous moment. This means that if `fan` is already true, `th` becomes 74.5 (otherwise 75.5). With such shifts of the threshold, we can avoid frequent changes of `fan` when `di` drifts around the threshold (75.0).

2.2 Execution Model

An XFRP program can be represented as a directed graph whose nodes and edges correspond to nodes (time-varying values) and their dependencies respectively. Figure 1 shows the graph representation of Listing 1, which consists of five nodes and five edges. We categorize the edges (dependencies) into two kinds: *past* and *present*. A past edge from node *m* to *n* means that *n* has *m*`@last` in its definition. A present edge from node *m* to *n*, in contrast, means that *n* directly refers to *m*. In Figure 1, the dotted arrow line from `fan` to `th` is the only past edge. All other edges are present.

By removing the past edges from the graph representation of the program, we should obtain a directed-acyclic graph (DAG). The topological sorting on the DAG gives a sequence of the nodes. For Figure 1, we have: `tmp`, `hmd`, `di`, `the`, `fan`.

The Emfrp runtime system updates the values of the nodes by repeatedly evaluating the elements of the sequence. We call a single evaluation cycle an *iteration*. The order of updates (scheduling) in an iteration must obey the partial order determined by the above mentioned DAG.

The value of *n*`@last` is the value of *n* in the last iteration. At the first iteration, where no nodes have their previous values, *n*`@last` refers to the initial value *c* specified with `init[`*c*`]` in the definition of *n*. In Figure 1, the initial value of `fan` is `False` (line 11).

3 DISTRIBUTED XFRP

We designed and implemented Distributed XFRP [15], a distributed dialect of XFRP. This section briefly introduces some important concepts of the language.

3.1 Glitches

This subsection describes the notion of *glitches* — temporal inconsistencies in the value propagation of FRP systems. Consider the program fragment below.

```
node x = a + a
node y = a * 2
node z = x == y
```

In a glitch-free system, all occurrences of the same node must have the same value at each moment. Thus the value of `z` should always be true. This property does not hold in a system with glitches. For example, consider a situation that the value of `a` changes. Due to possible differences in the propagation times of change, there may be a moment that `x` and `y` have different values.

Margara and Salvaneschi classified glitch-freedom into two types: *single-source* and *complete* [9]. 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. Distributed XFRP supports the single-source glitch freedom.

3.2 Execution Model

The runtime system of XFRP is based on the Actor model [1]. Each source, sink, or internal node corresponds to a single actor. Propagation of updates is performed as asynchronous message passing. When the value of a source changes, the source sends its updated value to other nodes that depend on it. Similarly, when a node receives a message that conveys the updated value, it updates its value and sends the result to nodes that depend on it.

The dependency between nodes is represented as a directed graph, where vertices are node actors and edges are reference relationship between them (for example, if node a has a reference to b, there is an edge from b to a). Note that a cycle in the graph is not permitted if the paths have only non-`@last` references. Every node has roots, which are sources that will effect on the node at their changes. In other words, if there is a path from a source i to a node n in the dependency graph, we can say the node n has the root i.

Each source has a counter that increments when it sends a new value, which is attached to propagation messages with the name of the source to keep track of the happened-before relation between changes. The attached information that is a pair of a source ID and its counter value is called *version*. The relation from version to the value of a node is surjective. We say "the value of node N is V at a version (s, i)" in the sense that the value of N results in V by receiving input messages with a version (s, i), where N is a node, V is a node value, and (s, i) is a pair of a source and its counter. The explicit use of the source IDs in versions enables the single-source glitch-freedom in our language [15].

Each actor has special internal values that represent the computational states. There are three elements: *Buffer*, *Last*, and *Deferred*. Buffer holds the received values before they are calculated. It is a map whose keys are versions and values are also maps which map the depending nodes to their value at the version. Last is a set of latest received input values used to complement values for different versions. Deferred is a list of versions which are received but their processing is postponed because the node lacks the inputs to calculate the expression.

The update algorithm is divided into two phases. One is the matching phase, where an actor collects inputs for calculating the expression from its Buffer. Every versions in the Buffer is scanned to find an entry that is sufficient to evaluate. The entry consists of input values that have the same source, therefore, at evaluation, the values are merged with inputs using other sources using Last and Deferred. For example, consider a node x has an expression $r + s + t$, where r and s have the same source and t has another source, and the Buffer in the node has an entry with the value of r and s in any version. Then x can be evaluated with the value of t in the Last field in x. Second phase is the receiving phase, where the actor waits for a new message from dependent nodes. An arriving message is stored in the Buffer by its attached version. In this time, the value referenced with `@last` is placed at a next Version in the Buffer, which means version $(s, i + 1)$ if the Version of the value is (s, i). The detailed description of the updating algorithm can be found in our previous paper [15].

4 CASE STUDY

This section presents a wireless sensor-actor network (WSAN) described in XFRP. The purpose of this case study is to demonstrate that the language is suitable for describing coordination.

Wireless sensor-actor networks (WSANs) [2, 7] are a variant of wireless sensor networks (WSNs) that contain *actor nodes* in addition to sensor nodes. The responsibilities of actor nodes include controlling actuators, making local decisions, and performing coordination tasks. Note that the term "actor" here is different from the one in the Actor-model.

4.1 WSAN Example

Figure 2 shows a wireless sensor-actor network (WSAN) that monitors and controls the temperature and humidity of the air in a long corridor. The corridor is divided into several segments that are numbered sequentially. Each odd-numbered (even-numbered) segment is equipped with a temperature (humidity) sensor and a temperature (humidity) controller. A temperature (humidity) controller here indicates a special kind of air-conditioner that controls air temperature (humidity). The purpose of this WSAN is to regulate the air environment of the corridor by lowering the difference in discomfort index among the corridor segments.

Now let us describe how it works. Let i (j) be a positive odd (even) integer, and k be a positive integer. The node[2] named $\mathtt{t}_i$ ($\mathtt{h}_j$) is the source node that is connected to the temperature (humidity) sensor located at the i-th (j-th) segment. Similarly, the node named $\mathtt{tc}_i$ ($\mathtt{hc}_j$) is the sink node that is connected to the temperature (humidity) controller located in the i-th (j-th) segment. Note that there is another source node named `th` — not shown in Figure 2 for simplicity — that represents the threshold for determining which output nodes should be activated. The node named $\mathtt{di}_{k(k+1)}$ represents the discomfort index of the in-between area of the k-th and $(k + 1)$-th segments. The node named $\mathtt{ddi}_k$ calculates $\mathtt{di}_{(k-1)k} - \mathtt{di}_{k(k+1)}$, which indicates the degree of imbalance in the data measured by the sensors located in the $(k - 1)$-th and $(k + 1)$-th segments. Thus, if it is larger (smaller) than `th` ($-$`th`), the controller with index $k - 1$ ($k + 1$) is activated to lower the difference.

4.2 WSAN Example in XFRP

Listing 2 shows the XFRP code for the example. The code is a straightforward implementation[3] of Figure 2. For simplicity reason, we omit host specifiers in the code. To deploy the nodes (time-varying values) to appropriate physical sensor/actor nodes (computers) in the WSAN, we can freely put host specifiers at the source node declarations (lines 3 and 5 in Listing 2) or at the node definitions (lines 18–22, 25–28 and 31–36). The single-source glitch-free property of the language guarantees that no temporal inconsistencies can be observed in any deployment configuration.

We demonstrate how the single-source glitch-free property is effective using a simple example scenario. In the following, we fix the value of `th` to 2.0 and assume that `ddi5` $\leq$ `th`. Suppose that `h2` = 70.0, `t3` = 24.0, and `h4` = 75.0. From the node definitions in Listing 2,

[2]We use the term "node" to indicate a time-varying value in XFRP rather than a physical sensor/actor node (computer) in the WSAN.

[3]The current version of the language does not allow indices in node names. So we should write, for example, `di23` for $\mathtt{di}_{23}$ and hence repeat similar definitions. It may not be difficult to add appropriate syntactic support in the future version.

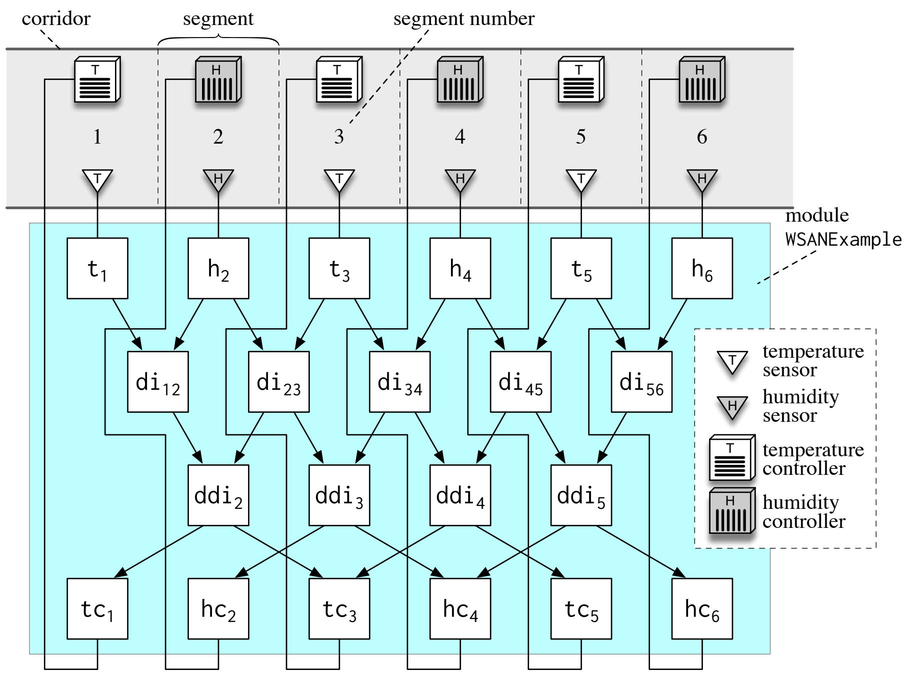

Figure 2: WSAN for Regulating the Air-Environment of a Corridor

we have di23 = 72.36, di34 = 72.84, and ddi3 = -0.47. Thus, neither hc2 nor hc4 is activated because $-0.47 \geq -$th. Now suppose that t3 changes to 25.0 but h2 and h4 remain the same in the next moment. At that moment, thanks to the single-source glitch-free property, both di23 and di34 are guaranteed to observe that t3 = 25.0. Thus di23 = 73.87 and di34 = 74.39 holds, and both hc2 and hc4 are still inactive because ddi3 = $-0.52 \geq -$th. However, without the single-source glitch-free property, it is possible that di23 sees t3 = 24.0 and di34 sees t3 = 25.0 at the same time, and hence di23 = 72.36 and di34 = 74.39 hold. In this case, we have ddi3 = $-2.03 < -$th, and thus hc4 is incorrectly activated.

4.3 Discussion

As the example shows, XFRP provides a declarative way to express inter-node (inter-computer) behaviors of WSNs and WSANs. In other words, the language can be used as a macroprogramming language[12]. Especially, the single-source glitch-freedom enables a convenient way to program WSANs. Because, when merging data from two or more actor nodes that directly or indirectly receive data from the same sensor node, we don't need to be bothered with the synchronization among the actor nodes. Moreover, the single-source glitch freedom is more efficient than complete glitch-freedom such as [8]. If we need the glitch-freedom with regard to multiple sensor nodes, we can use source unification feature provided by the language.

In addition, since the design of the language is based on Emfrp [14], we can also write internal behaviors of physical sensor/actor nodes in XFRP. Thus, the language enables a uniform way to express whole (inter- and intra-node) behaviors of WSANs. This sort of uniformity is important because it eases the development process of WSANs as follows. First, we can construct a prototype of a WSAN as a single module that defines the entire (inter- and intra node) behaviors of the WSAN. The module has no host specifiers initially and the source and sink nodes are connected to some debug/test stubs written in Erlang. After the local testing, we can gradually deploy nodes (time-varying values) to actual physical sensor/actor nodes by incrementally adding host specifiers and source unifiers to the module definition.

5 RELATED WORK

DREAM [8, 9] 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 [13] is a functional reactive library implemented in Scala. SID-UP (Source IDentifier Update Propagation) [4, 5] is an efficient propagation algorithm for distributed reactive programs in REScala and it supports complete glitch-freedom while the execution model is iterative.

Recently, a new method for REScala is proposed [10] and it provides fault tolerance for distributed reactive programming with reasonable performance. Besides, Myter et al. proposed another method for handling partial failures in distributed reactive systems [11]. However, these methods focus on node crashes rather than on network inconsistencies.

Regiment [12] is a functional macroprogramming language for wireless sensor networks (WSNs). The language enables us to write a WSN as a whole via functional reactive programming. However, it does not provide mechanisms that support actor nodes.

Listing 2: XFRP Code for Figure 2

```
module WSANExample
in  % temperature sensors
    t1 : Float, t3 : Float, t5 : Float,
    % humidity sensors
    h2 : Float, h4 : Float, h6 : Float,
    % threshold
    th : Float
out % temperature controllers
    tc1 : Bool, tc3 : Bool, tc5 : Bool,
    % humidity controllers
    hc2 : Bool, hc4 : Bool, hc6 : Bool

% discomfort index
fun di(t, h) = 0.81 * t
  + 0.01 * h * (0.99 * t - 14.3) + 46.3

% discomfort index nodes
node di12 = di(t1, h2)
node di23 = di(t3, h2)
node di34 = di(t3, h4)
node di45 = di(t5, h4)
node di56 = di(t5, h6)

% di-difference nodes
node ddi2 = di12 - di23
node ddi3 = di23 - di34
node ddi4 = di34 - di45
node ddi5 = di45 - di56

% controller nodes
node tc1 = ddi2 > th
node hc2 = ddi3 > th
node tc3 = ddi2 < -th || ddi4 > th
node hc4 = ddi3 < -th || ddi5 > th
node tc5 = ddi4 < -th
node hc6 = ddi5 < -th
```

6 CONCLUDING REMARKS

In this paper, we briefly describe a distributed dialect of the functional reactive programming language XFRP whose runtime system is based on the Actor-model. The updating algorithm of time-varying values employs asynchronous message passing for change propagation. The algorithm guarantees single-source glitch-freedom in a distributed system with the existence of out-of-order delivery of messages.

The case study showed that a wireless sensor-actor network (WSAN) application — typical application with coordination — can be straightforwardly written in the language. In addition, we discussed the possibility of an incremental development method for distributed applications.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press. http://mitpress.mit.edu/books/actors

[2] Ian F. Akyildiz and Ismail H. Kasimoglu. 2004. Wireless sensor and actor networks: research challenges. *Ad-hoc Networks* 2, 4 (2004), 351–367. https://doi.org/10.1016/j.adhoc.2004.04.003

[3] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A Survey on Reactive Programming. *Comput. Surveys* 45, 4 (2013), 52:1–52:34. https://doi.org/10.1145/2501654.2501666

[4] Joscha Drechsler and Guido Salvaneschi. 2014. Optimizing Distributed REScala. In *Workshop on Reactive and Event-based Languages and Systems (REBLS '14).*

[5] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2014).* ACM, ACM, 361–376. https://doi.org/10.1145/2660193.2660240

[6] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997).* ACM, 263–273. https://doi.org/10.1145/258949.258973

[7] Maryam Kamali, Linas Laibinis, Luigia Petre, and Kaisa Sere. 2014. Formal development of wireless sensor–actor networks. *Science of Computer Programming* 80 (2014), 25—49. https://doi.org/10.1016/j.scico.2012.03.002

[8] Alessandro Margara and Guido Salvaneschi. 2014. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *8th ACM International Conference on Distributed Event-Based Systems (DEBS 2014).* ACM, 142–153. https://doi.org/10.1145/2611286.2611290

[9] Alessandro Margara and Guido Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (Jul. 2018), 689–711. https://doi.org/10.1109/TSE.2018.2833109

[10] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (LIPIcs, Vol. 109).* Schloss Dagstuhl, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

[11] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2017. Handling Partial Failures in Distributed Reactive Programming. In *4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017).* ACM, 1–7. https://doi.org/10.1145/3141858.3141859

[12] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The Regiment Macroprogramming System. In *6th International Conference on Information Processing in Sensor Networks (IPSN '07).* ACM, 489–498. https://doi.org/10.1145/1236360.1236422

[13] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *13th International Conference on Modularity (Modularity 2014).* ACM, 25–36. https://doi.org/10.1145/2577080.2577083

[14] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems. In *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity.* ACM, 36–44. https://doi.org/10.1145/2892664.2892670

[15] Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed Functional Reactive Programming on Actor-Based Runtime. In *8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018).* ACM, 13–22. https://doi.org/10.1145/3281366.3281370

[16] Sam Van den Vonder, Joeri De Koster, Florian Myter, and Wolfgang De Meuter. 2017. Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model. In *4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017).* ACM, 27–33. https://doi.org/10.1145/3141858.3141863

[17] Takuo Watanabe. 2018. A Simple Context-Oriented Programming Extension to an FRP Language for Small-Scale Embedded Systems. In *10th International Workshop on Context-Oriented Programming (COP 2018).* ACM, 23–30. https://doi.org/10.1145/3242921.3242925

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