Towards a Functional Reactive Programming Model for Developing WSANs

Takuo Watanabe[1,a)] Kazuhiro Shibanai[1,2]

Abstract: Functional reactive programming (FRP) is a programming paradigm where a system is described using declarative abstractions of the change propagation of discrete events and continuous signals. This paper presents a purely functional reactive programming model that facilitates a uniform description of distributed coordination and per-node computation. A case study of a wireless sensor-actor network (WSAN) shows that both inter-node coordination and intra-node computation can uniformly be written as reactive behaviors. The paper also describes an implementation method of the model using Distributed XFRP, a pure FRP language for distributed systems.

Keywords: functional reactive programming, embedded systems, coordination, wireless sensor-actor networks

1. Introduction

Functional Reactive Programming (FRP) is a programming paradigm to support the development of reactive systems such as embedded systems and GUIs. In FRP, reactive behaviors are described as change propagation among *time-varying values* (aka *signals*) that represent continuously changing values or discrete events [3]. Originally, FRP is introduced to describe interactive animations in the purely functional language Haskell [6]. Until now, the paradigm and its non-functional variants have 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. We 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 [15]. 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 execution model of an FRP language as a concurrent runtime system for resource-constrained uniprocessor systems, Distributed XFRP actually supports distributed execution of pure FRP programs.

The main contribution of this paper is to show that FRP is suitable for describing distributed coordination via a case study on a wireless sensor-actor network (WSAN) that controls the air-environment of a long corridor. Since the program of the WSAN is written in Distributed XFRP, inter-node communication required to achieve the coordination is realized in a straightforward manner by time-varying values that support single-source glitch freedom. Moreover, the incremental development of such applications is discussed.

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, the execution model of Distributed XFRP is described briefly. 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. Before presenting how a WSAN example is developed using Distributed XFRP [15] in Section 4, this section briefly describes a non-distributed subset of the language.

[1] Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[2] Currently with BrainPad Inc.
[a)] takuo@acm.org

[*1] `https://github.com/45deg/distributed-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 = 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
 8
 9 % fan status
10 node init[False] fan = di >= th
11
12 % threshold
13 node th = 75.0 + if fan@last then -0.5 else 0.5
```

Fig. 1 Fan Controller in XFRP

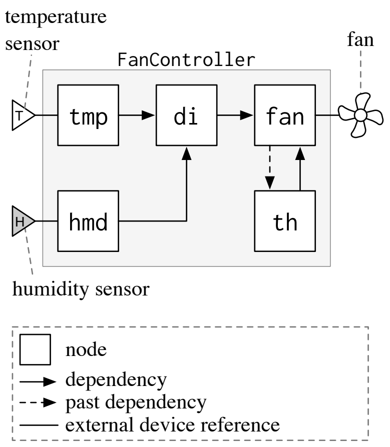

Fig. 2 Graph Representation of FanController

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 results and normally affect the outer world by, for example, 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 effects or mutable states. In summary, changes of sources propagate throughout internal nodes and finally reach sinks.

Fig. 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 relative humidity [%] measured by external sensors. The node `di` expresses the current discomfort index (the degree of discomfort experienced by 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 Fig. 1, `@last` is used to realize a simple hysteresis control that protects the fan motor from frequent switching. The definition of `th` (line 13 in Fig. 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. Fig. 2 shows the graph representation of Fig. 1, which consists of five nodes and five edges. The edges (dependencies) are categorized 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 Fig. 2, 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) (*i.e.*, no cycles consisting of present edges are allowed.). The topological sorting on the DAG gives a sequence of the nodes. For Fig. 2, a possible sequence is: `tmp`, `hmd`, `di`, `th`, `fan`.

The runtime system of the language updates the values of the nodes by repeatedly evaluating the elements of the sequence. A single evaluation cycle is called an *iteration*. The order of updates (scheduling) in an iteration is compatible with the partial order determined by the above mentioned DAG.

The expression *n*`@last` is implemented to denote 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 Fig. 2, the initial value of `fan` is specified as `False` (line 10).

3. Distributed XFRP

Distributed XFRP [15] is a distributed dialect of XFRP. This section briefly introduces some important concepts of the language.

3.1 Glitches

Glitches are 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
```

A glitch-free system guarantees that the values of the all occurrences of the same node (time-varying value) are the same at every moment. Thus, in a glitch-free system, the value of `z` is always true. In contrast, in a system with glitches, this property does not hold due to the possible differences in the time of change propagation. In such a system, the value of `z` may have chances to be false.

Margara and Salvaneschi classified glitch-freedom into two types: *single-source* and *complete* [9]. The former guarantees that the changes in a single source (node) are propagated to its dependent nodes without glitches but other sources are not considered. 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

This subsection briefly explains the execution model of Distributed XFRP, The detailed description can be found in our previous paper [15].

The runtime system of XFRP is based on the Actor model [1]. Each source, sink, or internal node is implemented as a single actor. Change propagation of time-varying values is realized using asynchronous messages. 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.

As described in Section 2.2, the dependency between nodes is represented as a directed graph. In the actor-based implementation, 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). As in XFRP, cycles consisting of present edges are not permitted. A *root* of a node is a source that has at least one present edge path to the node.

Each source has a counter that increments when it sends a new value. The pair of the ID of the source and the value of its counter is associated with the propagation messages to keep track of the happened-before relation between changes. The associated information is called *version*. The relation from version to the value of a node is surjective. If the value of a node n results in the value v by receiving a message with version (s, i), the value of n is said to be v at version (s, i). The use of the IDs of the sources in versions enables the single-source glitch-freedom in our language [15].

Each actor implementing a node has special internal values *Buffer*, *Last*, and *Deferred* to represent its computational states. Buffer holds the received values before they are processed. 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).

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

Fig. 3 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.

The behavior of the WSAN is described as follows: 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 Fig. 3 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

Fig. 4 shows the XFRP code for the example. The code is a straightforward implementation[*3] of Fig. 3. For simplicity reason, the host specifiers in the code is omitted. 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 Fig. 4) or at the node definitions (lines 18–22, 25–28 and 31–36). The single-source

[*2] The term "node" is used 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.

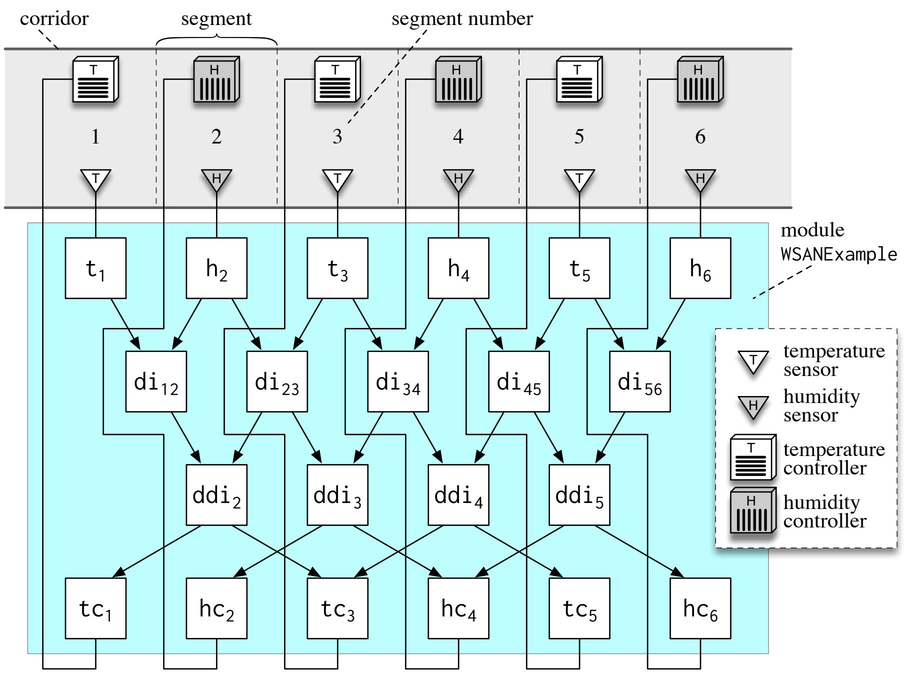

Fig. 3 WSAN for Regulating the Air-Environment of a Corridor

```
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
```

Fig. 4 XFRP Code for Fig. 3

glitch-free property of the language guarantees that no temporal inconsistencies can be observed in any deployment configuration.

Using a simple example scenario, the rest of this subsection demonstrates the effectiveness of the single-source glitch-free property in the WSAN. In the following, the value of `th` is fixed to 2.0 and assume that $\texttt{ddi5} \leq \texttt{th}$. Suppose that `h2` = 70.0, `t3` = 24.0, and `h4` = 75.0. From the node definitions in Fig. 4, we have `di23` = 72.36, `di34` = 72.84, and `ddi3` = -0.48. Thus, neither `hc2` nor `hc4` is activated because $\texttt{th} \geq \texttt{ddi3} \geq -\texttt{th}$ and $\texttt{ddi5} \leq \texttt{th}$. 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 $\texttt{ddi3} = -0.52 \geq -\texttt{th}$. However, without the single-source glitch-free property, it may be 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 may hold. In such case, `hc4` is incorrectly activated because $\texttt{ddi3} = -2.03 < -\texttt{th}$ hold.

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 macro-programming 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 the 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 supports multiple 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 realized in REScala and it supports complete glitch-freedom while the execution model is iterative.

Recently, a new propagation method for REScala is proposed [10]. The method 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.

6. Concluding Remarks

This paper emphasizes that, via a case study on a wireless sensor-actor network (WSAN) written in Distributed XFRP, pure functional reactive programming (FRP) with distribution support is suitable for developing coordinating distributed applications. 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. Besides, Distributed XFRP supports inter-node communication via time-varying values. This language feature enables us to write a distributed application as if it were a non-distributed program. Problems regarding synchronization are partially resolved thanks to the single-source glitch freedom realized by the language's runtime system. Also, an incremental development method for distributed applications is discussed.

Evaluation through the development of actual distributed applications remains for future work. One of the challenges is investigating the possibility of distributed FRP in coordinating distributed applications other than WSAN.

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

References

[1] Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press (1986).

[2] Akyildiz, I. F. and Kasimoglu, I. H.: Wireless sensor and actor networks: research challenges, *Ad-hoc Networks*, Vol. 2, No. 4, pp. 351–367 (online), DOI: 10.1016/j.adhoc.2004.04.003 (2004).

[3] 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, pp. 52:1–52:34 (online), DOI: 10.1145/2501654.2501666 (2013).

[4] Drechsler, J. and Salvaneschi, G.: Optimizing Distributed REScala, *Workshop on Reactive and Event-based Languages and Systems (REBLS '14)* (2014).

[5] 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, pp. 361–376 (online), DOI: 10.1145/2660193.2660240 (2014).

[6] Elliott, C. and Hudak, P.: Functional Reactive Animation, *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, ACM, pp. 263–273 (online), DOI: 10.1145/258949.258973 (1997).

[7] Kamali, M., Laibinis, L., Petre, L. and Sere, K.: Formal development of wireless sensor–actor networks, *Science of Computer Programming*, Vol. 80, pp. 25—49 (online), DOI: 10.1016/j.scico.2012.03.002 (2014).

[8] Margara, A. and Salvaneschi, G.: We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees, *8th ACM International Conference on Distributed Event-Based Systems (DEBS 2014)*, ACM, pp. 142–153 (online), DOI: 10.1145/2611286.2611290 (2014).

[9] 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, pp. 689–711 (online), DOI: 10.1109/TSE.2018.2833109 (2018).

[10] Mogk, R., Baumgärtner, L., Salvaneschi, G., Freisleben, B. and Mezini, M.: Fault-tolerant Distributed Reactive Programming, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, LIPIcs, Vol. 109, Schloss Dagstuhl, pp. 1:1–1:26 (online), DOI: 10.4230/LIPIcs.ECOOP.2018.1 (2018).

[11] Myter, F., Scholliers, C. and De Meuter, W.: Handling Partial Failures in Distributed Reactive Programming, *4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017)*, ACM, pp. 1–7 (online), DOI: 10.1145/3141858.3141859 (2017).

[12] Newton, R., Morrisett, G. and Welsh, M.: The Regiment Macroprogramming System, *6th International Conference on Information Processing in Sensor Networks (IPSN '07)*, ACM, pp. 489–498 (online), DOI: 10.1145/1236360.1236422 (2007).

[13] Salvaneschi, G., Hintz, G. and Mezini, M.: REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications, *13th International Conference on Modularity (Modularity 2014)*, ACM, pp. 25–36 (online), DOI: 10.1145/2577080.2577083 (2014).

[14] 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, pp. 36–44 (online), DOI: 10.1145/2892664.2892670 (2016).

[15] 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, pp. 13–22 (online), DOI: 10.1145/3281366.3281370 (2018).

[16] Van den Vonder, S., De Koster, J., Myter, F. and De Meuter, W.: Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model, *4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017)*, ACM, pp. 27–33 (online), DOI: 10.1145/3141858.3141863 (2017).

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

[18] 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)* (2016).